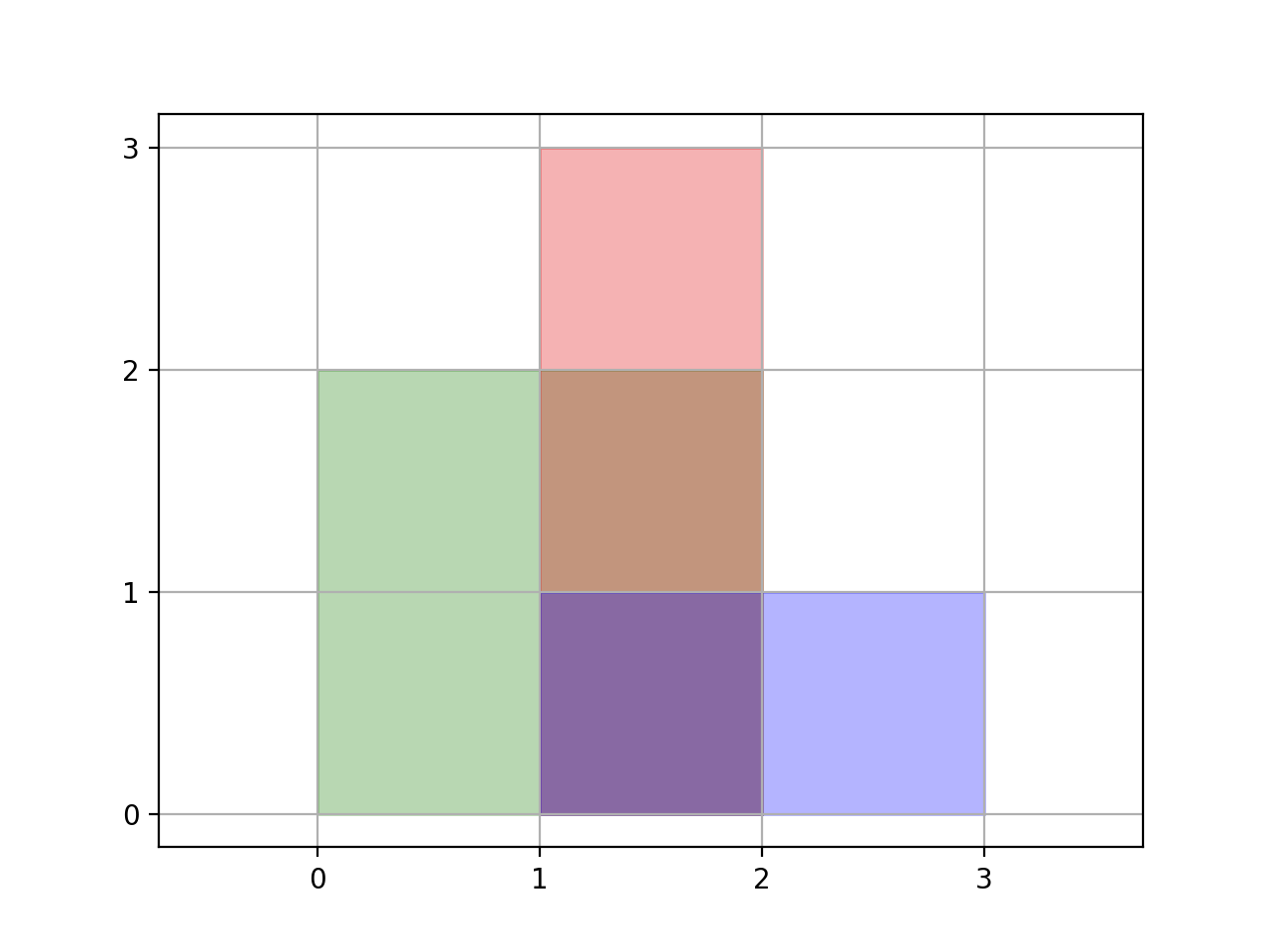
<https://leetcode.com/problems/rectangle-area-ii/description/>

You are given a 2D array of axis-aligned rectangles. Each rectangle[i] = [xi1, yi1, xi2, yi2] denotes the ith rectangle where (xi1, yi1) are the coordinates of the **bottom-left corner**, and (xi2, yi2) are the coordinates of the **top-right corner**.

Calculate the **total area** covered by all rectangles in the plane. Any area covered by two or more rectangles should only be counted **once**.

Return **the total area**. Since the answer may be too large, return it **modulo** 109 + 7.

**Example 1:**



**Input:** rectangles = [[0,0,2,2],[1,0,2,3],[1,0,3,1]]

**Output:** 6

**Explanation:** A total area of 6 is covered by all three rectangles, as illustrated in the picture.

From (1,1) to (2,2), the green and red rectangles overlap.

From (1,0) to (2,3), all three rectangles overlap.

**Example 2:**

**Input:** rectangles = [[0,0,1000000000,1000000000]]

**Output:** 49

**Explanation:** The answer is 1018 modulo (109 + 7), which is 49.

**Constraints:**

* 1 <= rectangles.length <= 200
* rectanges[i].length == 4
* 0 <= xi1, yi1, xi2, yi2 <= 10^9
* xi1 <= xi2
* yi1 <= yi2

**Attempt 1: 2023-12-20**

**Solution 1: Recursion (180min)**

class Solution {

    public int rectangleArea(int[][] rectangles) {

        int MOD = (int)1e9 + 7;

        List<int[]> list = new ArrayList<>();

        for(int[] rectangle : rectangles) {

            addRectangle(list, rectangle, 0);

        }

        long result = 0;

        for(int[] rectangle : list) {

            result = (result + (long)(rectangle[2] - rectangle[0]) \* (long)(rectangle[3] - rectangle[1])) % MOD;

        }

        return (int) result;

    }

    private void addRectangle(List<int[]> list, int[] curRec, int compareRecIndex) {

        // After comparing all exist rectangles(includes all new generated case

        // 1 to case 4 rectangles) on list, we will add the new rectangle(either

        // original one or split into one of 4 cases piece after comparison)

        if(compareRecIndex >= list.size()) {

            list.add(curRec);

            return;

        }

        int[] compareRec = list.get(compareRecIndex);

        // If no overlap with current compared rectangle, we need to move on the

        // comparison with next rectangle if any

        if(compareRec[0] >= curRec[2] || compareRec[2] <= curRec[0] || compareRec[1] >= curRec[3] || compareRec[3] <= curRec[1]) {

            addRectangle(list, curRec, compareRecIndex + 1);

            return;

        }

        // Case 1: Overlap with compared rectangle on left side

        if(compareRec[0] > curRec[0]) {

            addRectangle(list, new int[]{curRec[0], curRec[1], compareRec[0], curRec[3]}, compareRecIndex + 1);

            // Don't return directly because the interection may not be only

            // Case 1, we have to check all 4 Cases if any other overlap happens

        }

        // Case 2: Overlap with compared rectangle on right side

        if(compareRec[2] < curRec[2]) {

            addRectangle(list, new int[]{compareRec[2], curRec[1], curRec[2], curRec[3]}, compareRecIndex + 1);

        }

        // Case 3: Overlap with compared rectangle on bottom side

        if(compareRec[1] > curRec[1]) {

            addRectangle(list, new int[]{Math.max(compareRec[0], curRec[0]), curRec[1], Math.min(compareRec[2], curRec[2]), compareRec[1]}, compareRecIndex + 1);

        }

        // Case 4: Overlap with compared rectangle on top side

        if(compareRec[3] < curRec[3]) {

            addRectangle(list, new int[]{Math.max(compareRec[0], curRec[0]), compareRec[3], Math.min(compareRec[2], curRec[2]), curRec[3]}, compareRecIndex + 1);

        }

    }

}

Time Complexity: O(N^2), where N is the number of rectangles

Space Complexity: O(N)

**Refer to**

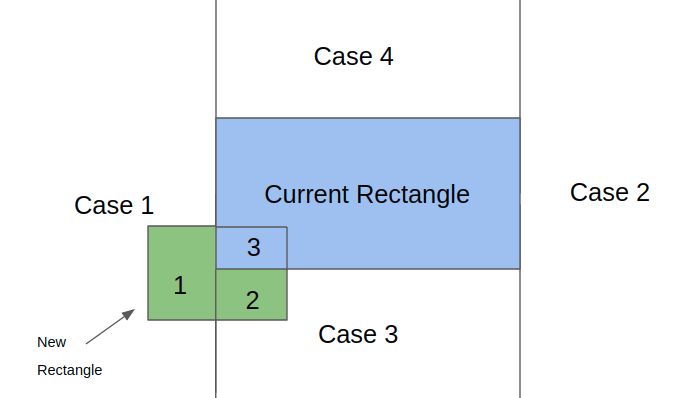
<https://leetcode.com/problems/rectangle-area-ii/solutions/138028/clean-recursive-solution-java/>

The idea here is to maintain a list of non-overlapping rectangles to calculate final area.

If a new rectangle does not overlap with any of the existing rectanlges, add it to the list.

If there is an overlap, split the non-overlapping regions of the rectangle into smaller rectangles and compare with the rest of the list.

For example, when a new rectangle (green) is compared with the current rectangle (blue), the non-overlapping regions can be split into two smaller rectangles. Rectangle 1 will be covered by the first overlapping case in addRectangle() and rectangle 2 will be covered by the third case. Rectangle 3 overlaps with the current rectangle and need not be considered.



    public int rectangleArea(int[][] rectangles) {

        int mod = (int)Math.pow(10,9)+7;

        long res = 0;

        List<int[]> recList = new ArrayList<>();

        for(int[] rec : rectangles)

            addRectangle(recList, rec, 0);

        for(int[] rec: recList)

            res = (res+((long)(rec[2]-rec[0])\*(long)(rec[3]-rec[1])))%mod;

        return (int) res%mod;

    }

    // Add new rectangle to the list. In case of overlap break up new rectangle into

    // non-overlapping rectangles. Compare the new rectanlges with the rest of the list.

    public void addRectangle(List<int[]> recList, int[] curRec, int start){

        if(start>=recList.size()){

            recList.add(curRec);

            return;

        }

        int[] r = recList.get(start);

        // No overlap

        if(curRec[2]<=r[0] || curRec[3]<=r[1] || curRec[0]>=r[2] || curRec[1]>=r[3]){

            addRectangle(recList, curRec, start+1);

            return;

        }

        if( curRec[0]<r[0])

            addRectangle(recList, new int[]{curRec[0],curRec[1],r[0],curRec[3]},start+1);

        if(curRec[2]>r[2])

            addRectangle(recList, new int[]{r[2],curRec[1],curRec[2],curRec[3]},start+1);

        if(curRec[1]<r[1])

            addRectangle(recList, new int[]{Math.max(r[0],curRec[0]),curRec[1],Math.min(r[2],curRec[2]),r[1]},start+1);

        if(curRec[3]>r[3])

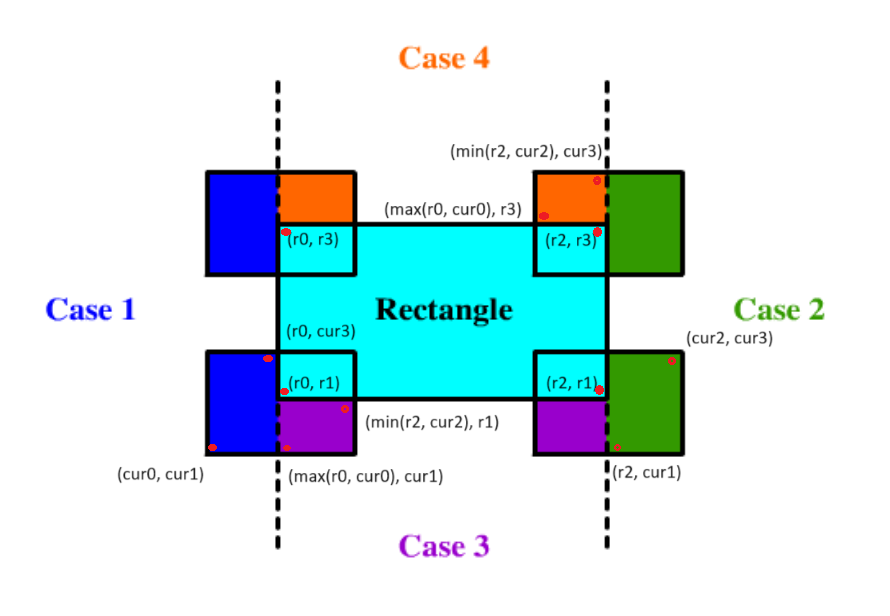
            addRectangle(recList, new int[]{Math.max(r[0],curRec[0]),r[3],Math.min(r[2],curRec[2]),curRec[3]},start+1);

    }

**Refer to**

<https://grandyang.com/leetcode/850/>

这道题是之前那道 [Rectangle Area](http://www.cnblogs.com/grandyang/p/4563153.html) 的拓展，那道题只有两个矩形重叠，而这道题有多个矩形可能同时重叠，整体难度一下就上来了，那么通过将所有矩形面积加起来再减去重叠区域的方法这里就不太适用了，因为多个矩形在同一区域重叠的话，都减去重叠面积是会错的，还得把多减的补回来，相当的麻烦。这里我们需要换一种解题的思路，不能一股脑儿的把所有的矩形都加起来，而是应该利用微积分的思想，将重叠在一起的区域拆分成一个个的小矩形，分别累加面积，因为这里的矩形是不会旋转的，所以是可以正常拆分的。思路有了，新建一个二维数组 all 来保存所有的矩形，然后遍历给定的矩形数组，对于每个遍历到的数组，调用一个子函数，将当前的矩形加入 all 中。下面主要来看一下这个子函数 helper 该如何实现？首先要明白这个函数的作用是将当前矩形加入 all 数组中，而且用的是递归的思路，所以要传入一个 start 变量，表示当前和 all 数组中正在比较的矩形的 index，这样在开始的时候，检查一下若 start 大于等于 all 数组长度，表示已经检测完 all 中所有的矩形了，将当前矩形加入 all 数组，并返回即可。否则的话则取出 start 位置上的矩形 rec，此时就要判断当前要加入的矩形和这个 rec 矩形是否有重叠，这在 LeetCode 中有专门一道题是考察这个的 [Rectangle Overlap](https://www.cnblogs.com/grandyang/p/10367583.html)，这里用的就是那道题的判断方法，假如判断出当前矩形 cur 和矩形 rec 没有交集，就直接对 all 数组中下一个矩形调用递归函数，并返回即可。假如有重叠的话，就稍微麻烦一点，由于重叠的部位不同，所以需要分情况讨论一下，参见下图所示：



对于一个矩形 Rectangle，若有另外一个矩形跟它有重叠的话，可以将重叠区域分为四个部分，如上图的 Case1，Case2，Case3，Case4 所示，非重叠部分一定会落在一个或多个区域中，则可以把这些拆开的小矩形全部加入到矩形数组 all 中。仔细观察上图可以发现，对于将矩形 cur 拆分的情况可以分为下面四种：

* 落入区间1，条件为 cur[0] < rec[0]，产生的新矩形的两个顶点为 {cur[0], cur[1], rec[0], cur[3]}。
* 落入区间2，条件为 cur[2] > rec[2]，产生的新矩形的两个顶点为 {rec[2], cur[1], cur[2], cur[3]}。
* 落入区间3，条件为 cur[1] < rec[1]，产生的新矩形的两个顶点为 {max(rec[0], cur[0]), cur[1], min(rec[2], cur[2]), rec[1]}。
* 落入区间4，条件为 cur[3] > rec[3]，产生的新矩形的两个顶点为 {max(rec[0], cur[0]), rec[3], min(rec[2], cur[2]), cur[3]}。

这样操作下来的话，整个所有的区域都被拆分成了很多个小矩形，每个矩形之间都不会有重复，最后只要分别计算每个小矩形的面积，并累加起来就是最终的结果了，参见代码如下：

解法一：

class Solution {

    public:

    int rectangleArea(vector<vector<int>>& rectangles) {

        long res = 0, M = 1e9 + 7;

        vector<vector<int>> all;

        for (auto rectangle : rectangles) {

            helper(all, rectangle, 0);

        }

        for (auto &a : all) {

            res = (res + (long)(a[2] - a[0]) \* (long)(a[3] - a[1])) % M;

        }

        return res;

    }

    void helper(vector<vector<int>>& all, vector<int> cur, int start) {

        if (start >= all.size()) {

            all.push\_back(cur); return;

        }

        auto rec = all[start];

        if (cur[2] <= rec[0] || cur[3] <= rec[1] || cur[0] >= rec[2] || cur[1] >= rec[3]) {

            helper(all, cur, start + 1); return;

        }

        if (cur[0] < rec[0]) {

            helper(all, {cur[0], cur[1], rec[0], cur[3]}, start + 1);

        }

        if (cur[2] > rec[2]) {

            helper(all, {rec[2], cur[1], cur[2], cur[3]}, start + 1);

        }

        if (cur[1] < rec[1]) {

            helper(all, {max(rec[0], cur[0]), cur[1], min(rec[2], cur[2]), rec[1]}, start + 1);

        }

        if (cur[3] > rec[3]) {

            helper(all, {max(rec[0], cur[0]), rec[3], min(rec[2], cur[2]), cur[3]}, start + 1);

        }

    }

};

**Solution 2: 2D Line Sweep (720min)**

**Style 1: ONLY with 'Event' helper class, but that leverage a trick on TreeMap customized comparator, the case is when we try to remove the 'event' from TreeMap when Line Sweep encounter 'close' type of 'event;, the custmozied comparator only use 'x\_start' and 'x\_end' as parameter to compare key, that will treat two even not same 'event' to the same one.**

**E.g if we define event = {x\_start, x\_end, y, type}, and we have two events as open rectangle event e2 = {11, 31, 83, 1} and close rectangle event e3 = {11, 31, 99, -1}, the case is if customized comparator in TreeMap only concern on 'x\_start' and 'x\_end', then since x\_start = 11 and x\_end = 31 in both e2 and e3, then these two events will recognized same, and the trick is we will able to add e2 onto TreeMap first, then remove e2 from TreeMap when we encounter e3.**

**To make it more strcit, we will roll out Style 2 wich includes another helper class 'Interval' to bypass the trick and make it standardable to restrict the key comparison**

class Solution {

    class Event {

        int x\_start;

        int x\_end;

        int y;

        // 'type' = 1 means open of rectangle

        // 'type' = -1 means close of rectangle

        int type;

        public Event(int x\_start, int x\_end, int y, int type) {

            this.x\_start = x\_start;

            this.x\_end = x\_end;

            this.y = y;

            this.type = type;

        }

    }

    public int rectangleArea(int[][] rectangles) {

        int MOD = (int)1e9 + 7;

        int open = 1;

        int close = -1;

        List<Event> events = new ArrayList<>();

        for(int[] rec : rectangles) {

            Event openRec = new Event(rec[0], rec[2], rec[1], open);

            Event closeRec = new Event(rec[0], rec[2], rec[3], close);

            events.add(openRec);

            events.add(closeRec);

        }

        // Sort events by y index

        Collections.sort(events, (a, b) -> a.y - b.y);

        // TreeMap 'delta' to simulate the Line Sweep process

        // Note: the sorting requirement is critical, first compare 'start',

        // if same 'start' then compare 'end'

        // e.g for rectangles = [[0,0,2,2],[1,0,2,3],[1,0,3,1]]

        // when y = 0, we will add three intervals {0,2}, {1,2}, {1,3} onto

        // 'delta', the correct order is {0,2}=1 -> {1,2}=1 -> {1,3}=1

        // If only sort by 'start' will error out on above example, since

        // TreeMap will not treat {1,3} different than {1,2}, the wrong

        // order is {0,2}=1 -> {1,2}=2, also only 'a.start - b.start' is

        // important to make sure sort all x-axis intervals with 'start',

        // when 'start' is same, the 'a.end - b.end' or 'b.end - a.end'

        // doesn't matter

        TreeMap<Event, Integer> delta = new TreeMap<>((a, b) -> a.x\_start == b.x\_start ? a.x\_end - b.x\_end : a.x\_start - b.x\_start);

        long result = 0;

        int prevY = 0;

        // Add event onto treemap 'delta' based on ascending order of y index

        // on y-axis to simulate similar behavior we usually did on x-axis

        for(Event event : events) {

            int curY = event.y;

            if(curY > prevY) {

                result = (result + calculateXIntervals(delta) \* (curY - prevY)) % MOD;

                prevY = curY;

            }

            if(event.type == open) {

                delta.put(event, delta.getOrDefault(event, 0) + 1);

            } else {

                delta.put(event, delta.getOrDefault(event, 0) - 1);

                if(delta.get(event) == 0) {

                    delta.remove(event);

                }

            }

        }

        return (int) result;

    }

    private long calculateXIntervals(TreeMap<Event, Integer> delta) {

        long result = 0;

        int cur = -1;

        for(Event e : delta.keySet()) {

            cur = Math.max(cur, e.x\_start);

            // Test out by:

            // Input rectangles = [[49,40,62,100],[11,83,31,99],[19,39,30,99]]

            // Use Testcase

            // Output = 1568, Expected = 1584

            //result += e.x\_end - cur;

            result += Math.max(e.x\_end - cur, 0);

            cur = Math.max(cur, e.x\_end);

        }

        return result;

    }

}

Time Complexity: O(N^2), where N is the number of rectangles

Space Complexity: O(N)

**Step by Step**

e.g

rectangles = {{49,40,62,100},{11,83,31,99},{19,39,30,99}}

                        {62,100}

     {30,99}{31,99} +-----+ e5 -> y = 100

  +-+--------+-+ e4/e3    |    -> y = 99    ^

  | |        | |    |     |                 |

{11,83}--------+ e2 |     |    -> y = 83    .

    |        |      |     |                 .

    |        |    {49,40} |                 .

  {19,39}    |      +-----+ e1 -> y = 40    .

    +--------+ e0              -> y = 39    .

  ^ ^        ^ ^    ^     ^                 |

  | |        | |    |     |               y-axis

 11 19      30 31   49    62  x-axis   Line Sweep from y = 39 to 100

======================================================================

 Events sorted by y: 39 -> 40 -> 83 -> 99 -> 99 -> 100

 e0 = {19,30,39,1} => {x\_start, x\_end, y, type}

 e1 = {49,62,40,1}

 e2 = {11,31,83,1}

 e3 = {11,31,99,-1}

 e4 = {19,30,99,-1}

 e5 = {49,62,100,-1}

 TreeMap<Event, Integer> delta = new TreeMap<>((a, b) -> a.x\_start == b.x\_start ? a.x\_end - b.x\_end : a.x\_start - b.x\_start);

 TreeMap sort first by x-axis start, if same then sort by x-axis end

 ======================================================================

 For e0:

 curY = 39, prevY = 0, 39 > 0

 TreeMap delta = {empty}

 calculateXIntervals(delta) =>

 result = 0

 prevY = curY = 39

 Total result += 0 \* (curY - prevY) = 0 + 0 \* (39 - 39) = 0

 event.type = open -> delta = {{e0,1}}

 So the 1st Line Sweep will add e0 onto delta as basement

----------------------------------------------------------------------

 For e1:

 curY = 40, prevY = 39, 40 > 39

 TreeMap delta = {{e0,1}}

 calculateXIntervals(delta) =>

 (1) check with e0

 cur = -1

 cur = Math.max(-1, 19) = 19

 result += Math.max(30 - 19, 0) = 11

 cur = Math.max(19, 30) = 30

 Total result += 11 \* (curY - prevY) = 0 + 11 \* (40 - 39) = 11

 prevY = curY = 40

 event.type = open -> delta = {{e0,1},{e1,1}}

 So the 2nd Line Sweep will add e1 onto delta, now delta has e0 and e1,

 the split out 1st piece size is 11 and exist between e0 and e1

 ----------------------------------------------------------------------

 For e2:

 curY = 83, prevY = 40, 83 > 40

 TreeMap delta = {{e0,1},{e1,1}}

 calculateXIntervals(delta) =>

 (1) check with e0

 cur = -1

 cur = Math.max(-1, 19) = 19

 result += Math.max(30 - 19, 0) = 11

 cur = Math.max(19, 30) = 30

 (2) check with e1

 cur = 30

 cur = Math.max(30, 49) = 49

 result += Math.max(62 - 49, 0) = 11 + 13 = 24

 cur = Math.max(49, 62) = 62

 Total result += 24 \* (curY - prevY) = 11 + 24 \* (83 - 40) = 1043

 prevY = curY = 83

 event.type = open -> delta = {{e2,1},{e0,1},{e1,1}}

 So the 3rd Line Sweep will add e2 onto delta, now delta has e0, e1 and e2,

 the split out 2nd + 3rd pieces size together is (11 + 13) \* (83 - 40) = 1032 and

 exist between e1 and e2

 ----------------------------------------------------------------------

 For e3:

 curY = 99, prevY = 83, 99 > 83

 TreeMap delta = {{e2,1},{e0,1},{e1,1}}

 calculateXIntervals(delta) =>

 (1) check with e2

 cur = -1

 cur = Math.max(-1, 11) = 11

 result += Math.max(31 - 11, 0) = 20

 cur = Math.max(11, 31) = 31

 (2) check with e0

 cur = 31

 cur = Math.max(31, 19) = 31

 result += Math.max(30 - 31, 0) = 20 + 0 = 20

 -> x-axis -1 special case happen, because we first proceed x-axis range

 calculation with e2, in e2 x-axis range is [11,31], then we proceed with

 e0, in e0 x-axis range is [19,30], since 11 < 19 < 30 < 31, e0 is fully

 cover by e2, so when we first get e2 end = 31, then get e0 end = 30, we

 will ignore e0 end

 cur = Math.max(31, 30) = 31

 (3) check with e1

 cur = 31

 cur = Math.max(31, 49) = 49

 result += Math.max(62 - 49, 0) = 20 + 13 = 33

 cur = Math.max(49, 62) = 62

 Total result += 33 \* (curY - prevY) = 1043 + 33 \* (99 - 83) = 1571

 prevY = curY = 99

 event.type = close -> delta = {{e2,1},{e0,1},{e1,1}} => delta = {{e0,1},{e1,1}}

 so when we proceed e3, {e2,1} removed by designiation as its a close of

 the rectangle when open with e2

 So the 4th Line Sweep will remove e2 from delta, now delta has e0 and e1,

 the split out 4th + 5th pieces size together is (20 + 13) \* (99 - 83) = 528 and

 exist between e2 and e3

 ----------------------------------------------------------------------

 Note: Why in TreeMap 'delta', it treat two events e2, e3 the same ?

 e2 = {11,31,83,1}

 e3 = {11,31,99,-1}

 because even for e2 and e3 their y and type are different, but in customized

 comparator, we only interest in x\_start and x\_end, and because 11, 31 are common

 for both e2 and e3, the TreeMap with customized comparator will treat e2, e3

 the same key

 TreeMap<Event, Integer> delta = new TreeMap<>((a, b) -> a.x\_start == b.x\_start ? a.x\_end - b.x\_end : a.x\_start - b.x\_start);

 ----------------------------------------------------------------------

 ...etc for e4, e5

**Style 2: wich includes another helper class 'Interval' to bypass the trick and make it standardable to restrict the key comparison**

class Solution {

    class Event {

        int x\_start;

        int x\_end;

        int y;

        // 'type' = 1 means open of rectangle

        // 'type' = -1 means close of rectangle

        int type;

        public Event(int x\_start, int x\_end, int y, int type) {

            this.x\_start = x\_start;

            this.x\_end = x\_end;

            this.y = y;

            this.type = type;

        }

    }

    class Interval {

        int x\_start;

        int x\_end;

        public Interval(int x\_start, int x\_end) {

            this.x\_start = x\_start;

            this.x\_end = x\_end;

        }

    }

    public int rectangleArea(int[][] rectangles) {

        int MOD = (int)1e9 + 7;

        int open = 1;

        int close = -1;

        List<Event> events = new ArrayList<>();

        for(int[] rec : rectangles) {

            Event openRec = new Event(rec[0], rec[2], rec[1], open);

            Event closeRec = new Event(rec[0], rec[2], rec[3], close);

            events.add(openRec);

            events.add(closeRec);

        }

        // Sort events by y index

        Collections.sort(events, (a, b) -> a.y - b.y);

        // TreeMap 'delta' to simulate the Line Sweep process

        // Note: the sorting requirement is critical, first compare 'start',

        // if same 'start' then compare 'end'

        // e.g for rectangles = [[0,0,2,2],[1,0,2,3],[1,0,3,1]]

        // when y = 0, we will add three intervals {0,2}, {1,2}, {1,3} onto

        // 'delta', the correct order is {0,2}=1 -> {1,2}=1 -> {1,3}=1

        // If only sort by 'start' will error out on above example, since

        // TreeMap will not treat {1,3} different than {1,2}, the wrong

        // order is {0,2}=1 -> {1,2}=2, also only 'a.start - b.start' is

        // important to make sure sort all x-axis intervals with 'start',

        // when 'start' is same, the 'a.end - b.end' or 'b.end - a.end'

        // doesn't matter

        TreeMap<Interval, Integer> delta = new TreeMap<>((a, b) -> a.x\_start == b.x\_start ? a.x\_end - b.x\_end : a.x\_start - b.x\_start);

        long result = 0;

        int prevY = 0;

        // Add event onto treemap 'delta' based on ascending order of y index

        // on y-axis to simulate similar behavior we usually did on x-axis

        for(Event event : events) {

            int curY = event.y;

            if(curY > prevY) {

                result = (result + calculateXIntervals(delta) \* (curY - prevY)) % MOD;

                prevY = curY;

            }

            if(event.type == open) {

                Interval interval = new Interval(event.x\_start, event.x\_end);

                delta.put(interval, delta.getOrDefault(interval, 0) + 1);

            } else {

                Interval interval = new Interval(event.x\_start, event.x\_end);

                delta.put(interval, delta.getOrDefault(interval, 0) - 1);

                if(delta.get(interval) == 0) {

                    delta.remove(interval);

                }

            }

        }

        return (int) result;

    }

    private long calculateXIntervals(TreeMap<Interval, Integer> delta) {

        long result = 0;

        int cur = -1;

        for(Interval e : delta.keySet()) {

            cur = Math.max(cur, e.x\_start);

            // Test out by:

            // Input rectangles = [[49,40,62,100],[11,83,31,99],[19,39,30,99]]

            // Use Testcase

            // Output = 1568, Expected = 1584

            //result += e.x\_end - cur;

            result += Math.max(e.x\_end - cur, 0);

            cur = Math.max(cur, e.x\_end);

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/rectangle-area-ii/solutions/188832/java-line-sweep-with-sub-class-interval/>This solution is borrowed from the leetcode official solution number 3: line sweep. But the styling of that solution is pretty bad, so I rewrote it, added a sub class called Interval, and used TreeMap for addition & removal, which should reduce both operations complexity to O(logn).

1. order every line from the rectangle by y index. mark start of rectangle line (bottom) as OPEN, mark end of rectangle line (top) as CLOSE.
2. sweep line from bottom to top, each time y coordinate changed, means all intervals on current y is sweeped, merge the length back together, multiply by the y coordinate diff.

class Solution {

    private static class Interval {

        public int start;

        public int end;

        public Interval(int start, int end) {

            this.start = start;

            this.end = end;

        }

    }

    public int rectangleArea(int[][] rectangles) {

        int OPEN = 0, CLOSE = 1;

        int[][] events = new int[rectangles.length \* 2][4];

        int t = 0;

        /\*\*

        open of rectangle: add to active set

        close of rectangle: remove from active set

        \*/

        for (int[] rec: rectangles) {

            // y, open\_or\_close, start, end

            events[t++] = new int[]{ rec[1], OPEN, rec[0], rec[2] };

            events[t++] = new int[]{ rec[3], CLOSE, rec[0], rec[2] };

        }

        /\*\*

        sort by current y index

        \*/

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

        TreeMap<Interval, Integer> active = new TreeMap<>((a, b) -> {

            if (a.start != b.start) return a.start - b.start;

            return a.end - b.end;

        });

        // first y coordinate at the bottom

        int currentY = events[0][0];

        long ans = 0;

        for (int[] event : events) {

            int y = event[0], typ = event[1], x1 = event[2], x2 = event[3];

            // Calculate sum of intervals in active set, that's the active intervals in prev line

            if (y > currentY) {

                ans += calculateInterval(active) \* (y - currentY);

                currentY = y;

            }

            /\*\*

            add or remove new interval to current active

            \*/

            if (typ == OPEN) {

                addInterval(active, x1, x2);

            } else {

                removeInterval(active, x1, x2);

            }

        }

        ans %= 1\_000\_000\_007;

        return (int) ans;

    }

    /\*\*

    using tree map, should be able to insert in logn time

    \*/

    private void addInterval(TreeMap<Interval, Integer> map, int x1, int x2) {

        Interval interval = new Interval(x1, x2);

        map.put(interval, map.getOrDefault(interval, 0) + 1);

    }

    /\*\*

    using tree map, should be able to remove in logn time

    \*/

    private void removeInterval(TreeMap<Interval, Integer> map, int x1, int x2) {

        Interval interval = new Interval(x1, x2);

        map.put(interval, map.getOrDefault(interval, 0) - 1);

        if (map.get(interval) == 0) map.remove(interval);

    }

    private long calculateInterval(TreeMap<Interval, Integer> map) {

        long query = 0;

        int cur = -1;

        for (Interval interval : map.keySet()) {

            cur = Math.max(cur, interval.start);

            query += Math.max(interval.end - cur, 0);

            cur = Math.max(cur, interval.end);

        }

        return query;

    }

}

**另一种风格，但是不像前一种风格那么直接**

**Refer to**

**Java TreeMap solution inspired by Skyline and Meeting Room**

<https://leetcode.com/problems/rectangle-area-ii/solutions/137941/java-treemap-solution-inspired-by-skyline-and-meeting-room/>

1. Sort the points in x order
2. For the points in the same x, calculate the current y (like the meeting room problem).
3. In the next x, calculate the area by preY \* (curX - preX)

The complexity in the worst case is O(N ^ 2) (all the rectangles have the same x)

class Solution {

    class Point {

        int x, y, val;

        Point(int x, int y, int val) {

            this.x = x;

            this.y = y;

            this.val = val;

        }

    }

    public int rectangleArea(int[][] rectangles) {

        int M = 1000000007;

        List<Point> data = new ArrayList<>();

        for (int[] r : rectangles) {

            data.add(new Point(r[0], r[1], 1));

            data.add(new Point(r[0], r[3], -1));

            data.add(new Point(r[2], r[1], -1));

            data.add(new Point(r[2], r[3], 1));

        }

        Collections.sort(data, (a, b) -> {

            if (a.x == b.x) {

                return b.y - a.y;

            }

            return a.x - b.x;

        });

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

        int preX = -1;

        int preY = -1;

        int result = 0;

        for (int i = 0; i < data.size(); i++) {

            Point p = data.get(i);

            map.put(p.y, map.getOrDefault(p.y, 0) + p.val);

            if (i == data.size() - 1 || data.get(i + 1).x > p.x) {

                if (preX > -1) {

                    result += ((long)preY \* (p.x - preX)) % M;

                    result %= M;

                }

                preY = calcY(map);

                preX = p.x;

            }

        }

        return result;

    }

    private int calcY(TreeMap<Integer, Integer> map) {

        int result = 0, pre = -1, count = 0;

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

            if (pre >= 0 && count > 0) {

                result += e.getKey() - pre;

            }

            count += e.getValue();

            pre = e.getKey();

        }

        return result;

    }

}

**Refer to**

<https://grandyang.com/leetcode/850/>

下面这种解法更是利用了微积分的原理，把x轴长度为1当作一个步长，然后计算每一列有多少个连续的区间，每个连续区间又有多少个小正方形，题目中给的例子每一个列都只有一个连续区间，但事实上是可以有很多个的，只要算出了每一列 1x1 小正方形的个数，将所有列都累加起来，就是整个区域的面积。这里求每列上小正方形个数的方法非常的 tricky，博主也不知道该怎么讲解，大致就是要求同一列上每个连续区间中的小正方形个数，再累加起来。对于每个矩形起始的横坐标，映射较低的y值到1，较高的y值到 -1，对于结束位置的横坐标，刚好反过来一下，映射较低的y值到 -1，较高的y值到1。这种机制跟之前那道 [The Skyline Problem](http://www.cnblogs.com/grandyang/p/4534586.html) 有些异曲同工之妙，都还是为了计算高度差服务的。要搞清楚这道题的核心思想，不是一件容易的事，博主的建议是就拿题目中给的例子带入到下面的代码中，一步一步执行，并分析结果，是能够初步的了解解题思路的，若实在有理解上的问题，博主可以进一步写些讲解，参见代码如下：

解法二：

class Solution {

    public:

    int rectangleArea(vector<vector<int>>& rectangles) {

        long res = 0, pre\_x = 0, height = 0, start = 0, cnt = 0, M = 1e9 + 7;

        map<int, vector<pair<int, int>>> groupMap;

        map<int, int> cntMap;

        for (auto &a : rectangles) {

            groupMap[a[0]].push\_back({a[1], 1});

            groupMap[a[0]].push\_back({a[3], -1});

            groupMap[a[2]].push\_back({a[1], -1});

            groupMap[a[2]].push\_back({a[3], 1});

        }

        for (auto &group : groupMap) {

            res = (res + (group.first - pre\_x) \* height) % M;

            for (auto &a : group.second) {

                cntMap[a.first] += a.second;

            }

            height = 0, start = 0, cnt = 0;

            for (auto &a : cntMap) {

                if (cnt == 0) start = a.first;

                cnt += a.second;

                if (cnt == 0) height += a.first - start;

            }

            pre\_x = group.first;

        }

        return res;

    }

};