<https://leetcode.com/problems/the-skyline-problem/description/>

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return **the skyline formed by these buildings collectively**.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

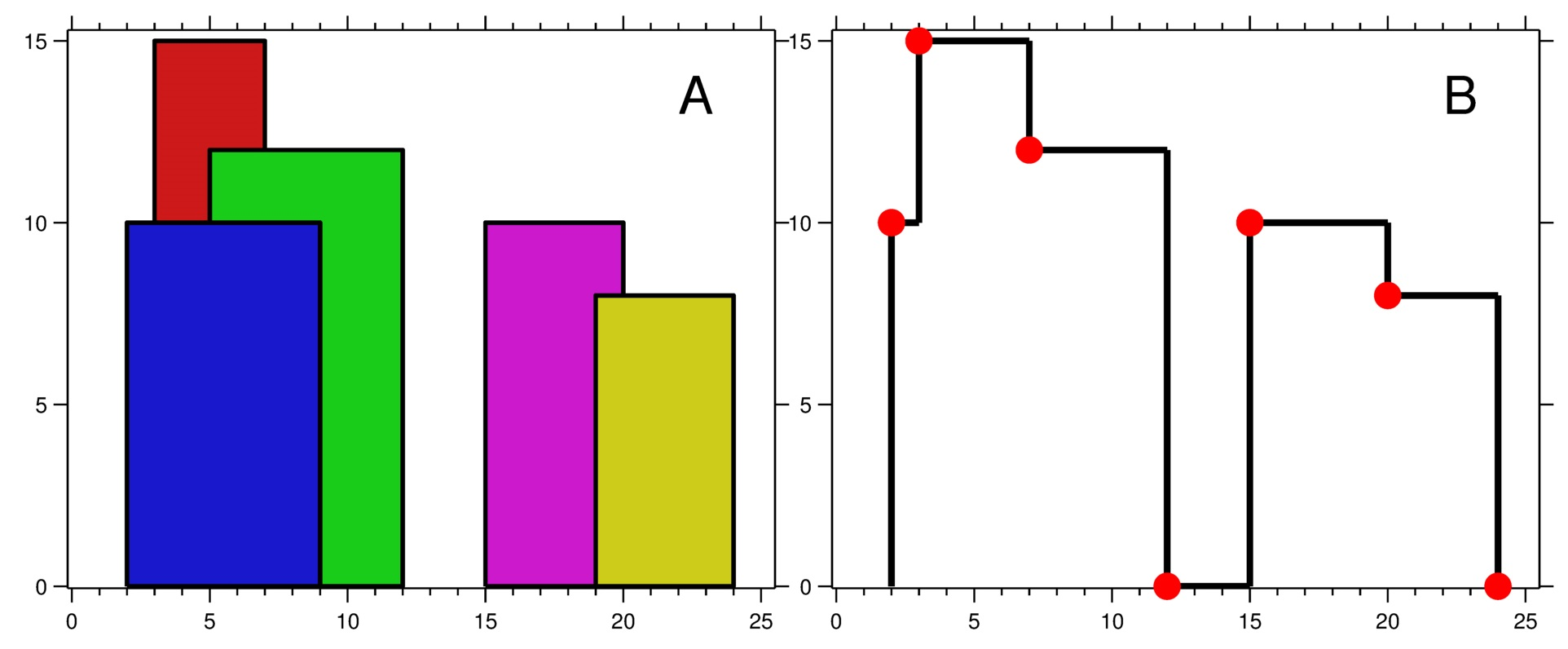
* lefti is the x coordinate of the left edge of the ith building.
* righti is the x coordinate of the right edge of the ith building.
* heighti is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

**Note:** There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

**Example 1:**



**Input:** buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

**Output:** [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]

**Explanation:**

Figure A shows the buildings of the input.

Figure B shows the skyline formed by those buildings. The red points in figure B represent the key points in the output list.

**Example 2:**

**Input:** buildings = [[0,2,3],[2,5,3]]

**Output:** [[0,3],[5,0]]

**Constraints:**

* 1 <= buildings.length <= 10^4
* 0 <= lefti < righti <= 2^31 - 1
* 1 <= heighti <= 2^31 - 1
* buildings is sorted by lefti in non-decreasing order.

**Attempt 1: 2023-12-21**

**Solution 1: Line Sweep + TreeMap (720min)**

**Wrong Solution**

**Test out by below, the issue is we cannot use TreeMap to create 'delta', in correct solution if given buildings = [[0,2,3],[2,5,3]], we will create 4 {index, height} pairs in basement scanning 'delta' array based on x-axis order, {0, -3}, {2, 3}, {2, -3}, {5, 3}, but if we use TreeMap to create 'delta', it will be only 3 pairs as {0, -3}, {2, 0}, {5, 3}, because it update on same key = 2 twice**

buildings = [[0,2,3],[2,5,3]]

java.util.NoSuchElementException

  at line 1637, java.base/java.util.TreeMap.key

  at line 302, java.base/java.util.TreeMap.firstKey

  at line 44, Solution.getSkyline

  at line 54, \_\_DriverSolution\_\_.\_\_helper\_\_

  at line 84, \_\_Driver\_\_.main

class Solution {

    public List<List<Integer>> getSkyline(int[][] buildings) {

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

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

        for(int[] building : buildings) {

            // Note: The trick is the left side of rectangle tagged with negative

            // number for identifying that's the first time add into 'heightMap'

            // e.g for buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

            // delta = {{2,-10},{3,-15},{5,-12},{7,15},{9,10},{12,12},{15,-20},{19,-8},{20,10},{24,8}}

            delta.put(building[0], delta.getOrDefault(building[0], 0) - building[2]);

            delta.put(building[1], delta.getOrDefault(building[1], 0) + building[2]);

        }

        // Instead of Priority Queue, use TreeMap and sort as descending order

        // then 'heightMap.firstKey()' will always return the highest height

        // at the moment

        TreeMap<Integer, Integer> heightMap = new TreeMap<>((a, b) -> b - a);

        // Define a previous height

        int prev\_h = 0;

        heightMap.put(prev\_h, 1);

        for(int index : delta.keySet()) {

            int h = delta.get(index);

            // Encounter left side of a rectangle

            if(h < 0) {

                // Turn negative 'h' into positive as '-h' and increase '-h' count

                heightMap.put(-h, heightMap.getOrDefault(-h, 0) + 1);

            // Encounte right side of a rectangle

            } else {

                heightMap.put(h, heightMap.get(h) - 1);

                if(heightMap.get(h) == 0) {

                    heightMap.remove(h);

                }

            }

            // Note: We cannot add 'h' into result directly, the actual height

            // on each index only based on 'heightMap.firstKey()', which is

            // working like a Priority Queue

            // e.g buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

            // when index = 5, we will get 'h' as 12, but that's not the actual

            // height on index = 5, because it has to 'prev\_h' comes from previous

            // index = 3 setup the top height, and since rectangle [3,7,15] not

            // finish at index = 5, reflect as 'prev\_h == cur\_h(heightMap.firstKey())',

            // so at index = 5 we don't need to setup height again, just record

            // a node [5,12] on heightMap but not at the first position for further

            // potential use

            int cur\_h = heightMap.firstKey();

            if(prev\_h != cur\_h) {

                result.add(Arrays.asList(index, cur\_h));

                prev\_h = cur\_h;

            }

        }

        return result;

    }

}

**Correct Solution**

**Style 1: TreeMap**

class Solution {

    public List<List<Integer>> getSkyline(int[][] buildings) {

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

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

        // Tricky point: tag start of a rectangle with negative height

        // and the end with positive height

        for(int[] building : buildings) {

            delta.add(new int[]{building[0], -building[2]});

            delta.add(new int[]{building[1], building[2]});

        }

        // If same x-axis index, sort based on corresponding height, negative height

        // should be first because that means the start index of a rectangle.

        // The physical meaning is a new rectangle may start between or at boundary

        // of previous rectangle x-axis interval, sequencially needs to consider

        // that new rectangle 'start' index before previous rectangle 'end' index

        // e.g buildings = [[0,2,3],[2,5,3]]

        // if we only sort based on x-axis, not consider same x-axis,

        // we will have delta = {{0,-3},{2,3},{2,-3},{5,3}}

        // but the correct delta is {{0,-3},{2,-3},{2,3},{5,3}}

        //Collections.sort(delta, (a, b) -> a[0] - b[0]); --> wrong

        Collections.sort(delta, (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

        // Instead of Priority Queue, use TreeMap and sort as descending order

        // then 'heightMap.firstKey()' will always return the highest height

        // at the moment

        TreeMap<Integer, Integer> heightMap = new TreeMap<>((a, b) -> b - a);

        // Define a previous height

        int prev\_h = 0;

        heightMap.put(prev\_h, 1);

        for(int[] d : delta) {

            int h = d[1];

            // Encounter left side of a rectangle

            if(h < 0) {

                // Turn negative 'h' into positive as '-h' and increase '-h' count

                heightMap.put(-h, heightMap.getOrDefault(-h, 0) + 1);

            // Encounte right side of a rectangle

            } else {

                heightMap.put(h, heightMap.get(h) - 1);

                if(heightMap.get(h) == 0) {

                    heightMap.remove(h);

                }

            }

            // Note: We cannot add 'h' into result directly, the actual height

            // on each index only based on 'heightMap.firstKey()', which is

            // working like a Priority Queue

            // e.g buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

            // when index = 5, we will get 'h' as 12, but that's not the actual

            // height on index = 5, because it has to 'prev\_h' comes from previous

            // index = 3 setup the top height, and since rectangle [3,7,15] not

            // finish at index = 5, reflect as 'prev\_h == cur\_h(heightMap.firstKey())',

            // so at index = 5 we don't need to setup height again, just record

            // a node [5,12] on heightMap but not at the first position for further

            // potential use

            int cur\_h = heightMap.firstKey();

            if(prev\_h != cur\_h) {

                result.add(Arrays.asList(d[0], cur\_h));

                prev\_h = cur\_h;

            }

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Style 2: Priority Queue**

class Solution {

    public List<List<Integer>> getSkyline(int[][] buildings) {

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

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

        // Tricky point: tag start of a rectangle with negative height

        // and the end with positive height

        // 拆解矩形，构建顶点的列表

        for(int[] building : buildings) {

            // 左顶点存为负数

            delta.add(new int[]{building[0], -building[2]});

            // 右顶点存为正数

            delta.add(new int[]{building[1], building[2]});

        }

        // If same x-axis index, sort based on corresponding height, negative height

        // should be first because that means the start index of a rectangle.

        // The physical meaning is a new rectangle may start between or at boundary

        // of previous rectangle x-axis interval, sequencially needs to consider

        // that new rectangle 'start' index before previous rectangle 'end' index

        // e.g buildings = [[0,2,3],[2,5,3]]

        // if we only sort based on x-axis, not consider same x-axis,

        // we will have delta = {{0,-3},{2,3},{2,-3},{5,3}}

        // but the correct delta is {{0,-3},{2,-3},{2,3},{5,3}}

        //Collections.sort(delta, (a, b) -> a[0] - b[0]); --> wrong

        // 根据横坐标对列表排序，相同横坐标的点纵坐标小的排在前面

        Collections.sort(delta, (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

        // 构建堆，按照纵坐标来判断大小

        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

        // prev\_h用于记录上次keypoint的高度

        int prev\_h = 0;

        // 将地平线0先加入堆中

        maxPQ.offer(prev\_h);

        for(int[] d : delta) {

            int h = d[1];

            // 将左顶点加入堆中

            if(h < 0) {

                maxPQ.offer(-h);

            // 将右顶点对应的左顶点移去

            } else {

                maxPQ.remove(h);

            }

            int cur\_h = maxPQ.peek();

            // 如果堆的新顶部和上个keypoint高度不一样，则加入一个新的keypoint

            if(prev\_h != cur\_h) {

                result.add(Arrays.asList(d[0], cur\_h));

                prev\_h = cur\_h;

            }

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://grandyang.com/leetcode/218/>

这道题一打开又是图又是这么长的题目的，看起来感觉应该是一道相当复杂的题，但是做完之后发现也就那么回事，虽然我不会做，是学习的别人的解法。这道求天际线的题目应该算是比较新颖的题，要是非要在之前的题目中找一道类似的题，也就只有 [Merge Intervals](http://www.cnblogs.com/grandyang/p/4370601.html)了吧，但是与那题不同的是，这道题不是求被合并成的空间，而是求轮廓线的一些关键的转折点，这就比较复杂了，通过仔细观察题目中给的那个例子可以发现，要求的红点都跟每个小区间的左右区间点有密切的关系，而且进一步发现除了每一个封闭区间的最右边的结束点是楼的右边界点，其余的都是左边界点，而且每个红点的纵坐标都是当前重合处的最高楼的高度，但是在右边界的那个楼的就不算了。在网上搜了很多帖子，发现网友[Brian Gordon的帖子](https://briangordon.github.io/2014/08/the-skyline-problem.html)图文并茂，什么动画渐变啊，横向扫描啊，简直叼到没朋友啊，但是叼到极致后就懒的一句一句的去读了，这里博主还是讲解另一位网友[百草园的博客](http://www.cnblogs.com/easonliu/p/4531020.html)吧。这里用到了 multiset 数据结构，其好处在于其中的元素是按堆排好序的，插入新元素进去还是有序的，而且执行删除元素也可方便的将元素删掉。这里为了区分左右边界，将左边界的高度存为负数，建立左边界和负高度的 pair，再建立右边界和高度的 pair，存入数组中，都存进去了以后，给数组按照左边界排序，这样就可以按顺序来处理那些关键的节点了。在 multiset 中放入一个0，这样在某个没有和其他建筑重叠的右边界上，就可以将封闭点存入结果 res 中。下面按顺序遍历这些关键节点，如果遇到高度为负值的 pair，说明是左边界，那么将正高度加入 multiset 中，然后取出此时集合中最高的高度，即最后一个数字，然后看是否跟 pre 相同，这里的 pre 是上一个状态的高度，初始化为0，所以第一个左边界的高度绝对不为0，所以肯定会存入结果 res 中。接下来如果碰到了一个更高的楼的左边界的话，新高度存入 multiset 的话会排在最后面，那么此时 cur 取来也跟 pre 不同，可以将新的左边界点加入结果 res。第三个点遇到绿色建筑的左边界点时，由于其高度低于红色的楼，所以 cur 取出来还是红色楼的高度，跟 pre 相同，直接跳过。下面遇到红色楼的右边界，此时首先将红色楼的高度从 multiset 中删除，那么此时 cur 取出的绿色楼的高度就是最高啦，跟 pre 不同，则可以将红楼的右边界横坐标和绿楼的高度组成 pair 加到结果 res 中，这样就成功的找到我们需要的拐点啦，后面都是这样类似的情况。当某个右边界点没有跟任何楼重叠的话，删掉当前的高度，那么 multiset 中就只剩0了，所以跟当前的右边界横坐标组成pair就是封闭点啦，具体实现参看代码如下：

class Solution {

    public:

    vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {

        vector<pair<int, int>> h, res;

        multiset<int> m;

        int pre = 0, cur = 0;

        for (auto &a : buildings) {

            h.push\_back({a[0], -a[2]});

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

        }

        sort(h.begin(), h.end());

        m.insert(0);

        for (auto &a : h) {

            if (a.second < 0) m.insert(-a.second);

            else m.erase(m.find(a.second));

            cur = \*m.rbegin();

            if (cur != pre) {

                res.push\_back({a.first, cur});

                pre = cur;

            }

        }

        return res;

    }

};

**Refer to**

**Priority Queue Solution**

<https://leetcode.com/problems/the-skyline-problem/solutions/61193/short-java-solution/>

class Solution {

    public List<int[]> getSkyline(int[][] buildings) {

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

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

        for(int[] b:buildings) {

            height.add(new int[]{b[0], -b[2]});

            height.add(new int[]{b[1], b[2]});

        }

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

            if(a[0] != b[0])

                return a[0] - b[0];

            return a[1] - b[1];

        });

        Queue<Integer> pq = new PriorityQueue<>((a, b) -> (b - a));

        pq.offer(0);

        int prev = 0;

        for(int[] h:height) {

            if(h[1] < 0) {

                pq.offer(-h[1]);

            } else {

                pq.remove(h[1]);

            }

            int cur = pq.peek();

            if(prev != cur) {

                result.add(new int[]{h[0], cur});

                prev = cur;

            }

        }

        return result;

    }

}

**TreeMap Solution**

<https://leetcode.com/problems/the-skyline-problem/solutions/61193/short-java-solution/comments/62419>

Thanks for the good solutions. However, there is a small thing that can be improved. pq.remove() is O(n) hence make it slower. I have modified it a little bit to use TreeMap instead of PriorityQueue and the run time is 2.5X faster.

public class Solution {

    public List<int[]> getSkyline(int[][] buildings) {

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

        for (int[] b: buildings) {

            heights.add(new int[]{b[0], - b[2]});

            heights.add(new int[]{b[1], b[2]});

        }

        Collections.sort(heights, (a, b) -> (a[0] == b[0]) ? a[1] - b[1] : a[0] - b[0]);

        TreeMap<Integer, Integer> heightMap = new TreeMap<>(Collections.reverseOrder());

        heightMap.put(0,1);

        int prevHeight = 0;

        List<int[]> skyLine = new LinkedList<>();

        for (int[] h: heights) {

            if (h[1] < 0) {

                Integer cnt = heightMap.get(-h[1]);

                cnt = ( cnt == null ) ? 1 : cnt + 1;

                heightMap.put(-h[1], cnt);

            } else {

                Integer cnt = heightMap.get(h[1]);

                if (cnt == 1) {

                    heightMap.remove(h[1]);

                } else {

                    heightMap.put(h[1], cnt - 1);

                }

            }

            int currHeight = heightMap.firstKey();

            if (prevHeight != currHeight) {

                skyLine.add(new int[]{h[0], currHeight});

                prevHeight = currHeight;

            }

        }

        return skyLine;

    }

}