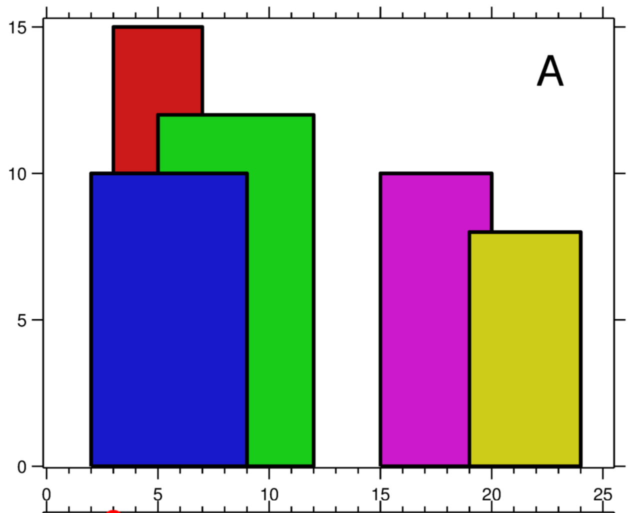
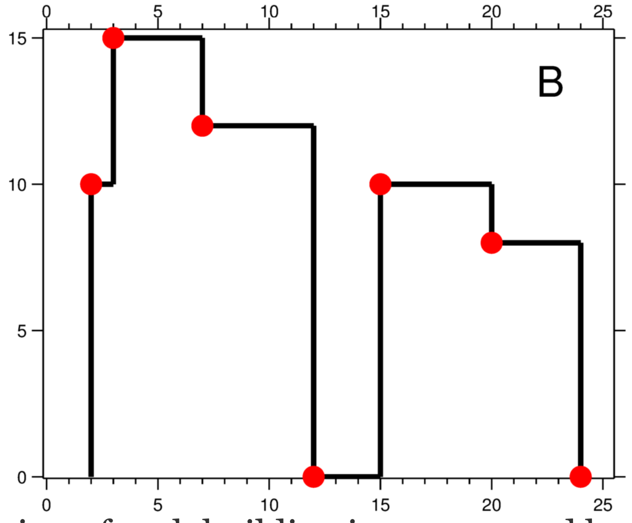
The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).





The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX`, `0 , and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.`

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` .

The output is a list of "**key points**" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as: `[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]`.
- The input list is already sorted in ascending order by the left x position `Li`.
- The output list must be sorted by the x position.
- 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], ...]`

Credits:
Special thanks to @stellari for adding this problem, creating these two awesome images and all test cases.

Solution 1

```
    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;
}
```

written by jinwu original link here

Solution 2

Detailed explanation: http://www.geeksforgeeks.org/divide-and-conquer-set-7-the-skyline-problem/

```java
public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
        if (buildings.length == 0)
            return new LinkedList<int[]>();
        return recurSkyline(buildings, 0, buildings.length - 1);
    }

    private LinkedList<int[]> recurSkyline(int[][] buildings, int p, int q) {
        if (p < q) {
            int mid = p + (q - p) / 2;
            return merge(recurSkyline(buildings, p, mid),
                    recurSkyline(buildings, mid + 1, q));
        } else {
            LinkedList<int[]> rs = new LinkedList<int[]>();
            rs.add(new int[] { buildings[p][0], buildings[p][2] });
            rs.add(new int[] { buildings[p][1], 0 });
            return rs;
        }
    }

    private LinkedList<int[]> merge(LinkedList<int[]> l1, LinkedList<int[]> l2) {
        LinkedList<int[]> rs = new LinkedList<int[]>();
        int h1 = 0, h2 = 0;
        while (l1.size() > 0 && l2.size() > 0) {
            int x = 0, h = 0;
            if (l1.getFirst()[0] < l2.getFirst()[0]) {
                x = l1.getFirst()[0];
                h1 = l1.getFirst()[1];
                h = Math.max(h1, h2);
                l1.removeFirst();
            } else if (l1.getFirst()[0] > l2.getFirst()[0]) {
                x = l2.getFirst()[0];
                h2 = l2.getFirst()[1];
                h = Math.max(h1, h2);
                l2.removeFirst();
            } else {
                x = l1.getFirst()[0];
                h1 = l1.getFirst()[1];
                h2 = l2.getFirst()[1];
                h = Math.max(h1, h2);
                l1.removeFirst();
                l2.removeFirst();
            }
            if (rs.size() == 0 || h != rs.getLast()[1]) {
                rs.add(new int[] { x, h });
            }
        }
        rs.addAll(l1);
        rs.addAll(l2);
        return rs;
    }
}
```

written by hscaizh original link here

Solution 3

The idea is to do line sweep and just process the buildings only at the start and end points. The key is to use a priority queue to save all the buildings that are still "alive". The queue is sorted by its height and end time (the larger height first and if equal height, the one with a bigger end time first). For each iteration, we first find the current process time, which is either the next new building start time or the end time of the top entry of the live queue. If the new building start time is larger than the top one end time, then process the one in the queue first (pop them until it is empty or find the first one that ends after the new building); otherswise, if the new building starts before the top one ends, then process the new building (just put them in the queue). After processing, output it to the resulting vector if the height changes. Complexity is the worst case O(NlogN)

Not sure why my algorithm is so slow considering others' Python solution can achieve 160ms, any commments?

```
class Solution {
public:
    vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {
        vector<pair<int, int>> res;
        int cur=0, cur_X, cur_H =-1,  len = buildings.size();
        priority_queue< pair<int, int>> liveBlg; // first: height, second, end time
        while(cur<len || !liveBlg.empty())
        { // if either some new building is not processed or live building queue is not empty
            cur_X = liveBlg.empty()? buildings[cur][0]:liveBlg.top().second; // next timing point to process

            if(cur>=len || buildings[cur][0] > cur_X)
            { //first check if the current tallest building will end before the next timing point
                  // pop up the processed buildings, i.e. those  have height no larger than cur_H and end before the top one
                while(!liveBlg.empty() && ( liveBlg.top().second <= cur_X) ) liveBlg.pop();
            }
            else
            { // if the next new building starts before the top one ends, process the new building in the vector
                cur_X = buildings[cur][0];
                while(cur<len && buildings[cur][0]== cur_X)  // go through all the new buildings that starts at the same point
                {  // just push them in the queue
                    liveBlg.push(make_pair(buildings[cur][2], buildings[cur][1]));
                    cur++;
                }
            }
            cur_H = liveBlg.empty()?0:liveBlg.top().first; // outut the top one
            if(res.empty() || (res.back().second != cur_H) ) res.push_back(make_pair(cur_X, cur_H));
        }
        return res;
    }
};
```

written by dong.wang.1694 original link here

From Leetcoder.