Sweep Line

391. Number of Airplanes in the Sky (LintCode)
252. Meeting Rooms
253. Meeting Rooms II
56. Merge Intervals
57. Insert Interval
1272. Remove Interval
435. Non-overlapping Intervals
1288. Remove Covered Intervals
352. Data Stream as Disjoint Intervals
1229. Meeting Scheduler
986. Interval List Intersections
759. Employee Free Time
218. The Skyline Problem

Sweep Line

Sweep Line is a vertical line (and/or, a horizontal line, in some cases) that is conceptually “swept” across the plane.

Sweep Line technique is often used to find intersections but can be extended to other scenarios like finding areas etc.

391. Number of Airplanes in the Sky (LintCode)

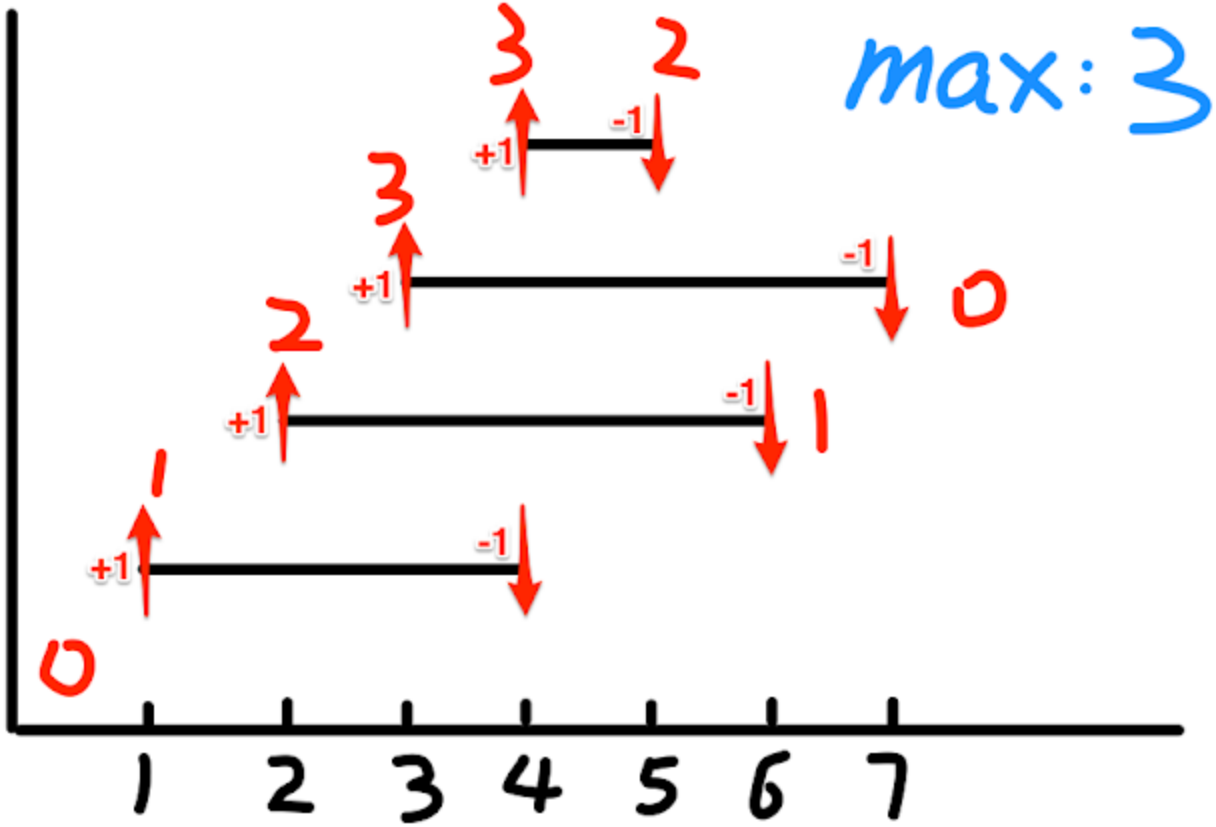
https://www.lintcode.com/problem/391/

Given an list interval, which are taking off and landing time of the flight. How many airplanes are there at most at the same time in the sky?

Input: [(1, 10), (2, 3), (5, 8), (4, 7)]
Output: 3
Explanation:
The first airplane takes off at 1 and lands at 10.
The second ariplane takes off at 2 and lands at 3.
The third ariplane takes off at 5 and lands at 8.
The forth ariplane takes off at 4 and lands at 7.
During 5 to 6, there are three airplanes in the sky.

391. Number of Airplanes in the Sky (LintCode)

[[1,4], [2,6],[3,7],[4,5]] Return 3



for every interval start time, count + 1, 。
interval end time count -1。

Sort all start and and end time in a single array.

Count from the first start time, when ever we see a start time count+1, see a end time count –1
maintain a max plane count .

```java
/**
 * Definition of Interval:
 * public class Interval {
 *     int start, end;
 *     Interval(int start, int
end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 * }
 */
```

```java
public class Solution {
    /**
     * @param airplanes: An interval array
     * @return: Count of airplanes are in the sky.
     */
    public int countOfAirplanes(List<Interval> airplanes) {
        // write your code here
        List<Interval> list = new ArrayList<>(airplanes.size() * 2);
        for(Interval i: airplanes){
            list.add(new Interval(i.start, 1));
            list.add(new Interval(i.end, -1));
        }
        Collections.sort(list, (Interval p1, Interval p2)->{
            if(p1.start == p2.start) return p1.end-p2.end;
            return p1.start - p2.start;
        });
        int cnt = 0;
        int ans = 0;
        for(Interval p: list){
            cnt += p.end;
            ans = Math.max(ans,cnt);
        }
        return ans;
    }
}
```

252. Meeting Rooms

Easy 560 36 Add to List Share

Given an array of meeting time intervals consisting of start and end times `[[s1, e1], [s2, e2], ...]` ($s_i < e_i$), determine if a person could attend all meetings.

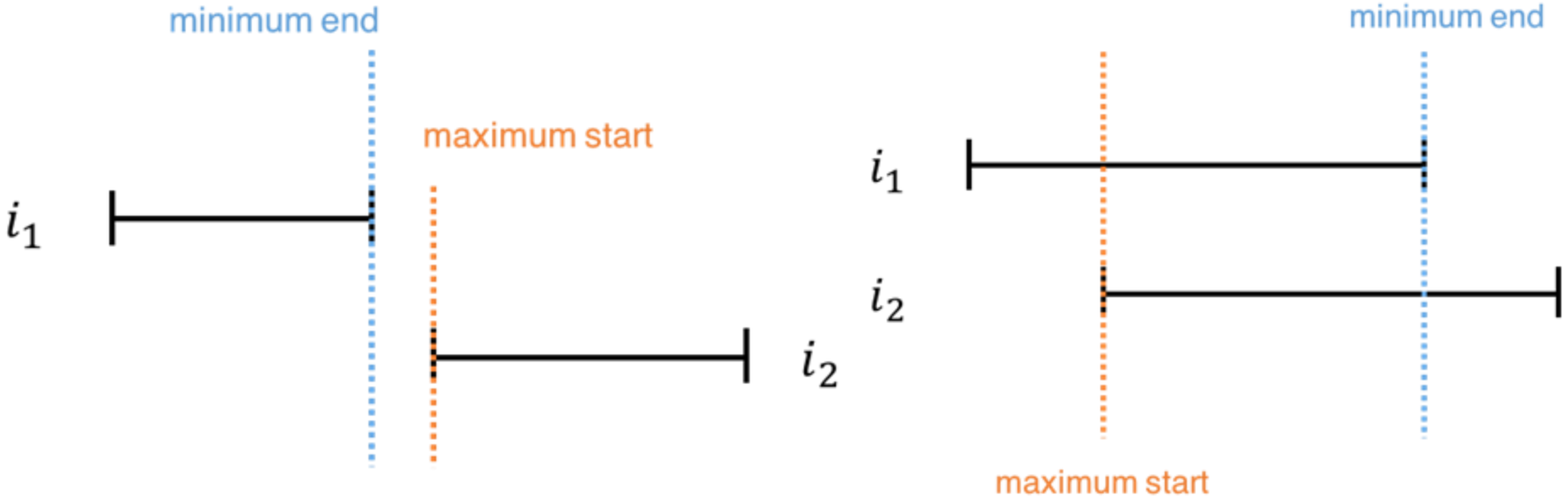
Example 1:

```
Input: [[0,30],[5,10],[15,20]]
Output: false
```

Example 2:

```
Input: [[7,10],[2,4]]
Output: true
```

252. Meeting Rooms




```java
public class Solution {
    public boolean canAttendMeetings(Interval[] intervals) {
        Arrays.sort(intervals, (a, b) -> a.start - b.start);
        for (int i = 0; i < intervals.length - 1; i++)
            if (intervals[i].end > intervals[i + 1].start) return false;
        return true;
    }
}
```

253. Meeting Rooms II

Medium 2469 41 Add to List Share

Given an array of meeting time intervals consisting of start and end times `[[s1, e1], [s2, e2], ...]` ($s_i < e_i$), find the minimum number of conference rooms required.

Example 1:

```
Input: [[0, 30],[5, 10],[15, 20]]
Output: 2
```

Example 2:

```
Input: [[7,10],[2,4]]
Output: 1
```

```java
public class Solution {
    public int minMeetingRooms(int[][] intervals) {
        List<int[]> list = new ArrayList<>();
        for (int[] interval : intervals) {
            list.add(new int[]{interval[0], 1});
            list.add(new int[]{interval[1], -1});
        }
        Collections.sort(list, (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);
        int res = 0, count = 0;
        for (int[] point : list) {
            count += point[1];
            res = Math.max(res, count);
        }
        return res;
    }
}
```

253. Meeting Rooms II

```java
public class Solution {
    public int minMeetingRooms(int[][] intervals) {
        int[] starts = new int[intervals.length], ends = new int[intervals.length];
        for (int i = 0; i < intervals.length; i++) {
            starts[i] = intervals[i][0];
            ends[i] = intervals[i][1];
        }
        Arrays.sort(starts);
        Arrays.sort(ends);
        int room = 0, end = 0;
        for (int i = 0; i < starts.length; i++) {
            if (starts[i] < ends[end])room++;
            else end++;
        }
        return room;
    }
}
```

56. Merge Intervals

Medium 3926 278 Add to List Share

Given a collection of intervals, merge all overlapping intervals.

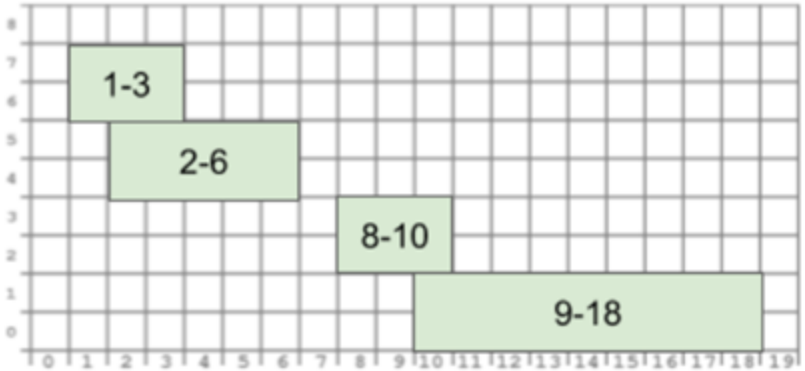
Example 1:

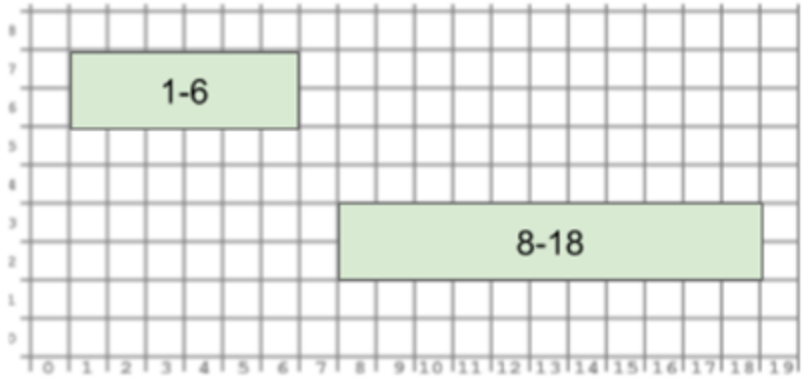
```
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Given [1,3],[2,6],[8,10],[9,18],
return [1,6],[8,18].





Sort intervals by its start

```
if curr.start <= last.end:
  Merge intervals
else:
  Insert a new interval
```

Time complexity: O(nlogn)
Space complexity: O(n)

```java
public class Solution {
    public int[][] merge(int[][] intervals) {
        List<int[]> res = new ArrayList<>();
        if (intervals == null || intervals.length == 0) return new int[0][];
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        int[] cur = intervals[0];
        for (int[] next : intervals) {
            if (cur[1] >= next[0]) cur[1] = Math.max(cur[1], next[1]);
            else {
                res.add(cur);
                cur = next;
            }
        }
        res.add(cur);
        return res.toArray(new int[0][]);
    }
}
```

```java
/unchecked/
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        // Make a new array of a's runtime type, but my contents:
        return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    System.arraycopy(elementData, srcPos: 0, a, destPos: 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

57. Insert Interval

Description | Hints | Submissions | Discuss | Solution

Pick One

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
```

```java
class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> res = new ArrayList<>();
        for (int[] cur : intervals)
            if (newInterval == null || cur[1] < newInterval[0]) res.add(cur);
            else if (cur[0] > newInterval[1]) {
                res.addAll(List.of(newInterval, cur));
                newInterval = null;
            } else {
                newInterval[0] = Math.min(newInterval[0], cur[0]);
                newInterval[1] = Math.max(newInterval[1], cur[1]);
            }
        if (newInterval != null) res.add(newInterval);
        return res.toArray(new int[res.size()][]);
    }
}
```

1272. Remove Interval

Description | Hints | Submissions | Discuss | Solution

Pick One

Given a **sorted** list of disjoint `intervals`, each interval `intervals[i] = [a, b]` represents the set of real numbers `x` such that `a <= x < b`.

We remove the intersections between any interval in `intervals` and the interval `toBeRemoved`.

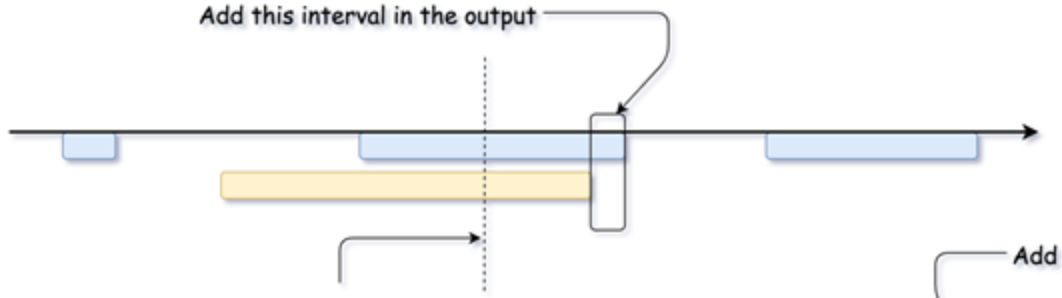
Return a **sorted** list of `intervals` after all such removals.

Example 1:

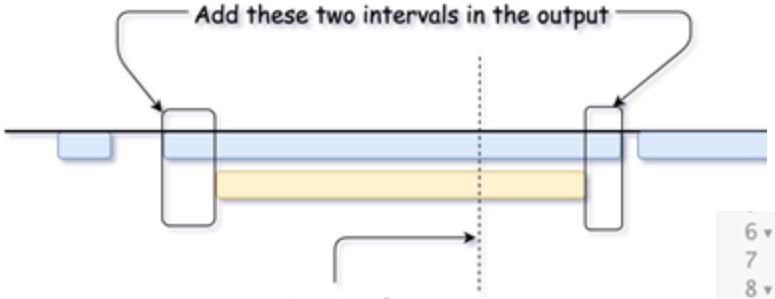
```
Input: intervals = [[0,2],[3,4],[5,7]], toBeRemoved = [1,6]
Output: [[0,1],[6,7]]
```

Example 2:

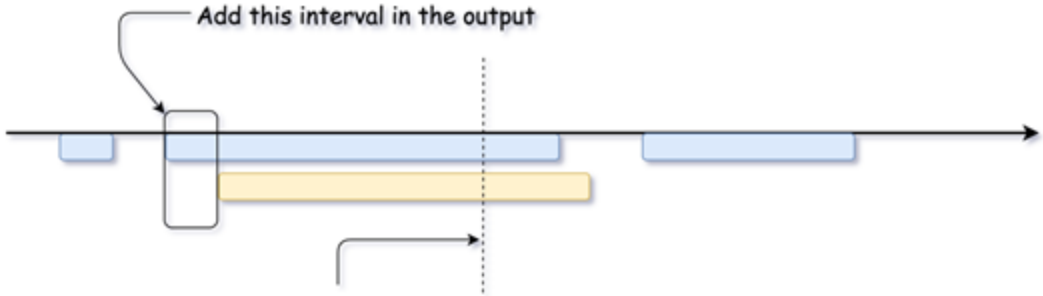
```
Input: intervals = [[0,5]], toBeRemoved = [2,3]
Output: [[0,2],[3,5]]
```



Situation 4.
"Right" overlap.



Situation 2.
toBeRemoved interval is
inside of the current one



Situation 3.
"Left" overlap.

```java
public List<List<Integer>> removeInterval(int[][] intervals, int[] toBeRemoved) {
    List<List<Integer>> res = new ArrayList<>();
    for (int[] i : intervals) {
        if (i[1] <= toBeRemoved[0] || i[0] >= toBeRemoved[1]) { // no overlap.
            res.add(Arrays.asList(i[0], i[1]));
        } else { // i[1] > toBeRemoved[0] && i[0] < toBeRemoved[1].
            if (i[0] < toBeRemoved[0]) // left end extra, we remain
                res.add(Arrays.asList(i[0], toBeRemoved[0]));
            if (i[1] > toBeRemoved[1]) // right end extra we remain
                res.add(Arrays.asList(toBeRemoved[1], i[1]));
        }
    }
    return res;
}
}
```

435. Non-overlapping Intervals

Description | Hints | Submissions | Discuss | Solution

Pick One

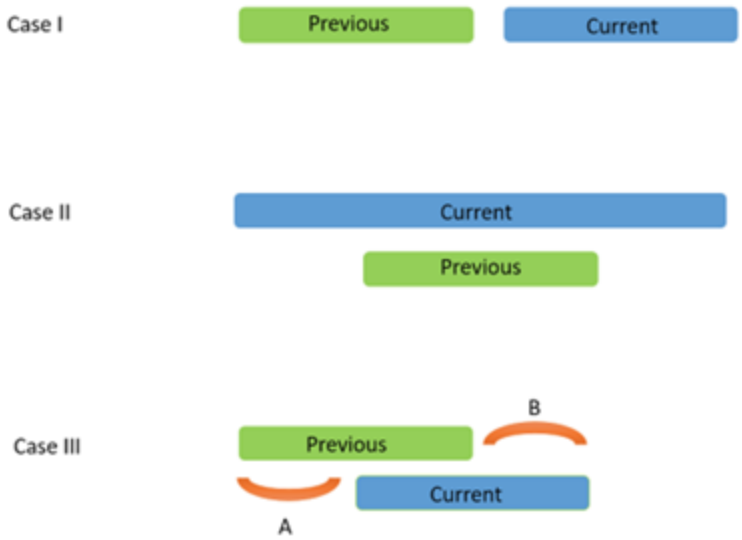
Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:

```
Input: [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.
```

Example 2:

```
Input: [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of intervals non-overla
```



If conflict always remove previous one, to leave more space for the later.

```java
class Solution {


    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals.length == 0) return 0;
        Arrays.sort(intervals, (a, b) -> a[1] - b[1]);
        int count = 0, end = Integer.MIN_VALUE;
        for (int[] cur : intervals) {
            if (end <= cur[0]) end = cur[1];
            else count++;
        }
        return count;
    }
}
```

1288. Remove Covered Intervals

Description | Hints | Submissions | Discuss | Solution

Pick One

Given a list of intervals, remove all intervals that are covered by another interval in the list. Interval `[a,b)` is covered by interval `[c,d)` if and only if `c <= a` and `b <= d`.

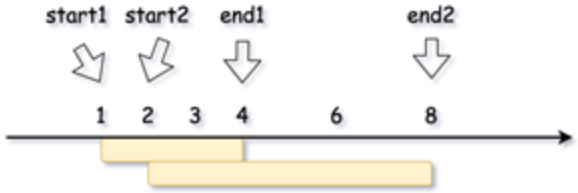
After doing so, return the number of remaining intervals.

Example 1:

```
Input: intervals = [[1,4],[3,6],[2,8]]
Output: 2
Explanation: Interval [3,6] is covered by [2,8], therefore it is removed.
```

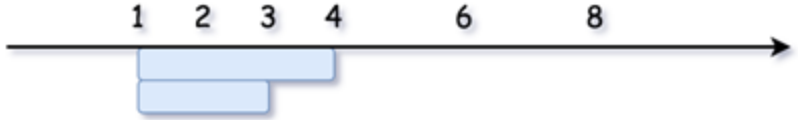
Sort by start time increasing, end time decreasing

- If `end1 < end2`, the intervals won't completely cover one another, though they have some overlapping.



The above algorithm will fail because it cannot distinguish these two situations as follows:

- If `end1 >= end2`, the interval 2 is covered by the interval 1.

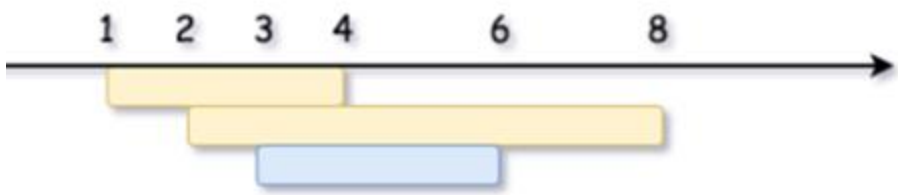


Iterate, count = 0

end > prev_end --> count++

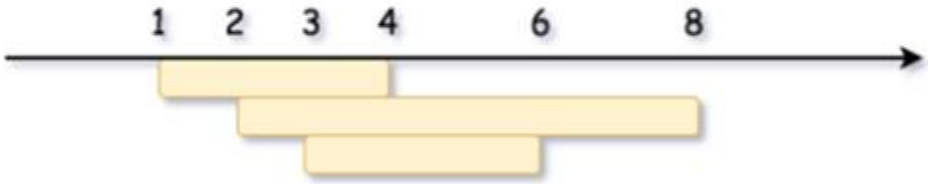
Iterate, count = 1

end > prev_end --> count++



Iterate, count = 2

end < prev_end --> do nothing




```java
public int removeCoveredIntervals(int[][] intervals) {
    Arrays.sort(intervals, (a, b) ->
                (a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]));
    int count = 0, cur = 0;
    for (int[] a : intervals)
        if (cur < a[1]) {
            cur = a[1];
            count++;
        }
    return count;
}
```

352. Data Stream as Disjoint Intervals

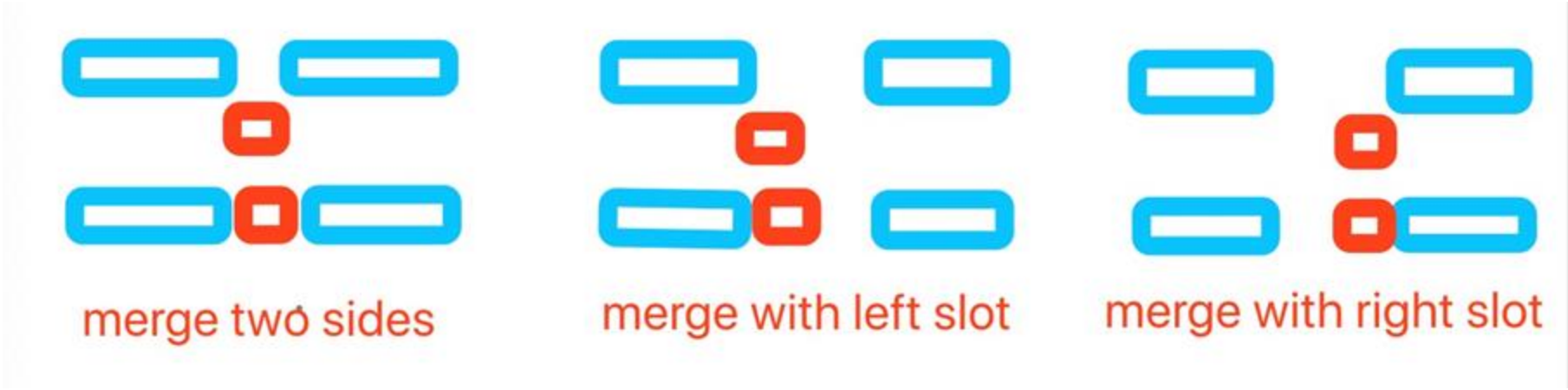
Description | Hints | Submissions | Discuss | Solution

Pick One

Given a data stream input of non-negative integers a_1, a_2, ..., a_n, ..., summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```




```java
class SummaryRanges {
    TreeSet<int[]> set = new TreeSet<>((a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

    public void addNum(int val) {
        int[] interval = new int[] {val, val};
        if (set.contains(interval)) return;
        int[] low = set.lower(interval), high = set.higher(interval);
        if (high != null && high[0] == val) return;
        if (low != null && low[1] + 1 == val && high != null && val + 1 == high[0]) {  //新插入的时间正好连接两个time slot
            low[1] = high[1];
            set.remove(high);
        }
        else if (low != null && low[1] + 1 >= val) low[1] = Math.max(low[1], val);     //新加入的时间和左边小的time slot merge
        else if (high != null && val + 1 == high[0]) high[0] = val;                    //新加入的时间和右边大的time slot merge
        else set.add(interval);                                                        //新加入的时间两边都不相邻，孤立插入
    }

    public int[][] getIntervals() {
        List<int[]> res = new ArrayList<>();
        for (int[] interval: set) res.add(interval);
        return res.toArray(new int[res.size()][]);
    }
}
```

986. Interval List Intersections

Description | Hints | Submissions | Discuss | Solution

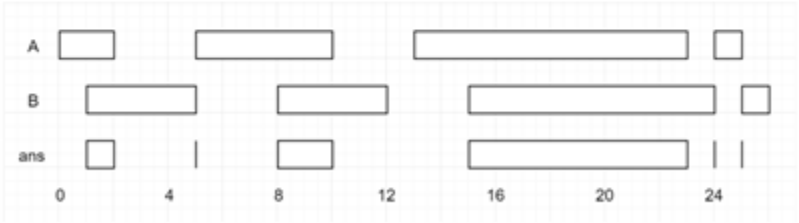
Pick One

Given two lists of **closed** intervals, each list of intervals is pairwise disjoint and in sorted order.

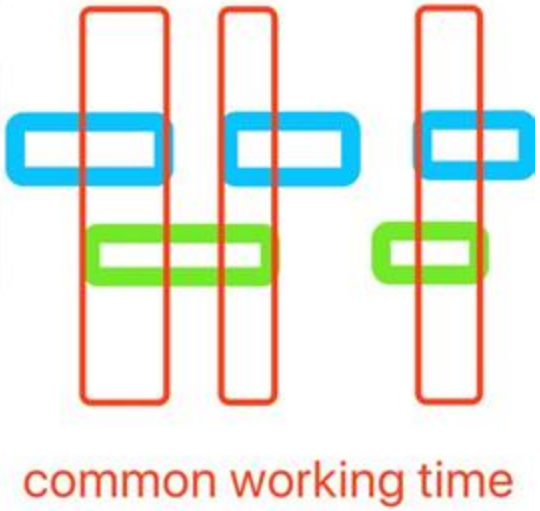
Return the intersection of these two interval lists.

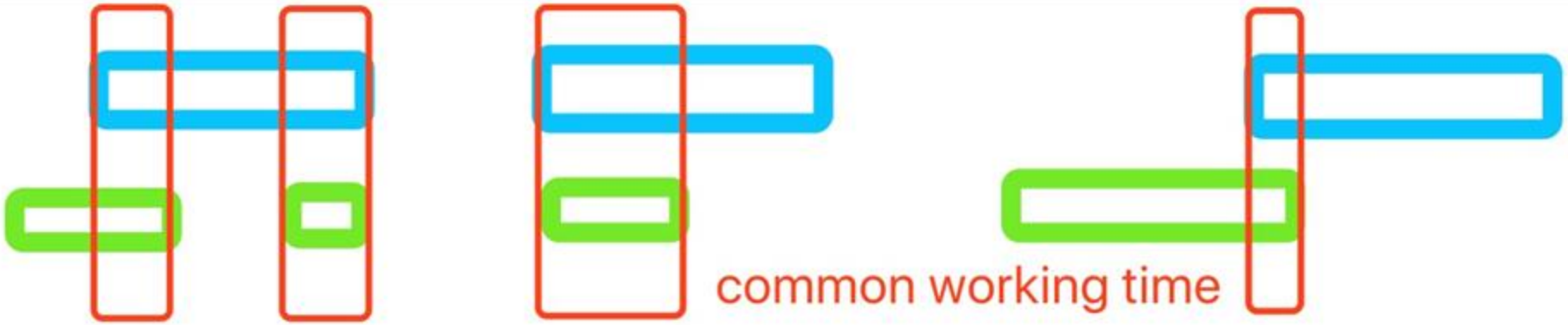
(Formally, a closed interval `[a, b]` (with `a <= b`) denotes the set of real numbers `x` with `a <= x <= b`. The intersection of two closed intervals is a set of real numbers that is either empty, or can be represented as a closed interval. For example, the intersection of [1, 3] and [2, 4] is [2, 3].)

Example 1:




```
Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```






```java
class Solution {
    public int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> res = new ArrayList<>();
        int i = 0, j = 0;
        while (i < A.length && j < B.length) {
            int low = Math.max(A[i][0], B[j][0]);
            int high = Math.min(A[i][1], B[j][1]);
            if (low <= high) res.add(new int[]{low, high});
            if (A[i][1] < B[j][1]) i++;
            else j++;
        }
        return res.toArray(new int[res.size()][]);
    }
}
```

759. Employee Free Time

Description | Hints | Submissions | Discuss | Solution

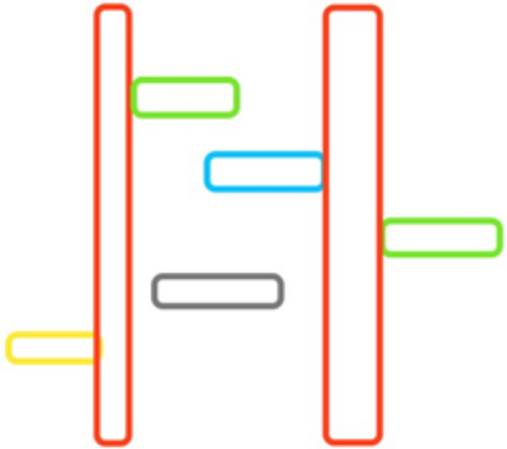
Pick One

We are given a list `schedule` of employees, which represents the working time for each employee.

Each employee has a list of non-overlapping `Intervals`, and these intervals are in sorted order.

Return the list of finite intervals representing **common, positive-length free time** for *all* employees, also in sorted order.

(Even though we are representing `Intervals` in the form `[x, y]`, the objects inside are `Intervals`, not lists or arrays. For example, `schedule[0][0].start = 1`, `schedule[0][0].end = 2`, and `schedule[0][0][0]` is not defined). Also, we wouldn't include intervals like [5, 5] in our answer, as they have zero length.

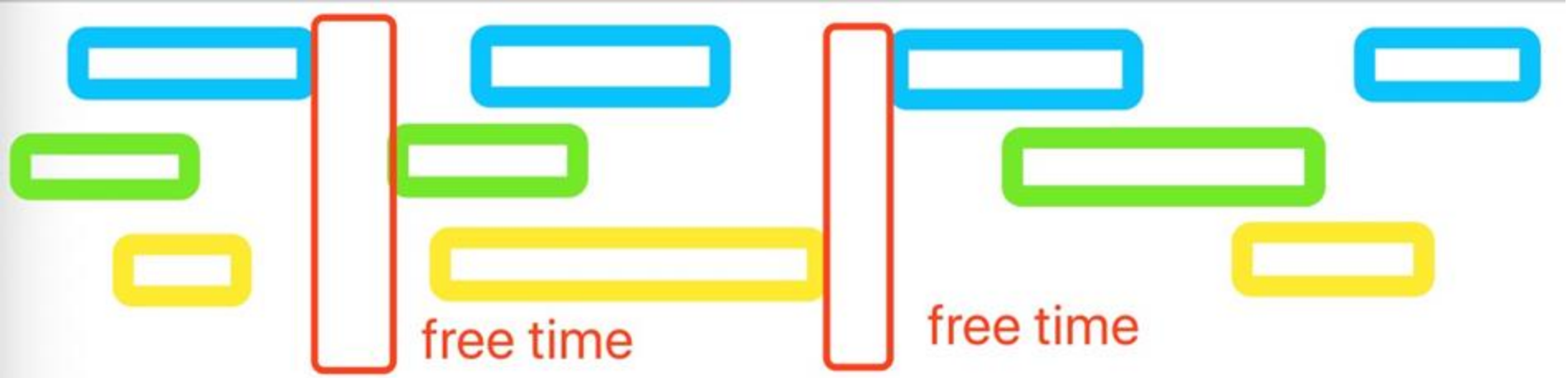


Example 1:

```
Input: schedule = [[[1,2],[5,6]],[[1,3]],[[4,10]]]
Output: [[3,4]]
Explanation: There are a total of three employees, and all common
free time intervals would be [-inf, 1], [3, 4], [10, inf].
We discard any intervals that contain inf as they aren't finite.
```

Example 2:

```
Input: schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]
Output: [[5,6],[7,9]]
```



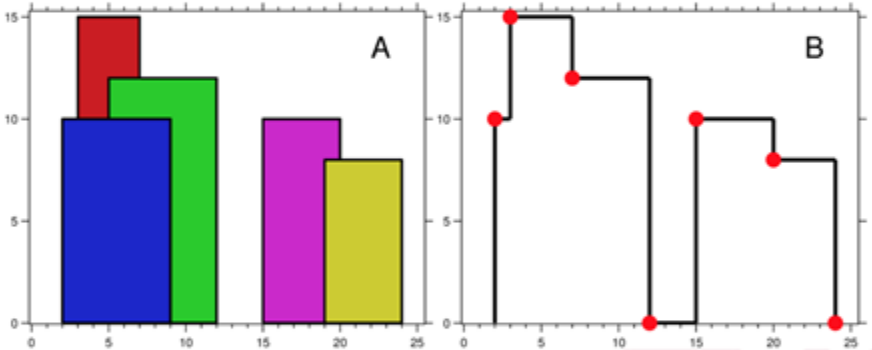

```java
class Solution {
    public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {
        List<Interval> res = new ArrayList<>();
        PriorityQueue<Interval> pq = new PriorityQueue<>((a, b) -> (a.start - b.start));
        for (List<Interval> list : schedule)
            for (Interval interval: list)
                pq.add(interval);
        Interval cur = pq.poll();
        while (!pq.isEmpty())
            if (cur.end >= pq.peek().start) {
                cur.end = Math.max(cur.end, pq.poll().end);       //两个工作interval交叉了，没有休息时间。我们合并工作时间
            } else {
                res.add(new Interval(cur.end, pq.peek().start));  //两个工作interval之间有空隙，我们加到答案中
                cur = pq.poll();
            }
        return res;
    }
}

```

218. The Skyline Problem

Description | Hints | Submissions | Discuss | Solution

Pick One

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).

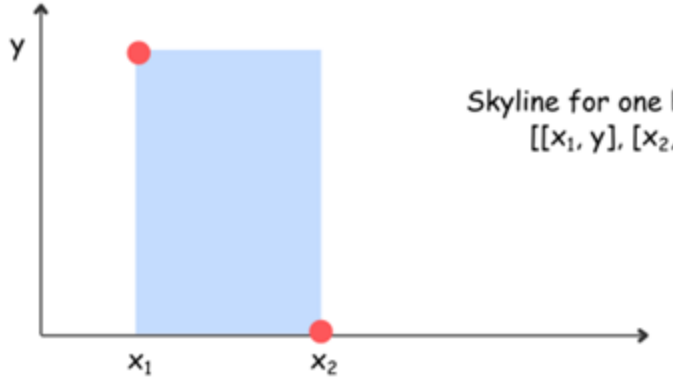


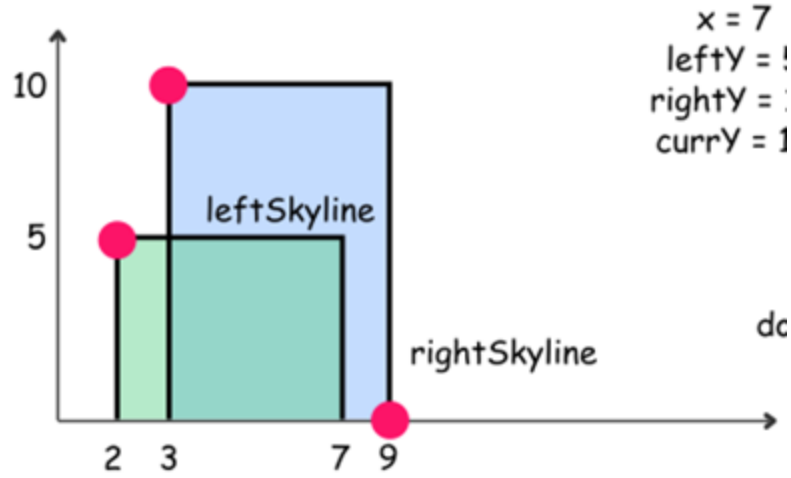
The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX`, `0 < Hi ≤ INT_MAX`, and `Ri - 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] ]`.

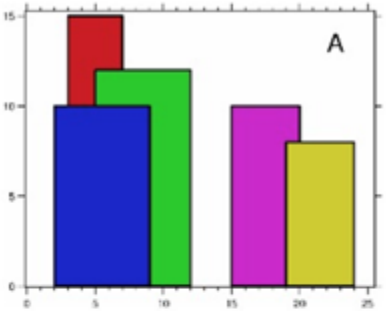
The output is a list of "**key points**" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to 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.

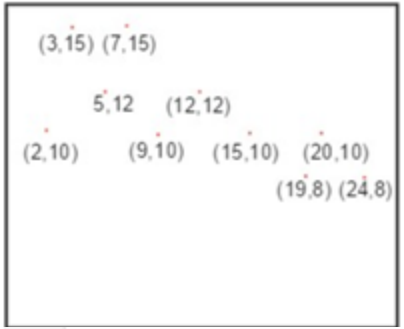
For instance, the skyline in Figure B should be represented as: `[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

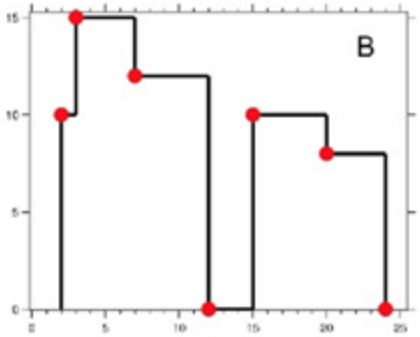




{2,9,10}, {3,7,15}, {5,12,12}, {15,20,10}, {19,24,8}







(x,y)	height queue(y)	key points (x,y)
	0	
(2,10)	10 0	(2,10)
(3,15)	15,10,0	(3,15)
(5,12)	15,12,10,0	
(7,-15)	12,10,0	(7,12)
(9,-10)	12,0	
(12,-12)	0	(12,0)
(15,10)	10,0	(15,10)
(19,8)	10,8,0	
(20,-10)	8,0	(20,8)
(24,-8)	0	(24,0)

```java
public class Solution {
    public List<List<Integer>> getSkyline(int[][] buildings) {
        List<List<Integer>> res = 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) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);
        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> (b - a));
        pq.offer(0);
        int preMax = 0;
        for (int[] h : height) {
            if (h[1] < 0) pq.offer(-h[1]);
            else pq.remove(h[1]);
            int curMax = pq.peek();
            if (curMax != preMax) {
                res.add(List.of(h[0], curMax));
                preMax = curMax;
            }
        }
        return res;
    }
}
```

pq.remove() Complexity : search O(n) delete log(n)