<https://leetcode.ca/all/759.html>

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.

**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]]

(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.

**Note:**

1. schedule and schedule[i] are lists with lengths in range [1, 50]
2. 0 <= schedule[i].start < schedule[i].end <= 10^8

**NOTE:** input types have been changed on June 17, 2019. Please reset to default code definition to get new method signature.

**Attempt 1: 2023-12-14**

**Solution 1: Sorting + Merge Interval (30 min)**

**这种解法和L56.P5.1.Merge Intervals的思路几乎一模一样**

class Solution {

    class Interval {

        int start;

        int end;

        public Interval() {

            this.start = 0;

            this.end = 0;

        }

        public Interval(int s, int e) {

            this.start = s;

            this.end = e;

        }

    }

    public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {

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

        List<Interval> list = new ArrayList<>();

        // Flattening the schedule

        for(List<Interval> tmp : schedule) {

            for(Interval interval : tmp) {

                list.add(interval);

            }

        }

        // Sorting by start of each Interval

        Collections.sort(list, (a, b) -> a.start - b.start);

        // Checking for free time between intervals

        Interval newInterval = list.get(0);

        for(Interval interval : list) {

            if(newInterval.end >= interval.start) {

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

            } else {

                result.add(new Interval(newInterval.end, interval.start));

                newInterval = interval;

            }

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Compleixty: O(N)

**Test data**

Test data set 1: schedule = [[[1,2],[5,6]],[[1,3]],[[4,10]]]

Expect Output: [[3,4]]

List<List<Interval>> schedule = new ArrayList<>();

List<Interval> list1 = new ArrayList<>();

Interval a1 = so.new Interval(1, 2);

Interval a2 = so.new Interval(5, 6);

list1.add(a1);

list1.add(a2);

List<Interval> list2 = new ArrayList<>();

Interval b1 = so.new Interval(1,3);

list2.add(b1);

List<Interval> list3 = new ArrayList<>();

Interval c1 = so.new Interval(4,10);

list3.add(c1);

schedule.add(list1);

schedule.add(list2);

schedule.add(list3);

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

Test data set 2: schedule = [[[1,4]],[[4,5]]]

Expect Output: empty

List<List<Interval>> schedule = new ArrayList<>();

List<Interval> list1 = new ArrayList<>();

Interval a1 = so.new Interval(1, 4);

list1.add(a1);

List<Interval> list2 = new ArrayList<>();

Interval b1 = so.new Interval(4,5);

list2.add(b1);

schedule.add(list1);

schedule.add(list2);

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

Test data set 3: schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]

Expect Output: [[5,6],[7,9]]

List<List<Interval>> schedule = new ArrayList<>();

List<Interval> list1 = new ArrayList<>();

Interval a1 = so.new Interval(1, 3);

Interval a2 = so.new Interval(6, 7);

list1.add(a1);

list1.add(a2);

List<Interval> list2 = new ArrayList<>();

Interval b1 = so.new Interval(2,4);

list2.add(b1);

List<Interval> list3 = new ArrayList<>();

Interval c1 = so.new Interval(2,5);

Interval c2 = so.new Interval(9,12);

list3.add(c1);

list3.add(c2);

schedule.add(list1);

schedule.add(list2);

schedule.add(list3);

**Refer to**

<https://grandyang.com/leetcode/759/>

这道题和之前那道[Merge Intervals](http://www.cnblogs.com/grandyang/p/4370601.html)基本没有太大的区别，那道题是求合并后的区间，这道题求合并后区间中间不相连的区间。那么只要我们合并好了区间，就很容易做了。那么我么首先应该给所有的区间排个序，按照起始位置从小到大来排。因为我们总不可能一会处理前面的，一会处理后面的区间。排好序以后，我们先取出第一个区间赋给t，然后开始遍历所有的区间内所有的区间，如果t的结束位置小于当前遍历到的区间i的起始位置，说明二者没有交集，那么把不相交的部分加入结果res中，然后把当前区间i赋值给t；否则如果区间t和区间i有交集，那么我们更新t的结束位置为二者中的较大值，因为按顺序遍历区间的时候，区间t的结束位置是比较的基准，越大越容易和后面的区间进行合并，参见代码如下：

解法一：

class Solution {

    public:

    vector<Interval> employeeFreeTime(vector<vector<Interval>>& schedule) {

        vector<Interval> res, v;

        for (auto a : schedule) {

            v.insert(v.end(), a.begin(), a.end());

        }

        sort(v.begin(), v.end(), [](Interval &a, Interval &b) {return a.start < b.start;});

        Interval t = v[0];

        for (Interval i : v) {

            if (t.end < i.start) {

                res.push\_back(Interval(t.end, i.start));

                t = i;

            } else {

                t = (t.end < i.end) ? i : t;

            }

        }

        return res;

    }

};

**Refer to**

<https://algo.monster/liteproblems/759>

Problem Explanation:

Given a list of employee work schedules with each employee having a list of non-overlapping intervals representing their working hours, we are tasked with finding the common free time for all employees, or in other words, the times when all employees are not working.

The input is a nested list of intervals, each interval as [start, end], with start < end. The intervals are non-overlapping and are already sorted in ascending order. The output should also be a list of sorted intervals.

For example, consider schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]. Here, Employee 1 works from 1 to 3 and 6 to 7. Employee 2 works from 2 to 4 and Employee 3 works from 2 to 5 and 9 to 12. The common free time for all employees is [5,6] and [7,9] as these are the intervals when all employees are free.

Solution Approach:

The proposed solution concatenates all work schedules into a single list, sorts this list and then identifies the gaps between work intervals, which represent the common free time for all employees.

First, the solution creates an empty array to store the result. Then, it goes through the work shifts of each employee and combines all the intervals to a single list.

Next, this combined list of work intervals is sorted using the start of the work intervals. This will ensure that the work schedules are in chronological order.

After that, the solution stores the end time of the first shift in a variable called prevEnd, and then iterates through each work interval.

If the start of a work interval is greater than prevEnd, this means there's a gap between the end of the previous work shift and the start of the current work shift. This gap represents a common free time, which is then added to the result list.

Finally, prevEnd is updated with the maximum end time of the current work interval or the prevEnd.

Through this approach, the algorithm successfully identifies the common free time of all employees.

/\*\*

 \* Definition for an interval.

 \* public class Interval {

 \*     int start;

 \*     int end;

 \*     Interval() { start = 0; end = 0; }

 \*     Interval(int s, int e) { start = s; end = e; }

 \* }

 \*/

public class Solution {

    public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {

        List<Interval> res = new ArrayList<>();

        List<Interval> intervals = new ArrayList<>();

        // Flattening the schedule

        for (List<Interval> employee : schedule)

            for (Interval interval : employee)

                intervals.add(interval);

        // Sorting by start of each Interval

        Collections.sort(intervals, (a, b) -> a.start - b.start);

        int end = intervals.get(0).end;

        // Checking for free time between intervals

        for (Interval interval : intervals) {

            if (interval.start > end)

                res.add(new Interval(end, interval.start));

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

        }

        return res;

    }

}

These solutions work by first flattening the schedule list, then sorting the intervals by their start times. They then keep track of the end time of the current interval. If the start of the next interval is greater than this end time, it means there is a gap where all employees are free, and this gap is then added to the results. These solutions have a time complexity of O(N log N) due to the sorting operation, where N is the total number of intervals. The space complexity is also O(N) for storing the intervals and the result.

**How to merge intervals ?**

**Refer to L56.P5.1.Merge Intervals**

class Solution {

    public int[][] merge(int[][] intervals) {

        if(intervals.length == 1) {

            return intervals;

        }

        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

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

        int[] newInterval = intervals[0];

        // Must add 'newInterval' into the result as a placeholder,

        // since 'int[] newInterval' is an object reference, we can

        // update it in further loop without delete old then add new

        // operation

        result.add(newInterval);

        for(int[] interval : intervals) {

            if(newInterval[1] >= interval[0]) {

                // Why have to compare recursively ? Because previous

                // interval may cover multiple later intervals if

                // compare the 'end'

                // e.g intervals = [[1,11],[2,6],[8,10],[15,18]]

                // if no Math.max() check and directly use 'interval[1]',

                // then the correct newInterval's end 11 will be mistakenly

                // ignored and only set as [8,10]'s end 10 when loop to

                // third element [8,10]

                newInterval[1] = Math.max(newInterval[1], interval[1]);

            } else {

                // Q1: Why update 'newInterval' with 'interval' first and

                // then add 'newInterval' into result ?

                // Because 'newInterval' is a reference and will be used

                // for 'end' comparison logic in further iterations

                // Q2: Why "result.add(interval)" won't work ?

                // Because 'interval' is an isoldated reference than 'newInterval',

                // won't able to support further iterations 'end' comparison logic

                // Q3: Why cannot converse these two statements order ?

                // Because we need to refresh the value of 'newInterval'

                // since we find previous merged or not merged interval should

                // terminate, the round only can start with new 'interval' value,

                // but the 'end' comparison reference and logic still based on

                // 'newInterval', hence update 'newInterval' with 'interval''s value

                newInterval = interval;

                result.add(newInterval);

                //newInterval = interval;

            }

        }

        return result.toArray(new int[result.size()][]);

    }

}

**Solution 2: Line Sweep + Two Pointers (30 min)**

**Two Pointers handling way exactly same as L1943.Describe the Painting**

class Solution {

    class Interval {

        int start;

        int end;

        public Interval() {

            this.start = 0;

            this.end = 0;

        }

        public Interval(int s, int e) {

            this.start = s;

            this.end = e;

        }

    }

    public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {

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

        for(List<Interval> employee : schedule) {

            for(Interval interval : employee) {

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

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

            }

        }

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

        // Below Two Pointers handling way exactly same as L1943.Describe the Painting

        // 'count' is the running presum, the accumulative sum of delta array

        int count = 0;

        // 'start' record the start of all employee common free interval start timestamp

        // 'end' record the start of all employee common free interval end timestamp

        int start = -1;

        int end = -1;

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

            end = e.getKey();

            // The only difference between L1943.Describe the Painting is

            // second condition 'count == 0' (in L1943 is 'count != 0')

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

                result.add(new Interval(start, end));

            }

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

            count += e.getValue();

            start = end;

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Compleixty: O(N)

**Step by Step**

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.

Wrong way to '-' at [end + 1]

                       +                            -     -> [4,10+1)

           +           -                                  -> [1,3+1)

                           +       -                      -> [5,6+1)

           +       -                                      -> [1,2+1)

time   0   1   2   3   4   5   6   7   8   9   10   11

Delta  0   2   0  -1   0   1   0  -1   0   0    0   -1

presum 0   2   2   1   1   2   2   1   1   1    1    0

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

Correct way to '-' at [end]

                       +                       -         -> [4,10]

           +       -                                     -> [1,3]

                           +   -                         -> [5,6]

           +   -                                         -> [1,2]

time   0   1   2   3   4   5   6   7   8   9   10

Delta  0   2  -1  -1   1   1  -1   0   0   0   -1

presum 0   2   1   0   1   2   1   1   1   1    0

                   ^   #

The presum(running accumulate sum of delta) '0' at timestamp 3 (tag with ^)

is what we intreset with: it means all employees free interval start at

this moment, and all employees free interval end at timestamp 4 (tag with #)

which set by coming interval 'start' timestamp (e.g [4,10] is the coming

interval right after timestamp 3), so all employee free interval is [3,4]

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

Example 2:

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

Output: [[5,6],[7,9]]

Wrong way to '-' at [end + 1]

                                           +                  -  -> [9,12+1)

               +               -                                 -> [2,5+1)

               +           -                                     -> [2,4+1)

                               +       -                         -> [6,7+1)

           +           -                                         -> [1,3+1)

time   0   1   2   3   4   5   6   7   8   9   10   11   12   13

Delta  0   1   2   0  -1  -1   0   0  -1   1    0    0    0   -1

presum 0   1   3   3   2   1   1   1   0   1    1    1    1    0

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

Correct way to '-' at [end]

                                           +             -       -> [9,12]

               +           -                                     -> [2,5]

               +       -                                         -> [2,4]

                               +   -                             -> [6,7]

           +       -                                             -> [1,3]

time   0   1   2   3   4   5   6   7   8   9   10   11   12

Delta  0   1   2  -1  -1  -1   1  -1   0   1   0    0    -1

presum 0   1   3   2   1   0   1   0   0   1   1    1    0

                           ^   #   ^       #

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

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/data-structure/759.employee-free-time>

Approach 1: Events (Line Sweep)

**Complexity Analysis**

Time Complexity: O\_(\_C\_log\_C), where C is the number of intervals across all employees.

Space Complexity: O\_(\_C).

/\*

// Definition for an Interval.

class Interval {

    public int start;

    public int end;

    public Interval() {}

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

        start = \_start;

        end = \_end;

    }

};

\*/

class Solution {

  public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {

        int OPEN = 0, CLOSE = 1;

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

    for (List<Interval> employee : schedule) {

      for (Interval iv : employee) {

        events.add(new int[] {iv.start, OPEN});

        events.add(new int[] {iv.end, CLOSE});

      }

    }

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

    List<Interval> ans = new ArrayList();

    int prev = -1, bal = 0;

    for (int[] event : events) {

      if (bal == 0 && prev >= 0) {

        ans.add(new Interval(prev, event[0]));

      }

      bal += (event[1] == OPEN ? 1 : -1);

      prev = event[0];

    }

    return ans;

  }

}

**Refer to**

<https://grandyang.com/leetcode/759/>

我们再来看一种解法，这种解法挺巧妙的，我们使用TreeMap建立一个位置和其出现次数之间的映射，对于起始位置，进行正累加，对于结束位置，进行负累加。由于TreeMap具有自动排序的功能，所以我们进行遍历的时候，就是从小到大进行遍历的。定义一个变量cnt，初始化为0，我们对于每个遍历到的数，都加上其在TreeMap中的映射值，即该数字出现的次数，起始位置的话就会加正数，结束位置就是加负数。开始的时候，第一个数字一定是个起始位置，那么cnt就是正数，那么接下来cnt就有可能加上正数，或者减去一个负数，我们想，如果第一个区间和第二个区间没有交集的话，那么接下来遇到的数字就是第一个区间的结束位置，所以会减去1，这样此时cnt就为0了，这说明一定会有中间区域存在，所以我们首先把第一个区间当前起始位置，结束位置暂时放上0，组成一个区间放到结果res中，这样我们在遍历到下一个区间的时候更新结果res中最后一个区间的结束位置。语言描述难免太干巴巴的，我们拿题目中的例1来说明，建立好的TreeMap如下所示：

1 -> 2

2 -> -1

3 -> -1

4 -> 1

5 -> 1

6 -> -1

10 -> -1

那么开始遍历这所有的映射对，cnt首先为2，然后往后遍历下一个映射对2 -> -1，此时cnt为1了，不进行其他操作，再往下遍历，下一个映射对3 -> -1，此时cnt为0了，说明后面将会出现断层了，我们将(3, 0)先存入结果res中。然后遍历到4 -> 1时，cnt为1，此时将结果res中的(3, 0)更新为 (3, 4)。然后到5 -> 1，此时cnt为2，不进行其他操作，然后到6 -> -1，此时cnt为1，不进行其他操作，然后到10 -> -1，此时cnt为0，将(10, 0)加入结果res中。由于后面再没有任何区间了，所以res最后一个区间不会再被更新了，我们应该将其移出结果res，因为题目中限定了区间不能为无穷，参见代码如下：

解法二：

class Solution {

    public:

    vector<Interval> employeeFreeTime(vector<vector<Interval>>& schedule) {

        vector<Interval> res;

        map<int, int> m;

        int cnt = 0;

        for (auto employee : schedule) {

            for (Interval i : employee) {

                ++m[i.start];

                --m[i.end];

            }

        }

        for (auto a : m) {

            cnt += a.second;

            if (!cnt) res.push\_back(Interval(a.first, 0));

            if (cnt && !res.empty() && !res.back().end) res.back().end = a.first;

        }

        if (!res.empty()) res.pop\_back();

        return res;

    }

};