## Employee Free Time

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`.

## Solution 1

The idea is to just add all the intervals to the priority queue. (NOTE that it is not matter how many different people are there for the algorithm. becuase we just need to find a gap in the time line.

1. priority queue - sorted by start time, and for same start time sort by either largest end time or smallest (it is not matter).
2. Everytime you poll from priority queue, just make sure it doesn't intersect with previous interval.
   This mean that there is no coomon interval. Everyone is free time.

```
    public List<Interval> employeeFreeTime(List<List<Interval>> avails) {

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

        PriorityQueue<Interval> pq = new PriorityQueue<>((a, b) -> a.start - b.star
t);
        avails.forEach(e -> pq.addAll(e));

        Interval temp = pq.poll();
        while(!pq.isEmpty()) {
            if(temp.end < pq.peek().start) { // no intersect
                result.add(new Interval(temp.end, pq.peek().start));
                temp = pq.poll(); // becomes the next temp interval
            }else { // intersect or sub merged
                temp = temp.end < pq.peek().end ? pq.peek() : temp;
                pq.poll();
            }
        }
        return result;
    }
```

@Majcpatrick ask why priority queue. I think it's not needed. Below is just array list.

```
    public List<Interval> employeeFreeTime(List<List<Interval>> avails) {
        List<Interval> result = new ArrayList<>();
        List<Interval> timeLine = new ArrayList<>();
        avails.forEach(e -> timeLine.addAll(e));
        Collections.sort(timeLine, ((a, b) -> a.start - b.start));

        Interval temp = timeLine.get(0);
        for(Interval each : timeLine) {
            if(temp.end < each.start) {
                result.add(new Interval(temp.end, each.start));
                temp = each;
            }else{
                temp = temp.end < each.end ? each : temp;
            }
        }
        return result;
    }
```

written by wavy original link here

## Solution 2

```
class Solution {
public:
    vector<Interval> employeeFreeTime(vector<vector<Interval>>& a) {
        vector<Interval> res;
        map<int, int> timeline;
        for (int i = 0; i < a.size(); i++) {
            for (int j = 0; j < a[i].size(); j++) {
                timeline[a[i][j].start]++;
                timeline[a[i][j].end]--;
            }
        }
        int workers = 0;
        for (pair<int, int> p : timeline) {
            workers += p.second;
            if (!workers) res.push_back(Interval(p.first, 0));
            if (workers && !res.empty() && !res.back().end) res.back().end = p.firs
t;
        }
        if (!res.empty()) res.pop_back();
        return res;
    }
};
```

written by alexander original link here

## Solution 3

The idea is to sort all the intervals based on the starting time. this gives a set of busy intervals. After merging all the busy times, the gaps in between, form the free time common to everyone in the list.
Using a stack, we can start merging the intervals, and whenever a new interval is pushed to the stack, the time between the last seen interval and the new interval is a free time that can be added to the result.

```
def employeeFreeTime(self, avails):
    avails = list(sorted(chain(*avails), key=lambda interval: interval.start))
    stack = [avails[0]]
    res = list()
    for cur in avails[1:]:
        top = stack.pop()
        if top.end < cur.start:
            res.append(Interval(top.end, cur.start))
            stack.append(cur)
        else:
            if cur.end <= top.end:
                stack.append(top)
            else:
                stack.append(cur)
    return res
```

written by johnyrufus16 original link here