

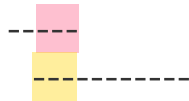
Merge Interval

use cases : meeting room , train arrival departure problems.

conditions :

1. If intervals overlaps, means end of current interval is greater than start of next interval. Means intervals are overlapping. exam : 3 is greater than 2

1 2 3 4 5 6



color overlapping region.

In this case, end of current interval should be replaced with end of next interval

Exam : $[[1,3], [2,6]]$: result should be $[1,6]$: intervals are overlapped.

2. If one interval can consume another interval completely, then no merging is required

Exam: $[[1,6],[2,4]] = [1,6]$

3. If two intervals are not overlapping means end of current interval is less than start of next interval. Then new interval is found and **No merging** will be performed.

Exam: $[[1,4], [6,8]] = [[1,4], [6,8]]$

1 2 3 4 5 6 7 8



no overlapping

PS: <https://leetcode.com/problems/merge-intervals/>

```
class Solution {
    static class Pair implements Comparable<Pair>{
        int start;
        int end;

        public Pair(int start , int end){
            this.start = start;
            this.end = end;
        }
        @Override
        public int compareTo(Pair other){
            if(this.start != other.start){
                return this.start - other.start;
            }else{
```

```

        return this.end - other.end;
    }
}

public int[][] merge(int[][] intervals) {
    Pair[] pairs = new Pair[intervals.length];
    for(int i=0; i < intervals.length; i++){
        pairs[i] = new Pair(intervals[i][0], intervals[i][1]);
    }
    Arrays.sort(pairs);
    Stack<Pair> sp = new Stack<>();
    sp.push(pairs[0]);
    for(int i=1; i < pairs.length; i++){
        Pair prev = sp.peek();
        Pair curr = pairs[i];

        if(prev.end > curr.start && prev.end > curr.end){
            continue;
        }else if(prev.end >= curr.start && (prev.end < curr.end)){
            sp.pop();
            prev.end = curr.end;
            sp.push(prev);
        }else if(prev.end < curr.start){
            sp.push(curr);
        }
    }

    Stack<Pair> fsp = new Stack<>();

    while(!sp.isEmpty()){
        fsp.push(sp.pop());
    }
    int[][] res = new int[fsp.size()][2];

    int i = 0;
    while(!fsp.isEmpty()){
        Pair p = fsp.pop();
        res[i][0] = p.start;
        res[i][1] = p.end;
        i++;
    }
    return res;
}
}

```

1. Meeting room 1:

<https://leetcode.com/problems/meeting-rooms/>

PS : Given an array of meeting time intervals where `intervals[i] = [starti, endi]`, determine if a person could attend all meetings.

solution : A person can attend a meet if meetings are not overlapping.

If one meeting time can be consumed by other meeting completely, means person can attend that meeting too.

```
class Solution {
    public boolean canAttendMeetings(int[][] intervals) {
        if(intervals.length == 0){
            return true;
        }
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        int fs = intervals[0][0]; // 7
        int fe = intervals[0][1]; // 10

        for(int i = 1; i < intervals.length; i++){
            int ns = intervals[i][0]; // 2
            int ne = intervals[i][1]; // 4
            if(fs > ns && fe > ne){
                return true;
            } else if(fe > ns){
                return false;
            }
            fs = ns;
            fe = ne;
        }
        return true;
    }
}
```

2. Meeting room 2:

<https://leetcode.com/problems/meeting-rooms-ii/>

```
public int minMeetingRooms(Interval[] intervals) {
    if (intervals == null || intervals.length == 0)
        return 0;

    // Sort the intervals by start time
    Arrays.sort(intervals, new Comparator<Interval>() {
        public int compare(Interval a, Interval b) { return a.start - b.start; }
    });

    // Use a min heap to track the minimum end time of merged intervals
    PriorityQueue<Interval> heap = new PriorityQueue<Interval>(intervals.length,
        new Comparator<Interval>() {
            public int compare(Interval a, Interval b) { return a.end - b.end; }
        });
}
```

```

});

// start with the first meeting, put it to a meeting room
heap.offer(intervals[0]);

for (int i = 1; i < intervals.length; i++) {
    // get the meeting room that finishes earliest
    Interval interval = heap.poll();

    if (intervals[i].start >= interval.end) {
        // if the current meeting starts right after
        // there's no need for a new room, merge the interval
        interval.end = intervals[i].end;
    } else {
        // otherwise, this meeting needs a new room
        heap.offer(intervals[i]);
    }

    // don't forget to put the meeting room back
    heap.offer(interval);
}

return heap.size();
}

```