题目描述（中等难度）

56. Merge Intervals

Medium 1826 139 Favorite 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.
```

给定一个列表，将有重叠部分的合并。例如[[1 3] [2 6]] 合并成 [1 6] 。

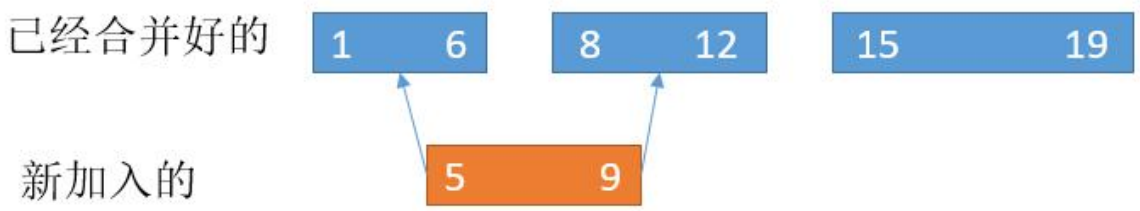
解法一

常规的思想，将大问题化解成小问题去解决。

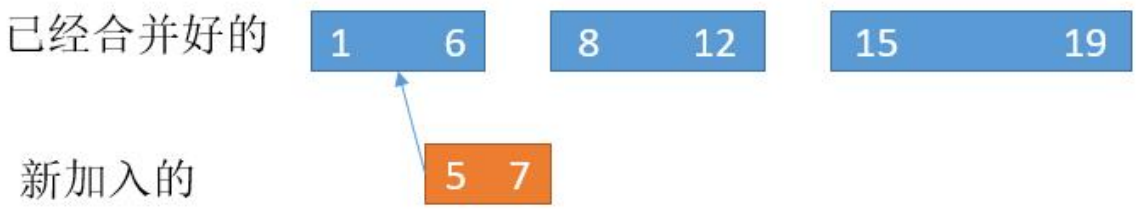
假设给了一个大小为 n 的列表，然后我们假设 n - 1 个元素的列表已经完成了全部合并，我们现在要解决的就是剩下的 1 个，怎么加到已经合并完的 n -1 个元素中。

这样的话分下边几种情况， 我们把每个范围叫做一个节点，节点包括左端点和右端点。

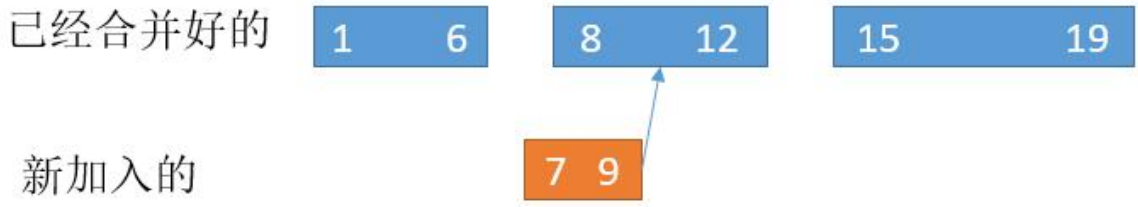
1. 如下图，新加入的节点左端点和右端点，分别在两个节点之间。这样，我们只要删除这两个节点，并且使用左边节点的左端点，右边的节点的右端点作为一个新节点插入即可。也就是删除 [1 6] 和 [8 12] ，加入 [1 12] 到合并好的列表中。



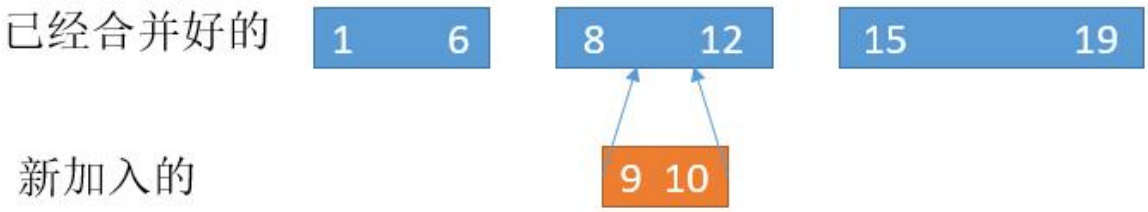
2. 如下图，新加入的节点只有左端点在之前的一个节点之内，这样的话将这个节点删除，使用删除的节点的左端点，新加入的节点的右端点，作为新的节点插入即可。也就是删除 [1 6]，加入 [1 7] 到合并好的列表中。



3. 如下图，新加入的节点只有右端点在之前的一个节点之内，这样的话将这个节点删除，使用删除的节点的右端点，新加入的节点的左端点，作为新的节点插入即可。也就是删除 [8 12]，加入 [7 12] 到合并好的列表中。

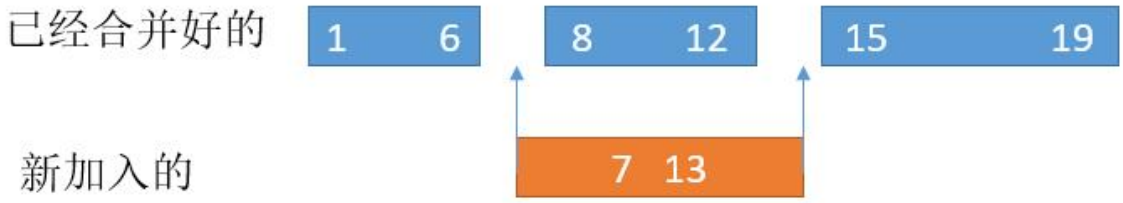


4. 如下图，新加入的节点的左端点和右端点在之前的一个节点之内，这样的话新加入的节点舍弃就可以了。



5. 如下图，新加入的节点没有在任何一个节点之内，那么将它直接作为新的节点加入到合并好的节点之内就可以了。

6. 如下图，还有一种情况，就是新加入的节点两个端点，将之前的节点囊括其中，这种的话，我们只需要将囊括的节点删除，把新节点加入即可。把 [8 12] 删除，将 7 13 加入即可。并且，新加入的节点可能会囊括多个旧节点，比如新加入的节点是 [1 100]，那么下边的三个节点就都包括了，就需要都删除掉。



以上就是所有的情况了，可以开始写代码了。

```
public class Interval {
    int start;
    int end;

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

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

public List<Interval> merge(List<Interval> intervals) {
    List<Interval> ans = new ArrayList<>();
    if (intervals.size() == 0)
        return ans;
    //将第一个节点加入，作为合并好的节点列表
    ans.add(new Interval(intervals.get(0).start, intervals.get(0).end));
    //遍历其他的每一个节点
    for (int i = 1; i < intervals.size(); i++) {
        Interval start = null;
        Interval end = null;
        //新加入节点的左端点
        int i_start = intervals.get(i).start;
        //新加入节点的右端点
        int i_end = intervals.get(i).end;
        int size = ans.size();
        //情况 6，保存囊括的节点，便于删除
        List<Interval> in = new ArrayList<>();
        //遍历合并好的每一个节点
        for (int j = 0; j < size; j++) {
            //找到左端点在哪个节点内
            if (i_start >= ans.get(j).start && i_start <= ans.get(j).end) {
                start = ans.get(j);
            }
            //找到右端点在哪个节点内
```

```
            if (i_end >= ans.get(j).start && i_end <= ans.get(j).end) {
                end = ans.get(j);
            }
            //判断新加入的节点是否囊括当前旧节点，对应情况 6
            if (i_start < ans.get(j).start && i_end >ans.get(j).end) {
                in.add(ans.get(j));
            }

        }
        //删除囊括的节点
        if (in.size() != 0) {
            for (int index = 0; index < in.size(); index++) {
                ans.remove(in.get(index));
            }
        }
        //equals 函数作用是在 start 和 end 有且只有一个 null，或者 start 和 end 是同一个节点返回 true，相当于情况 2 3 4 中的一种
        if (equals(start, end)) {
            //如果 start 和 end 都不等于 null 就代表情况 4

            // start 等于 null 的话相当于情况 3
            int s = start == null ? i_start : start.start;
            // end 等于 null 的话相当于情况 2
            int e = end == null ? i_end : end.end;
            ans.add(new Interval(s, e));
            // start 和 end 不是同一个节点，相当于情况 1
        } else if (start!= null && end!=null) {
            ans.add(new Interval(start.start, end.end));
            // start 和 end 都为 null，相当于情况 5 和 情况 6 ，加入新节点
        }else if (start == null) {
            ans.add(new Interval(i_start, i_end));
        }
        //将旧节点删除
        if (start != null) {
            ans.remove(start);
        }
        if (end != null) {
            ans.remove(end);
        }

    }
    return ans;
}

private boolean equals(Interval start, Interval end) {
    if (start == null && end == null) {
        return false;
    }
    if (start == null || end == null) {
```

```
            return true;
        }
        if (start.start == end.start && start.end == end.end) {
            return true;
        }
        return false;
    }
```

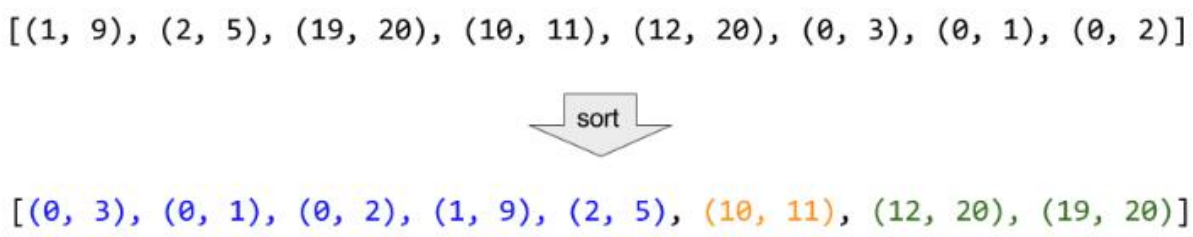
时间复杂度：O（n^2）。

空间复杂度：O （n），用来存储结果。

解法二

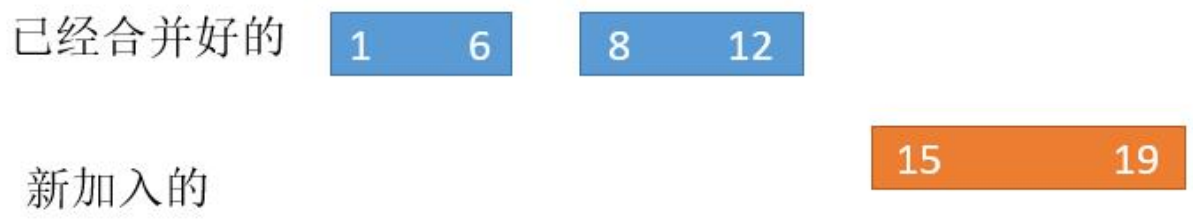
参考这里的解法二。

在解法一中，我们每次对于新加入的节点，都用一个 for 循环去遍历已经合并好的列表。如果我们把之前的列表，按照左端点进行从小到大排序了。这样的话，每次添加新节点的话，我们只需要和合并好的列表最后一个节点对比就可以了。

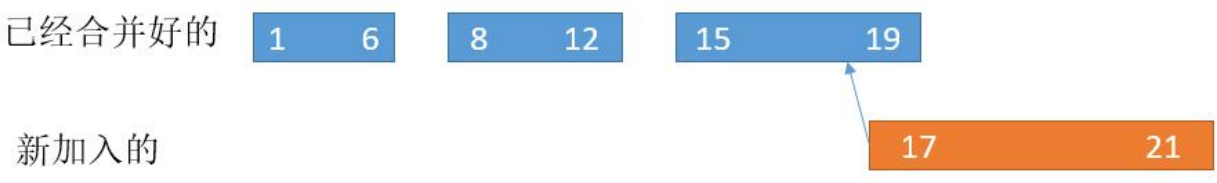


排好序后我们只需要把新加入的节点和最后一个节点比较就够了。

情况 1，如果新加入的节点的左端点大于合并好的节点列表的最后一个节点的右端点，那么我们只需要把新节点直接加入就可以了。



情况 2，如果新加入的节点的左端点不大于合并好的节点列表的最后一个节点的右端点，那么只需要判断新加入的节点的右端点和最后一个节点的右端点哪个大，然后更新最后一个节点的右端点就可以了。



```
private class IntervalComparator implements Comparator<Interval> {
    @Override
    public int compare(Interval a, Interval b) {
        return a.start < b.start ? -1 : a.start == b.start ? 0 : 1;
    }
}

public List<Interval> merge(List<Interval> intervals) {
    Collections.sort(intervals, new IntervalComparator());

    LinkedList<Interval> merged = new LinkedList<Interval>();
    for (Interval interval : intervals) {
        //最开始是空的，直接加入
        //然后对应情况 1，新加入的节点的左端点大于最后一个节点的右端点
        if (merged.isEmpty() || merged.getLast().end < interval.start) {
            merged.add(interval);
        }
        //对于情况 2 ，更新最后一个节点的右端点
        else {
            merged.getLast().end = Math.max(merged.getLast().end,
interval.end);
        }
    }

    return merged;
}
```

时间复杂度：O（n log（n））， 排序算法。

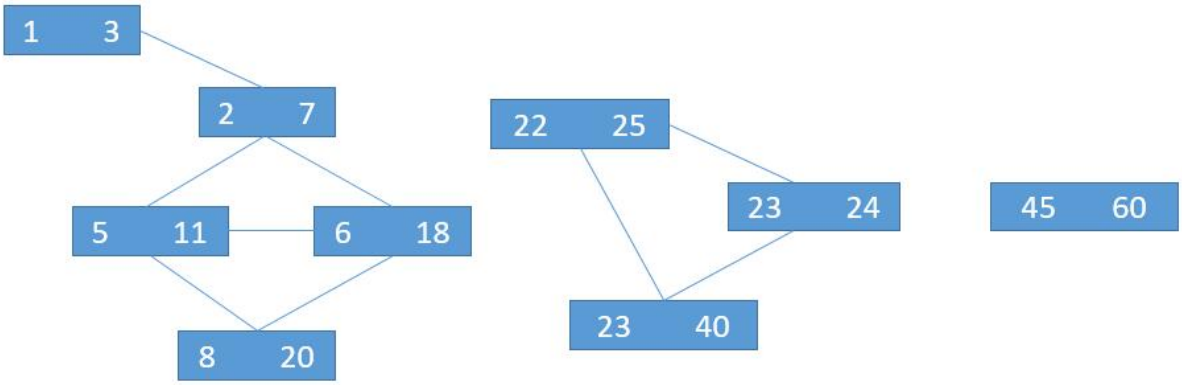
空间复杂度：O（n），存储结果。另外排序算法也可能需要。

解法三

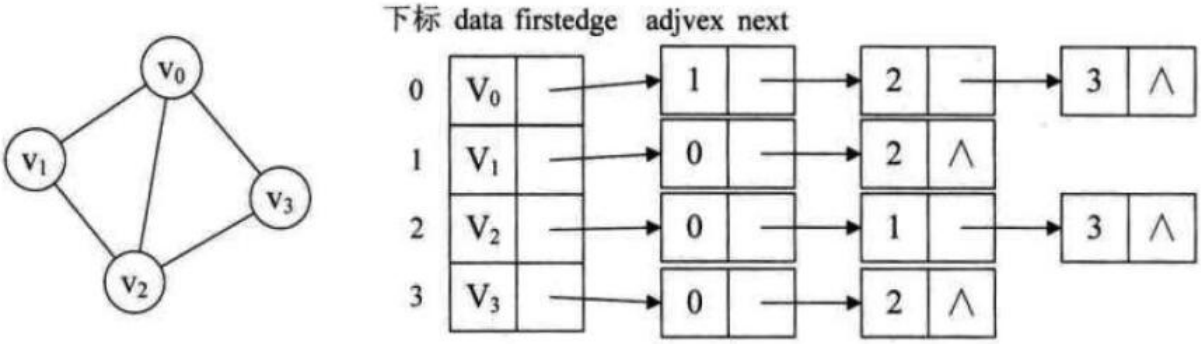
参考这里的解法 1。

刷这么多题，第一次利用图去解决问题，这里分享下作者的思路。

如果每个节点如果有重叠部分，就用一条边相连。



我们用一个 HashMap，用邻接表的结构来实现图，类似于下边的样子。



图存起来以后，可以发现，最后有几个连通图，最后合并后的列表就有几个。我们需要把每个连通图保存起来，然后在每个连通图中找最小和最大的端点作为一个节点加入到合并后的列表中就可以了。最后，我们把每个连通图就转换成下边的图了。


```
class Solution {
    private Map<Interval, List<Interval> > graph; //存储图
    private Map<Integer, List<Interval> > nodesInComp; ///存储每个有向图
    private Set<Interval> visited;

   //主函数
    public List<Interval> merge(List<Interval> intervals) {
        buildGraph(intervals); //建立图
        buildComponents(intervals); //单独保存每个有向图


        List<Interval> merged = new LinkedList<>();
        //遍历每个有向图，将有向图中最小最大的节点加入到列表中
        for (int comp = 0; comp < nodesInComp.size(); comp++) {
```

```
            merged.add(mergeNodes(nodesInComp.get(comp)));
        }

        return merged;
    }

    // 判断两个节点是否有重叠部分
    private boolean overlap(Interval a, Interval b) {
        return a.start <= b.end && b.start <= a.end;
    }

    //利用邻接表建立图
    private void buildGraph(List<Interval> intervals) {
        graph = new HashMap<>();
        for (Interval interval : intervals) {
            graph.put(interval, new LinkedList<>());
        }

        for (Interval interval1 : intervals) {
            for (Interval interval2 : intervals) {
                if (overlap(interval1, interval2)) {
                    graph.get(interval1).add(interval2);
                    graph.get(interval2).add(interval1);
                }
            }
        }
    }

   // 将每个连接图单独存起来
    private void buildComponents(List<Interval> intervals) {
        nodesInComp = new HashMap();
        visited = new HashSet();
        int compNumber = 0;

        for (Interval interval : intervals) {
            if (!visited.contains(interval)) {
                markComponentDFS(interval, compNumber);
                compNumber++;
            }
        }
    }

    //利用深度优先遍历去找到所有互相相连的边
    private void markComponentDFS(Interval start, int compNumber) {
        Stack<Interval> stack = new Stack<>();
        stack.add(start);

        while (!stack.isEmpty()) {
            Interval node = stack.pop();
```

```
            if (!visited.contains(node)) {
                visited.add(node);

                if (nodesInComp.get(compNumber) == null) {
                    nodesInComp.put(compNumber, new LinkedList<>());
                }
                nodesInComp.get(compNumber).add(node);

                for (Interval child : graph.get(node)) {
                    stack.add(child);
                }
            }
        }
    }


    // 找出每个有向图中最小和最大的端点
    private Interval mergeNodes(List<Interval> nodes) {
        int minStart = nodes.get(0).start;
        for (Interval node : nodes) {
            minStart = Math.min(minStart, node.start);
        }

        int maxEnd = nodes.get(0).end;
        for (Interval node : nodes) {
            maxEnd= Math.max(maxEnd, node.end);
        }

        return new Interval(minStart, maxEnd);
    }
}
```

时间复杂度：

空间复杂度：O（n^2），最坏的情况，每个节点都互相重合，这样每个都与其他节点相连，就会是 n^2 的空间存储图。

可惜的是，这种解法在 leetcode 会遇到超时错误。

总

开始的时候，使用最常用的思路，将大问题化解为小问题，然后用递归或者直接用迭代实现。解法二中，先对列表进行排序，从而优化了时间复杂度，也不是第一次看到了。解法三中，利用图解决问题很新颖，是我刷题第一次遇到的，又多了一种解题思路。