区间交集题目描述：

给定一组闭区间，其中部分区间存在交集。任意两个给定区间的交集，称为公共区间（如：[1,2],[2,3]的公共区间为[2,2]，[3,5],[3,6]的公共区间为[3,5]）。公共区间之间若存在交集，则需要合并（如：[1,3],[3,5]区间存在交集[3,3]，须合并为[1,5])。按升序排列输出合并后的区间列表。

输入描述：

一组区间列表，

区间数为 N：

0 <= N <= 1000；

区间元素为 X：

-10000 <= X <= 10000。

输出描述：

升序排列的合并后区间列表

补充说明：

1、区间元素均为数字，不考虑字母、符号等异常输入。

2、单个区间认定为无公共区间。

示例 1

输入：

```
0 3

1 3

3 5

3 6
```

输出：

```
1 5
```

说明：

[0,3]和[1,3]的公共区间为[1,3]，[0,3]和[3,5]的公共区间为[3,3]，[0,3]和[3,6]的公共区间为[3,3]，[1,3]和[3,5]的公共区间为[3,3]，[1,3]和[3,6]的公共区间为[3,3]，[3,5]和[3,6]的公共区间为[3,5]，公共区间列表为[[1,3],[3,3],[3,5]]；

[1,3],[3,3],[3,5]存在交集，须合并为[1,5]。

示例 2

输入：

```
0 3

1 4

4 7

5 8
```

输出：

```
1 3

4 4

5 7
```

说明：

示例 3

输入：

```
1 2
3 4
```

输出：

```
None
```

说明：

[1,2]和[3,4]无交集

```
import java.util.*;

// 注意类名必须为 Main, 不要有任何 package xxx 信息
public class Main {
        public static void main(String[] args) {
                Scanner in = new Scanner(System.in);
                // 注意 hasNext 和 hasNextLine 的区别
                List<List<Integer>> list = new LinkedList<>();

                while (in.hasNextLine()) { // 注意 while 处理多个 case
                        String line = in.nextLine();
                        if (line.isEmpty()) {
                                break;
                        }
                        String[] split = line.split(" ");
                        int start = Integer.parseInt(split[0]);
                        int end = Integer.parseInt(split[1]);
                        List<Integer> tempList = new LinkedList();
                        tempList.add(start);
                        tempList.add(end);
//                        int[] arr = {start, end};
                        list.add(tempList);
                }
                int[][] intervals = new int[list.size()][];
                for (int i = 0; i < list.size(); i++) {
                        List<Integer> tempList = list.get(i);
                        intervals[i] = tempList.stream().mapToInt(Integer::intValue).toArray();
                }

                int[][] mergedIntervals = mergeIntervals(intervals);
                if (mergedIntervals.length == 0)
                        System.out.println("None");
                for (int[] interval : mergedIntervals) {
                        System.out.println(interval[0] + " " + interval[1]);
                }
        }
        public static int[][] mergeIntervals(int[][] intervals) {
                if (intervals == null || intervals.length == 0) {
                        return new int[0][];
                }
                List<int[]> merged = new ArrayList<>();
                for (int i = 0; i < intervals.length; i++) {
```

```
            for (int j = i + 1; j < intervals.length; j++) {
                int[] interval1 = intervals[i];
                int[] interval2 = intervals[j];
                if (hasCommonInterval(interval1, interval2)) {
                    int[] commonInterval = getCommonInterval(interval1, interval2);
                    merged.add(commonInterval);
                }
            }
        }
        merged = mergeOverLappingInterval(merged);
        Collections.sort(merged, Comparator.comparingInt(a->a[0]));
        return merged.toArray(new int[merged.size()][]);
    }
    public static boolean hasCommonInterval(int[] inter1, int[] inter2) {
        return inter1[1] >= inter2[0] && inter2[1] >= inter1[0];
    }

    public static int[] getCommonInterval(int[] inter1, int[] inter2) {
        int strat = Math.max(inter1[0], inter2[0]);
        int end = Math.min(inter1[1], inter2[1]);
        return new int[] {strat, end};
    }

    public static List<int[]> mergeOverLappingInterval(List<int[]> interval) {
        List<int[]> merged = new ArrayList<>();
        if (interval.size() == 0) {
            return merged;
        }
        Collections.sort(interval, Comparator.comparingInt(a->a[0]));
        int[] currentInterval = interval.get(0);
        merged.add(currentInterval);
        for (int[] inter : interval) {
            int currentEnd = currentInterval[1];
            int nextStart = inter[0];
            int nextEnd = inter[1];
            if (nextStart <= currentEnd) {
                currentInterval[1] = Math.max(currentEnd, nextEnd);
            } else {
                currentInterval = inter;
                merged.add(currentInterval);
            }
        }
        return merged;
    }
```

```
}
```