**1. Problem Description :**

You are given a number n, representing the number of time intervals. In the next n lines, you are given a pair of space-separated numbers. The pair of numbers represents the start time and end time of a meeting (the first number is the start time and the second number is end time) Your task is to merge the meetings and print the output of the merged meeting in increasing order of start time.

Note: The given input may not be sorted by start-time.

Example) Let us say there are 6 meetings :

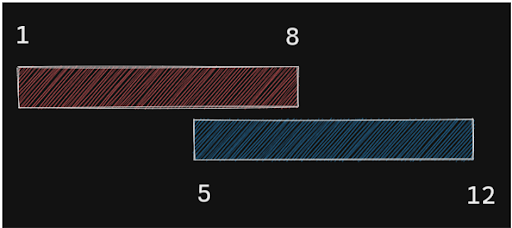
1 8 5 12 14 19 22 28 25 27 27 30

Then the output of merged meetings will be :

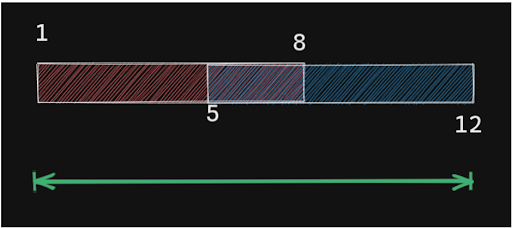
1 12 14 19 22 30

**2. Approach :**

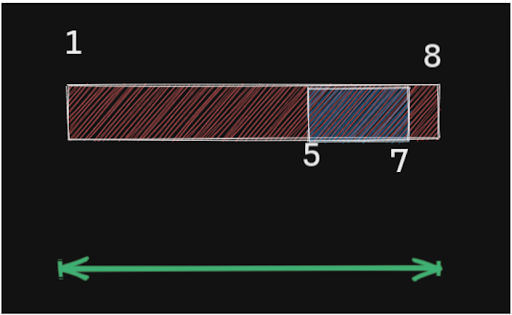
Before trying to form an approach let’s try to analyze how the meetings are being merged. From the given example we can observe that if the start time of one meeting is in between start and end time of another meeting, then they can be merged. So, one thing is for sure. If we sort the intervals in the order of their starting value. Then the two intervals that can be merged will be consecutive. This will make our task easier. So that will be our first step Sort in increasing order on the basis of the first element (starting value) Let's say there are two intervals. 1-8 and 5-12.

****

Clearly, there is an overlap. Why? Because 5 < 8. So if a start value of one is lesser than the end value of the previous then there is an overlap. So, we will merge them. Then what would be the final merged interval?

****

It will be from 1 to 12. How did we get 12? Simply max of ending of the previous and the ending of the current. This will also cover the case where the intervals are 1-8 and 5-7.

****

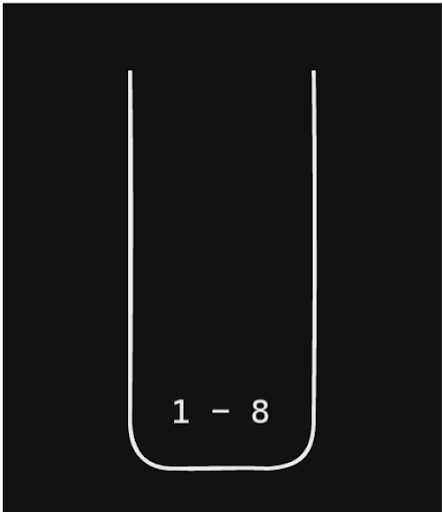
Then the final interval would still be 1-8. Because 8 is the maximum of 7 and 8. But what about the start time? Will it not be updated? Ideally the merged interval should be between this range min(start1, start2) <-> max(end1, end2) [ start1, end1 ] => is the first interval [ start2, end2 ] => is the second interval Now min(start1, start2) will always be start1. Since the array is already sorted, so we are good to go here. Also, when can we know that the intervals cannot overlap? Simple, when the start2 is greater than the end1. In simple words, if the start time of the current interval is after the end time of the previous, it will never overlap. So, basically, at every point, we will be making updates with the last element. So we need a data structure that can help make quick updates only to the last inserted item. And also push new items easily. What do you think will be the best fit? Yes, a stack. But there is one issue, the output should be in increasing order of start time. But here it will be the opposite. Then? No worries, we know to reverse a stack using another stack. That's it then. So, let's review once.

Pseudo Code :

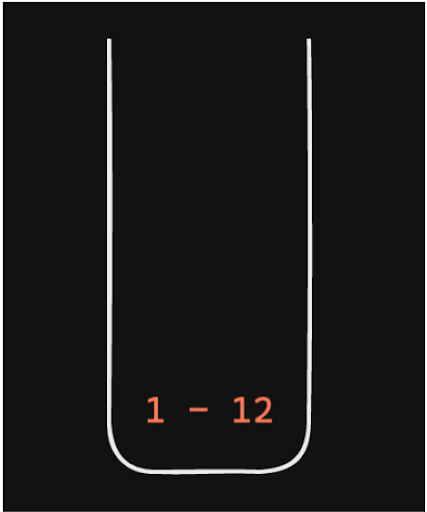
1• Sort the intervals. 2• Put the first interval in the stack. 3• Check if the next interval overlaps with the interval at the top of the stack: a. If it does overlap, update the end time of the interval at the top of the stack. b. If it doesn't simply push it into the stack. 4• Now reverse the contents of the stack to make it opposite.

Dry Run :

The first step we follow will be that we will be sorting this array. 1 8 5 12 14 19 22 28 25 27 27 30 Now we will push the first element into the stack.

****

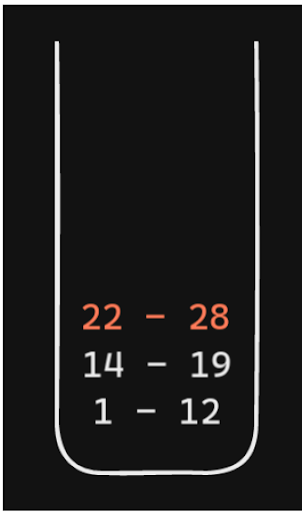
For the second element, we can see that 5<8 and 12>8 so it will merge with 1-8 and we will get the interval as 1-12 which will be pushed into the stack.

****

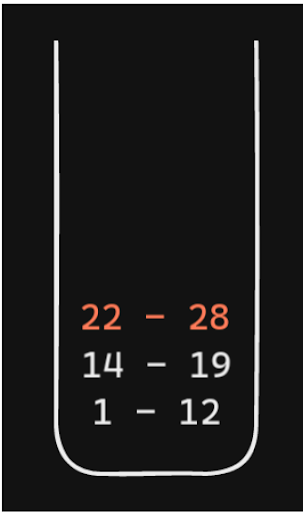
Now for 14-19, we can clearly see that 14>12 so it will be directly pushed into the stack.

****

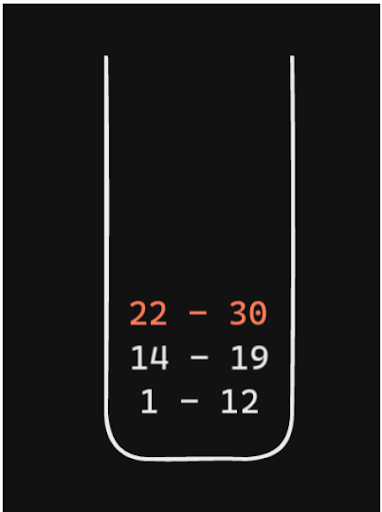
Again 22>19, it will be directly pushed.

****

Now we have 25-27. We can see that 25 < 28 but 27 < 28 too so this interval is already present in 22-18 and here no change in the end time will take place.

****

Lastly, we have 27-30. 27<28 and 30>28 so it will merge and will become 22-30 and will be pushed into the stack.

****

Now, to reverse we will pop the items from here and push them to a new stack. Then finally pop it from there to get the correct order.

**3. Code :**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static void main(String[] args) throws Exception {

// write your code here

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.parseInt(br.readLine());

int[][] arr = new int[n][2];

for (int j = 0; j < n; j++) {

String line = br.readLine();

arr[j][0] = Integer.parseInt(line.split(" ")[0]);

arr[j][1] = Integer.parseInt(line.split(" ")[1]);

}

mergeOverlappingIntervals(arr);

}

public static void mergeOverlappingIntervals(int[][] arr) {

Pair[] pairs = new Pair[arr.length];

for (int i = 0; i < arr.length; i++) {

pairs[i] = new Pair(arr[i][0], arr[i][1]);

}

Arrays.sort(pairs);

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

for (int i = 0; i < pairs.length; i++) {

if (i == 0) {

st.push(pairs[i]);

} else {

Pair top = st.peek();

if (pairs[i].st > top.et) {

st.push(pairs[i]);

} else {

top.et = Math.max(top.et, pairs[i].et);

}

}

}

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

while (st.size() > 0) {

rs.push(st.pop());

}

while (rs.size() > 0) {

Pair p = rs.pop();

System.out.println(p.st + " " + p.et);

}

}

public static class Pair implements Comparable< Pair> {

int st;

int et;

Pair(int st, int et) {

this.st = st;

this.et = et;

}

public int compareTo(Pair other) {

if (this.st != other.st) {

return this.st - other.st;

} else {

return this.et - other.et;

}

}

}

}

**4. Analysis :**

Time Complexity : O(n log(n))

Sorting will take O(nlogn) and the operations with the stack will be O(n). Because all we are doing is iterating through all the elements but making single-stack operations. So overall time complexity is: O(n + n log(n)) = O(n log(n))

Space Complexity : O(n)

We just used two extra stacks which in the worst case will have n items. The worst case is when all the intervals are disjoint i.e non-overlapping. So space complexity is O(n)