**QPREP2- Merge intervals**

**Module Introduction**

Write a program to take a set of intervals as input, merge the overlapping intervals and output the modified set of intervals.

#### Objective

Given a collection of intervals as input, merge overlapping intervals and output the result.

The input may not specify intervals in sorted order i.e a lower interval could be specified after a higher interval. The output intervals have to be in sorted order.

Note: Handle cases where an interval could overlap with more than one other interval.

#### Examples

**Example 1**

Input: [1,4],[2,5],[12,15],[8,10]

Output: [1,5],[8,10],[12,15]

Intervals [1,4] and [2,5] have overlap and have been merged into [1,5]

**Example 2**

Input: [1,3],[4,5]

Output: [1,3],[4,5]

No overlap seen; not merged.

***SOLUTION STEPS FROM NEXT PAGE:***

**Write down at least 3 examples in the following format. Kindly stick to the format.**

**Suggestion:**

EXAMPLE#1

INPUT:

5

1 2

2 4

6 10

9 10

10 200

OUTPUT:

1 4

6 200

EXAMPLE#2

INPUT:

2

1 2

3 4

OUTPUT:

1 2

3 4

EXAMPLE#3

INPUT:

5

2 5

8 20

7 23

33 50

49 70

OUTPUT:

2 5

7 23

33 70

**Detail your problem understanding here**

**Suggestion:**

There are multiple ranges provided as input. Each range will have a starting and ending number.

The solution should go through each of the ranges. For any overlapping ranges, they should be merged into a single range. The merged range will have the starting number as the minimum of all the starting numbers of the overlapping ranges and the ending number as the maximum of all the ending numbers.

We may need to create multiple consolidated ranges created across the ranges provided.

There may be multiple ranges that converge to a single range. If there is no overlap for some of the ranges, keep them as is.

Output the consolidated ranges.

**Does this problem follow a known algorithmic pattern or standard application of a data structure? If there are multiple approaches, which one would you choose and why? Write down your chosen approach in 2-3 sentences like you would explain to a 10 year old.**

The problem seems to involve standard sorting + some merge logic.

Sorting the intervals based on the start value; Traverse the intervals and combine multiple intervals together if they have overlapping ranges.

**Write the pseudocode here in plain English**

Read the input intervals as

class Interval {

int startPoint;

int endPoint;

}

Sort the intervals based on startPoint

With sorting, all intervals that need to be merged will be contiguous

Loop through the sorted intervals {

Pick the startPoint of current interval

checkAndExtendRange(currentInterval.endPoint, nextInterval)

}

Return the intervals

Comparator comes in handy to do the sorting.

SortIntervals(List of Intervals) {

Sort the intervals based on their start value and update the list

}

The function details need not be written upfront, the detail can always be hashed out when you get to this function.

checkAndExtendRange(interval1, interval2) {

If interval1 completely overlaps interval2, remove interval2 from list.

If interval1 partially overlaps with interval2, extend interval1’s interval by using max(interval1.endPoint, interval2.endPoint) and remove interval2 from list.

If interval1 doesn’t overlap with interval2, don’t do anything

}

**Can you specify a few boundary or edge cases here?**

EXAMPLE#1

INPUT:

1

1 2

OUTPUT:

1 2

EXAMPLE#2

INPUT:

5

1 5

6 10

11 15

16 20

5 16

OUTPUT:

1 20

EXAMPLE#3

INPUT:

5

5 10

16 20

15 16

10 11

12 15

OUTPUT:

5 11

12 20

**Write the functions you would create here**

List mergeIntervals (List intervals)

checkAndExtendRangeCheckOverlap(Interval interval1, Interval interval2)

List sortIntervals(List)

#### Summary

Starting with a brief explanation of the problem statement followed by pseudocode and then implementing the solution helps you approach the problem in a systematic way. This methodology helps with easy as well as hard problems.

**Time Complexity: O(n log n)**

Sorting takes (n log(n)) and an iteration through the sorted list takes (n), which brings it to (n + n log(n)). However, the runtime is dominated by the sort here and overall complexity is O(n log(n))

**Space Complexity: O(1)or O(n)**

If the sort is taking place without any additional space, then O(1). If we need a copy of the array to sort, then O(n).

#### Concepts

Concepts covered in this Module

* Array
* Sort

Similar problems

* <https://leetcode.com/problems/meeting-rooms/>
* <https://leetcode.com/problems/meeting-rooms-ii/>
* <https://leetcode.com/problems/insert-interval/>
* <https://leetcode.com/problems/employee-free-time/>

#### Good habits

Think about these for your solution:

* Comments - have you used comments in a way that others can understand this code?
* Test Cases - Are most of the scenarios/corner cases/boundary conditions handled in the solution?
* Naming Convention - Are the variables and functions named sensibly and with uniform convention?
* Modular Functions - Has the solution been addressed using concise functions? Will these functions work without any changes if they are to be used in another problem?
* Optimization - Analyze the Time Complexity and Space Complexity for your solution. Has the solution been optimized or did it use the brute force method? Is further optimization desirable/possible?
* Data Structures - Has the optimal/appropriate data structure been used?

SOLUTION:

APPROACH 1: CAUSES TIMEOUT

import java.io.\*;

import java.util.\*;

class MergeIntervals {

/\* public int[][] merge(int[][] intervals) {

int[][] results;

return results;

}\*/

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

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

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

nums[i][0] = scanner.nextInt();

nums[i][1] = scanner.nextInt();

}

scanner.close();

// System.out.println("Original");

/\* for (int i = 0; i < nums.length; ++i)

System.out.println(String.valueOf(nums[i][0]) + " " + String.valueOf(nums[i][1]));\*/

int temp;

//1) Sort the rows of the array so that always the smaller element is at 0th location of a row

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

if(nums[i][0]>nums[i][1])

{

temp = nums[i][0];

nums[i][0] = nums[i][1];

nums[i][1] = temp;

}

}

/\*System.out.println("columns in rows sorted!");

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

System.out.println(String.valueOf(nums[i][0]) + " " + String.valueOf(nums[i][1]));\*/

//2) Sort the columns now according to the row values, such that smallest 0th column row is at top

int zero,one;

for (int i = 0; i < n-1; i++) {

for (int j = i+1 ; j < n;j++)

if(nums[i][0]>nums[j][0])

{

zero = nums[i][0];

one = nums[i][1];

nums[i][0] = nums[j][0];

nums[i][1] = nums[j][1];

nums[j][0] = zero;

nums[j][1] = one;

}

}

// System.out.println("Array is sorted to need!");

// int[][] results = new MergeIntervals().merge(nums);

// for (int i = 0; i < results.length; ++i)

/\* for (int i = 0; i < nums.length; ++i)

System.out.println(String.valueOf(nums[i][0]) + " " + String.valueOf(nums[i][1]));\*/

int max,min;

//3)for each row,

for(int i = 0;i<n-1;i++)

{

// set row[0 ]to min, row[1] to max

min = nums[i][0];

max = nums[i][1];

// if any row has 0 and 1 index as -1, ignore it

if(max != -1 && min != -1 )

{

// check all other rows after it for

for(int j = i+1; j<n;j++)

{

// iii)0th element of any such row which belongs within min, max

if(nums[j][0]>=min && nums[j][0]<=max)

{

// a) check if its [1] index is greater than max, then update max, and change the values of 0 and 1 index of this row to -1.

if(nums[j][1]>max)

{

max = nums[j][1];

nums[j][0] = -1;

nums[j][1] = -1;

}

else if(nums[j][1]<=max)

{

nums[j][0] = -1;

nums[j][1] = -1;

}

}

}

//after inner loop ends, update ith row 1 index with max

if (max > nums[i][1])

nums[i][1] = max;

}

}

// after both the for loop ends, copy elements of this matrix which dont have -1 as values of entire row to a result matrix

//System.out.println("FINALLLLLLLLLLLLLLLL SOLUTION");

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

{

if(nums[i][0]!=-1 && nums[i][1]!=-1)

System.out.println(String.valueOf(nums[i][0]) + " " + String.valueOf(nums[i][1]));

}

}

}

**Complexity Analysis:**

* **Time Complexity:**
* **Space Complexity:**

APPROACH 2: Solution Works

public int[][] merge(int[][] intervals) {

if(intervals == null || intervals.length == 0)return new int[][]{};

Arrays.sort(intervals,(a,b) -> a[0]-b[0]);//sort them by starting time

HashSet<Integer> set = new HashSet<Integer>();//to store the index which needs to be deleted

for(int i=0;i<intervals.length-1;i++){

int []interval\_1 = intervals[i];

int []interval\_2 = intervals[i+1];

if(interval\_2[0] <= interval\_1[1]){ //start time of the second interval will be less then end time then overlap exist

interval\_2[0] = interval\_1[0];//set the start time as of the first interval

interval\_2[1] = Math.max(interval\_2[1],interval\_1[1]);//set the larger end time

set.add(i);//store index to delete the item

}

}

//answer intervals will be total input size - intervals marked for deletion as they are merged

int[][]ans = new int[intervals.length-set.size()][2];

int index = 0;

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

if(!set.contains(i)){//if index exist do not consider that

ans[index++] = intervals[i];

}

}

return ans;

}

**Complexity Analysis:**

* **Time Complexity:**
* **Space Compelxity:**

APPROACH 3:

public int[][] merge(int[][] intervals) {

if(intervals.length<=1 || intervals == null){

return intervals;

}

Arrays.sort(intervals,(int [] newArr, int [] alreadyPresent)->(newArr[0] - alreadyPresent[0]));

List<int []> result = new ArrayList<>();

result.add(intervals[0]);

int [] top = result.get(0);

for(int [] interval : intervals){

if(interval[0]<=top[1]){

/\*merge here\*/

top[1] = Math.max(top[1],interval[1]);

}else{

/\* cannot merge with top \*/

/\* (1,6) (8,10) earlier top was (1,6) now we push (8.10 ) in list and top becomes (8,10) \*/

top = interval;

result.add(top);

}

}

return result.toArray(new int [result.size()][]);

}

**Complexity Analysis:**

* **Time Complexity:**
* **Space Complexity:**

**Approach 4: works with more modularity**

private class IntervalComparator implements Comparator<int[]> {

@Override

public int compare(int[] a, int[] b) {

return a[0] < b[0] ? -1 : a[0] == b[0] ? 0 : 1;

}

}

public int[][] merge(int[][] intervals) {

Collections.sort(Arrays.asList(intervals), new IntervalComparator());

LinkedList<int[]> merged = new LinkedList<>();

for (int[] interval : intervals) {

// if the list of merged intervals is empty or if the current

// interval does not overlap with the previous, simply append it.

if (merged.isEmpty() || merged.getLast()[1] < interval[0]) {

merged.add(interval);

}

// otherwise, there is overlap, so we merge the current and previous

// intervals.

else {

merged.getLast()[1] = Math.max(merged.getLast()[1], interval[1]);

}

}

return merged.toArray(new int[merged.size()][]);

}