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

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

**Example 1:**

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

**Example 2:**

Input: intervals = [[1,4],[4,5]]

Output: [[1,5]]

Explanation: Intervals [1,4] and [4,5] are considered overlapping.

**Constraints:**

* 1 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= starti <= endi <= 104

**Attempt 1: 2023-03-01**

**Solution 1: Sorting (60 min)**

**Style 1: Add first element reference onto list initially**

class Solution {

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

if(intervals.length == 1) {

return intervals;

}

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

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

int[] newInterval = intervals[0];

// Must add 'newInterval' into the result as a placeholder,

// since 'int[] newInterval' is an object reference, we can

// update it in further loop without delete old then add new

// operation

result.add(newInterval);

for(int[] interval : intervals) {

if(newInterval[1] >= interval[0]) {

// Why have to compare recursively ? Because previous

// interval may cover multiple later intervals if

// compare the 'end'

// e.g intervals = [[1,11],[2,6],[8,10],[15,18]]

// if no Math.max() check and directly use 'interval[1]',

// then the correct newInterval's end 11 will be mistakenly

// ignored and only set as [8,10]'s end 10 when loop to

// third element [8,10]

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

} else {

// Q1: Why update 'newInterval' with 'interval' first and

// then add 'newInterval' into result ?

// Because 'newInterval' is a reference and will be used

// for 'end' comparison logic in further iterations

// Q2: Why "result.add(interval)" won't work ?

// Because 'interval' is an isoldated reference than 'newInterval',

// won't able to support further iterations 'end' comparison logic

// Q3: Why cannot converse these two statements order ?

// Because we need to refresh the value of 'newInterval'

// since we find previous merged or not merged interval should

// terminate, the round only can start with new 'interval' value,

// but the 'end' comparison reference and logic still based on

// 'newInterval', hence update 'newInterval' with 'interval''s value

newInterval = interval;

result.add(newInterval);

//newInterval = interval;

}

}

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

}

}

**Refer to**

<https://leetcode.com/problems/merge-intervals/solutions/21222/a-simple-java-solution/>

The idea is to sort the intervals by their starting points. Then, we take the first interval and compare its end with the next intervals starts. As long as they overlap, we update the end to be the max end of the overlapping intervals. Once we find a non overlapping interval, we can add the previous "extended" interval and start over.

Sorting takes O(n log(n)) and merging the intervals takes O(n). So, the resulting algorithm takes O(n log(n)).

I used a lambda comparator (Java 8) and a for-each loop to try to keep the code clean and simple.

EDIT: The function signature changed in April 2019, here is a new version of the algorithm with arrays. To make more memory efficient, I reused the initial array (sort of "in-place") but it would be easy to create new subarrays if you wanted to keep the initial data, it takes less memory than 99% of the other solutions (sometimes 90% depending on the run) and is more than 10 times faster than the previous version with lists.

class Solution {

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

if (intervals.length <= 1)

return intervals;

// Sort by ascending starting point

Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0], i2[0]));

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

int[] newInterval = intervals[0];

result.add(newInterval);

for (int[] interval : intervals) {

if (interval[0] <= newInterval[1]) // Overlapping intervals, move the end if needed

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

else { // Disjoint intervals, add the new interval to the list

newInterval = interval;

result.add(newInterval);

}

}

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

}

}

**Style 2: Using 'prev' dummy placeholder instead of adding first element onto list initially**

class Solution {

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

if(intervals.length == 1) {

return intervals;

}

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

int[] prev = null;

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

for(int[] interval : intervals) {

// If previous interval not defined(=null) or current

// interval.start > prev.end means no overlapping between

// two intervals, add current interval onto the result

if(prev == null || prev[1] < interval[0]) {

result.add(interval);

prev = interval;

// else if current interval.start <= prev.end, modify the element

// which already in list by extending previous interval end

// Two tips:

// 1.Update 'prev' not 'interval', because 'prev' should update

// to "current interval value" and prepare for next iteration

// comparison between "current interval value" and "new interval value"

// 2.only when current interval.end > previous interval end

// e.g intervals = [[1,11],[2,6],[8,10],[15,18]]

// if no Math.max() check and directly use 'interval[1]',

// then the correct newInterval's end 11 will be mistakenly

// ignored and only set as [8,10]'s end 10 when loop to

// third element [8,10]

} else {

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

}

}

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

}

}

**Refer to**

<https://leetcode.com/problems/merge-intervals/solutions/21222/a-simple-java-solution/comments/263749>

Using prev makes so much sense, and readable, instead of newInterval in the OP which confused me. A good variable name really matters.

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

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

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

int[] prev = null;

for (int[] inter : intervals) {

//if prev is null or curr.start > prev.end, add the interval

if (prev==null || inter[0] > prev[1]) {

ret.add(inter);

prev = inter;

} else if (inter[1] > prev[1]) {

// curr.end > prev.end, modify the element already in list

prev[1] = inter[1];

}

}

return ret.toArray(new int[ret.size()][2]);

}

**Style 3: Use the iterator to iterate through the original list and then directly modify it without create new object (60min, required by Facebook interview)**

**Why we have to use Collections.sort() instead of Arrays.sort() when using Iterator ?**

**The Iterator interface in Java is a part of the Collections framework in ‘java.util’ package and is a cursor that can be used to step through the collection of objects. Also it requires the raw input is List<Interval> or List<int[]> instead of int[][]**

**Refer to**

<https://leetcode.com/problems/merge-intervals/solutions/21222/a-simple-java-solution/comments/21220>

Mine is similar, but one difference is I use the iterator to iterate through the original list and then directly modify it, So the final results are already in the "intervals" list.

public List<Interval> merge(List<Interval> intervals) {

if (intervals == null || intervals.isEmpty())

return intervals;

Collections.sort(intervals, new Comparator<Interval>() {

public int compare(Interval i1, Interval i2) {

if (i1.start != i2.start) {

return i1.start - i2.start;

}

return i1.end - i2.end;

}

});

ListIterator<Interval> it = intervals.listIterator();

Interval cur = it.next();

while (it.hasNext()) {

Interval next = it.next();

if (cur.end < next.start) {

cur = next;

continue;

} else {

cur.end = Math.max(cur.end, next.end);

it.remove();

}

}

return intervals;

}

<https://leetcode.com/problems/merge-intervals/solutions/21222/a-simple-java-solution/comments/21194>

Was asked to solve this question by FB without using the new operator (ie creating new objects). Below is my solution in O(1) space.

public class Solution {

public List<Interval> merge(List<Interval> itv) {

if (itv == null) throw new IllegalArgumentException();

Collections.sort(itv, new Comparator<Interval>(){

@Override

public int compare(Interval a, Interval b) {

if (a.start == b.start) return b.end - a.end;

return a.start - b.start;

}

});

int i = 0;

while (i < itv.size() - 1) {

Interval a = itv.get(i), b = itv.get(i + 1);

if (a.end >= b.start) {

a.end = Math.max(a.end, b.end);

itv.remove(i + 1);

}

else i++;

}

return itv;

}

}