**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

/\*\*

\* @param {number[][]} intervals

\* @return {number[][]}

\*/

var merge = function(intervals) {

if (!intervals || !intervals.length) {

return [];

}

intervals.sort((a, b) => {

return a[0] - b[0]

});

let res = [intervals[0]];

console.log('test',res.length)

console.log(intervals);

for (let i = 1; i < intervals.length; i++) {

let start1 = res[res.length - 1][0];

let end1 = res[res.length - 1][1];

let start2 = intervals[i][0];

let end2 = intervals[i][1];

if (end1 >= start2) {

res[res.length - 1] = [start1, Math.max(end1, end2)];

} else {

res.push(intervals[i]);

}

}

return res;

};

Approach 1: Connected Components

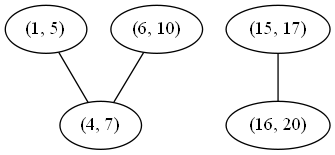
**Intuition**

If we draw a graph (with intervals as nodes) that contains undirected edges between all pairs of intervals that overlap, then all intervals in each *connected component* of the graph can be merged into a single interval.

**Algorithm**

With the above intuition in mind, we can represent the graph as an adjacency list, inserting directed edges in both directions to simulate undirected edges. Then, to determine which connected component each node is it, we perform graph traversals from arbitrary unvisited nodes until all nodes have been visited. To do this efficiently, we store visited nodes in a Set, allowing for constant time containment checks and insertion. Finally, we consider each connected component, merging all of its intervals by constructing a new Interval with start equal to the minimum start among them and end equal to the maximum end.

This algorithm is correct simply because it is basically the brute force solution. We compare every interval to every other interval, so we know exactly which intervals overlap. The reason for the connected component search is that two intervals may not directly overlap, but might overlap indirectly via a third interval. See the example below to see this more clearly.



Although (1, 5) and (6, 10) do not directly overlap, either would overlap with the other if first merged with (4, 7). There are two connected components, so if we merge their nodes, we expect to get the following two merged intervals:

(1, 10), (15, 20)

class Solution {

private Map<int[], List<int[]>> graph;

private Map<Integer, List<int[]>> nodesInComp;

private Set<int[]> visited;

// return whether two intervals overlap (inclusive)

private boolean overlap(int[] a, int[] b) {

return a[0] <= b[1] && b[0] <= a[1];

}

// build a graph where an undirected edge between intervals u and v exists

// iff u and v overlap.

private void buildGraph(int[][] intervals) {

graph = new HashMap<>();

for (int[] interval : intervals) {

graph.put(interval, new LinkedList<>());

}

for (int[] interval1 : intervals) {

for (int[] interval2 : intervals) {

if (overlap(interval1, interval2)) {

graph.get(interval1).add(interval2);

graph.get(interval2).add(interval1);

}

}

}

}

// merges all of the nodes in this connected component into one interval.

private int[] mergeNodes(List<int[]> nodes) {

int minStart = nodes.get(0)[0];

for (int[] node : nodes) {

minStart = Math.min(minStart, node[0]);

}

int maxEnd = nodes.get(0)[1];

for (int[] node : nodes) {

maxEnd = Math.max(maxEnd, node[1]);

}

return new int[] {minStart, maxEnd};

}

// use depth-first search to mark all nodes in the same connected component

// with the same integer.

private void markComponentDFS(int[] start, int compNumber) {

Stack<int[]> stack = new Stack<>();

stack.add(start);

while (!stack.isEmpty()) {

int[] 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 (int[] child : graph.get(node)) {

stack.add(child);

}

}

}

}

// gets the connected components of the interval overlap graph.

private void buildComponents(int[][] intervals) {

nodesInComp = new HashMap<>();

visited = new HashSet<>();

int compNumber = 0;

for (int[] interval : intervals) {

if (!visited.contains(interval)) {

markComponentDFS(interval, compNumber);

compNumber++;

}

}

}

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

buildGraph(intervals);

buildComponents(intervals);

// for each component, merge all intervals into one interval.

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

for (int comp = 0; comp < nodesInComp.size(); comp++) {

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

}

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

}

}

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2)

Building the graph costs O(V + E) = O(V) + O(E) = O(n) + O(n^2) = O(n^2)*O*(*V*+*E*)=*O*(*V*)+*O*(*E*)=*O*(*n*)+*O*(*n*2)=*O*(*n*2) time, as in the worst case all intervals are mutually overlapping. Traversing the graph has the same cost (although it might appear higher at first) because our visited set guarantees that each node will be visited exactly once. Finally, because each node is part of exactly one component, the merge step costs O(V) = O(n)*O*(*V*)=*O*(*n*) time. This all adds up as follows:

O(n^2) + O(n^2) + O(n) = O(n^2)*O*(*n*2)+*O*(*n*2)+*O*(*n*)=*O*(*n*2)

* Space complexity : O(n^2)*O*(*n*2)

As previously mentioned, in the worst case, all intervals are mutually overlapping, so there will be an edge for every pair of intervals. Therefore, the memory footprint is quadratic in the input size.

#### Approach 2: Sorting

**Intuition**

If we sort the intervals by their start value, then each set of intervals that can be merged will appear as a contiguous "run" in the sorted list.

**Algorithm**

First, we sort the list as described. Then, we insert the first interval into our merged list and continue considering each interval in turn as follows: If the current interval begins after the previous interval ends, then they do not overlap and we can append the current interval to merged. Otherwise, they do overlap, and we merge them by updating the end of the previous interval if it is less than the end of the current interval.

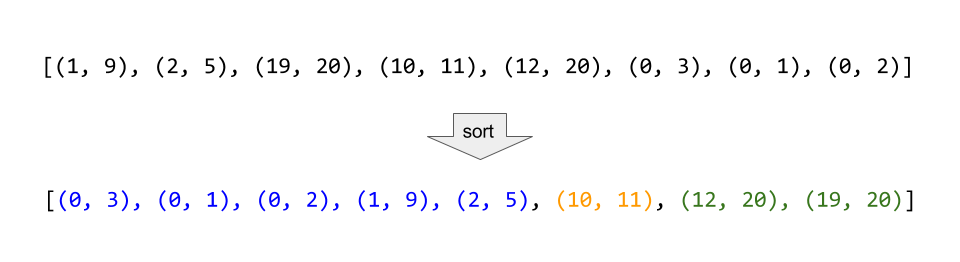
A simple proof by contradiction shows that this algorithm always produces the correct answer. First, suppose that the algorithm at some point fails to merge two intervals that should be merged. This would imply that there exists some triple of indices i*i*, j*j*, and k*k* in a list of intervals \text{ints}ints such that i < j < k*i*<*j*<*k* and (\text{ints[i]}ints[i], \text{ints[k]}ints[k]) can be merged, but neither (\text{ints[i]}ints[i], \text{ints[j]}ints[j]) nor (\text{ints[j]}ints[j], \text{ints[k]}ints[k]) can be merged. From this scenario follow several inequalities:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \\ \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \\ \end{aligned}ints[i].end<ints[j].startints[j].end<ints[k].startints[i].end≥ints[k].start​

We can chain these inequalities (along with the following inequality, implied by the well-formedness of the intervals: \text{ints[j].start} \leq \text{ints[j].end}ints[j].start≤ints[j].end) to demonstrate a contradiction:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \leq \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \end{aligned}ints[i].end<ints[j].start≤ints[j].end<ints[k].startints[i].end≥ints[k].start​

Therefore, all mergeable intervals must occur in a contiguous run of the sorted list.



Consider the example above, where the intervals are sorted, and then all mergeable intervals form contiguous blocks.

class Solution {

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

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

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()][]);

}

}

**Complexity Analysis**

* Time complexity : O(n\log{}n)*O*(*n*log*n*)

Other than the sort invocation, we do a simple linear scan of the list, so the runtime is dominated by the O(n\log{}n)*O*(*n*log*n*) complexity of sorting.

* Space complexity : O(\log N)*O*(log*N*) (or O(n)*O*(*n*))

If we can sort intervals in place, we do not need more than constant additional space, although the sorting itself takes O(\log n)*O*(log*n*) space. Otherwise, we must allocate linear space to store a copy of intervals and sort that.