

Solution: Merge Intervals

Let's solve the Merge Intervals problem using the Merge Intervals pattern.

We'll cover the following

- Statement
- Solution
 - Naive approach
 - Optimized approach using merge intervals
 - Step-by-step solution construction
 - Just the code
 - Solution summary
 - Time complexity
 - Space complexity

Statement

We are given an array of **closed intervals** `intervals`, where each interval has a start time and an end time. The input array is sorted with respect to the start times of each interval. For example, `intervals = [[1, 4], [3, 6], [7, 9]]` is sorted in terms of start times 1, 3, and 7.

Your task is to merge the overlapping intervals and return a new output array consisting of only the non-overlapping intervals.

Constraints:

- $1 \leq \text{intervals.length} \leq 10^4$
- `intervals[i].length = 2`
- $0 \leq \text{start time} \leq \text{end time} \leq 10^4$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow, based on considerations such as time complexity and any implementation constraints.

Naive approach

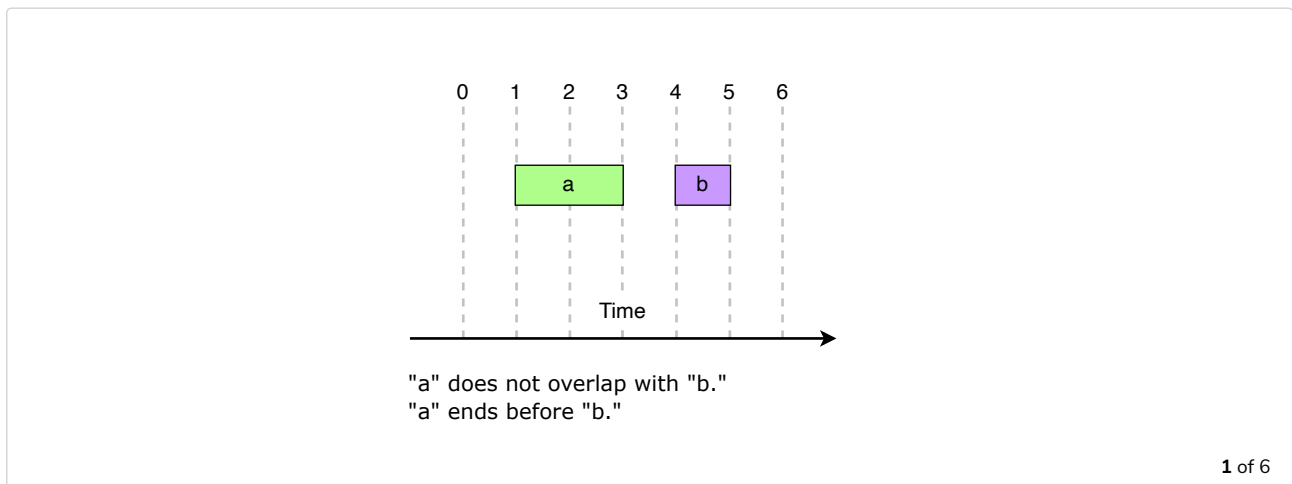
The naive approach is to start from the first interval in the input list and check for any other interval in the list that overlaps it. If there is an overlap, merge the other interval into the first interval and then remove the other interval from the list. Repeat this for all the remaining intervals in the list.

In the solution above, we traversed the given input array twice, at most, so the time complexity is $O(n^2)$, which is not optimal. However, the space complexity for this solution is $O(1)$, since we are not using any extra processing space.

Optimized approach using merge intervals



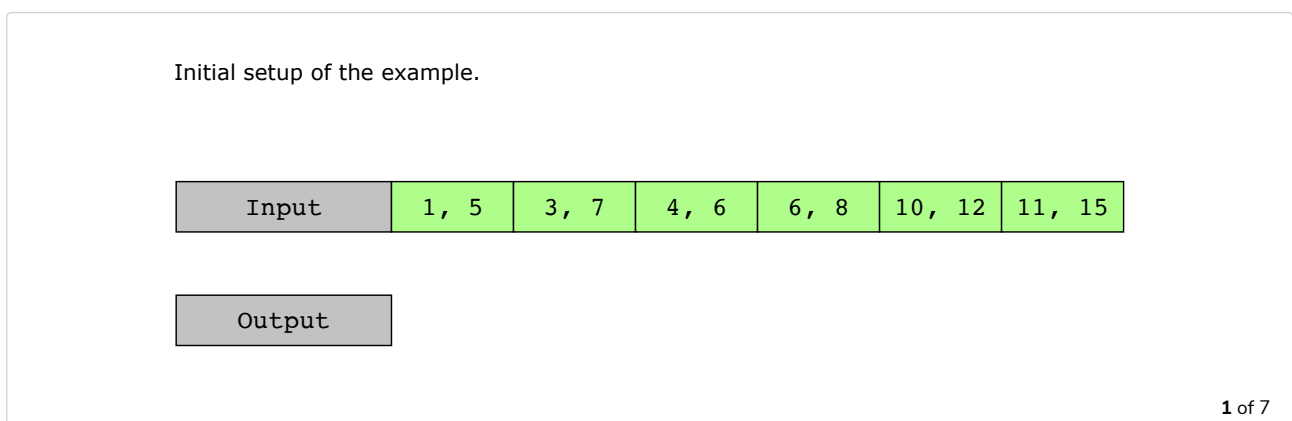
This problem illustrates the characteristic features of the merge intervals pattern. Therefore, to solve this problem, we need to handle all the six ways in which any two given intervals relate to each other. Before jumping onto the solution, let's review all these ways using the illustration from the introduction to this pattern:



Note: In the slides above, if we notice, there is something common among all the scenarios of overlapping intervals. If we call 'a' the first interval and 'b' the second interval, then in all the overlapping scenarios, the start time of the second interval always occurs before the end time of the first interval, that is, $b's\ start\ time < a's\ end\ time$.

In this solution, we use the merge intervals pattern with a simple linear scan to merge the overlapping intervals. First, we create an output list and copy the first interval of the input list to it. Next, we traverse the remaining intervals of the input list and check whether any interval overlaps with the interval present in the output list. If they overlap, update the interval in the output list. Otherwise, add the current input interval to the output list. Repeat this for all the intervals in the input list. Please note that when we have more than one interval in the output list, we compare the input intervals with the last interval of the output list.

Now, let's look at the following illustration to get a better understanding of the solution:



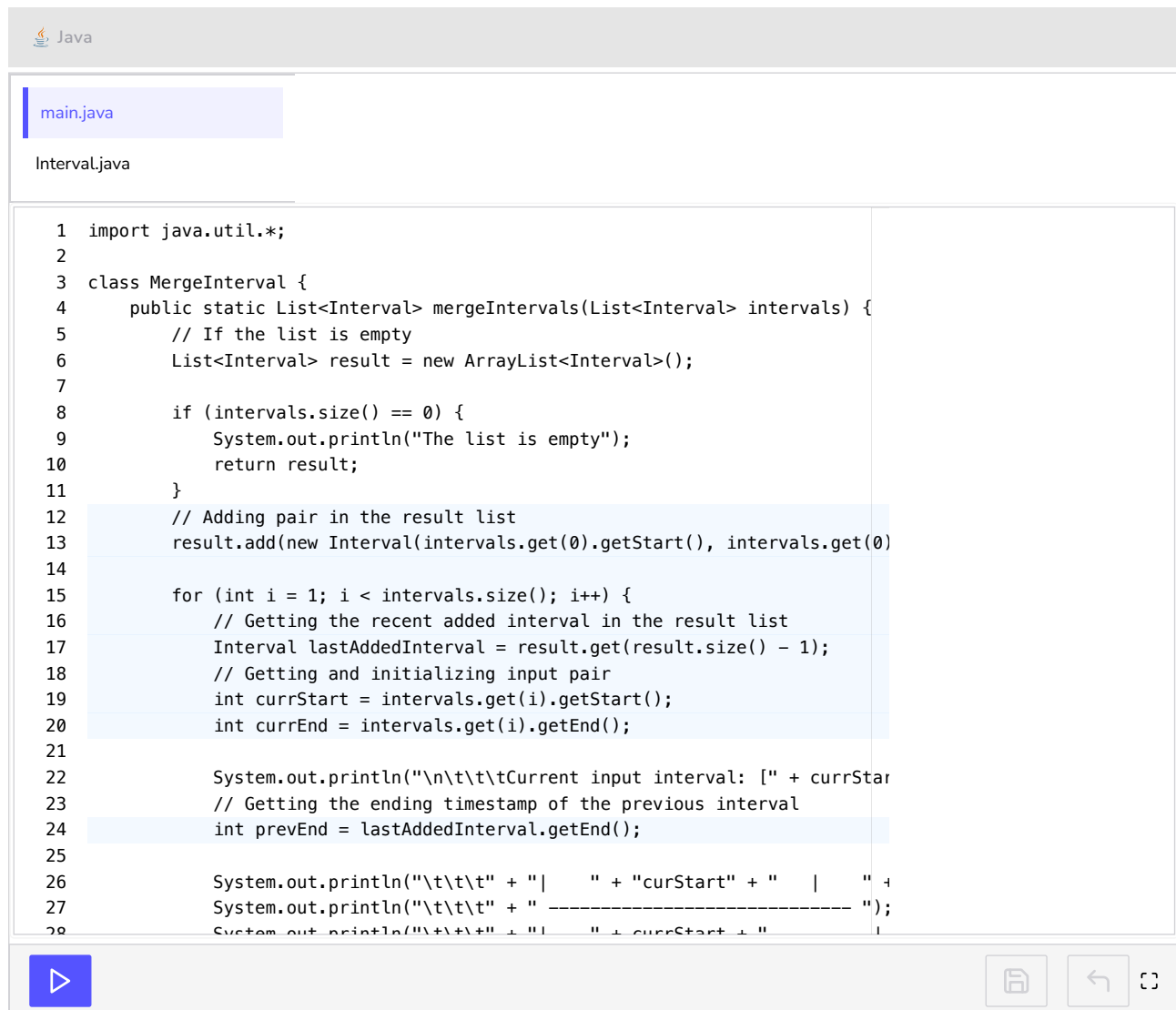
Note: In the following section, we will gradually build the solution. Alternatively, you can skip straight

to the “Just the code” section.

Step-by-step solution construction

The first step is to copy the first interval from the input list to the output list. Next, add a loop to traverse the remaining intervals in the input list. In each loop, we will use the variable `currStart` to store the start time and the variable `currEnd` to store the end time of the current input interval. Moreover, to keep track of the end time of the last interval in the output list, we create a variable, `prevEnd`. These variables will help us identify any overlapping intervals in the later steps.

The code below implements the steps above:



```
1 import java.util.*;
2
3 class MergeInterval {
4     public static List<Interval> mergeIntervals(List<Interval> intervals) {
5         // If the list is empty
6         List<Interval> result = new ArrayList<Interval>();
7
8         if (intervals.size() == 0) {
9             System.out.println("The list is empty");
10            return result;
11        }
12        // Adding pair in the result list
13        result.add(new Interval(intervals.get(0).getStart(), intervals.get(0).getEnd()));
14
15        for (int i = 1; i < intervals.size(); i++) {
16            // Getting the recent added interval in the result list
17            Interval lastAddedInterval = result.get(result.size() - 1);
18            // Getting and initializing input pair
19            int currStart = intervals.get(i).getStart();
20            int currEnd = intervals.get(i).getEnd();
21
22            System.out.println("\n\t\t\t\tCurrent input interval: [" + currStart + " " + currEnd + "]");
23            // Getting the ending timestamp of the previous interval
24            int prevEnd = lastAddedInterval.getEnd();
25
26            System.out.println("\t\t\t\t" + " | " + "curStart" + " | " + "curEnd" + " | " + "prevEnd" + " |");
27            System.out.println("\t\t\t\t" + "-----");
28            System.out.println("\t\t\t\t" + " | " + currStart + " | " + currEnd + " | " + prevEnd + " |");
```

Merge Intervals

Inside the loop, we check each interval of the input list against the last interval of the output list. For each interval in the input list, we do the following:

- If the current input interval is overlapping with the last interval in the output list, we merge these two intervals and replace the last interval of the output list with the newly merged interval.
- Otherwise, we add the input interval to the output list.

To check if the current input interval and the last interval in the output list overlap, we'll check the start time of the current interval and the end time of the last interval in the output list. If the start time of the current interval is less than the end time of the last interval in the output list, that is, `currStart <= prevEnd`, the two

intervals overlap. Otherwise, they don't overlap. Since the intervals are sorted in terms of their start times, we won't encounter cases where the current interval's start and end times are less than the start time of the last interval in the output list.

For example, if we have an input interval list $[[1, 5], [3, 7]]$, we will first add the interval $[1, 5]$ to the output list. Then, we will check $[3, 7]$ from the input list against the last interval in the output list, $[1, 5]$, to check if they overlap or not. The start time, 3, of the current interval is less than the end time, 5, of the last interval in the output list, so the two intervals overlap. Since these two intervals overlap, we will merge them and update the last interval of our output list to $[1, 7]$.





The lines highlighted below implement this logic:

Java

main.java

Interval.java

```
1 import java.util.*;
2
3 class MergeInterval {
4
5     public static List<Interval> mergeIntervals(List<Interval> intervals) {
6         // If the list is empty
7         List<Interval> result = new ArrayList<Interval>();
8         if (intervals.size() == 0) {
9             System.out.println("The list is empty");
10            return result;
11        }
12        // Adding pair in the result list
13        result.add(new Interval(intervals.get(0).getStart(), intervals.get(0).getEnd()));
14        for (int i = 1; i < intervals.size(); i++) {
15            // Getting the recently added interval in the result list
16            Interval lastAddedInterval = result.get(result.size() - 1);
17            // Getting and initializing input pair
18            int currStart = intervals.get(i).getStart();
19            int currEnd = intervals.get(i).getEnd();
20            System.out.println("\n\t\t\t\tCurrent input interval: [" + currStart + " " + currEnd + "]");
21            // Getting the ending timestamp of the previous interval
22            int prevEnd = lastAddedInterval.getEnd();
23            System.out.println("\t\t\t\tLast output interval: [" + lastAddedInterval.getStart() + " " + lastAddedInterval.getEnd() + "]");
24            // Overlapping condition
25            if (currStart <= prevEnd) {
26                lastAddedInterval.setEnd(Math.max(currEnd, prevEnd));
27                System.out.println("\t\t\t\tOverlapping condition true. Updating last output interval to [" + lastAddedInterval.getStart() + " " + lastAddedInterval.getEnd() + "]");
28            } else {
29                result.add(new Interval(currStart, currEnd));
30            }
31        }
32        return result;
33    }
34}
```



Merge Intervals

Just the code


Here's the complete solution to this problem:

Java

main.java

Interval.java

```
1 import java.util.*;
2
3 class MergeInterval {
```



```

3 class MergeInterval {
4
5     public static List<Interval> mergeIntervals(List<Interval> intervals) {
6         List<Interval> result = new ArrayList<Interval>();
7         if (intervals.size() == 0) {
8             return result;
9         }
10        result.add(new Interval(intervals.get(0).getStart(), intervals.get(0).getEnd()));
11        for (int i = 1; i < intervals.size(); i++) {
12            Interval lastAddedInterval = result.get(result.size() - 1);

```

```

16            if (currStart <= prevEnd) {
17                lastAddedInterval.setEnd(Math.max(currEnd, prevEnd));
18            } else {
19                result.add(new Interval(currStart, currEnd));
20            }
21        }
22        return result;
23    }
24
25    public static String intervalListToStr(List<Interval> l1) {
26        String resultStr = "[";
27        for (int i = 0; i < l1.size() - 1; i++) {
28            resultStr += "[" + l1.get(i).getStart() + " " + l1.get(i).getEnd()

```



Merge Intervals

Solution summary

To recap, the solution to this problem can be divided into the following two parts:

1. Insert the first interval from the input list into the output list.
2. Traverse the input list of intervals. For each interval in the input list, we do the following:
 - If the input interval is overlapping with the last interval in the output list, merge these two intervals and replace the last interval of the output list with this merged interval.
 - Otherwise, add the input interval to the output list.

Time complexity

The time complexity of this solution is $O(n)$, where n is the number of intervals in the input list.

Space complexity

The space complexity of this solution is $O(1)$, since we only use constant space other than the input and output data structures.

← Back

Merge Intervals

Next →

Insert Interval

☒ Mark as Completed



