Solution: Insert Interval

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

We'll cover the following

Statement

Given a sorted list of **nonoverlapping** intervals and a new interval, your task is to insert the new interval into the correct position while ensuring that the resulting list of intervals remains sorted and nonoverlapping. Each interval is a pair of nonnegative numbers, the first being the start time and the second being the end time of the interval.

Constraints:

- $0 \leq$ `existing_intervals.length` $\leq 10^4$
- `existing_intervals[i].length`, `new_interval.length` == 2
- $0 \leq$ *start time*, *end time* $\leq 10^4$
- The first number should always be less than the second number in each interval.
- The list of intervals is sorted in ascending order based on the first element in every interval.

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.

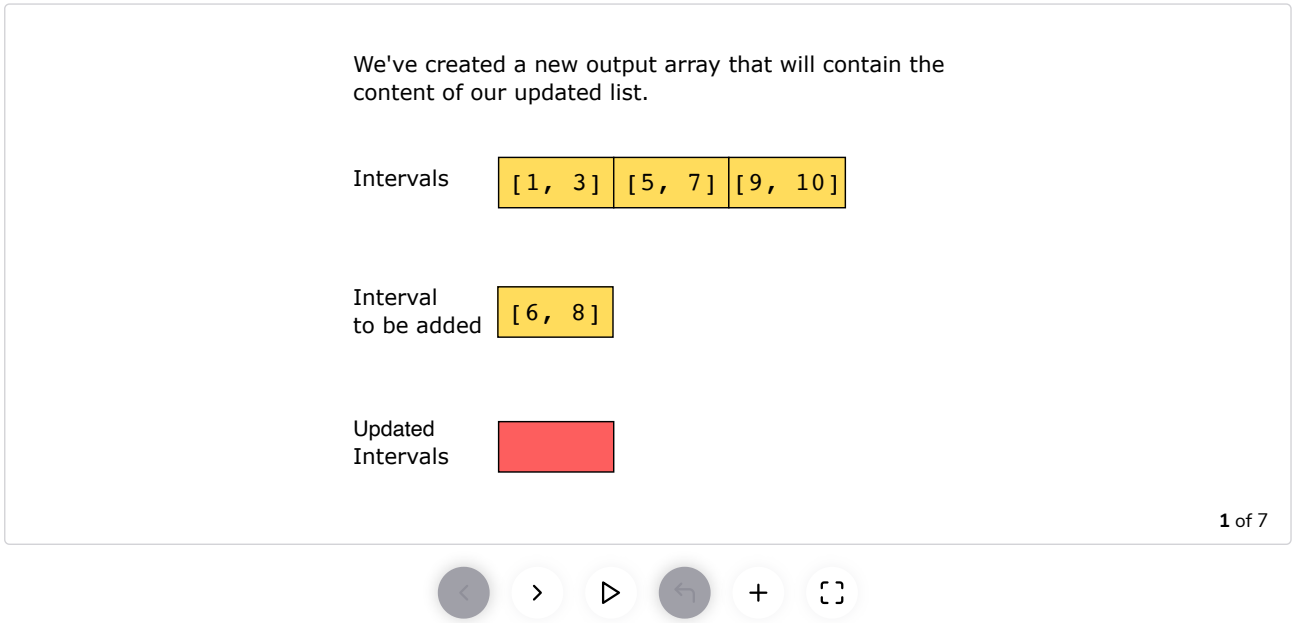
Naive approach

The naive approach to this problem is to iterate through the existing intervals. For each interval, we check if the current interval overlaps the existing interval. If an overlap is detected, the intervals are merged by adjusting the start and end times. Additionally, a flag is set to TRUE, indicating that the new interval is added to the existing intervals. After iterating through all existing intervals, if the flag remains FALSE, the new interval does not overlap with any existing interval. In such a case, the new interval is appended to the end of the output list. Finally, the output list is sorted based on the start times of the intervals to ensure that the new interval is placed in the correct order.

The time complexity is $O(n \log(n))$.

Optimized approach using the merge intervals pattern

The problem requires us to do two things. First, we need to insert a new interval into a sequence of nonoverlapping intervals. Second, the result should also be a list of nonoverlapping intervals. This requires merging the new interval with any overlapping interval(s) in the input. Therefore, the merge interval pattern applies.



> **Note:** In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

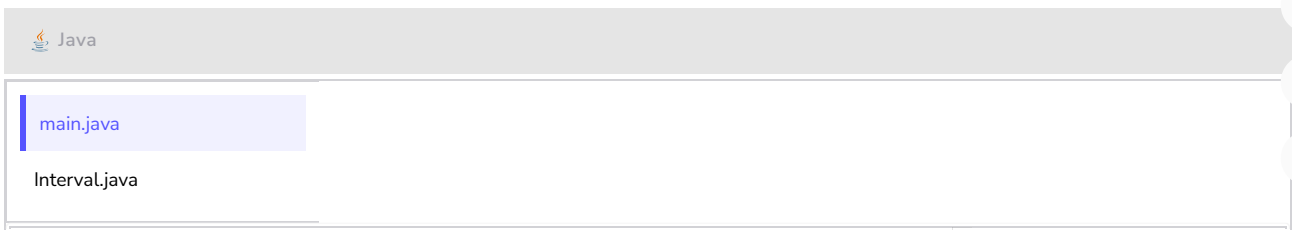
Step-by-step solution construction

First, we process the intervals that start before the new interval and then insert the new interval itself. Additionally, we need to merge the new interval with the existing intervals in the input.

Here's how we'll implement this feature:

- Since the intervals are sorted, we'll create an empty list, `output`, and start adding the intervals that start before the new interval.
- Next, we'll add the new interval to the `output` list and merge it with the previously added interval if there is an overlap.
- Finally, we'll add the remaining intervals to the `output` list, merging them with the previously added intervals when they overlap.

We'll look at these steps and how they work individually to have a more comprehensive understanding of our solution. The first step is to see the new interval to be added and append all the intervals that start before it to a new list which we will use as our output later.



```java
import java.util.*;

class InsertInterval {
    public static void insertInterval(List<Interval> existingIntervals, Inter
        // Read the starting and ending time of the new interval, into separa
        int newStart = newInterval.getStart();
        int newEnd = newInterval.getEnd();
        System.out.println("The new interval starts at " + newStart + " and e

        // Initialize variables to help in iterating over the existing interv
        int i = 0;
        int n = existingIntervals.size();
        System.out.println("There are " + n + " intervals present in the list

        // Initialize an empty list to store the output
        List<Interval> output = new ArrayList<>();

        // Append all intervals that start before the new interval to the out
        System.out.println("Let's start adding these intervals into our outpu
        System.out.println(new String(new char[100]).replace('\0', '-'));
        while (i < n && existingIntervals.get(i).getStart() < newStart) {
            output.add(existingIntervals.get(i));
            System.out.println("We can add " + (i + 1) + " intervals in our r
            System.out.println(display(output));
            i += 1;
            System.out.println(new String(new char[100]).replace('\0', '-'));
        }
        System.out.println();
```

Insert Interval

After appending the intervals that start before the new interval to the output list, we check if any intervals in the output list overlap with the new interval. If an overlap is found, we merge the intervals by updating the end time of the last interval in the output list to the maximum of its current end time and the end time of the new interval.

In this step, we handle the merging of intervals if there is an overlap with the new interval.

Java

main.java

Interval.java

```java
import java.util.*;

class InsertInterval {
    public static void insertInterval(List<Interval> existingIntervals, Inter
        // Read the starting and ending time of the new interval into separat
        int newStart = newInterval.getStart();
        int newEnd = newInterval.getEnd();
        System.out.println("The new interval starts at " + newStart + " and e

        // Initialize variables to help in iterating over the existing interv
        int i = 0;
        int n = existingIntervals.size();
        System.out.println("There are " + n + " intervals present in the list

        // Initialize an empty list to store the output
        List<Interval> output = new ArrayList<Interval>();

        // Append all intervals that start before the new interval to the out
        System.out.println("Let's start adding these intervals into our outpu
        System.out.println(new String(new char[100]).replace('\0', '-'));

        while (i < n && existingIntervals.get(i).getStart() < newStart) {
```

```
23              output.add(existingIntervals.get(i));
24              i += 1;
25          }
26
27          System.out.println("The current output list of intervals without any
28
```

Insert Interval

In this step, we handle the remaining intervals in the existing interval list after adding the new interval. We iterate through the remaining intervals and merge them with the intervals in the output list if there is an overlap. By comparing start and end times, we determine if an interval should be appended or merged. This process ensures that all overlapping intervals are correctly merged into the output list.

Finally, we obtain the final output list representing the merged intervals from the existing and new intervals.

Java

main.java

Interval.java

```
1  import java.util.*;
2
3  class InsertInterval {
4      public static List<Interval> insertInterval(List<Interval> existingInterv
5          // Read the starting and ending time of the new interval into separat
6          int newStart = newInterval.getStart();
7          int newEnd = newInterval.getEnd();
8          System.out.println("The new interval starts at " + newStart + " and e
9
10         // Initialize variables to help in iterating over the existing interv
11         int i = 0;
12         int n = existingIntervals.size();
13         System.out.println("There are " + n + " intervals present in the list
14
15         // Initialize an empty list to store the output
16         List<Interval> output = new ArrayList<Interval>();
17
18         // Append all intervals that start before the new interval to the out
19         while (i < n && existingIntervals.get(i).getStart() < newStart) {
20             output.add(existingIntervals.get(i));
21             i += 1;
22         }
23
24         System.out.println("The output before we add the new interval is: " +
25
26         // If the new interval starts after the end of the last interval appe
27         // just append the new interval to the output list.
28         if (output.size() == 0 || output.get(output.size() - 1).getEnd() < ne
```

Insert Interval

Just the code

Here's the complete solution to this problem:

Java

main.java

Interval.java

```java
import java.util.*;

class InsertInterval {
    public static List<Interval> insertInterval(List<Interval> existingInterv
        int newStart = newInterval.getStart();
        int newEnd = newInterval.getEnd();
        int i = 0;
        int n = existingIntervals.size();
        List<Interval> output = new ArrayList<Interval>();
        while (i < n && existingIntervals.get(i).getStart() < newStart) {
        }
        if (output.size() == 0 || output.get(output.size() - 1).getEnd() < ne
            output.add(newInterval);
        } else {
            output.get(output.size() - 1).setEnd(Math.max(output.get(output.s
        }
        while (i < n) {
            Interval ei = existingIntervals.get(i);
            int start = ei.getStart();
            int end = ei.getEnd();
            if (output.get(output.size() - 1).getEnd() < start) {
                output.add(ei);
            } else {
                output.get(output.size() - 1).setEnd(Math.max(output.get(outp
            }
            i += 1;
```

Insert Interval

Solution summary

- Append all intervals occurring before the new interval to the `output` list until we find an interval that starts after the starting point of the new interval.
- If there is an overlap between the last interval in the `output` list and the new interval, merge them by updating the end value of the last interval. Otherwise, append the new interval to the `output` list.
- Continue iterating through the remaining intervals and merge the overlapping intervals with the last interval in the `output` list.
- Return the final `output` list containing the merged intervals.

Time complexity

The time complexity is $O(n)$, where n is the number of intervals in the input list. This is because we iterate through the list once, checking and merging intervals as necessary.

Space complexity

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