# Solution: Interval List Intersections

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

We'll cover the following



## Statement

For two lists of closed intervals given as input, `intervalLista` and `intervalListb`, where each interval has its own start and end time, write a function that returns the intersection of the two interval lists.

For example, the intersection of $[3, 8]$ and $[5, 10]$ is $[5, 8]$.

**Constraints**

- $0 \leq$ `intervalLista.length`, `intervalListb.length` $\leq 1000$
- $0 \leq$ `start[i]` $<$ `end[i]` $\leq 10^9$, where $i$ is used to indicate `intervalLista`
- `end[i]` $<$ `start[i + 1]`
- $0 \leq$ `start[j]` $<$ `end[j]` $\leq 10^9$, where $j$ is used to indicate `intervalListb`
- `end[j]` $<$ `start[j + 1]`

## 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 for this problem is to use a nested loop for finding intersecting intervals.

- The outer loop will iterate for every interval in `intervalLista` and the inner loop will search for any intersecting interval in the `intervalListb`.
- If such an interval exists, we add it to the intersections list.

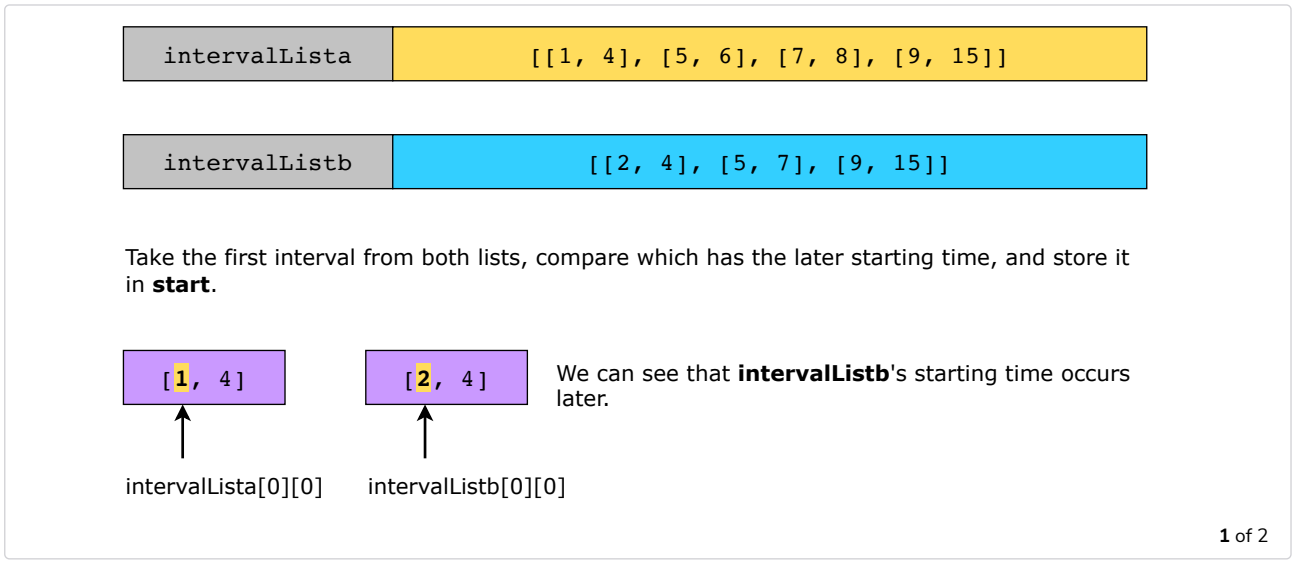
The time complexity for this naive approach will be $O(n^2)$ since we are using nested loops.

### Optimized approach using merge intervals

This problem shares two features with the merge intervals pattern: the lists of intervals are sorted and the result requires comparing intervals to check overlap. Taking advantage of the sorted array of the lists, we can safely compare pairs of intervals (one from List A and one from List B), knowing that after every comparison, we need only move forward in the lists, without having to re-check either list from the start.

The algorithm to solve this problem is as follows:

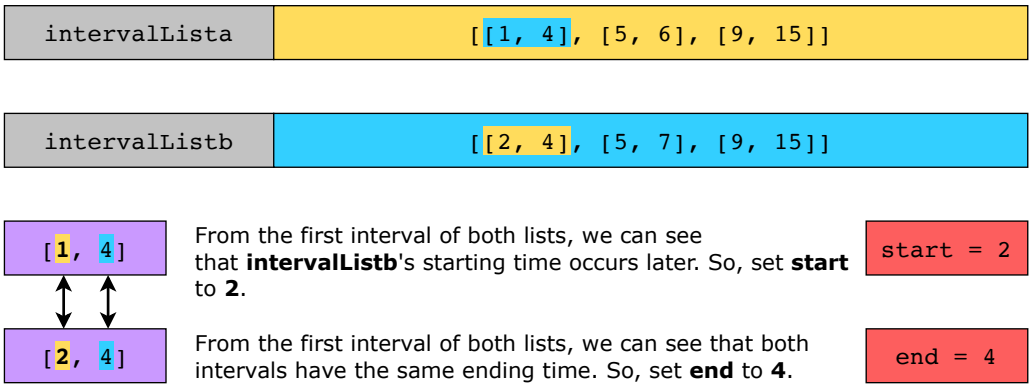
- We'll use two indices, `i` and `j`, to iterate through the intervals in both lists, that is, `intervalLista` and `intervalListb` respectively.
- To check whether there's any intersecting point among the given intervals:
  - Take the starting times of the first pair of interval from both lists and check which occurs later, storing it in a variable, say `start`.
  - Also compare the ending times of the same pair of intervals from both lists and store the minimum end time in another variable, say `end`.




- Next, we will check if `intervalLista[i]` and `intervalListb[j]` overlap by comparing the `start` and `end` times.
  - If the times overlap, then the intersecting time interval will be added to the resultant list, that is, `intersections`.
  - After the comparison, we need to move forward in one of the two input lists. The decision is taken based on which of the two intervals being compared *ends* earlier. If the interval that ends first is in `intervalLista`, we move forward in that list, else, we move forward in `intervalListb`.

The slide deck below illustrates the key steps of the solution.

Take intervals from both lists and compare their starting and ending times to find any intersection. Suppose, that the starting time of **intervalLista** arrives earlier or is equal to **intervalListb**'s ending time. Then, an intersection is found.






Let's look at the code for this solution below:

Java

main.java

interval.java

```
1  import java.util.*;
2
3  class Intersection {
4
5      public static String display(List<Interval> l1) {
6          String resultStr = "[";
7          for (int i = 0; i < l1.size() - 1; i++) {
8              resultStr += "[" + l1.get(i).getStart() + ", " + l1.get(i).getEnd
9          }
10         resultStr += "[" + l1.get(l1.size() - 1).getStart() + ", " + l1.get(l
11         resultStr += "]";
12         return resultStr;
13     }
14
15     public static List<Interval> intervalsIntersection(List<Interval> interva
16         List<Interval> intersections = new ArrayList<>(); // to store all int
17         // index "i" to iterate over the length of list a and index "j"
18         // to iterate over the length of list b
19         int i = 0, j = 0;
20         // while loop will break whenever either of the lists ends
21         while (i < intervalLista.size() && j < intervalListb.size()) {
22             // Let's check if intervalLista[i] intervalListb[j]
23             // 1. start - the potential startpoint of the intersection
24             // 2. end - the potential endpoint of the intersection
25             int start = Math.max(intervalLista.get(i).getStart(), intervalLis
26             int end = Math.min(intervalLista.get(i).getEnd(), intervalListb.g
27             if (start <= end) // if this is an actual intersection
28                 intersections.add(new Interval(start, end)); // add it to ths
```

### Solution summary

Let's briefly discuss the approach that we have used to solve the above mentioned problem:

- Compare the starting and ending times of a given interval from A and B.
- If the start time of the current interval in A is less than or equal to the end time of the current interval in B, or vice versa, we have found an intersection. Add it to a resultant list.
- Move forward in the list whose current interval ends earlier and repeat comparison and moving forward steps to find all intersecting intervals.
- Return the resultant list of intersecting intervals.

### Time complexity

The time complexity is $O(n + m)$, where $n$ and $m$ are the number of meetings in `intervalLista` and `intervalListb`, respectively.

### Space complexity

The space complexity is $O(1)$ as only a fixed amount of memory is consumed by a few temporary variables for computations performed by the algorithm.