educative

Conflicting Appointments (medium)

We'll cover the following

Problem Statement

Given an array of intervals representing 'N' appointments, find out if a person can **attend all the appointments**.

Example 1:

```
Appointments: [[1,4], [2,5], [7,9]]
Output: false
Explanation: Since [1,4] and [2,5] overlap, a person cannot attend both of these appointments.
```

Example 2:

```
Appointments: [[6,7], [2,4], [8,12]]
Output: true
Explanation: None of the appointments overlap, therefore a person can attend all of them.
```

Example 3:

```
Appointments: [[4,5], [2,3], [3,6]]
Output: false
Explanation: Since [4,5] and [3,6] overlap, a person cannot attend both of these appointments.
```

Try it yourself

Try solving this question here:

Java Python3 JS C++

```java
import java.util.*;

class Interval {
  int start;
  int end;

  public Interval(int start, int end) {
    this.start = start;
    this.end = end;
  }
};

class ConflictingAppointments {

  public static boolean canAttendAllAppointments(Interval[] intervals) {
    // TODO: Write your code here
    return true;
  }

  public static void main(String[] args) {
    Interval[] intervals = { new Interval(1, 4), new Interval(2, 5), new Interval(7, 9) };
    boolean result = ConflictingAppointments.canAttendAllAppointments(intervals);
    System.out.println("Can attend all appointments: " + result);

    Interval[] intervals1 = { new Interval(6, 7), new Interval(2, 4), new Interval(8, 12) };
    result = ConflictingAppointments.canAttendAllAppointments(intervals1);
    System.out.println("Can attend all appointments: " + result);

```

Run Save Reset

Solution

The problem follows the Merge Intervals pattern. We can sort all the intervals by start time and then check if any two intervals overlap. A person will not be able to attend all appointments if any two appointments overlap.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class Interval {
  int start;
  int end;

  public Interval(int start, int end) {
    this.start = start;
    this.end = end;
  }
};

class ConflictingAppointments {

  public static boolean canAttendAllAppointments(Interval[] intervals) {
    // sort the intervals by start time
    Arrays.sort(intervals, (a, b) -> Integer.compare(a.start, b.start));

    // find any overlapping appointment
    for (int i = 1; i < intervals.length; i++) {
      if (intervals[i].start < intervals[i - 1].end) {
        // please note the comparison above, it is "<" and not "<="
        // while merging we needed "<=" comparison, as we will be merging the two
        // intervals having condition "intervals[i].start == intervals[i - 1].end" but
        // such intervals don't represent conflicting appointments as one starts right
        // after the other
        return false;
      }
```

Run | Save | Reset

Time complexity

The time complexity of the above algorithm is $O(N * logN)$, where 'N' is the total number of appointments. Though we are iterating the intervals only once, our algorithm will take $O(N * logN)$ since we need to sort them in the beginning.

Space complexity

The space complexity of the above algorithm will be $O(N)$, which we need for sorting. For Java, `Arrays.sort()` uses Timsort, which needs $O(N)$ space.

Similar Problems

Problem 1: Given a list of appointments, find all the conflicting appointments.

Example:

```
Appointments: [[4,5], [2,3], [3,6], [5,7], [7,8]]
Output:
[4,5] and [3,6] conflict.
[3,6] and [5,7] conflict.
```