Interview Cake

Your company built an in-house calendar tool called HiCal. You want to add a feature to see the times in a day when *everyone* is available.

To do this, you'll need to know when *any* team is having a meeting. In HiCal, a meeting is stored as an object of a `Meeting` class with integer variables `startTime` and `endTime`. These integers represent the number of 30-minute blocks past 9:00am.

Java

```
public class Meeting {

    private int startTime;
    private int endTime;

    public Meeting(int startTime, int endTime) {
        // number of 30 min blocks past 9:00 am
        this.startTime = startTime;
        this.endTime   = endTime;
    }

    public int getStartTime() {
        return startTime;
    }

    public void setStartTime(int startTime) {
        this.startTime = startTime;
    }

    public int getEndTime() {
        return endTime;
    }

    public void setEndTime(int endTime) {
        this.endTime = endTime;
    }
}
```

For example:

Java

```
new Meeting(2, 3);  // meeting from 10:00 – 10:30 am
new Meeting(6, 9);  // meeting from 12:00 – 1:30 pm
```

Write a method `mergeRanges()` that takes a list of meeting time ranges and returns a list of condensed ranges.

For example, given:

```
[Meeting(0, 1), Meeting(3, 5), Meeting(4, 8), Meeting(10, 12), Meeting(9, 10)]
```
Java

your method would return:

```
[Meeting(0, 1), Meeting(3, 8), Meeting(9, 12)]
```
Java

Do not assume the meetings are in order. The meeting times are coming from multiple teams.

Write a solution that's efficient even when we can't put a nice upper bound on the numbers representing our time ranges. Here we've simplified our times down to the number of 30-minute slots past 9:00 am. But we want the method to work even for very large numbers, like Unix timestamps. In any case, the spirit of the challenge is to merge meetings where `startTime` and `endTime` don't have an upper bound.

Gotchas

Look at this case:

```
[Meeting(1, 2), Meeting(2, 3)]
```
Java

These meetings should probably be merged, although they don't exactly "overlap"—they just "touch." Does your method do this?

Look at this case:

```
[Meeting(1, 5), Meeting(2, 3)]
```
Java

Notice that although the second meeting starts later, it ends before the first meeting ends. Does your method correctly handle the case where a later meeting is "subsumed by" an earlier meeting?

Look at this case:

```
[Meeting(1, 10), Meeting(2, 6), Meeting(3, 5), Meeting(7, 9)]
```
Java

Here *all* of our meetings should be merged together into just `Meeting(1, 10)`. We need keep in mind that after we've merged the first two we're not done with the result—the result of that merge *may itself need to be merged into other meetings as well.*

Make sure that your method won't "leave out" the *last* meeting.

We can do this in $O(n \lg n)$ time.

Breakdown

What if we only had two ranges? Let's take:

```
[Meeting(1, 3), Meeting(2, 4)]
```
java

These meetings clearly overlap, so we should merge them to give:

```
[Meeting(1, 4)]
```
java

But how did we know that these meetings overlap?

We could tell the meetings overlapped because the *end time* of the first one was after the *start time* of the second one! But our ideas of "first" and "second" are important here—this only works after we ensure that we treat the meeting that *starts earlier* as the "first" one.

How would we formalize this as an algorithm? **Be sure to consider these edge cases:**

1. The end time of the first meeting and the start time of the second meeting are equal. For example: `[Meeting(1, 2), Meeting(2, 3)]`
2. The second meeting ends before the first meeting ends. For example: `[Meeting(1, 5), Meeting(2, 3)]`

Here's a formal algorithm:

1. We treat the meeting with earlier start time as "first," and the other as "second."
2. If the end time of the first meeting is *equal to or greater than* the start time of the second meeting, we merge the two meetings into one time range. The resulting time range's start time is the first meeting's start, and its end time is *the later of* the two meetings' end times.

3. Else, we leave them separate.

So, we could compare *every* meeting to *every other* meeting in this way, merging them or leaving them separate.

Comparing *all pairs* of meetings would take $O(n^2)$ time. We can do better!

If we're going to beat $O(n^2)$ time, maybe we're going to get $O(n)$ time? Is there a way to do this in one pass?

It'd be great if, for each meeting, we could just try to merge it with the *next* meeting. But that's definitely not sufficient, because the ordering of our meetings is random. There might be a non-next meeting that the current meeting could be merged with.

What if we sorted our list of meetings by start time?

Then any meetings that could be merged would always be adjacent!

So we could sort our meetings, then walk through the sorted list and see if each meeting can be merged with the one after it.

Sorting takes $O(n \lg n)$ time in the worst case. If we can then do the merging in one pass, that's another $O(n)$ time, for $O(n \lg n)$ overall. That's not as good as $O(n)$, but it's better than $O(n^2)$.

Solution

First, we sort our input list of meetings by start time so any meetings that might need to be merged are now next to each other.

Then we walk through our sorted meetings from left to right. At each step, either:

1. We *can* merge the current meeting with the previous one, so we do.
2. We *can't* merge the current meeting with the previous one, so we know the previous meeting can't be merged with any future meetings and we throw the current meeting into `mergedMeetings`.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public static List<Meeting> mergeRanges(List<Meeting> meetings) {

    // make a copy so we don't destroy the input
    List<Meeting> sortedMeetings = new ArrayList<>();
    for (Meeting meeting: meetings) {
        Meeting meetingCopy = new Meeting(meeting.getStartTime(), meeting.getEndTime());
        sortedMeetings.add(meetingCopy);
    }

    // sort by start time
    Collections.sort(sortedMeetings, new Comparator<Meeting>() {
        @Override
        public int compare(Meeting m1, Meeting m2) {
            return m1.getStartTime() - m2.getStartTime();
        }
    });

    // initialize mergedMeetings with the earliest meeting
    List<Meeting> mergedMeetings = new ArrayList<>();
    mergedMeetings.add(sortedMeetings.get(0));

    for (Meeting currentMeeting : sortedMeetings) {

        Meeting lastMergedMeeting = mergedMeetings.get(mergedMeetings.size() - 1);

        // if the current and last meetings overlap, use the latest end time
        if (currentMeeting.getStartTime() <= lastMergedMeeting.getEndTime()) {
            lastMergedMeeting.setEndTime(Math.max(lastMergedMeeting.getEndTime(), currentMeeting.get

        // add the current meeting since it doesn't overlap
        } else {
            mergedMeetings.add(currentMeeting);
        }
    }
```

```
        return mergedMeetings;
    }
```

Complexity

$O(n \lg n)$ time and $O(n)$ space.

Even though we only walk through our list of meetings once to merge them, we *sort* all the meetings first, giving us a runtime of $O(n \lg n)$. It's worth noting that *if* our input were sorted, we could skip the sort and do this in $O(n)$ time!

We create a new list of merged meeting times. In the worst case, none of the meetings overlap, giving us a list identical to the input list. Thus we have a worst-case space cost of $O(n)$.

Bonus

1. What if we *did* have an upper bound on the input values? Could we improve our runtime? Would it cost us memory?
2. Could we do this "in-place" on the input list and save some space? What are the pros and cons of doing this in-place?

What We Learned

This one arguably uses a greedy approach as well, except this time we had to *sort* the list first.

How did we figure that out?

We started off trying to solve the problem in one pass, and we noticed that it wouldn't work. We then noticed the *reason* it wouldn't work: to see if a given meeting can be merged, we have to look at *all* the other meetings! That's because the order of the meetings is random.

That's what got us thinking: what if the list *were* sorted? We saw that *then* a greedy approach would work. We had to spend $O(n \lg n)$ time on sorting the list, but it was better than our initial brute force approach, which cost us $O(n^2)$ time!