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 arrays

An **array** is a data structure that holds an **ordered collection of elements**. Each position in the array has an **index**, starting with 0.

In a low-level array, you must specify the size of your array when you instantiate it. Under the hood, arrays are stored in a contiguous chunk of memory large enough to hold that many elements.

C

```
// declare an array that holds 3 integers
int gasPrices[3];

gasPrices[0] = 346;
gasPrices[1] = 360;
gasPrices[2] = 354;
```

Arrays are efficient for looking up the element at an index, because if you know the address where an array starts in memory, it's simple math to find the address of any index. This gives arrays an $O(1)$ lookup time.

Arrays are the foundation of many other data structures, like dynamic arrays, stacks, and hashes.

Some languages, including Ruby, have dynamic arrays, which are usually easier to work with. Confusingly, in Ruby, these are just called "arrays."

A **dynamic array (/concept/dynamic-array-amortized-analysis)** doesn't require you to specify its length and allows you to seamlessly (although sometimes with time and space costs) insert and delete elements at any index.

In Ruby, you can simply say:

Ruby

```
gas_prices = []

gas_prices[0] = 346
gas_prices[1] = 360
gas_prices[2] = 354
```

Here, the details about the length are abstracted out for you. You can add as many prices as you'd like.

> Fun fact: **strings** are almost always implemented as arrays of characters.

of integers `[start_time, end_time]`. These integers represent the number of 30-minute blocks past 9:00am.

For example:

```ruby
[2, 3] # meeting from 10:00 - 10:30 am
[6, 9] # meeting from 12:00 - 1:30 pm
```

Write a function `merge_ranges()` that takes an array of meeting time ranges and returns an array of condensed ranges.

For example, given:

```ruby
[[0, 1], [3, 5], [4, 8], [10, 12], [9, 10]]
```

your function would return:

```ruby
[[0, 1], [3, 8], [9, 12]]
```

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 function to work even for very large numbers, like Unix timestamps. In any case, the spirit of the challenge is to merge meetings where `start_time` and `end_time` don't have an upper bound.

Gotchas

Look at this case:

```ruby
[[1, 2], [2, 3]]
```

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

Look at this case:

```ruby
[[1, 5], [2, 3]]
```

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

Look at this case:

```ruby
[[1, 10], [2, 6], [3, 5], [7, 9]]
```

Here *all* of our meetings should be merged together into just `[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 function 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:

```ruby
[[1, 3], [2, 4]]
```

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

```ruby
[[1, 4]]
```

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: `[[1, 2], [2, 3]]`
2. The second meeting ends before the first meeting ends. For example: `[[1, 5], [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 array 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 array 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 array 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 `merged_meetings`.

```ruby
def merge_ranges(meetings)

    # sort by start times
    sorted_meetings = meetings.sort

    # initialize merged_meetings with the earliest meeting
    merged_meetings = [sorted_meetings[0]]

    sorted_meetings[1..-1].each do |current_meeting_start, current_meeting_end|

        last_merged_meeting_start, last_merged_meeting_end = merged_meetings[-1]

        # if the current and last meetings overlap, use the latest end time
        if current_meeting_start <= last_merged_meeting_end
            merged_meetings[-1] = [last_merged_meeting_start, [last_merged_meeting_end, cur

        # add the current meeting since it doesn't overlap
        else
            merged_meetings.push([current_meeting_start, current_meeting_end])
        end
    end

    return merged_meetings
end
```

Complexity

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

Even though we only walk through our array 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 array of merged meeting times. In the worst case, none of the meetings overlap, giving us an array identical to the input array. 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 array and save some space? What are the pros and cons of doing this in-place?

What We Learned

This one arguably uses a greedy⤵

> A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.
>
> The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

approach as well, except this time we had to *sort* the array 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 array *were* sorted? We saw that *then* a greedy approach would work. We had to spend $O(n \lg n)$ time on sorting the array, but it was better than our initial brute force approach, which cost us $O(n^2)$ time!
