Interview Cake

In order to win the prize for most cookies sold, my friend Alice and I are going to merge our Girl Scout Cookies orders and enter as one unit.

Each order is represented by an "order id" (an integer).

We have our lists of orders sorted numerically already, in arrays. Write a function to merge our arrays of orders into one sorted array.

For example:

Ruby ▾

```
my_array     = [3, 4, 6, 10, 11, 15]
alices_array = [1, 5, 8, 12, 14, 19]

puts merge_arrays(my_array, alices_array)
# prints [1, 3, 4, 5, 6, 8, 10, 11, 12, 14, 15, 19]
```

Gotchas

We can do this in $O(n)$ time and space.

If you're running a built-in sorting function, your algorithm probably takes $O(n \lg n)$ time for that sort.

Think about edge cases! What happens when we've merged in all of the elements from one of our arrays but we still have elements to merge in from our other array?

Breakdown

We could simply concatenate (join together) the two arrays into one, then sort the result:

Ruby ▾

```ruby
def merge_sorted_arrays(arr1, arr2)
    return (arr1 + arr2).sort
end
```

What would the time cost be?

$O(n \lg n)$, where n is the total length of our output array (the sum of the lengths of our inputs).

We can do better. With this algorithm, we're not really taking advantage of the fact that the input arrays are themselves *already sorted*. How can we save time by using this fact?

A good general strategy for thinking about an algorithm is to try writing out a sample input and performing the operation by hand. If you're stuck, try that!

Since our arrays are sorted, we know they each have their smallest item in the 0th index. **So the smallest item overall is in the 0th index of one of our input arrays!**

Which 0th element is it? Whichever is smaller!

To start, let's just write a function that chooses the $0th$ element for our sorted array.

Ruby ▾

```ruby
def merge_arrays(my_array, alices_array)

    # make an array big enough to fit the elements from both arrays
    merged_array_size = my_array.length + alices_array.length
    merged_array = [nil] * merged_array_size

    head_of_my_array = my_array[0]
    head_of_alices_array = alices_array[0]

    # case: 0th comes from my array
    if head_of_my_array < head_of_alices_array
        merged_array[0] = head_of_my_array

    # case: 0th comes from Alice's array
    else
        merged_array[0] = head_of_alices_array
    end

    # eventually we'll want to return the merged array
    return merged_array
end
```

Okay, good start! That works for finding the 0th element. Now how do we choose the next element?

Let's look at a sample input:

Ruby ▾

```ruby
[3,  4,  6, 10, 11, 15] # my_array
[1,  5,  8, 12, 14, 19] # alices_array
```

To start we took the 0th element from `alices_array` and put it in the 0th slot in the output array:

Ruby ▾

```ruby
[3,  4,  6, 10, 11, 15] # my_array
[1,  5,  8, 12, 14, 19] # alices_array
[1,  x,  x,  x,  x,  x] # merged_array
```

We need to make sure we don't try to put that 1 in `merged_array` again. We should mark it as "already merged" somehow. For now, we can just cross it out:

Ruby ▾

```
[3,  4,  6, 10, 11, 15] # my_array
[x,  5,  8, 12, 14, 19] # alices_array
[1,  x,  x,  x,  x,  x] # merged_array
```

Or we could even imagine it's removed from the array:

Ruby ▾

```
[3,  4,  6, 10, 11, 15] # my_array
[5,  8, 12, 14, 19]     # alices_array
[1,  x,  x,  x,  x,  x] # merged_array
```

Now to get our next element we can use the same approach we used to get the 0th element—it's the smallest of the *earliest unmerged elements* in either array! In other words, it's the smaller of the leftmost elements in either array, assuming we've removed the elements we've already merged in.

So in general we could say something like:

1. We'll start at the beginnings of our input arrays, since the smallest elements will be there.
2. As we put items in our final `merged_array`, we'll keep track of the fact that they're "already merged."
3. At each step, each array has a *first* "not-yet-merged" item.
4. At each step, the next item to put in the `merged_array` is the smaller of those two "not-yet-merged" items!

Can you implement this in code?

Ruby ▾

```ruby
def merge_arrays(my_array, alices_array)

    merged_array_size = my_array.length + alices_array.length
    merged_array = [nil] * merged_array_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_array_size
        first_unmerged_alices = alices_array[current_index_alices]
        first_unmerged_mine = my_array[current_index_mine]

        # case: next comes from my array
        if first_unmerged_mine < first_unmerged_alices
            merged_array[current_index_merged] = first_unmerged_mine
            current_index_mine += 1

        # case: next comes from Alice's array
        else
            merged_array[current_index_merged] = first_unmerged_alices
            current_index_alices += 1
        end

        current_index_merged += 1
    end

    return merged_array
end
```

Okay, this algorithm makes sense. To wrap up, we should think about edge cases and check for bugs. What edge cases should we worry about?

Here are some edge cases:

1. One or both of our input arrays is 0 elements or 1 element
2. One of our input arrays is longer than the other.
3. One of our arrays runs out of elements before we're done merging.

Actually, 3 will *always* happen. In the process of merging our arrays, we'll certainly exhaust one before we exhaust the other.

Does our function handle these cases correctly?

If both arrays are empty, we're fine. But for all the other edge cases, at some point `current_index_mine` or `current_index_alices` will be `nil` because there won't be an element at one of those indices. So we'll either get a `NoMethodError` for calling "less than" on `nil`, or an `ArgumentError` for calling "less than" on an integer but passing `nil` as an argument! (Remember, `9 < 1` is just shorthand for `9.<(1)`)

How can we fix this?

We can probably solve these cases at the same time. They're not so different—they just have to do with handling empty arrays.

To start, we could treat each of our arrays being out of elements as a separate case to handle, in addition to the 2 cases we already have. So we have 4 cases total. Can you code that up?

Be sure you check the cases in the right order!

Ruby ▾

```ruby
def merge_arrays(my_array, alices_array)
    merged_array_size = my_array.length + alices_array.length
    merged_array = [nil] * merged_array_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_array_size

        # case: my array is exhausted
        if current_index_mine >= my_array.length
            merged_array[current_index_merged] = alices_array[current_index_alices]
            current_index_alices += 1

        # case: Alice's array is exhausted
        elsif current_index_alices >= alices_array.length
            merged_array[current_index_merged] = my_array[current_index_mine]
            current_index_mine += 1

        # case: my item is next
        elsif my_array[current_index_mine] < alices_array[current_index_alices]
            merged_array[current_index_merged] = my_array[current_index_mine]
            current_index_mine += 1

        # case: Alice's item is next
        else
            merged_array[current_index_merged] = alices_array[current_index_alices]
            current_index_alices += 1
        end

        current_index_merged += 1
    end

    return merged_array
end
```

Cool. This'll work, but it's a bit repetitive. We have these two lines twice:

```ruby
merged_array[current_index_merged] = my_array[current_index_mine]
current_index_mine += 1
```

Same for these two lines:

```ruby
merged_array[current_index_merged] = alices_array[current_index_alices]
current_index_alices += 1
```

That's not DRY↴ . Maybe we can avoid repeating ourselves by bringing our code back down to just 2 cases.

See if you can do this in just one "if else" by combining the conditionals.

You might try to simply squish the middle cases together:

```ruby
if is_alices_array_exhausted || \
        my_array[current_index_mine] < alices_array[current_index_alices])

    merged_array[current_index_merged] = my_array[current_index_mine]
    current_index_mine += 1
```

But what happens when `my_array` is exhausted?

We'll get a `NoMethodError` when we try calling "less than" on `my_array[current_index_mine]` because it'll be `nil`!

How can we fix this?

Solution

First, we allocate our answer array, getting its size by adding the size of `my_array` and `alices_array`.

We keep track of a current index in `my_array`, a current index in `alices_array`, and a current index in `merged_array`. So at each step, there's a "current item" in `alices_array` and in `my_array`. The smaller of those is the next one we add to the `merged_array`!

But careful: we also need to account for the case where we exhaust one of our arrays and there are still elements in the other. To handle this, we say that the current item in `my_array` is the next item to add to `merged_array` only if `my_array` is *not* exhausted AND, either:

1. `alices_array` is exhausted, or
2. the current item in `my_array` is less than the current item in `alices_array`

Ruby ▾

```ruby
def merge_arrays(my_array, alices_array)

    # set up our merged_array
    merged_array_size = my_array.length + alices_array.length
    merged_array = [nil] * merged_array_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_array_size

        is_my_array_exhausted = current_index_mine >= my_array.length
        is_alices_array_exhausted = current_index_alices >= alices_array.length

        # case: next comes from my array
        # my array must not be exhausted, and EITHER:
        # 1) Alice's array IS exhausted, or
        # 2) the current element in my array is less
        #    than the current element in Alice's array
        if !is_my_array_exhausted and (is_alices_array_exhausted || \
                (my_array[current_index_mine] < alices_array[current_index_alices]))

            merged_array[current_index_merged] = my_array[current_index_mine]
            current_index_mine += 1

        # case: next comes from Alice's array
        else
            merged_array[current_index_merged] = alices_array[current_index_alices]
            current_index_alices += 1
        end

        current_index_merged += 1
    end

    return merged_array
end
```

The if statement is carefully constructed to avoid indexing into an empty array, because Ruby would give us `nil` and we'd get a `NoMethodError` or an `ArgumentError` when we tried comparing `nil` with an integer. We take advantage of Ruby's lazy evaluation↴ and check *first* if the arrays are exhausted.

Complexity

$O(n)$ time and $O(n)$ additional space, where n is the number of items in the merged array.

The added space comes from allocating the `merged_array`. There's no way to do this " in-place↴ " because neither of our input arrays are necessarily big enough to hold the merged array.

But if our inputs were linked lists, we could avoid allocating a new structure and do the merge by simply adjusting the `next` pointers in the list nodes!

In our implementation above, we could avoid tracking `current_index_merged` and just compute it on the fly by adding `current_index_mine` and `current_index_alices`. This would only save us one integer of space though, which is hardly anything. It's probably not worth the added code complexity.

Bonus

What if we wanted to merge *several* sorted arrays? Write a function that takes as an input *an array of sorted arrays* and outputs a single sorted array with all the items from each array.

What We Learned

We spent a lot of time figuring out how to cleanly handle edge cases.

Sometimes it's easy to lose steam at the end of a coding interview when you're debugging. But keep sprinting through to the finish! Think about edge cases. Look for off-by-one errors.