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:

```javascript
var myArray     = [3, 4, 6, 10, 11, 15];
var alicesArray = [1, 5, 8, 12, 14, 19];

console.log(mergeArrays(myArray, alicesArray));
// logs [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:

```javascript
function mergeSortedArrays(myArray, alicesArray) {
    var mergedArray = myArray.concat(alicesArray);
    return mergedArray.sort(function(a, b) {return a - b});
}
```

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.

```JavaScript
function mergeArrays(myArray, alicesArray) {

    var mergedArray = [];

    var headOfMyArray = myArray[0];
    var headOfAlicesArray = alicesArray[0];

    // case: 0th comes from my array
    if (headOfMyArray < headOfAlicesArray) {
        mergedArray[0] = headOfMyArray;

    // case: 0th comes from Alice's array
    } else {
        mergedArray[0] = headOfAlicesArray;
    }

    // eventually we'll want to return the merged array
    return mergedArray;
}
```

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:

```JavaScript
[3,  4,  6, 10, 11, 15] // myArray
[1,  5,  8, 12, 14, 19] // alicesArray
```

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

```JavaScript
[3,  4,  6, 10, 11, 15] // myArray
[1,  5,  8, 12, 14, 19] // alicesArray
[1,  x,  x,  x,  x,  x] // mergedArray
```

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

```JavaScript
[3,  4,  6, 10, 11, 15] // myArray
[x,  5,  8, 12, 14, 19] // alicesArray
[1,  x,  x,  x,  x,  x] // mergedArray
```

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

```JavaScript
[3,  4,  6, 10, 11, 15] // myArray
[5,  8, 12, 14, 19]     // alicesArray
[1,  x,  x,  x,  x,  x] // mergedArray
```

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 `mergedArray`, 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 `mergedArray` is the smaller of those two "not-yet-merged" items!

Can you implement this in code?

```JavaScript
function mergeArrays(myArray, alicesArray) {

    var mergedArray = [];

    var currentIndexAlices = 0;
    var currentIndexMine   = 0;
    var currentIndexMerged = 0;

    while (currentIndexMerged < (myArray.length + alicesArray.length)) {
        var firstUnmergedAlices = alicesArray[currentIndexAlices];
        var firstUnmergedMine = myArray[currentIndexMine];

        // case: next comes from my array
        if (firstUnmergedMine < firstUnmergedAlices) {
            mergedArray[currentIndexMerged] = firstUnmergedMine;
            currentIndexMine++;

        // case: next comes from Alice's array
        } else {
            mergedArray[currentIndexMerged] = firstUnmergedAlices;
            currentIndexAlices++;
        }

        currentIndexMerged++;
    }

    return mergedArray;
}
```

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 `currentIndexMine` or `currentIndexAlices` will be `undefined` because there won't be an element at one of those indices. Then JavaScript will compare `undefined` with a number, which will always be false, and `mergedArray` might be out of order or contain `undefined`!

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!

```JavaScript
function mergeArrays(myArray, alicesArray) {

    var mergedArray = [];

    var currentIndexAlices = 0;
    var currentIndexMine   = 0;
    var currentIndexMerged = 0;

    while (currentIndexMerged < (myArray.length + alicesArray.length)) {

        // case: my array is exhausted
        if (currentIndexMine >= myArray.length) {
            mergedArray[currentIndexMerged] = alicesArray[currentIndexAlices];
            currentIndexAlices++;

        // case: Alice's array is exhausted
        } else if (currentIndexAlices >= alicesArray.length) {
            mergedArray[currentIndexMerged] = myArray[currentIndexMine];
            currentIndexMine++;

        // case: my item is next
        } else if (myArray[currentIndexMine] < alicesArray[currentIndexAlices]) {
            mergedArray[currentIndexMerged] = myArray[currentIndexMine];
            currentIndexMine++;

        // case: Alice's item is next
        } else {
            mergedArray[currentIndexMerged] = alicesArray[currentIndexAlices];
            currentIndexAlices++;
        }

        currentIndexMerged++;
    }

    return mergedArray;
}
```

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

```
mergedArray[currentIndexMerged] = myArray[currentIndexMine];
currentIndexMine++;
```

Same for these two lines:

```
mergedArray[currentIndexMerged] = alicesArray[currentIndexAlices];
currentIndexAlices++;
```

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:

```
if (isAlicesArrayExhausted ||
        (myArray[currentIndexMine] < alicesArray[currentIndexAlices])) {

    mergedArray[currentIndexMerged] = myArray[currentIndexMine];
    currentIndexMine++;
```

But what happens when `myArray` is exhausted?

`myArray[currentIndexMine]` will be `undefined`. But our code will still work! JavaScript gives `false` for any comparison with `undefined`, and we *want* false here—we're checking if the element in `myArray` comes first, and it *doesn't* if `myArray` is empty. But this code wouldn't work in many other languages—we'd get an error when we tried to access an element in an empty array or compare a number with a non-numerical value like `undefined`.

Even though our code *works*, it's messy to access and compare an element that doesn't exist. Let's adjust our code so we're more explicit and don't rely on JavaScript's uncommon and nonobvious behavior of giving `false` in comparisons with `undefined`.

Solution

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

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

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 `myArray` is the next item to add to `mergedArray` only if `myArray` is *not* exhausted AND, either:

1. `alicesArray` is exhausted, or
2. the current item in `myArray` is less than the current item in `alicesArray`

JavaScript

```
function mergeArrays(myArray, alicesArray) {

    // set up our mergedArray
    var mergedArray = [];

    var currentIndexAlices = 0;
    var currentIndexMine   = 0;
    var currentIndexMerged = 0;

    while (currentIndexMerged < (myArray.length + alicesArray.length)) {

        var isMyArrayExhausted = currentIndexMine >= myArray.length;
        var isAlicesArrayExhausted = currentIndexAlices >= alicesArray.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 (!isMyArrayExhausted && (isAlicesArrayExhausted ||
                (myArray[currentIndexMine] < alicesArray[currentIndexAlices]))) {

            mergedArray[currentIndexMerged] = myArray[currentIndexMine];
            currentIndexMine++;

        // case: next comes from Alice's array
        } else {
            mergedArray[currentIndexMerged] = alicesArray[currentIndexAlices];
            currentIndexAlices++;
        }

        currentIndexMerged++;
    }

    return mergedArray;
}
```

The if statement is carefully constructed to avoid indexing into an empty array, because JavaScript would give us `undefined` and we don't want to compare `undefined` with an integer. We take advantage of JavaScript'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 `mergedArray`. 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 `currentIndexMerged` and just compute it on the fly by adding `currentIndexMine` and `currentIndexAlices`. 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.