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:

```java
int[] myArray     = new int[]{3, 4, 6, 10, 11, 15};
int[] alicesArray = new int[]{1, 5, 8, 12, 14, 19};

System.out.println(mergeArrays(myArray, alicesArray));
// 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:

```java
public int[] mergeSortedArrays(int[] myArray, int[] alicesArray) {
    int[] mergedArray = Arrays.copyOf(myArray, myArray.length + alicesArray.length);
    System.arraycopy(alicesArray, 0, mergedArray, myArray.length, alicesArray.length);
    Arrays.sort(mergedArray);
    return mergedArray;
}
```

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.

```java
public int[] mergeArrays(int[] myArray, int[] alicesArray) {

    // make an array big enough to fit the elements from both arrays
    int[] mergedArray = new int[myArray.length + alicesArray.length];

    int headOfMyArray = myArray[0];
    int 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:

```Java
[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:

```Java
[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:

```Java
[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:

```Java
[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?

Java

```
public int[] mergeArrays(int[] myArray, int[] alicesArray) {

    int[] mergedArray = new int[myArray.length + alicesArray.length];

    int currentIndexAlices = 0;
    int currentIndexMine   = 0;
    int currentIndexMerged = 0;

    while (currentIndexMerged < mergedArray.length) {
        int firstUnmergedAlices = alicesArray[currentIndexAlices];
        int 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?

We'll get `an ArrayIndexOutOfBoundsException` in all three cases!

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!

Java

```
public int[] mergeArrays(int[] myArray, int[] alicesArray) {

    int[] mergedArray = new int[myArray.length + alicesArray.length];

    int currentIndexAlices = 0;
    int currentIndexMine   = 0;
    int currentIndexMerged = 0;

    while (currentIndexMerged < mergedArray.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:

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

Same for these two lines:

```java
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:

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


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

But what happens when `myArray` is exhausted?

We'll get an `ArrayIndexOutOfBoundsException` when we try to access `myArray[currentIndexMine]`!

How can we fix this?

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`

Java

```
public int[] mergeArrays(int[] myArray, int[] alicesArray) {

    // set up our mergedArray
    int[] mergedArray = new int[myArray.length + alicesArray.length];

    int currentIndexAlices = 0;
    int currentIndexMine   = 0;
    int currentIndexMerged = 0;

    while (currentIndexMerged < mergedArray.length) {

        boolean isMyArrayExhausted = currentIndexMine >= myArray.length;
        boolean 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 an `ArrayIndexOutOfBoundsException` from indexing into an empty array. We take advantage of Java's lazy evaluation.

> **Lazy evaluation** (called **short-circuit evaluation** in compiled languages) is a strategy some programming languages use to save work for the last minute or avoid unnecessary work altogether. For example, suppose we had a conditional like this:
>
> ```java
> if (itIsFriday && itIsRaining) {
>     System.out.println("board games at my place!");
> }
> ```
>
> Suppose `itIsFriday` was false. Because Java short-circuits evaluation, it wouldn't bother checking the value of `itIsRaining`—it knows that either way the result of our `&&` will be false, so we won't print the invitation to board game night.
>
> We can use this to our advantage. For example, suppose we have a check like this:
>
> ```java
> if (friends.get("Becky").isFreeThisFriday()) {
>     inviteToBoardGameNight(friends.get("Becky"));
> }
> ```
>
> What happens if 'Becky' isn't in our `friends` hash map? In Java, we'll get a `NullPointerException` (Python would similarly raise a `KeyError`, but Ruby and JavaScript would just give us a null object).

Instead, we could first confirm that 'Becky' and I are still on good terms:

```java
if (friends.containsKey("Becky") && friends.get("Becky").isFreeThisFriday()) {
    inviteToBoardGameNight(friends.get("Becky"));
}
```

This way, if 'Becky' *isn't* in `friends`, Java will skip the second check about Becky being free and avoid throwing the `NullPointerException`!

> This is all hypothetical, of course. It's not like things with Becky are weird or anything. We're totally cool. She's still in my friends dictionary for sure and I hope I'm still in hers and Becky if you're reading this I just want you to know you're still in my friends dicitionary.

Python's **generators** are also an example of lazy evaluation. For example, the function `range()` in Python generates a list of numbers in a specific range:

```python
print range(1, 11)
# prints [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# (the first argument to range()
# is inclusive, and the second is exclusive)
```

This is commonly used for looping. For example, if we wanted to count to `some_high_number`, we could do this:

```python
for i in range(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

But this will generate a list in memory whose size is order of `some_high_number`! That could be a lot of space.

So instead, we could use a generator. It behaves like a list in that we can loop through it, but instead of building up all of its contents at once, it simply generates the *next* element right when it's needed (lazily)!

There's a generator version of `range()` in Python: `xrange()`:

Python

```
# much more memory efficient!
for i in xrange(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

In Python 3 they went ahead and made `range()` a generator, so there is no `xrange()`.

We can also take a **lazy approach** in system design. For example, suppose we had a class for tracking temperatures:

Java

OOPS! WE'RE MISSING A JAVA TRANSLATION. SHOWING C# FOR NOW.

```
public class TempTracker
{
    private List<int> _recordedTemps = new List<int>();

    public void Record(int temp)
    {
        _recordedTemps.Add(temp);
    }
}
```

Suppose we wanted to add a feature for getting the the highest temperature we've seen so far. We could "eagerly" keep the max up to date whenever we insert a new temperature:

Java

OOPS! WE'RE MISSING A JAVA TRANSLATION. SHOWING C# FOR NOW.

```
public class TempTrackerEager
{
    private List<int> _recordedTemps = new List<int>();
    private int _maxTemp;

    public void Record(int temp)
    {
        _recordedTemps.Add(temp);
        if (temp > _maxTemp)
            _maxTemp = temp;
    }

    public int GetMax()
    {
        return _maxTemp;
    }
}
```

Or we could lazily (or "just in time") calculate the max whenever it's requested:

Java

OOPS! WE'RE MISSING A JAVA TRANSLATION. SHOWING C# FOR NOW.

```
public class TempTrackerLazy
{
    private List<int> _recordedTemps = new List<int>();

    public void Record(int temp)
    {
        _recordedTemps.Add(temp);
    }

    public int GetMax()
    {
        return _recordedTemps.Max();
    }
}
```

The best choice depends on how often you expect to run `getMax()`!

> Becky, I haven't hosted another board game night since the incident. I know we both said things we didn't really mean and anyway Becky just if you're reading this please know that I've been cake free for 3 whole days now and it's hard but I'm doing it for you PLEASE Becky. Please.

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

An **in-place** algorithm operates *directly* on its input and *changes* it, instead of creating and returning a *new* object. This is sometimes called **destructive**, since the original input is "destroyed" when it's edited to create the new output.

Careful: "In-place" does *not* mean "without creating any additional variables"! Rather, it means "without creating a new copy of the input." In general, an in-place function will only create additional variables that are $O(1)$ space.

Here are two functions that do the same operation, except one is in-place and the other is out-of-place:

```java
public int[] squareArrayInPlace(int[] intArray) {

    for (int i = 0; i < intArray.length; i++) {
        intArray[i] *= intArray[i];
    }

    // NOTE: we don't *need* to return anything
    // this is just a convenience
    return intArray;
}

public int[] squareArrayOutOfPlace(int[] intArray) {

    // we allocate a new array with the length of the input array
    int[] squaredArray = new int[intArray.length];

    for (int i = 0; i < intArray.length; i++) {
        squaredArray[i] = (int) Math.pow(intArray[i], 2);
    }

    return squaredArray;
}
```

Working in-place is a good way to save space. An in-place algorithm will generally have $O(1)$ space cost.

But be careful: an in-place algorithm can cause side effects. Your input is "destroyed" or "altered," which can affect code *outside* of your function. For example:

```java
int[] originalArray = new int[]{2, 3, 4, 5};
int[] squaredArray  = squareArrayInPlace(originalArray);

System.out.println("squared: " + Arrays.toString(squaredArray));
// prints: squared: [4, 9, 16, 25]

System.out.println("original array: " + Arrays.toString(originalArray));
// prints: original array: [4, 9, 16, 25], confusingly!

// and if squareArrayInPlace() didn't return anything,
// which it could reasonably do, squaredArray would be null!
```

Generally, out-of-place algorithms are considered safer because they avoid side effects. You should only use an in-place algorithm if you're very space constrained or you're *positive* you don't need the original input anymore, even for debugging.

" 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.
