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 lists. Write a function to merge our lists of orders into one sorted list.

For example:

```python
my_list     = [3, 4, 6, 10, 11, 15]
alices_list = [1, 5, 8, 12, 14, 19]

print merge_lists(my_list, alices_list)
# 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 lists but we still have elements to merge in from our other list?

Breakdown

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

```python
def merge_sorted_lists(arr1, arr2):
    return sorted(arr1 + arr2)
```

What would the time cost be?

$O(n \lg n)$, where n is the total length of our output list (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 lists 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 lists 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 lists!**

Which 0th element is it? Whichever is smaller!

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

```python
def merge_lists(my_list, alices_list):

    # make a list big enough to fit the elements from both lists
    merged_list_size = len(my_list) + len(alices_list)
    merged_list = [None] * merged_list_size

    head_of_my_list = my_list[0]
    head_of_alices_list = alices_list[0]

    # case: 0th comes from my list
    if head_of_my_list < head_of_alices_list:
        merged_list[0] = head_of_my_list

    # case: 0th comes from Alice's list
    else:
        merged_list[0] = head_of_alices_list

    # eventually we'll want to return the merged list
    return merged_list
```

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:

```python
[3,  4,  6, 10, 11, 15] # my_list
[1,  5,  8, 12, 14, 19] # alices_list
```

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

```python
[3,  4,  6, 10, 11, 15] # my_list
[1,  5,  8, 12, 14, 19] # alices_list
[1,  x,  x,  x,  x,  x] # merged_list
```

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

```python
[3,  4,  6, 10, 11, 15] # my_list
[x,  5,  8, 12, 14, 19] # alices_list
[1,  x,  x,  x,  x,  x] # merged_list
```

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

```python
[3,  4,  6, 10, 11, 15] # my_list
[5,  8, 12, 14, 19]     # alices_list
[1,  x,  x,  x,  x,  x] # merged_list
```

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 list! In other words, it's the smaller of the leftmost elements in either list, 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 lists, since the smallest elements will be there.
2. As we put items in our final `merged_list`, we'll keep track of the fact that they're "already merged."
3. At each step, each list has a *first* "not-yet-merged" item.

4. At each step, the next item to put in the `merged_list` is the smaller of those two "not-yet-merged" items!

Can you implement this in code?

Python ▾

```python
def merge_lists(my_list, alices_list):

    merged_list_size = len(my_list) + len(alices_list)
    merged_list = [None] * merged_list_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_list_size:
        first_unmerged_alices = alices_list[current_index_alices]
        first_unmerged_mine = my_list[current_index_mine]

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

        # case: next comes from Alice's list
        else:
            merged_list[current_index_merged] = first_unmerged_alices
            current_index_alices += 1

        current_index_merged += 1

    return merged_list
```

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

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

Does our function handle these cases correctly?

We'll get `an IndexError` 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 lists.

To start, we could treat each of our lists 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!

Python ▾

```python
def merge_lists(my_list, alices_list):
    merged_list_size = len(my_list) + len(alices_list)
    merged_list = [None] * merged_list_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_list_size:

        # case: my list is exhausted
        if current_index_mine >= len(my_list):
            merged_list[current_index_merged] = alices_list[current_index_alices]
            current_index_alices += 1

        # case: Alice's list is exhausted
        elif current_index_alices >= len(alices_list):
            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1

        # case: my item is next
        elif my_list[current_index_mine] < alices_list[current_index_alices]:
            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1

        # case: Alice's item is next
        else:
            merged_list[current_index_merged] = alices_list[current_index_alices]
            current_index_alices += 1

        current_index_merged += 1

    return merged_list
```

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

Python ▾

```python
merged_list[current_index_merged] = my_list[current_index_mine]
current_index_mine += 1
```

Same for these two lines:

```python
merged_list[current_index_merged] = alices_list[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:

```python
if is_alices_list_exhausted or \
        my_list[current_index_mine] < alices_list[current_index_alices]):

    merged_list[current_index_merged] = my_list[current_index_mine]
    current_index_mine += 1
```

But what happens when `my_list` is exhausted?

We'll get an `IndexError` when we try to access `my_list[current_index_mine]`!

How can we fix this?

Solution

First, we allocate our answer list, getting its size by adding the size of `my_list` and `alices_list`.

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

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

1. `alices_list` is exhausted, or

2. the current item in `my_list` is less than the current item in `alices_list`

Python ▾

```python
def merge_lists(my_list, alices_list):

    # set up our merged_list
    merged_list_size = len(my_list) + len(alices_list)
    merged_list = [None] * merged_list_size

    current_index_alices = 0
    current_index_mine = 0
    current_index_merged = 0

    while current_index_merged < merged_list_size:

        is_my_list_exhausted = current_index_mine >= len(my_list)
        is_alices_list_exhausted = current_index_alices >= len(alices_list)

        # case: next comes from my list
        # my list must not be exhausted, and EITHER:
        # 1) Alice's list IS exhausted, or
        # 2) the current element in my list is less
        #    than the current element in Alice's list
        if not is_my_list_exhausted and (is_alices_list_exhausted or \
                (my_list[current_index_mine] < alices_list[current_index_alices])):

            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1

        # case: next comes from Alice's list
        else:
            merged_list[current_index_merged] = alices_list[current_index_alices]
            current_index_alices += 1

        current_index_merged += 1

    return merged_list
```

The if statement is carefully constructed to avoid an `IndexError` from indexing into an empty array. We take advantage of Python's lazy evaluation↴ and check *first* if the lists are exhausted.

Complexity

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

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

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.

Trivia! Python's native sorting algorithm is called `Timsort`. It's actually *optimized* for sorting lists where subsections of the lists are already sorted. For this reason, a more naive algorithm:

```python
def merge_sorted_lists(arr1, arr2):
    return sorted(arr1 + arr2)
```

is actually *faster* until n gets *pretty* big. Like 1,000,000.

Also, in Python 2.6+, there's a built-in function for merging sorted lists into one sorted list: `heapq.merge()`.

Bonus

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

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.