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

For example:

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

# Prints [1, 3, 4, 5, 6, 8, 10, 11, 12, 14, 15, 19]
print(merge_lists(my_list, alices_list))
```

Python 3.6

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 3.6

```
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 3.6

```
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]

    if head_of_my_list < head_of_alices_list:
        # Case: 0th comes from my list
        merged_list[0] = head_of_my_list
    else:
        # Case: 0th comes from Alice's list
        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:

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

Python 3.6

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

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

Python 3.6

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:

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

Python 3.6

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

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

Python 3.6

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 3.6

```
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]

        if first_unmerged_mine < first_unmerged_alices:
            # Case: next comes from my list
            merged_list[current_index_merged] = first_unmerged_mine
            current_index_mine += 1
        else:
            # Case: next comes from Alice's list
            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?

If both lists are empty, we're fine. But for all other edge cases, we'll get `an IndexError`.

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 indexing past the end of 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 3.6

```
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:
        if current_index_mine >= len(my_list):
            # Case: my list is exhausted
            merged_list[current_index_merged] = alices_list[current_index_alices]
            current_index_alices += 1
        elif current_index_alices >= len(alices_list):
            # Case: Alice's list is exhausted
            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1
        elif my_list[current_index_mine] < alices_list[current_index_alices]:
            # Case: my item is next
            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1
        else:
            # Case: Alice's item is next
            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 3.6

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

Same for these two lines:

Python 3.6

```
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 3.6

```
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 3.6

```
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)
        if (not is_my_list_exhausted and
                (is_alices_list_exhausted or
                 my_list[current_index_mine] < alices_list[current_index_alices])):
            # 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
            merged_list[current_index_merged] = my_list[current_index_mine]
            current_index_mine += 1
        else:
            # Case: next comes from Alice's list
            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 past the end of a list. We take advantage of Python 3.6's short circuit evaluation.

Short-circuit evaluation is a strategy most programming languages (including Python 3.6) use to avoid unnecessary work. For example, say we had a conditional like this:

Python 3.6

```
if it_is_friday and it_is_raining:
    print("board games at my place!")
```

Let's say `it_is_friday` is false. Because Python 3.6 short-circuits evaluation, it wouldn't bother checking the value of `it_is_raining`—it knows that either way the condition is false and we won't print the invitation to board game night.

We can use this to our advantage. For example, say we have a check like this:

```
if friends['Becky'].is_free_this_friday():
    invite_to_board_game_night(friends['Becky'])
```
Python 3.6

What happens if `'Becky'` isn't in our `friends` dictionary? We'll get a `KeyError` when we run `friends['Becky']`.

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

```
if 'Becky' in friends and friends['Becky'].is_free_this_friday():
    invite_to_board_game_night(friends['Becky'])
```
Python 3.6

This way, if `'Becky'` *isn't* in `friends`, Python will ignore the rest of the conditional and avoid throwing the `KeyError`.

> 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 dictionary.

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"↴

> An **in-place** function modifies data structures or objects outside of its own stack frame↴ (i.e.: stored on the process heap or in the stack frame of a calling function). Because of this, the changes made by the function remain after the call completes.

> In-place algorithms are sometimes called **destructive**, since the original input is "destroyed" (or modified) during the function call.

> **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.

An **out-of-place** function doesn't make any changes that are visible to other functions. Usually, those functions copy any data structures or objects before manipulating and changing them.

In many languages, **primitive** values (integers, floating point numbers, or characters) are copied when passed as arguments, and more complex **data structures** (lists, heaps, or hash tables) are passed by reference. This is what Python does.

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

Python 3.6

```
def square_list_in_place(int_list):
    for index, element in enumerate(int_list):
        int_list[index] *= element

    # NOTE: no need to return anything - we modified
    # int_list in place


def square_list_out_of_place(int_list):
    # We allocate a new list with the length of the input list
    squared_list = [None] * len(int_list)

    for index, element in enumerate(int_list):
        squared_list[index] = element ** 2

    return squared_list
```

Working in-place is a good way to save time and space. An in-place algorithm avoids the cost of initializing or copying data structures, and it usually has an $O(1)$ 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:

Python 3.6

```
original_list = [2, 3, 4, 5]
square_list_in_place(original_list)

print("original list: %s" % original_list)
# Prints: original list: [4, 9, 16, 25], confusingly!
```

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

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 2.7

```
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.

Do we absolutely have to allocate a new list to use for the merged output? Where else could we store our merged list? How would our function need to change?

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.