Interview Cake

← course home (/table-of-contents)

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 vectors. Write a function to merge our vectors of orders into one sorted vector.

For example:

C++

```
const vector<int> myVector {3, 4, 6, 10, 11, 15};
const vector<int> alicesVector {1, 5, 8, 12, 14, 19};

vector<int> mergedVector = mergeVectors(myVector, alicesVector);

cout << "[";
for (size_t i = 0; i < mergedVector.size(); ++i) {
    if (i > 0) {
        cout << ", ";
    }
    cout << mergedVector[i];
}
cout << "]" << endl;
// 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 vectors but we still have elements to merge in from our other vector?

Breakdown

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

C++

```cpp
vector<int> mergeSortedVectors(const vector<int>& myVector, const vector<int>& alicesVector)
{
    vector<int> mergedVector;
    mergedVector.insert(mergedVector.end(), myVector.cbegin(), myVector.cend());
    mergedVector.insert(mergedVector.end(), alicesVector.cbegin(), alicesVector.cend());
    sort(mergedVector.begin(), mergedVector.end());
    return mergedVector;
}
```

What would the time cost be?

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

Which 0th element is it? Whichever is smaller!

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

C++

```
vector<int> mergeVectors(const vector<int>& myVector, const vector<int>& alicesVector)
{
    // make a vector big enough to fit the elements from both vectors
    vector<int> mergedVector(myVector.size() + alicesVector.size());

    int headOfMyVector = myVector[0];
    int headOfAlicesVector = alicesVector[0];

    // case: 0th comes from my vector
    if (headOfMyVector < headOfAlicesVector) {
        mergedVector[0] = headOfMyVector;
    }

    // case: 0th comes from Alice's vector
    else {
        mergedVector[0] = headOfAlicesVector;
    }

    // eventually we'll want to return the merged vector
    return mergedVector;
}
```

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:

C++

```
[3,  4,  6, 10, 11, 15]  // myVector
[1,  5,  8, 12, 14, 19]  // alicesVector
```

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

```
[3,  4,  6, 10, 11, 15]  // myVector
[1,  5,  8, 12, 14, 19]  // alicesVector
[1,  x,  x,  x,  x,  x]  // mergedVector
```

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

```
[3,  4,  6, 10, 11, 15]  // myVector
[x,  5,  8, 12, 14, 19]  // alicesVector
[1,  x,  x,  x,  x,  x]  // mergedVector
```

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

```
[3,  4,  6, 10, 11, 15]  // myVector
[5,  8, 12, 14, 19]      // alicesVector
[1,  x,  x,  x,  x,  x]  // mergedVector
```

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

Can you implement this in code?

C++

```
vector<int> mergeVectors(const vector<int>& myVector, const vector<int>& alicesVector)
{
    vector<int> mergedVector(myVector.size() + alicesVector.size());

    size_t currentIndexAlices = 0;
    size_t currentIndexMine   = 0;
    size_t currentIndexMerged = 0;

    while (currentIndexMerged < mergedVector.size()) {
        int firstUnmergedAlices = alicesVector[currentIndexAlices];
        int firstUnmergedMine = myVector[currentIndexMine];

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

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

        ++currentIndexMerged;
    }

    return mergedVector;
}
```

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

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

Does our function handle these cases correctly?

If both vectors are empty, we're fine. But for all other edge cases, we'll get `a segfault`.

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 vectors.

To start, we could treat each of our vectors 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!

C++ ▾

```
vector<int> mergeVectors(const vector<int>& myVector, const vector<int>& alicesVector)
{
    vector<int> mergedVector(myVector.size() + alicesVector.size());

    size_t currentIndexAlices = 0;
    size_t currentIndexMine   = 0;
    size_t currentIndexMerged = 0;

    while (currentIndexMerged < mergedVector.size()) {

        // case: my vector is exhausted
        if (currentIndexMine >= myVector.size()) {
            mergedVector[currentIndexMerged] = alicesVector[currentIndexAlices];
            ++currentIndexAlices;
        }

        // case: Alice's vector is exhausted
        else if (currentIndexAlices >= alicesVector.size()) {
            mergedVector[currentIndexMerged] = myVector[currentIndexMine];
            ++currentIndexMine;
        }

        // case: my item is next
        else if (myVector[currentIndexMine] < alicesVector[currentIndexAlices]) {
            mergedVector[currentIndexMerged] = myVector[currentIndexMine];
            ++currentIndexMine;
        }

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

        ++currentIndexMerged;
    }

    return mergedVector;
}
```

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

C++

```
mergedVector[currentIndexMerged] = myVector[currentIndexMine];
++currentIndexMine;
```

Same for these two lines:

C++

```
mergedVector[currentIndexMerged] = alicesVector[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:

C++

```
if (isAlicesVectorExhausted
        || (myVector[currentIndexMine] < alicesVector[currentIndexAlices])) {

    mergedVector[currentIndexMerged] = myVector[currentIndexMine];
    ++currentIndexMine;
```

But what happens when `myVector` is exhausted?

We'll get a `segfault` when we try to access `myVector[currentIndexMine]`!

How can we fix this?

Solution

First, we allocate our answer vector, getting its size by adding the size of `myVector` and `alicesVector`.

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

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

1. `alicesVector` is exhausted, or
2. the current item in `myVector` is less than the current item in `alicesVector`

C++

```cpp
vector<int> mergeVectors(const vector<int>& myVector, const vector<int>& alicesVector)
{
    // set up our mergedVector
    vector<int> mergedVector(myVector.size() + alicesVector.size());

    size_t currentIndexAlices = 0;
    size_t currentIndexMine   = 0;
    size_t currentIndexMerged = 0;

    while (currentIndexMerged < mergedVector.size()) {

        bool isMyVectorExhausted = currentIndexMine >= myVector.size();
        bool isAlicesVectorExhausted = currentIndexAlices >= alicesVector.size();

        // case: next comes from my vector
        // my vector must not be exhausted, and EITHER:
        // 1) Alice's vector IS exhausted, or
        // 2) the current element in my vector is less
        //    than the current element in Alice's vector
        if (!isMyVectorExhausted && (isAlicesVectorExhausted
                || (myVector[currentIndexMine] < alicesVector[currentIndexAlices]))) {

            mergedVector[currentIndexMerged] = myVector[currentIndexMine];
            ++currentIndexMine;

        // case: next comes from Alice's vector
        }
        else {
            mergedVector[currentIndexMerged] = alicesVector[currentIndexAlices];
            ++currentIndexAlices;
        }

        ++currentIndexMerged;
    }

    return mergedVector;
}
```

The if statement is carefully constructed to avoid undefined behavior from indexing past the end of the vector. We take advantage of C++ short circuit evaluation↴ and check *first* if the vectors are exhausted.

Complexity

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

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

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 vectors? Write a function that takes as an input *a vector of sorted vectors* and outputs a single sorted vector with all the items from each vector.

Do we absolutely have to allocate a new vector to use for the merged output? Where else could we store our merged vector? 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.

🡄 course home (/table-of-contents)

Next up: Cafe Order Checker 🡆 (/question/cafe-order-checker?course=fc1&section=array-and-string-manipulation)