Finding the next-largest entry

Consider the following problem:

> Write a function that takes two arguments, an array of double and its capacity. Return the next-largest entry in the array, and if the largest entry is repeated, return the largest entry. You may assume the array has at least two entries.

We call it the next-largest and not the second-largest, as the next-largest may be equal to the largest. How can we tackle this problem? We will do so in three different ways.

Three problems, three loops

We can start by describing the algorithm as follows where there are three separate problems to solve, and this will require possibly three loops:

1. Go through the array to find the largest entry in the array.
2. Go through the array, and count how often the largest entry appears in the array.
3. If the largest entry appears more than once, return that value.
4. Otherwise, go through the array, and find the next-largest entry.

Here is an implementation of the description above:

```
double next_largest3( double array[], std::size_t capacity ) {
    assert( capacity >= 2 );
    // Find the largest entry of the array.
    double largest{ array[0] };

    for ( std::size_t k{1}; k < capacity; ++k ) {
        if ( array[k] > largest ) {
            largest = array[k];
        }
    }

    // We will go through the array and count how often the largest entry
    // appears. If it appears twice, we will return the largest entry, as
    // required in the specifications.

    int count{0};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] == largest ) {
            if ( count == 0 ) {
                count = 1;
            } else {
                assert( count == 1 );
                return largest;
            }
        }
    }
```

```
        // Sanity check: we should have found at least one entry equaling
        //               the largest value
        assert( count == 1 );

        // At this point, we are guaranteed there is only one entry in the
        // array containing the largest value. Thus, go through the array
        // and find the next-largest entry.

        double next_largest{ array[0] };

        for ( std::size_t k{1}; k < capacity; ++k ) {
            if ( (array[k] > next_largest) && (array[k] != largest) ) {
                next_largest = array[k];
            }
        }

        return next_largest;
    }
```

There is a bug in this program. Can you find it and fix it? It makes an assumption that is not necessarily valid. To fix it, you will be using the assumption that the array has a capacity of at least two.

Only two loops

Now, from a point of view of efficiency, this requires us to step through the loop three times, so if it takes 1 ms to go through a very large array, then it will take 3 ms to execute this program. Can we reduce this so that we only go through the array twice:

1. Go through the array to find the largest entry in the array and determine if it appears at least twice.
2. If the largest entry appears more than once, return that value.
3. Otherwise, go through the array, and find the next-largest entry.

Here is an implementation of the description above:

```
double next_largest2( double array[], std::size_t capacity ) {
    assert( capacity >= 2 );

    // Find the largest entry of the array.
    double largest{ array[0] };
    // As we have only checked the first entry, the current largest
    // can only be determined to have been found once.
    double count{ 1 };

    for ( std::size_t k{1}; k < capacity; ++k ) {
        if ( array[k] > largest ) {
            // If we find something larger than the largest current
            // largest, update the largest and reset the count to 1.
            largest = array[k];
            count = 1;
        } else if ( array[k] == largest ) {
            // We have now seen the largest entry one more time
            //   - Why can we not just break now and return 'largest'?
            ++count;
        }
    }

    if ( count >= 2 ) {
        // The largest appears at least twice, so return this value
        return largest;
    }

    // At this point, we are guaranteed there is only one entry in the
    // array containing the largest value. Thus, go through the array
    // and find the next-largest entry.

    double next_largest{ array[0] };

    for ( std::size_t k{1}; k < capacity; ++k ) {
        if ( (array[k] > next_largest) && (array[k] != largest) ) {
            next_largest = array[k];
        }
    }

    return next_largest;
}
```

So, now, our new program only goes through the array twice, reducing the time it takes to run this program by approximately 30%.

One loop

Can we do better? Can we find the next-largest entry at the same time we are finding the largest entry?

For this, we would have to have two local variables:

```
// 'largest' and 'next_largest' sound better
//     than 'largest' and 'next_largest' or something like that...
double largest;
double next_largest;
```

What should they be initialized with?

If you've thought about this, you may have considered the following:

```
if ( array[0] >= array[1] ) {
    largest      = array[0];
    next_largest = array[1];
} else {
    largest      = array[1];
    next_largest = array[0];
}
```

Now, when we check `array[k]`, if that entry is now larger than the current largest, then the current largest must become the current next-largest entry, and the new largest must be updated:

```
// In the for loop:
if ( array[k] > largest ) {
    next_largest = largest;
    largest = array[k];
}
```

If, however, `array[k]` equals the currently recorded largest entry, this means that that largest entry occurs at least twice. In this case, all we must do is set the next-largest value to this largest-currently-found value:

```
// Continue from previous conditional statement
} else if ( array[k] == largest ) {
    next_largest = largest;
}
```

If, however, `array[k]` is less than the current largest, but still larger than the current next-largest value, then this new value becomes a candidate for the next-largest entry of the array:

```
// Continue from previous conditional statement
} else if ( array[k] > next_largest ) {
    // At this point, array[k] can no longer be equal to the
    // largest value, so make sure this is correct.
    assert( array[k] < largest );
    next_largest = array[k];
}
```

At this point, we can simply return the next-largest entry:

```
return next_largest;
```

Thus, our program is:

```
double next_largest1( double array[], std::size_t capacity ) {
    assert( capacity >= 2 );

    // These two are to be initialized in
    // the next conditional statement.
    double largest;
    double next_largest;

    if ( array[0] >= array[1] ) {
        largest      = array[0];
        next_largest = array[1];
    } else {
        largest      = array[1];
        next_largest = array[0];
    }

    for ( std::size_t k{1}; k < capacity; ++k ) {
        // In the for loop:
        if ( array[k] > largest ) {
            next_largest = largest;
            largest = array[k];
        } else if ( array[k] == largest ) {
            next_largest = largest;
        } else if ( array[k] > next_largest ) {
            // At this point, array[k] can no longer be equal to the
            // largest value, so make sure this is correct.
            assert( array[k] < largest );
            next_largest = array[k];
        }
    }

    return next_largest;
}
```

Note, if you wanted to, you could *clean up* the code at the top by making it somewhat easier to understand, if a little more expensive to run:

```
    double largest{      array[0] };
    double next_largest{ array[1] };

    if ( largest < next_largest ) {
        std::swap( largest, next_largest );

    }
```

This final function does not suffer from the same issue as our first two programs. Did you fix the problem in the first two programs? If not, find the next largest entry in the array here:

```
double data[8]{23.32, 17.95, 18.47, 19.23, 13.33, 15.98, 11.23, 16.99};
```

Does either of the first two functions find the next-largest entry?

Online

Once you have tried this yourself, you can look at the complete solution at `repl.it` with the URL

https://repl.it/@dwharder/Next-largest-entry-of-an-array.