Represent the worst-case running time of your sortQueue algorithm as a mathe- matical function. Then, use the mathematical definition of big-O to determine an asymptotic bound on this function (that is, you should compute the values of c and n0 to show that the bound exists). You may assume the runtime complexities of the standard queue methods are as described in lectures.

"n" means the size of the input.
Use "n" to indicate the mathematical function of running time.

The worst-case runtime:

```
public static <T extends Comparable<T>> void sortQueue(Queue<T> queue) {
    for (int i = 0; i < queue.size(); i++) {

        T firstValue = queue.peek();
        queue.poll();
        int size = queue.size();
        T secondValue = queue.peek();

        for (int j = 0; j < size; j++) {

            // Compare two values and always put the larger one at the end of the queue.
            if (firstValue.compareTo(secondValue) >= 0) {
                queue.offer(secondValue);
                queue.poll();
                secondValue = queue.peek();
            } else {
                queue.offer(firstValue);
                firstValue = secondValue;
                queue.poll();
                secondValue = queue.peek();
            }
        }
        queue.offer(firstValue);
    }
}
```

Operations
1 + 2(n + 1)
n
n
n
n
n * (1 + 2(n + 1))
n * 2n
n * n
n * n
n * n
n * n
n

Total: $8n^2 + 10n + 3$

Show that $8n^2 + 10n + 3$ is $O(n^2)$

$f(n) = 8n^2 + 10n + 3$
$g(n) = n^2$
$8n^2 + 10n + 3 \leq c * n^2$
$(c - 8)n^2 - 10n - 3 \geq 0$
Since we need c and n_0 to be positive constants and $n = \frac{10 \pm \sqrt{10^2 + 12(c-8)}}{2(c-8)}$,
pick $c = 9$, $n_0 = 11$

Express the worst-case running time of your findMissingNumber algorithm as a mathematical recurrence. State an asymptotic bound in big-O notation for this recurrence. Explain how you determined this bound.

```
if (numbers[startIndex] == numbers[endIndex]){
    return numbers[startIndex];
}
if(midIndex - startIndex <= 1 || endIndex - midIndex <= 1) {
    // Consider there are only two elements in the sequence.
    if (numbers.length == 2) {
        return (numbers[startIndex] + commonDifference);
    }
    // Consider the missing number is between number[midIndex] and number[midIndex + 1].
    else if (numbers[midIndex + 1] - numbers[midIndex] != commonDifference) {
        return (numbers[midIndex] + commonDifference);
    }
    // Consider the missing number is between number[midIndex - 1] and number[midIndex].
    else if (numbers[midIndex] - numbers[midIndex - 1] != commonDifference) {
        return (numbers[midIndex] - commonDifference);
    }
}
// Calculate the common difference from the first element to the middle element.
// If the calculation result is equal to the common difference, this means the missing
// element is on the other side. So, the start position updates to midIndex plus 1.
if ((numbers[midIndex] - numbers[0]) / midIndex == commonDifference) {
    return findMissingNumberHelper(numbers, startIndex: midIndex + 1, endIndex);
}
// Same as above. However, this time the missing element is on the left side.
// So, change the end position to midIndex minus 1.
return findMissingNumberHelper(numbers, startIndex, endIndex: midIndex - 1);
```

$O(1)$

"n" means the size of input.

$$T(\text{n}) = \begin{cases} O(1) & \text{if (n} \leq 3 \text{ or (numbers[n] – numbers[0] == 0))} \\ T(\text{n} / 2) & \text{if (n} > 3 \text{ and (numbers[n] – numbers[0] != 0))} \end{cases}$$

$$\begin{aligned} T(\text{n}) &= T(\tfrac{n}{2}) \\ &= T(\tfrac{n}{2^2}) \\ &= T(\tfrac{n}{2^3}) \\ &\cdots \end{aligned}$$

2, 4, 6, 8, 10, 12, … , n	
2, 4, 6, 8 … , $\frac{n}{2}$	1 call
2, 4, 6, … , $\frac{n}{2^2}$	2 calls
2, 4, … , $\frac{n}{2^3}$	3 calls

To find the missing number in an arithmetic sequence, we cut off half of the elements after each call. So, the total number of calls will be less than or equal to $\log_2 n$.
So, we can say $T(\text{n})$ is $O(\log_2 n)$.