1. Assume the mathematical function is F(n).
   Assume the numbers of operations of compareTo(), queue.offer(), queue.poll(), queue.size(), and queue.peek() are 1.
   The method of sortQueue is shown below:

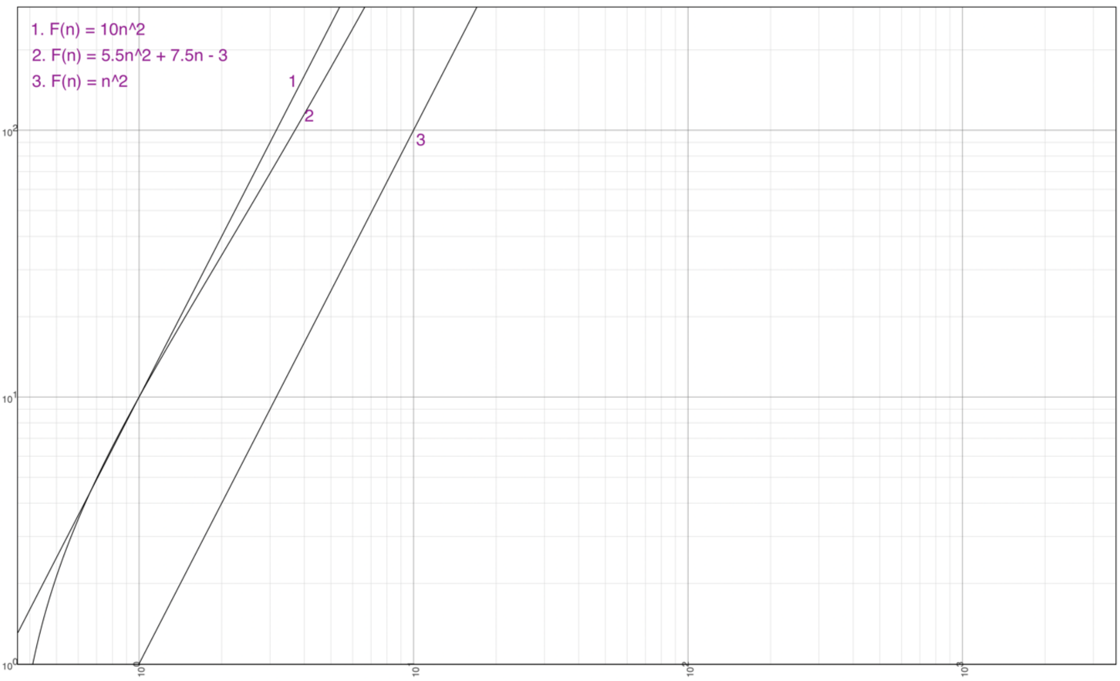
```java
public static <T extends Comparable<T>> void sortQueue(Queue<T> queue) {
    int size = queue.size();                  // 2 operations
    int counter = 0;                          // 1 operations

    //Total operations for the whole loop:
    //(6n − 2) + 7 ∗ (n + (n − 1) + … + 1) + 4 ∗ (2 + 3 + 4 + (n − 1))
    for (int k = 1; k < size; k++) {        // Assume the size is n, so we have 1 + n operations
        T max = queue.poll();               // 2 ∗ (n − 1) operations

        // Total operations for the first nested loop are :
        // 7 ∗ (n + (n − 1) + (n − 2) + ... + 1) + 1
        for (int i = 0; i < size − k; i++) {
            if (max.compareTo(queue.peek()) >= 0) {
                queue.offer(queue.poll());
            }else {
                T temp = max;
                max = queue.poll();
                queue.offer(temp);
            }
        }
        counter++;                          // (n − 1) operations
        queue.add(max);                     // (n − 1) operations


        // Total operations for the second nested loop are:
        // 4 ∗ (2 + 3 + 4 + ... + (n − 1)) + 1 operations
        for (int j = counter; j >= 2; j−−) {
            queue.offer(queue.poll());
        }
    }
}
```

   From the diagram above, when we add all operations, we can get(in worst case):
   F(n) = (6n - 2) + 7 * (n + (n – 1) + … + 1) + 4 * (2 + 3 + 4 + (n – 1)) + 3
   Therefore, F(n) = $5.5n^2 + 7.5n - 3$, and we pick n_0 = 1 and c = 10.
   When $n \geq 1$, we know $F(n) = 5n^2 + 7n - 3 \leq 10n^2$, so F(n) is O($10n^2$).
   The diagram is shown below:



   Hence, the runtime complexity of my sortQueue algorithm is O(n^2).

2. Assume the time complexity of the implementation is F(n)
   Assume the number of operation of numbers.length is 1.
   The implementation is shown below:

```java
public static int findMissingNumber(int[] numbers) {
    int commonDifference = (numbers[numbers.length - 1] - numbers[0]) / numbers.length;   //6 operations
    return helpFindMissingNumber(numbers, 0, numbers.length - 1, commonDifference);       // 1 operations
}

private static int helpFindMissingNumber(int[] numbers, int start, int end, int commonDifference) {
    if (end - start + 1 == 2) {                          // 3 operations
        return numbers[start] + commonDifference;        // 2 operations
    }

    int middle = (start + end) / 2;                      // 3 operations
    if ((numbers[middle] - numbers[start]) / (middle - start) != commonDifference) {
        return helpFindMissingNumber(numbers, start, middle, commonDifference);
    }else {
        return helpFindMissingNumber(numbers, middle, end, commonDifference);
    }
}
```

From the diagram above, we can easily know that the asymptotic bound in Big O for findMissingNumber is O(1).

From the method of helpFindMissingNumber, we know that this method is recursive method. Assume the time complexity of the helpFindMissingNumber method is T(n), so we have the recursive expression below:

$$T(n) = \begin{cases} O(1) & (n = 2) \\ T\left(\frac{n}{2}\right) + O(1) & (n \geq 3) \end{cases}$$

Therefore, $\mathrm{T(n)} = \mathrm{O}(1) + \mathrm{T}\left(\frac{\mathrm{n}}{2}\right) = \mathrm{O}(1) + \mathrm{O}(1) + \mathrm{T}\left(\frac{\mathrm{n}}{4}\right) = \cdots = \log_2 \mathrm{n} * \mathrm{O}(1)$

Hence, the asymptotic bound in Big O for helpFindMissingNumber is $\mathrm{O}(\log_2 \mathrm{n})$.
Hence, the time complexity of whole implementation F(n) is $\mathrm{O}(\log_2 \mathrm{n})$ + O(1), which is $\mathrm{O}(\log_2 \mathrm{n})$.