

# COMP3506 – Assignment 2

Kenton Lam

Due 23/08/2019 5:00 pm

## Question 2

### Analysis of sortQueue

The algorithm implemented is a variant of bubble sort (the algorithm won't be explained, see code comments). The runtime of this algorithm depends on the number of elements out of order. Also, note that bubble sort moves elements at different speeds, depending if they need to move up or down. Suppose we have a queue of  $n$  elements

$$A = [a_1, a_2, \dots, a_n].$$

We define  $k$ , the number of elements out of order, as below.  $\#\{\dots\}$  denotes the number of elements in the given set.

$$k = \#\{a_i \mid \exists i < j \leq n \text{ s.t. } a_j < a_i\}$$

For example,

$$A = [1, 2, 3, 4] \implies k = 0$$

$$A = [4, 1, 2, 3] \implies k = 1$$

$$A = [4, 3, 2, 1] \implies k = 4$$

Note that in bubble sort, exactly  $k$  iterations of 'bubbling' are needed to sort an array with  $k$  unordered elements.

With this in mind, we count the worst-case number of primitive operations, noted in the comments below. Assume  $i++$  is compiled to one primitive operation. Also assume queue methods are 1 primitive operation, as we are looking for an asymptotic limit. This could change depending on the implementation of queue used. We assume Java's *LinkedList*.

```

1 static <T extends Comparable<T>> void sortQueue(Queue<T> queue) {
2     // INIT section
3     int size = queue.size(); // 2: .size(), assignment
4     if (size <= 1)           // 1: comparison
5         return;              // 1: return
6     T a, b, prev;            // 0: compiler declarations
7     boolean repeat = true;   // 1: assignment
8
9     // WHILE section (executes k times)
10    while (repeat) {          // 1: comparison
11        repeat = false;      // 1: assignment
12
13        a = queue.remove();   // 2: .remove(), assignment
14        b = null;             // 1: assignment
15        prev = null;         // 1: assignment
16
17        // 1: initial assignment i = 0
18        // FOR section (executes n-1 times)
19        for (int i = 0; i < size - 1; i++) {
20            if (a == null) {   // 1: comparison
21                a = queue.remove(); // 2: .remove(), assignment
22            } else {
23                b = queue.remove(); // 2: .remove(), assignment
24            }
25
26            // 3: .compareTo(), comparison, assignment
27            T toPush = a.compareTo(b) < 0 ? a : b;
28            // 2: comparison, assignment
29            T other = toPush == a ? b : a;
30            // 1: .add()
31            queue.add(toPush);
32
33            if (a == toPush) { // 1: comparison
34                a = null;      // 1: assignment
35            } else {
36                b = null;      // 1: assignment
37            }
38            // on first run, prev = null so
39            // 1: comparison
40            // on subsequent runs, prev != null so
41            // 3: comparison, .compareTo(), comparison
42            if (prev != null && prev.compareTo(toPush) > 0) {
43                repeat = true; // 1: assignment
44            }
45            prev = toPush;     // 1: assignment
46        }
47        // 3: .add(), comparison
48        queue.add(a != null ? a : b);
49    }
50 }

```

Let  $T_i(n, k)$ ,  $T_w(n, k)$  and  $T_f(n, k)$  denote the primitive operations used in each iteration of the ‘INIT’, ‘WHILE’ and ‘FOR’ sections respectively. We seek  $T(n, k)$ , the total number of primitive operations given  $n$  and  $k$ . From here, we assume  $n \geq 2$  otherwise  $T(n, k) = 3$  trivially.

- Firstly,  $T_i(n, k) = 5$  because there are a constant number of operations.
- Then, working inside-out, we consider the inner for loop. Because we are looking for an upper bound, that the branch of the final if statement is taken every iteration. Then,  $T_f(n, k) = 16$ .
- The while loop contains 10 primitive operations and  $n - 1$  iterations of the for loop, so  $T_w(n, k) = 10 + (n - 1)T_f(n, k)$ .
- Finally, the function contains  $k$  iterations of the while loop, so

$$\begin{aligned}
 T(n, k) &= T_i(n, k) + kT_w(n, k) \\
 &= 5 + k(10 + (n - 1)16) \\
 &= 5 + 10k + k(n - 1)16 \\
 &= 5 - 6k + 16nk
 \end{aligned}$$

Because we are looking for the worst-case upper bound, we assume  $k = n$ . Then, we have  $T(n) = 5 - 6n + 16n^2$ . We claim this is  $O(n^2)$ . That is, there exists  $c$  and  $n_0$  such that

$$n \geq n_0 \implies T(n) \leq cn^2.$$

Assume  $n \geq 1$ . Then,

$$T(n) = 5 - 6n + 16n^2 \leq 5 + 16n^2 \leq 5n^2 + 16n^2 = 21n^2.$$

So  $n_0 = 1$  and  $c = 21$  fulfil this requirement and  $T(n) \in O(n^2)$ .

## Question 4

### Analysis of findMissingNumber