COMP3506 Homework 2

Daniel Nathan

March 13, 2021

The code for the implementation of sortQueue (with comments taken out) is shown below. The number of primitive operations required for each line is given, as is the expected number of times a loop will run. Note that the runtime for queue methods is calculated from the code for LinkedList to be 1 for queue.size(), 3 for queue.remove() and 8 for queue.add() (2, 4, and 9 respectively including an operation for the function call). compareTo is assumed to be one primitive operation.

```
1    public static <T extends Comparable<T>> void
2            sortQueue(Queue<T> queue) {
3        T selected;
4        // 1 initial operation (assignment of i)
5        // runs n - 1 times, with 5 operations each time (compare,
6        // increment, queue.size function call, subtract)
7        for (int i = 0; i < queue.size() - 1; i++) {
8            // 5 operations (1 for assignment, 4 for queue.remove)
9            selected = queue.remove();
10           // runs n - 1 times, with 4 operations each time
11           for (int j = 0; j < queue.size(); j++) {
12               // 5 operations (1 for assignment, 4 for queue.remove)
13               T front = queue.remove();
14               // compareTo method & comparison - 2
15               if (front.compareTo(selected) > 0) {
16                   queue.add(selected); // 9 operations
17                   selected = front; // 1 operations
18               } else {
19                   queue.add(front); // 9 operations
20               }
21           }
22           queue.add(selected); // 9 operations
23       }
24   }
```

Let n be the number of elements in the queue. In the worst case, the if statement will be true every time, resulting in $4 + 5 + 9 + 1 = 19$ elementary operations per loop of the inner loop. This loop has one assignment operation for j, and is then run $n - 1$ times, resulting in $19(n - 1) + 1 = 19n - 18$ operations. From this, every time the outer loop runs, it will take $5+5+(19n-18)+9 = 19n+1$ operations. Running this $n - 1$ times, plus once more to assign i results in a total operations required ($T(n)$) of

$$\begin{aligned} T(n) &= (19n + 1)(n - 1) + 1 \\ &= 19n^2 - 18n - 1 + 1 \\ &= 19n^2 - 18n \end{aligned}$$

Choosing $c = 19, n_0 = 1$, it can be shown that this run time is $O(n^2)$. For $n \geq 1$:

$$19n^2 - 18n \leq 19n^2$$

For findMissingNumber, the code is:

```
1    public static int findMissingNumber(int[] numbers) {
2        return fMNRecursion(numbers, 0, numbers.length);
3    }
```

This is a method call, and a return from a method (2 operations). for fMNRecursion:

```
1    private static int fMNRecursion(int[] numbers, int start, int end) {
2        if (end - start == 2) { // 2 operations (subtract, comparison)
3            // 6 operations (return, 2 indexing, 3 arithmetic)
4            return (numbers[start] + numbers[end - 1]) / 2;
5        }
6        // 6 operations (assignment, math.abs, 2 indexing, 2 arithmetic)
7        int difference1 = Math.abs(numbers[start + 1] - numbers[start]);
8        // 7 operations (assignment, math.abs, 2 indexing, 3 arithmetic)
9        int difference2 = Math.abs(numbers[end - 1] - numbers[end - 2]);
10       if (difference1 > difference2) { // 1 operation (comparison)
11           // 6 operations (return, 2 indexing, 3 arithmetic)
12           return (numbers[start + 1] + numbers[start]) / 2;
13       } else if (difference2 > difference1) { // 1 operation
14           // 7 operations (return, 2 indexing, 4 arithmetic)
15           return (numbers[end - 1] + numbers[end - 2]) / 2;
16       } else {
17           // 4 (method call, return, 2 arithmetic)
18           return fMNRecursion(numbers, start + 1, end - 1);
19       }
20   }
```

Let n be the number of elements in the array. The first if statement required 2 primitive operations. In the end $-$ start $==$ 2 base case, $2 + 6 = 8$ primitive operations are required. The two variable assignments take 13 elementary operations between them. If difference1 $>$ difference2, the total primitive operations is $2+13+1+6 = 22$. If, difference2 $>$ difference 1, $2+13+1+1+7 = 24$ total primitive operation are required. In the non-base case $2 + 13 + 1 + 1 + 4 = 21$ total primitive operations are required. The size (n) is also decreased by 2 in the non-base case. From this, a mathematical reccurence for this running time $(T(n))$ is:

$$T(n) = \begin{cases} 8, n = 2 \\ 22, \text{difference1} > \text{difference2} \\ 24, \text{difference2} > \text{difference1} \\ 22 + T(n-2), \text{non-base case} \end{cases}$$

This is $O(n)$, as a constant amount of operations are required for a constant decrease in size of n in the non-base case (resulting in a linear relationship between size and running time), and the base cases all run in constant time.

Note: everything on this page was done before I realised that I only had to give the recurrence and deduce the asymptotic efficiency from that (rather than getting a function for running time). I'm not deleting what I've already done, but my answer is complete without this page (so please don't mark me down for my answer being too long!)

This recurrence can also yield a function for runtime. An array of odd length in the worst case ($n = 3$ for base case, difference2 > difference1) will take

$$\begin{aligned} 24 + 22 \times \frac{n-3}{2} &= 22 + 11(n-3) \\ &= 11n - 11 \end{aligned}$$

operations, as each call other than the base case of n = 3 will remove 2 from the length of the array. For an array of even length, the worst case ($n = 2$)

$$\begin{aligned} 8 + 22 \times \frac{n-2}{2} &= 8 + 11(n-2) \\ &= 11n - 14 \end{aligned}$$

operations. Therefore the worst case is for an odd array length. Adding this to the 2 operations from the initial function call, the worst case run time is $11n - 9$. Choosing $c = 11, n_0 = 1$, it can be verified that this run time is $O(n)$. For $n \geq 1$:

$$11n - 9 \leq 11n$$