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 wont 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

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

We define k, the number of elements out of order, as below. $\#\{\ldots\}$ 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,

$$\begin{aligned} A = [1,2,3,4] &\implies k = 0 \\ A = [4,1,2,3] &\implies k = 1 \\ A = [4,3,2,1] &\implies k = 4 \end{aligned}$$

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*.

```java
static <T extends Comparable<T>> void sortQueue(Queue<T> queue) {
    // INIT section
    int size = queue.size(); // 2: .size(), assignment
    if (size <= 1)           // 1: comparison
        return;              // 1: return
    T a, b, prev;            // 0: compiler declarations
    boolean repeat = true;   // 1: assignment

    // WHILE section (executes k times)
    while (repeat) {         // 1: comparison
        repeat = false;      // 1: assignment

        a = queue.remove();  // 2: .remove(), assignment
        b = null;            // 1: assignment
        prev = null;         // 1: assignment

        // 1: initial assignment i = 0
        // FOR section (executes n-1 times)
        for (int i = 0; i < size - 1; i++) {
            if (a == null) {        // 1: comparison
                a = queue.remove(); // 2: .remove(), assignment
            } else {
                b = queue.remove(); // 2: .remove(), assignment
            }

            // 3: .compareTo(), comparison, assignment
            T toPush = a.compareTo(b) < 0 ? a : b;
            // 2: comparison, assignment
            T other = toPush == a ? b : a;
            // 1: .add()
            queue.add(toPush);

            if (a == toPush) { // 1: comparison
                a = null;      // 1: assignment
            } else {
                b = null;      // 1: assignment
            }
            // on first run, prev = null so
            //   1: comparison
            // on subsequent runs, prev != null so
            //   3: comparison, .comparTo(), comparison
            if (prev != null && prev.compareTo(toPush) > 0) {
                repeat = true; // 1: assignment
            }
            prev = toPush;     // 1: assignment
        }
        // 3: .add(), comparison
        queue.add(a != null ? a : b);
    }
}
```

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

It can be seen that the algorithm implemented performs a constant number of operations, then a recursive call on $n/2$ of the input size. Then if n is the input size, then the running time of findMissingNumber is

$$T(n) = T(n/2) + O(1).$$

Thus, the call tree looks like a sequence (a tree where nodes have at most one child) of length $\lceil \log_2 n \rceil$. Each function call takes $O(1)$ operations, so the total worst case runtime is

$$T(n) = O(1)\lceil \log_2 n \rceil \in O(\log_2 n).$$