THE UNIVERSITY OF
SYDNEY

Assignment 1

COMP2123: Data Structures and Algorithms
510415022

Problem 1. (20 points)

We can upperbound the running time of the algorithm by counting the total number of steps the algorithm requires to run. We begin with the inner-most `for`-loop on lines `6-7`. The loop runs for a total of k iterations undertaking a sum and assignment operation each time. Both of these run in $O(1)$ and hence total steps $= 2k \approx k$.

Now, each k-step iteration taking place in the second `for`-loop occurs $n-k$ times due to the outer `for`-loop starting on line `4`. There are also three assignment operations on lines `2`, `3` and `5` which run in constant time. Hence, total steps $= k(n-k)+3 \approx k(n-k)$.

Therefore our upperbound of the running time for this algorithm in terms of n and k is $O(k(n-k))$. Note that if we use the assumption of $k < n/2$, we can also find an expression only in terms of n:

$$\begin{aligned}
\text{Total steps} &= k(n-k) && (1)\\
&< \frac{n}{2}\left[n - \frac{n}{2}\right] && (2)\\
&= \frac{n}{2} \cdot \frac{n}{2} && (3)\\
&= \frac{n^2}{4} && (4)\\
\implies \text{Run time} &= O(n^2) && (5)
\end{aligned}$$

Problem 2. (40 points)

a) Recall from the lectures that a queue is an ADT which supports `enqueue(e)`, `dequeue()`, `first()`, `size()`, and `isEmpty()`.

We wish to add a new `seeSaw()` operation to the queue which returns the sum of integers in even positions and reciprocals of integers in odd positions in the queue, and does this in $O(1)$ time. To do so, we will simply maintain two new variables in our data structure, `sum1` and `sum2`, where for the queue holding elements $Q_0, Q_1, \ldots, Q_{n-1}$,

$$\texttt{sum1} = Q_0 + \frac{1}{Q_1} + Q2 + \frac{1}{Q_3} + \ldots \tag{6}$$

$$\texttt{sum2} = \frac{1}{Q_0} + Q1 + \frac{1}{Q_2} + Q_3 + \ldots \tag{7}$$

These two sums will need to be toggled between each time `enqueue()` or `dequeue()` are called as:

- On any two successive `enqueue()`'s, the data structure will need to swap between adding e or $1/e$ to each list, otherwise `sum1` would be the sum of all integers and `sum2` the sum of all reciprocals.
- The `dequeue()` operation flips each element from an odd position to an even position in the queue and vice-versa. This means that, for example, an element which previously was outputted in `seeSaw()` as an integer will now need to be outputted as its reciprocal. Hence we need two sums as we will be computing and storing two values: Q and $1/Q$.

Note that we will be implementing the data structure using a linked list rather than an array, with one pointer at the head and one at the tail. This will allow us to perform both `enqueue(e)` and `dequeue()` in constant time as there is no need for expansion of the array once the queue is full, or for shifting of elements in the array each time a user calls `dequeue()`.

Below is some pseudocode describing the above data structure.

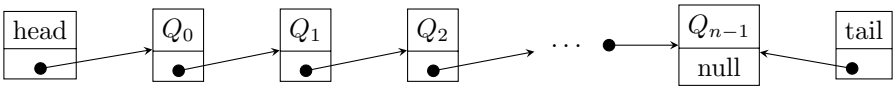

Figure 1: The node structure of the elements in our queue data structure

```
1: class NODE(int e):
2:     int value = e
3:     Node next = null
```

```
1: class Queue:
2:     Node head
3:     Node tail
4:     int size
5:     double sum0
6:     double sum1
7:     int whichSumEnqueue = 0          ▷ toggle variable for sum0 or sum1 ∈ [0, 1]
8:     int whichSumDequeue = 0
```

```
1: function newEnqueue(e):
2:     n ← new node with value e, pointing to null
3:     if queue.size = 0 then
4:         queue.head = n
5:         queue.tail = n
6:     else
7:         queue.tail.next = n
8:         queue.tail = n
9:
10:    if queue.whichSumEnqueue = 0 then
11:        queue.sum0 += e
12:        queue.sum1 += 1/e
13:    else
14:        queue.sum0 += 1/e
15:        queue.sum1 += e
16:    queue.whichSumEnqueue = 1−queue.whichSumEnqueue
17:    queue.size += 1
```

```
1:  function newDequeue:
2:      if queue.isEmpty() then
3:          return error
4:
5:      n ← queue.head
6:      queue.head = n.next                ▷ n is freed from memory and so is "deleted"
7:
8:      if queue.whichSumDequeue = 0 then
9:          queue.sum0 −= e
10:         queue.sum1 −= 1/e
11:     else
12:         queue.sum0 −= 1/e
13:         queue.sum1 −= e
14:     queue.whichSumDequeue = 1− queue.whichSumDequeue
15:     queue.size −= 1
16:     return n
```

```
1:  function seeSaw:
2:      if queue.whichSumDequeue = 0 then
3:          return queue.sum0
4:      else
5:          return queue.sum1
```

b) Let sum1_i be the value of `sum1` at an arbitrary i-th state of the queue. Let sum2_i be defined similarly.

$$\text{sum1}_i = Q_0 + \frac{1}{Q_1} + Q2 + \frac{1}{Q_3} + \cdots + Q_n \tag{8}$$

$$\text{sum2}_i = \frac{1}{Q_0} + Q1 + \frac{1}{Q_2} + Q_3 + \cdots + \frac{1}{Q_n} \tag{9}$$

i.e. we assume that at a certain point in time, these two sums have been correctly maintained. We now show that the data structure and operations described above affect these sums correctly for the `enqueue(e)` and `dequeue()` operations.

Enqueue

When we add an element v to the queue, we ensure to always add v and $1/v$ to the sums which last added a reciprocal and an integer respectively. This is done by using a boolean operator which switches each time the function is called (pseudocode of this can be seen in lines 10 to 16 of the `newEnqueue(e)` function above). Hence, when we enqueue Q_{n+1}, our sums will look like:

$$\text{sum1}_{i+1} = Q_0 + \frac{1}{Q_1} + Q2 + \frac{1}{Q_3} + \cdots + Q_n + \frac{1}{Q_{n+1}} \tag{10}$$

$$\text{sum2}_{i+1} = \frac{1}{Q_0} + Q1 + \frac{1}{Q_2} + Q_3 + \cdots + \frac{1}{Q_n} + Q_{n+1} \tag{11}$$

which is correct. Note that the dequeue boolean which the `seeSaw()` operation is based on has not been touched, so calling the `seeSaw()` function will still return the correct sum i.e. if calling `seeSaw()` at state i would have returned `sum1`, then it would do the same at state $i+1$.

Dequeue

When we dequeue, we first subtract the value v or $1/v$ from each sum, depending on the state of the boolean operator. Note that having a separate boolean operator here is necessary as it is tracking which sum is currently "active" which is solely based on the number of times each element has been flipped from odd to even or vice versa. Therefore our sums after dequeueing look like:

$$\text{sum1}_{i+1} = \frac{1}{Q_1} + Q2 + \frac{1}{Q_3} + \cdots + Q_n + \frac{1}{Q_{n+1}} \tag{12}$$

$$\text{sum2}_{i+2} = Q1 + \frac{1}{Q_2} + Q_3 + \cdots + \frac{1}{Q_n} + Q_{n+1} \tag{13}$$

The `seeSaw()` operation has now also automatically been updated to return the alternate sum to as it would prior to the dequeue, which is correct.

Hence we can generalise this to prove that regardless of the current state of the queue, the `seeSaw()` operation will always return the correct value, which is based on the correct maintenance of the two sums `sum1` and `sum2`.

c) The reason we choose to use a linked list to implement the queue is to improve the running time and space complexity of our data structure and operations. In an array-based implementation, a queue would take up size N bytes of memory regardless of the number of elements stored in it. Now, if the queue was full and a user attempted to `enqueue()` a new element, we would need to create a new, larger array of size $2N$ and then copy each old element over to the new array, a process which would take $O(n)$ time. We don't have this issue with a linked list implementation, as linked lists do not require a single contiguous block of memory to be stored in. Hence the space complexity of our queue is $O(n)$ where n is the number of nodes currently in the queue.

Furthermore, when we run `dequeue()`, an array-based implementation would call for index 0 to be removed and then all remaining elements to be shifted over to the left by 1. This process, which also runs in $O(n)$, is saved by using linked lists as we simply have to point the head pointer of the list to the second node, a reassignment which takes constant time.

All lines in the functions `newEnqueue(e)` and `newDequeue()` are either assignments or trivial mathematical operations such as addition and subtraction, both of which run in constant

time. The space complexity is still $O(n)$ as we are only storing two additional sum variables in the data structure.

Overall, the data structure and its operations have space and time complexity of $O(n)$.

Problem 3. (40 points)

a) A stack is an ADT which supports the following operations: `push(e)`, `pop()`, `top()`, `size()` and `isEmpty()`. We wish to design a stack which, in addition to these operations, is also able to return the maximum number of consecutive elements in the stack in $O(1)$ time, using just $O(n)$ space.

This data structure can be implemented using a linked list-implementation of a stack, where each node in the stack stores 4 variables:

1. The colour pushed onto the stack.
2. The local maximum number of consecutive elements i.e. for the current node with colour c, how many previous consecutive elements also had the colour c?
3. The global maximum number of consecutive elements i.e. the output of `MaxMono()` — what is the maximum number of consecutive elements in the stack up to the current node?
4. The next node in the stack.

Whenever a colour c is to be pushed to the stack on node n, we will first compare it with the colour c' contained in the node n' previously on top of the stack. If $c = c'$ then the local max of n will be the local max of n' incremented by 1. We will then compare the local max variable to the global max variable contained in n'. If the new local max is greater than the global max variable in n', then the new local max will become both the local and global max at n.

Some pseudocode will no doubt help explain this algorithm[1]:

```
1:  function PUSH(c):
2:      n ← new node storing colour c
3:      n.next = stack.top;
4:      if size = 0 then                          ▷ base case for the stack
5:          n.localMax = 1
6:          n.globalMax = 1
7:      else
8:          if c ≠ stack.top.colour then
9:              n.localMax = 1
10:         else
11:             n.localMax = stack.top.localMax + 1
12:         n.globalMax = max(stack.top.globalMax, n.localMax)
13:     stack.top = n
14:     stack.size = stack.size + 1
```

[1] A full Java implementation of this code can be found in the appendix.

```
1: function POP:
2:     if stack.isEmpty() then
3:         return error
4:
5:     n ← stack.top
6:     stack.top = stack.top.next
7:     stack.size = stack.size - 1
8:
9:     return n
```

```
1: function MAXMONO:
2:     return stack.top.globalMax
```

b) Because the values in each new node on the stack are dependent on the previous node's values, most sly edge cases are taken care of in this algorithm, including when a user pop's several values off the stack and then push's the same colour as the current `top` onto the stack. Other edge cases such as when the stack is empty are taken care of by setting the node's local and global maxima to 1.

c) For similar reasons as described in Problem 2 c), here we used a linked list implementation of a stack. This allows our data structure to have a space complexity not based on N, the total size of an array. Instead, our space complexity is simply $O(4n) \approx O(n)$ where n is the number of nodes in the stack. This is because for each node we are storing exactly 4 values which is still linear in space.

The time complexity of all operations in the stack are $O(1)$. This is because each line in the algorithm is either an assignment or simple mathematical operations (including the `max()` operation) which run in $O(1)$ time.

A Implementation of Problem 3 in Java

```
enum Colour {
    ERROR, RED, GREEN, BLUE
}

class Node {
    public Colour colour;
    public int localMax;
    public int globalMax;
    public Node next;
}

public class Stack {

    Node top;
    int size;

    Stack() {
        this.top = null;
        this.size = 0;
    }

    public void push(Colour colour) {
        Node temp = new Node();
        temp.colour = colour;
        temp.next = this.top;

        if (size == 0) {
            temp.localMax = 1;
            temp.globalMax = 1;
        } else {
            if (colour != this.top.colour) {
                temp.localMax = 1;
            } else {
                temp.localMax = this.top.localMax + 1;
            }
            temp.globalMax = Math.max(this.top.globalMax, temp.localMax);
        }

        this.top = temp;
        this.size++;

        return;
    }

    public Colour pop() {
        if (this.isEmpty()) {
            System.out.println("Stack is empty!");
            return Colour.ERROR;
        }

        Node oldTop = this.top;
```

```
52          this.top = this.top.next;
53          this.size--;
54
55          return oldTop.colour;
56      }
57
58      public boolean isEmpty() {
59          return top == null;
60      }
61
62      public int size() {
63          return this.size;
64      }
65
66      public Colour top() {
67          if (this.isEmpty()) {
68              System.out.println("Stack is empty!");
69              return Colour.ERROR;
70          }
71
72          return this.top.colour;
73      }
74
75      public void display() {
76          if (this.isEmpty()) {
77              System.out.println("Stack is empty!");
78              return;
79          }
80
81          Node temp = top;
82          while (temp != null) {
83              System.out.println("(" + temp.colour + "," + temp.localMax + "," +
    temp.globalMax + ")");
84              temp = temp.next;
85          }
86      }
87
88      public int maxMono() {
89          return this.top.globalMax;
90      }
91  }
```

Listing 1: An implementation of Problem 3 in Java