Programming and Problem Solving
Assignment # 2
PART 1

SUBMITTED BY
Name: Manan Dineshbhai Paruthi
Student ID: 40192620

Pseudo Code - Question 1 - Part I

```
INPUT arr
INITIALIZE repeatingEle = arr[0] , repeatingEleStartIndex = 0 , repeatingEleCount = 1

FOR i = 1 to arr.length ; INCREMENT by 1
        IF arr[i] == repeatingEle  THEN
                repeatingEleCount++
        ELSE
                IF repeatingEleCount > 1 THEN
                        OUTPUT "Value " + repeatingEle + " is repeated " + repeatingEleCount +
                                "times starting at index " + repeatingEleStartIndex
                END IF
                repeatingEle = arr[i];
                repeatingEleStartIndex = i;
                repeatingEleCount = 1;
        END IF
END FOR
IF repeatingEleCount > 1 THEN
        OUTPUT "Value " + repeatingEle + " is repeated " + repeatingEleCount +
                "times starting at index " + repeatingEleStartIndex
END IF
```

a) **Motive behind design** : Check the current value with the previous value and increment the counter if it matches otherwise print the output and reset the previous value
b) **Big-O complexity** : O(n) - looping through the array only once by storing previous element value and comparing the current element
c) **Big-Ω complexity** : Best case scenario is that array is empty then complexity will be O(1) otherwise if elements is non empty then it will be O(n)
d) **Big-O space complexity :** O(1)

Pseudo Code - Question 1 - Part II

```
INPUT arr
INITIALIZE repeatingEle = q.dequeue() , repeatingEleStartIndex = 0 , repeatingEleCount = 1

FOR i = 1 to q.getCapacity() ; INCREMENT by 1
        INITIALIZE currEle = q.dequeue()
        IF currEle == repeatingEle  THEN
```

```
            repeatingEleCount++
        ELSE
            IF repeatingEleCount > 1 THEN
                OUTPUT "Value " + repeatingEle + " is repeated " + repeatingEleCount +
                    "times starting at index " + repeatingEleStartIndex
            END IF
            repeatingEle = currEle;
            repeatingEleStartIndex = i;
            repeatingEleCount = 1;
        END IF
    END FOR
    IF repeatingEleCount > 1 THEN
        OUTPUT "Value " + repeatingEle + " is repeated " + repeatingEleCount +
            "times starting at index " + repeatingEleStartIndex
    END IF
```

a) **Big-O complexity** : O(n) - dequeuing all the elements from the queue once by storing previous element value and comparing the current element
b) **Big-Ω complexity** : Best case scenario is that queue is empty then complexity will be O(1) otherwise if elements is non empty then it will be O(n)
c) **Big-O space complexity of the utilized queue** : all the elements are directly enqueued in the queue and then dequeued for calculation so it will be O(n)

Pseudo Code - Question 2 - Case I

```
// Stack Class
INITIALIZE startIndex, endIndex, top

Constructor with 3 params arr, startIndex, endIndex
    top = startIndex-1

Push method takes 1 param - value to be inserted
    IF top == endIndex THEN
        OUTPUT “Stack is Full”
    ELSE
        top += 1;
        arr[top] = val;
    END IF

Pop method with no param - returns popped value
    IF top == startIndex-1 THEN
        OUTPUT “Stack is Empty”
        RETURN -1
    ELSE
        removedVal = stack[top]
        stack[top] = -1
        top -= 1
        RETURN removedVal
    END IF

Is Full method
```

```
        IF top == endIndex THEN
            RETURN true
        ELSE
            RETURN false
        END IF

    Is Empty method
        IF top == startIndex-1 THEN
            RETURN true
        ELSE
            RETURN false
        END IF

    Size method
        RETURN top - startIndex

    // Make two objects of stack class
    Stack s1 = new Stack(arr, 0, arr.length/2-1);
    Stack s2 = new Stack(arr, arr.length/2, arr.length-1);
```

a) **Describe algorithm** : divided the stack into two parts - one stack starts from start to middle and other starts from middle to end
b) **Big-O Complexity for all methods** : for push, pop, isEmpty and isFull it is O(1) - as traversing is not required in any method only variable checking is required and fetching value based on index from array
c) **Big-Ω complexity for all methods** : for best and worst case both it will be O(1)

Pseudo Code - Question 2 - Case II

```
    INITIALIZE topOfStack1 = -1 , topOfStack2 = arr.length

    pushToStack1 method takes 1 param - value to be inserted
        IF topOfStack1+1 < topOfStack2 THEN
            topOfStack1 += 1
            arr[topOfStack1] = val
        ELSE
            OUTPUT “Stack is Full”
        END IF

    pushToStack2 method takes 1 param - value to be inserted
        IF topOfStack2-1 > topOfStack1 THEN
            topOfStack2 -= 1
            arr[topOfStack2] = val
        ELSE
            OUTPUT “Stack is Full”
        END IF

    popFromStack1 method with no param - returns popped value
        IF topOfStack1 > -1 THEN
            poppedVal = arr[topOfStack1]
            arr[topOfStack1] = -1
```

```
            topOfStack1 -= 1
            RETURN poppedVal
        ELSE
            RETURN -1
        END IF

popFromStack2 method with no param - returns popped value
        IF topOfStack2 < arr.length THEN
            poppedVal = arr[topOfStack2]
            arr[topOfStack2] = -1
            topOfStack2 += 1
            RETURN poppedVal
        ELSE
            RETURN -1
        END IF

Is Full method - for both stack
        IF topOfStack1+1 >= topOfStack2 THEN
            RETURN true
        ELSE
            RETURN false
        END IF


Is Empty method - Stack 1
        IF topOfStack1 == -1 THEN
            RETURN true
        ELSE
            RETURN false
        END IF

Is Empty method - Stack 2
        IF topOfStack2 == arr.length THEN
            RETURN true
        ELSE
            RETURN false
        END IF

Size method - Stack 1
        RETURN topOfStack1 + 1

Size method - Stack 2
        RETURN endIndex - topOfStack2 + 1
```

a) **Describe algorithm** : divided the stack into two parts - one stack starts from start to middle and other starts from end to middle in order to use maximum possible space
b) **Big-O Complexity for all methods** : for push, pop, isEmpty and isFull it is O(1) - as traversing is not required in any method only variable checking is required and fetching value based on index from array
c) **Big-Ω complexity for all methods** : for best and worst case both it will be O(1)
d) Implementing 3 stacks is not possible in this scenario as with 3 stacks we'll not be able to utilize maximum possible space

e) **Pseudo Code - Question 3**

```
// push method - takes 1 param - value to be inserted
    IF top == size-1 THEN
        OUTPUT "Stack is full"
    ELSE
        IF top == 0 || val >  maxTrackStack[top-1] THEN
            maxTrackStack[top] = val
        ELSE
            maxTrackStack[top] =  maxTrackStack[top-1]
    END IF

// pop method - with no param - returns popped value
    IF top == -1 THEN
        OUTPUT "Stack is Empty"
        RETURN -1
    ELSE
        removedVal = stack[top]
        stack[top] = -1
        top -= 1
        RETURN removedVal
    END IF


// max method - returns max value in the stack
    IF top > -1 THEN
        RETURN maxTrackStack[top]
    ELSE
        RETURN -1
    END IF
```

a) **Big-O complexity** : O(1) - for push, pop and max methods as we have made a diff stack to track the max value

Pseudo Code - Question 4

Answer :

It is not possible to have a single tree with the given preorder and postorder traversal

preorder ==> EKDMJGIACFHBL

postorder ==> DJIGAMKFLBHCE

'GIA' preorder is not matching 'IGA' postorder => to match postorder should be 'IAG'

&

'HBL' preorder is not matching 'LBH' postorder => to match postorder should be 'BLH'

Algorithm behind making a tree from preorder and postorder is

1) The 1st element of preorder should be same as last element of postorder => match root
2) Find the matching value of 2nd element of preorder in the postorder
3) The length of left child will be from the distance from the 1st element in postorder till the matching element with 2nd element of preorder
4) Split into left and right child and recursively call the method to find out the invalid element