Matthew Jednak
40133030
COMP 352
Assignment 2
November 4th 2021

# Written questions

## Question 1

I)
Input: an array of elements a
Output: displays the start indices where values start to repeat, as well as the values of these elements
Algorithm repeatCount(a):

```
    Int index <- 0
    Int nbTimes <- 1
    Create prev variable of same type as the array elements

    For(int i <- 0; i < a.length; i++)
        If i == 0
            then prev <- a[i]
        else
            if prev == a[i] then
                nbTimes <- nbTimes + 1
                if index == 0 then
                    index = I – 1
            else
                if index != 0 then
                    print value and number of times it was repeated
                    with starting index
                    index <- 0
                    nbTimes <- 1
                prev <- a[i]

    if index != 0 then
        print value and number of times it was repeated with starting index
```

A) I chose to do it this way because I simply had to use a single for loop to achieve the desired result

B) The Big-O time complexity for this would be O(n) because there is only 1 loop and we are iterating until the end of the array of length n

C) The Big-Ω time complexity would be Ω-(n) for the same reason as the Big-O time complexity. There is a single loop and at best we still need to iterate through the entire array to find the repeating elements.

D) The space complexity is just O(n) because the only thing used is the original array and a few variables.

II)
Input: an array of elements a
Output: displays the start indices where values start to repeat, as well as the values of these elements
Algorithm repeatCountStack(a):

```
    Create values stack S
    Create indexes stack Idx
    int tempVal <- 0
    int tempIndex <- 0
    int size <- 0
    boolean repeat <- false
    nbTimes <- 1

    For(int i <- 0; i < a.length; i++)
        If values.isEmpty then
            values.push(a[i])
        else
            if values.peek() == a[i] then
                values.push(a[i])
                if !repeat then
                    indexes.push([i-1])
                    repeat <- true
            else
                if !repeat then
                    values.pop()
                values.push(a[i])
                repeat <- false
```

```
size <- values.size()
while !values.isEmpty()
    if size == values.size then
        tempVal <- values.pop
    else
        if values.peek() == tempVal then
            nbTimes <- nbTimes + 1
            values.pop()

        else
            tempIndex <- indexes.pop()
            **print value and number of times it was repeated with
            starting index**
            tempVal <- values.pop()
            nbTimes <- 1

tempIndex <- indexes.pop()
**print value and number of times it was repeated with starting index**
```

A) I basically needed one stack to keep track of repeating values and another stack to keep track of at what index those repeating values began.

B) The Big-O time complexity for this would be O(n) even though there is a for loop and a while loop, they only run one after the other with the for loop having a O(n) runtime and the while loop also having a O(n) runtime. This gives us our overall runtime of O(n)

C) The Big-Ω time complexity would be Ω(n). This is the case because even if we don’t have any repeating values, we will still at the very least loop through the entire array of length n.

D) The space complexity is just O(n) because we use the original array, a few variables and 2 stacks of single elements.

## Question 2

I)
Input: an array of integers a, the value of the mod we are searching for x
Output: displays the indexes and values of pairs whose modulo gives x
Algorithm modFinder(a. x):

```
    Create values queue modValues
    Create indexes queue modIndexes
    tempIdx1, tempIdx2 <- 0
    tempVal1, tempVal2 <- 0

    For(int i <- 0; i < a.length; i++)
        For(int j <- 0; j < a.length; j++)
            If a[i] % a[j] == x
                modValues.add(a[i])
                modValues.add(a[j])
                modIndexes.add(i)
                modIndexes.add(j)

    while(!modValues.isEmpty())
        tempIdx1 <- modIndexes.remove();
        tempIdx2 <- modIndexes.remove();
        tempVal1 <- modValues.remove();
        tempVal2 <- modValues.remove();
        print indexes and value pair that give the desired modulo result
```

II) I basically have 2 queues being used in parallel so that if we find a pair of values that give us the result we want, we simply have to push both indexes to the index queue and both values to the value queue. Similarly when it comes time to display it all, we simply have to dequeue 2 at a time from each queue to get the values required to display each pair properly

III) The Big-O time complexity here is $O(n^2)$ because we have a loop inside another loop and both of them are iterating n-1 times where n is the length of the array passed into the function. In terms of the space complexity, it is $O(n)$ because the array is still single dimensional and both queues have only single values where the size of each queue cannot exceed 2n

IV) The Big-Ω time complexity would be $\Omega(n^2)$ because again even if we find no pairs we still need to loop through the entire array in both the inner and outer loops. The space complexity is also $\Omega(n)$ because the array is again still a 1D array and the queues can only have 2n values at most.

## Question 3

We must keep this in mind
$O(1) \subset O(\log n) \subset O(n^{½}) \subset O(n) \subset O(n \log n) \subset O(n^3) \subset O(n^3) \subset \dots \subset O(2^n) \subset \dots O(n^n)$

I) O(g(n))
This is the case because $\log^3 n$ < sqrt(n) for every value of n >0

II) Ω(g(n))
n * sqrt(n) + logn > $\log n^2$ for every n > 0

III) Ω(g(n))
n > $\log^2 n$ for every value of n > 0.3

IV)Ω(g(n))
Sqrt(n) > $2^{\text{sqrt(logn)}}$ for every value of n > 2.5

V) Ω(g(n))
n! increases much faster than n does and so $2^{n!} > 3^n$

VI) O(g(n))
$n^n$ will grow exponentially faster than 2 to the power of any exponent

VII) O(g(n))
$(n^n)^5 = n^{5n}$
If we look at just the exponents, we have 5n and $n^5$
$n^5 > 5n$ for every value of n > 2