Andrew Ha – 40088148
Nora Houari
COMP 352
Assignment 2 - Theory

Question 1:

Pseudo Code:

```
Algorithm findMultiplesOfX(A, x)
    Input: array A of integers
           integer value x to compare to see if it is a multiple

    Output: res, a two dimensional array containing indexes and value being stored

    //Initialize Variables

        //Initialize array
        res ← new Array[A.length()]

        //Looping in and finding multiples

        for i ← 0 to A.length() - 1 do

            //If the value is multiple of X then add the number and index, otherwise keep
            //it null

            if A[i] % x = 0 then

                //If the value is a multiple then store it the number and index

                res[i] ← A[i] //Storing number

        end for //end for loop


        return res
```

****How to use to returned array****

```
//Lets assume that you have received the array 'res' back from your method

        for i ← 0 to res.length() - 1 do
        if res[i] !=  null then
                //If the value is null, then it is not a multiple, if it is then
                System.out.println("Indexes are:  " +  i + "Values are: res[i] " );
    end for //end for loop
```

1A) The motive behind my design is that first of all, I understand that I will have to return two values back in the array, you could use the index of the array instead of making it two dimension. Which makes my Big-O Complexity O(n). So in order to reduce my complexity, we are going to assume that res[i] represents the value, and its index in the array represents its actual index. I have decided to purposely leave null values inside the array (which is another bad practice and comparing to **null**), this is because I would need to resize my array. I could've also calculated the multiples before, and declared an array of that size, but that would mean I would need to go through the loop twice.

1B) The Big-O Complexity of my solution is O(n). I have obtained this because in my algorithm, I only have one loop that goes to the length of '**n**' (the array). I do have a one dimensional array. Assuming that the indexes are the same between the input array and my array can help reduce space complexity,

1C) The Big-Ω of my solution is still O(n). This is because the 'best case' of my algorithm will still go to the length of '**n**' no matter what. Therefore, my Big-Ω of my solution will also be Ω(n).

1D) The Big-O of my space complexity is O(1). This is because I only have an instantiation of an array that will not grow in size.

Queue Implementation Pseudo Code:

Pseudo Code:

Algorithm findMultiplesOfX(A, x)

Input: array A of integers
integer value x to compare to see if it is a multiple

Output: res, Queue containing indexes and value being stored. They are to be stored in two consecutive pairs. Example: res.add(value); res.add(index)

//Initialize Variables

//Initialize queue
res ← new Queue()

//Looping in and checking for multiples

for i ← 0 **to** A.length() - 1 **do**

//If the value is multiple of X then add the number and index to the queue
if A[i] % x = 0 **then**

res.enqueue(i)
res.enqueue(A[i]) // **end if**

end for //end for loop

return res

****How to use to returned array****

//Lets assume that you have received the queue 'res' back from your method

While res.isEmpty = false do

if A[i][0] != null **then**

//If the value is null, then it is not a multiple, if it is then

System.out.println("Indexes are: " + res.dequeue() + "Values are: " + res.dequeue();

end while //end while loop

2A) The motive behind my design, is that a Queue is dynamic and is able to be resized. Since it is resizable we will store both the index and value and just dequeue them both. As long as it is well documented and is known that the queue is storing the data in this format, then there should be no issues storing this.

2B) The Big-O Complexity of my algorithm is still O(n), since in the algorithm I am only looping at the length of ‘n’ once. Therefore, the Big-O of my algorithm is O(n)

2C) Just like the Big Ω for the previous algorithm above. The Big-Ω Complexity of my algorithm is still Ω (n), since my algorithm still needs to run through the entirety of the array ‘n’ once no matter what.

2D) The Big-O space complexity of my algorithm is O(n), this is because I am just going to be continually adding elements to the queue.

Question 2:

2A)

for i=0 to n-1 do	n-1
Res[i]=0	1
end for	
for i=0 to n-2 do	n-2
for j=i+1 to n-1 do	n-1
if A[i]≤A[j] then	1
Res [j]= Res [j]+1	
else	
Res[i]= Res[i]+1	
end if	
end for	
end for	
for i=0 to n-1 do	n-1
B[Res [i]]= A[i]	1
end for	
Return B	

Time Complexity:

= n-1 + (n-2)(n-2) + n-1
= (2n – 2) + (n^2 – 4n + 4)
= (n^2 – 2n + 2)

Big – O Notation = O(n^2), since n^2 is the biggest term in the calculation

Big – Omega Notation = Ω(n^2) since the best case, the algorithm will still run in a double loop

2B) A = (88, 12, 94, 17, 2, 36, 69), N = 7

Comparison	i	j	Res
88 > 12	0	1	[1, 0, 0, 0, 0, 0, 0]
88 < 94	0	2	[1, 0, 1, 0, 0, 0, 0]
88 > 17	0	3	[2, 0, 1, 0, 0, 0, 0]
88 > 2	0	4	[3, 0, 1, 0, 0, 0, 0]
88 > 36	0	5	[4, 0, 1, 0, 0, 0, 0]
88 > 69	0	6	[5, 0, 1, 0, 0, 0, 0]
12 < 94	1	2	[5, 0, 2, 0, 0, 0, 0]
12 < 17	1	3	[5, 0, 2, 1, 0, 0, 0]
12 > 2	1	4	[5, 1, 2, 1, 0, 0, 0]
12 < 36	1	5	[5, 1, 2, 1, 0, 1, 0]
12 < 69	1	6	[5, 1, 2, 1, 0, 1, 1]
94 > 17	2	3	[5, 1, 3, 1, 0, 1, 1]
94 > 2	2	4	[5, 1, 4, 1, 0, 1, 1]
94 > 36	2	5	[5, 1, 5, 1, 0, 1, 1]
94 > 69	2	6	[5, 1, 6, 1, 0, 1, 1]
17 > 2	3	4	[5, 1, 6, 2, 0, 1, 1]
17 < 36	3	5	[5, 1, 6, 2, 0, 2, 1]
17 < 69	3	6	[5, 1, 6, 2, 0, 2, 2]
2 < 36	4	5	[5, 1, 6, 2, 0, 3, 2]
2 < 69	4	6	[5, 1, 6, 2, 0, 3, 3]
36 < 69	5	6	[5, 1, 6, 2, 0, 3, 4]

Now our final loop

i	b
Before Loop	[0, 0, 0, 0, 0, 0, 0]
0	[0, 0, 0, 0, 0, 88, 0]
1	[0, 12, 0, 0, 0, 88, 0]
2	[0, 12, 0, 0, 0, 88, 94]
3	[0, 12, 17, 0, 0, 88, 94]
4	[2, 12, 17, 0, 0, 88, 94]
5	[2, 12, 17, 36, 0, 88, 94]
6	[2, 12, 17, 36, 69, 88, 94]

2C) What does algorithm does it that it first creates an array of the same length as the array that is passed in. Afterwards, the all values of the newly created array will be set to 0. This array acts as a ‘counter’. So, we have a nested loop afterwards, the outer one starts at the first element to the second last. The nested starts from the second to the last. The purpose of these loops is that they are comparing the numbers to each other in the loop. Depending on which number is bigger, the ‘counter’ array will then get its value incremented at the same index for which the number is bigger. For example, if array[3] was bigger than array[5]. Then counter[3] gets its value incremented by 1. Essentially, the

purpose of incrementing the counter is that it represents the new index of the numbers from A. For example the biggest number is incremented the most during the comparisons, and subsequently will get the last index in the array. These indexes represent the numbers new positions from smallest to biggest.

So in conclusion what this Algorithm does, is that it takes an array, and will resort it to be smallest to biggest.

2D)

for i=0 to n-1 do	N – 1
Res[i]=0	
end for	
for i=0 to n-1 do	N – 1
Res[A[i]] = Res[A[i]] + 1;	
end for	
for i=0 to n-1 do	N – 1
Res[A[i]] += A[I - 1]];	
end for	
for i = n-1 to 0 do	N – 1
B[Res[A[i]] - 1] = A[i]	
Res[A[i]] = --Res[A[i]] - 1	
end for	
Return B	

2D) Yes our running time can actually be improved by using a Count Sort. We will only run in O(n) as implementing this algorithm will remove the nested loop.

2E) The space complexity can be improved; however, it will require rewriting the entire algorithm to use less memory. This will change the intended behavior and potentially the complexity of the algorithm depending on how it is implemented. Therefore, it might be a better idea to leave it as is.

Question 3:

i) f(n) log^3n (Can be rewritten as (logn)^3) and g(n) = sqrt(n)
First testing to see if it fits the definition of Big O
f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 20
f(n) <= 1 * g(n) where n >= 20
log(n)^3 <= 4 * sqrt(n)
log(20)^3 <= 4 * sqrt(20)
log(20) <= 4 * sqrt(20) => True

Therefore, the relationship for this function is f(n) is O(g(n)) when k > 20 and c > 1

ii) f(n) n(sqrt(n)) + logn and g(n) = logn^2
First testing to see if it fits the definition of Big O
f(n) = O(g(n)) if f(n) <= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 5000
f(n) <= 1 * g(n) where n >= 5000
n(sqrt(n)) <= 1 * logn^2 => False
logn <= 1 * logn^2 => True
Therefore, 5(sqrt(5)) + log(5) <= 0.1 * (log5)^2 => False

This means that f(n) cannot be O(g(n)) however by the definition of Big Omega:
f(n) = Ω (g(n)) if f(n) >= C* g(n) for all n>= K where C > 0 and K > 0
f(n) >= 1 * g(n) where n >= 5000

n(sqrt(n)) >= 1 * logn^2 => True
logn >= 1* log5000^2 => True whenever C >= 1 and n >= 5000

Therefore, the relationship of these functions are f(n) is Ω(g(n)) when C >= 1 and K >= 5000

iii) f(n) = n and g(n) = (logn)^2
Let's use limits to prove this relationship: $\lim_{x\to\infty} n/(logn)^2$
Applying L'hopitals rule: $\lim_{x\to\infty} 1/(2/\mathrm{x})$
= $\lim_{x\to\infty} x/2$
= ∞
Since the limit is infinity, it is sufficient to say that the relationship between these two functions are f(n) is Ω(g(n))

Using the definition of Omega to prove this:
f(n) = Ω (g(n)) if f(n) >= C* g(n) for all n >= K where C > 0 and K > 0

n >= log(n)^2

Let n = 5000 and C = 3
5000 >= log(5000)^2 => True

Therefore, this relationship is valid for all C >= 3 and N >= 5000

iv) f(n) = sqrt(n) and g(n) is 2^(sqrt(logn))
Let's first try to use the definition of Big-O for this problem.
f(n) = O(g(n)) if f(n) <= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 100
f(n) <= 1 * g(n) where n >= 100
f(100) <= 1 * g(100) where n >= 100
sqrt(100) <= 1 * 2^(sqrt(log(100))) where n >= 100 and C >= 1 is False

Let's use the definition of Big-Omega then.
f(n) = Ω (g(n)) if f(n) >= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 100
f(n) >= 1 * g(n) where n >= 100
f(100) >= 1 * g(100) where n >= 100
sqrt(100) >= 1 * 2^(sqrt(log(100))) where n >= 100 and C >= 1 is True.

Therefore, the relationship between these functions is f(n) is Ω(g(n))

v) f(n) = 2^(n!) and g(n) = 3^n
Let's first try to use the definition of Big-O for this problem.
f(n) = O(g(n)) if f(n) <= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 4
f(n) <= 1 * g(n) where n >= 4
f(4) <= 1 * g(4) where n >= 4
2 ^ (4!) <= 1 * 3^4 where n>= 4 and C >= 1 is False

Therefore, let's definition of the Big-Omega definition:
Let's use the definition of Big-Omega then.
f(n) = Ω (g(n)) if f(n) >= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 4
f(n) >= 1 * g(n) where n >= 4
f(4) >= 1 * g(4) where n >= 4
2 ^ (4!) >= 1 * 3^4 where n>= 4 and C >= 1 is True

Therefore, the relationship between these functions is f(n) is Ω(g(n))

vi) f(n) = 2^10n and g(n) = n^n
Let's solve this using limits.
= $\lim_{n\to\infty} 2\text{^}10n/\text{n^n}$
= $\lim_{n\to\infty} 1024\text{^}n/n\text{^}n$
= 0, Since the limit is 0, it is sufficient to say that the relationship is f(n) is O(g(n)).

Now actual proof, using definitions of Big O:
f(n) = O(g(n)) if f(n) <= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 10000
2^100000 <= 1 * 10000 ^ 10000
Therefore this proof is valid where C > 1 and K >= 10000

vii) f(n) = (n^n)^5 and g(n) = n^(n^5)
Let's solve this using limits.
= $\lim_{n\to\infty} (\text{n^n})\text{^}(5)\ /\text{n^}(\text{n}^5)$
= $\lim_{n\to\infty} (n\text{^}5n)/(n\text{^}(n\text{^}5))$
= 0, Since the limit is 0, it is sufficient to say that the relationship is f(n) is O(g(n)).

Solving using definition of Big O

f(n) = O(g(n)) if f(n) <= C* g(n) for all n >= K where C > 0 and K > 0

Let C = 1 and K = 10000
(10000^10000) <= 1 * 10000 ^ (10000 ^ 5)
Therefore this proof is valid where C > 1 and K >= 10000