COMP 352: Assignment 2
Name: Lise Delesalle
Id: 40160434

Written Part:

Q1.
a.

Algorithm PegDisks(num, from_peg, to_peg, aux_peg)

Input: an integer num and three characters from_peg, to_peg and aux_peg

Output: prints a protocole of all disk movements

difference ← from_peg - to_peg

if difference is -1 or 1 **then**

    if num=1 **then**

        print “Disk 1 moved from ”, from_peg, “to ”, to_peg

        return

    END if

PegDisks(num-1, from_peg, aux_peg, to_peg)

print“Disk“, num, “moved from”, from_peg, “ to “, to_peg

PegDisks(num-1, aux_peg, to_peg, from_peg)

END if

else

    PegDisks(num, from_peg, aux_peg, to_peg)

    print “Disk“, num, “moved from”, from_peg, “ to “, to_peg

    PegDisks(num, aux_peg, to_peg, from_peg)

b.

For n=3, here is the output:

Disk 1 moved from A to B

Disk 1 moved from B to C

Disk 2 moved from A to B

Disk 1 moved from C to B

Disk 1 moved from B to A

Disk 2 moved from B to C

Disk 1 moved from A to B

Disk 1 moved from B to C

Disk 3 moved from A to B

Disk 1 moved from C to B

Disk 1 moved from B to A

Disk 2 moved from C to B

Disk 1 moved from A to B

Disk 1 moved from B to C

Disk 2 moved from B to A

Disk 1 moved from C to B

Disk 1 moved from B to A

Disk 3 moved from B to C

Disk 1 moved from A to B

Disk 1 moved from B to C

Disk 2 moved from A to B

Disk 1 moved from C to B

Disk 1 moved from B to A

Disk 2 moved from B to C

Disk 1 moved from A to B

Disk 1 moved from B to C

c.

The time complexity of the PegDisks algorithm can be analysed recursively based on the number of disks (num) being moved.

For num=1, the algorithm returns immediately, so it's a constant time, O(1).

For num>1, the algorithm makes two recursive calls to PegDisks with num-1 disks, followed by some constant time operations. The two recursive calls are independent of each other and move num-1 disks each. Therefore, the time complexity can be expressed as follows: **O(**2^{num}**).**

Q2.

a.

Algorithm RecursivePermutation(A, n)

Input: An array A, an integer n

Output: Generates all the permutations of the numbers of length n

if n=1 **then**

    return A

else

    for i ← 0 **to** n-1 **do**

        A[i], A[n-1] = A[n-1], A[i]

        RecursivePermutation(A, n-1)

        A[i], A[n-1] = A[n-1], A[i]

The time complexity of this algorithm can be calculated as follows:

- The outer loop iterates n times

- For each outer loop iteration, the function recursively calls itself with a smaller n value of n-1
- Each recursive call has an inner loop that iterates n times
- The time complexity of this algorithm can therefore be written as O(n*(n-1)). In Big-oh notation this is equivalent to **O(n!)**.

b.

Algorithm NonRecursivePermutation(n)

Input: An integer n

Output: Generates all the permutations of the numbers of length n

Queue = []

for i←1 **to** n **do**

    Append i in Queue[]

    while Queue[] is not empty

        if length of permutation = n **then**

            return permutation

        else

            for i← 1 **to** n **do**

              if i is not in permutation **then**

                  Append permutation concatenated with i to Queue[]

At each iteration of the while loop, the algorithm dequeues a permutation from the queue and enqueues up to n-1 new permutations.

Therefore, the total number of permutations that the algorithm generates is **(n-1) * (n-2) * ... * 2 * 1 = (n-1)!**

Therefore, the time complexity of the algorithm is **O((n-1)!)**

Q3.

a.

Algorithm generatePowerSet(T)

Input: an array of integers T

Output: list of sets of integers

powerSet ← empty List of Sets of Integers

n ← length of T

// Add empty set to power set

powerSet.add(new empty Set of Integers)

// Create stack and queue for generating subsets

stack ← empty Stack of Sets of Integers

queue ← empty Queue of Sets of Integers

// Enqueue all singleton sets into the queue

For i ← 0 **to** n-1 **do**

set ← new Set of Integers

set.add(T[i])

queue.offer(set)

END For

// Generate subsets using stack and queue

generatedSubsets ← empty Set of Sets of Integers

While queue is not empty **do**

subset ← queue.poll()

```
      If subset is not in generatedSubsets then
         powerSet.add(subset)
         generatedSubsets.add(subset)
      END if
      For i ← 0 to n-1 do
         if T[i] is not in subset then
            newSubset ← copy of subset
            newSubset.add(T[i])
            stack.push(newSubset)
         END if
      END For
      While stack is not empty do
        //takes a set into a queue
         queue.offer(stack.pop())
      END while
   END while


   Return powerSet
```

b.

The time complexity of the given pseudocode can be analysed as follows:

- The initialization of the powerSet list, stack, and queue take constant time.
- The for loop that adds all singleton sets into the queue runs n times, where n is the length of the input array T. Each iteration of the loop creates a new set and adds it to the queue, which takes constant time. Therefore, the total time complexity of this loop is O(n).
- The main loop that generates subsets runs until the queue is empty. In each iteration, it performs the following steps:

- Dequeue a subset from the queue (constant time).
- Check if the subset has been generated before by checking if it is in the generatedSubsets set (constant time if a good hash function is used). If it has not been generated before, add it to the powerSet list and the generatedSubsets set (constant time for each operation).
- Iterate over all elements of T. For each element that is not already in the subset, create a new subset by copying the existing subset and adding the new element. This takes O(|subset|) time, where |subset| is the size of the subset. Since the size of the largest subset in the power set is 2^n, this loop runs in O(n*2^n) time.
- Push all newly created subsets onto the stack (constant time for each operation).
- Transfer all subsets from the stack to the queue (constant time for each operation).

The overall time complexity of the algorithm is therefore **O(n*2^n)**, where n is the length of the input array T. This is because the size of the power set is 2^n, and the algorithm generates all of these subsets by iterating over each element of T for each subset, and performing a constant amount of work for each operation.

Q4.

We will start by computing the time complexity in big-Oh notation for each operation:

$n^4 + (\log n)^2$ has a time complexity of **O(**n^4)

log log n has a time complexity of **O(log log n)**

$\sqrt{n}$ has a time complexity of **O(**$\sqrt{n}$ **)**

n! + n has a time complexity of **O(n!)**

$\frac{n}{2}$ has a time complexity of **O(n)**

$\binom{n}{2} = \frac{n!}{2\,(n-2)!} = \frac{1*2*3*...*n-2*n-1*n}{1*2*3*...*n-2} = \frac{(n-1)n}{2}$ and therefore, has a time complexity of

O(n^2)

2^n has a time complexity of **O(**2^n**)**

n log n has a time complexity of **O(n log n)**

n^n has a time complexity of **O(**n^n **)**

$2^{log\ n}$ = n has a time complexity of **O(n)**

$2^{n!}$ + n^2 has a time complexity of **O(**$2^{n!}$**)**

2^{2^n} has a time complexity of **O(**2^{2^n} **)**

Therefore, below is the non-decreasing order of the functions according to their big-Oh time complexities:

log log n ≤ $\sqrt{n}$ ≤ n log n ≤ $\frac{n}{2}$ ≤ $2^{log\ n}$ ≤ $\binom{n}{2}$ ≤ 2^n ≤ $n^4 + (log\ n)^2$ ≤ n! + n ≤ n^n ≤ 2^{2^n} ≤ $2^{n!}$ + n^2

<u>Programming part:</u>

valstk ← Stack which holds the values

opstk ← Stack which holds the operators

Algorithm operatorPrec(op)

if op is "^" **then**

return 5

else if op is "*" **or** op is "/" **then**

return 4

else if op is "+" **or** op is "-" **then**

return 3

else if op is ">" **or** op is "<" **or** op is "<=" **or** op is ">=" **then**

return 2

```
    else if op is "=" or op is "!=" then
        return 1
    else
        return 0

Algorithm calculate(x, y, op)
    result ← " "
    if op is "^" then
        result += power(x, y)
    else if op is "*" then
        result += multiplication(x, y)
    else if op is "/" then
        result += division(x, y)
    else if op == "+" then

        result += addition(x, y)
    else if op is "-" then
        result += subtraction(x, y)
    else if op is "<" then
        result += lessThan(x, y)
    else if op is ">" then
        result += moreThan(x, y)
    else if op is "<=" then
        result += lessThanOrEqual(x, y)
    else if op is ">=" then
        result += moreThanOrEqual(x, y)
```

```
    else if op is "==" then
        result += Equals(x, y)
    else if op is "!=" then
        result += notEquals(x, y)
    return result

Algorithm operation(op)
    y ← valstk.pop()
    x ← valstk.pop()
    if op is "=" then
        op ← opstk.pop() + op
    result = calculate(x, y, op)
    valstk.push(result)

Algorithm precedenceOp(input)
    if input is "(" then
        opstk.push(input)
        return
    if opstk.isEmpty() then
        inner ← 0
    else
        inner = operatorPrec(opstk.top())
    outer = operatorPrec(input)
    if inner < outer then
        opstk.push(input)
    else
```

```
if input is ")" and opstk.top() is "(" then
    opstk.pop()
    return
else if input is "=" and (opstk.top() is "=" or opstk.top() is "!" or opstk.top() == "<" or
opstk.top() is ">") then
    opstk.push(input)
    return
else
    operation(opstk.pop())
    precedenceOp(input)
```

Algorithm arithmeticCalculator(equation)

```
numHolder ← " "
for each character i in equation do
    if i is ' ' then
        continue
    if i isNumeric then
        numHolder ← numHolder + i
    else
        if numHolder isEmpty then
            values ← push(numHolder)
            numHolder ← ""
        precedenceOp(i)

if numHolder is not Empty then
    values ← push(numHolder)
```

while operations is not Empty **do**

doOp(operations ← pop())

finalResult ← values ← pop()

print(finalResult)

c.

The operatorPrec algorithm has a time complexity of

O(1) → No loops, No recursion.

and a space complexity of

O(1) → Only 1 variable gets used.

The calculate algorithm has a time complexity of

O(1) → No loops, No recursion, performs only once when called.

and a space complexity of

O(1) → Only a constant number of variables are created every time.

The precedenceOp has a time complexity of

O(n) → This function can call itself n/2 times, going character by character, as it might have to call itself up to n/2 times.

and a space complexity of

O(n) → While the code is recursive, nothing is returned by the function, as such, the space taken will be a constant times the amount of recursions that takes place.

The arithmeticCalculator has a time complexity of

O(n^2) → there is a loop going character by character (up to n times), and the precedenceOp that might call itself n times.

and a space complexity of

O(n) → Gotten from the precedenceOp, this function itself will only have a constant number of variables.