Sridevi Nataraj
Student ID: 40005572
Assignment # 2 - Part 1 & 2
Comp 352- Section X
Date: Monday, February 20 2017

Written Questions (50 marks):

Question 1
Consider the following scenario: you have a machine hall containing three pegs named A, B, and C. Each peg can hold many discs, but a disc with a larger diameter can never be placed on top of a disc with a smaller diameter and all discs have different diameters. If there are *n* discs, then the discs are numbered from 1 to *n*, where 1 is the smallest disc and *n* is the largest disk. There exists a robot arm that can move exactly one disc at a time from peg A to B or B to A, or from B to C or C to B. The robot arm cannot do any other move. Initially, all *n* discs are on peg A stacked by increasing diameters from top to bottom, with the bottommost one of the largest diameter. The goal is to move all discs from peg A to C by using only the robot arm based on the rules and restrictions described above. You are

required to submit the following deliverables:

Algorithm: TowerOfHanoi
Input: Number of disks
Output: The possible movements from A,B, and C

```
TowerOfHanoi(N, Src, Aux, Dst)
if N is 0
  exit
else
 TowerOfHanoi (N-1, source, Aux, Dst)    // moves (N-1) small disks from source to Dst
  Move from source to Aux                 // moves the largest disk from source to Aux
 TowerOfHanoi (N-1, Dst, Aux, source)   // moves (N-1) small disks from Dst to source
  Move from Aux to Dst                    // moves the largest disk from Aux to Dst
 TowerOfHanoi (N-1, source, Aux, Dst)  // moves (N-1) small disks from source to Dst
End if
```

B) A hand-run of your algorithm for *n* = 4.

If n=4, then then we could equal the formula to T(4) = 2T(3)+1= 15,
Hence the Tower of Hanoi code would move the disks 15 times.

C) Formulation and calculation of its complexity using the big-Oh notation. Justify your answers.

We Know that when n=1, the code would move the 1 disk directly. Which means T(1)=1 // which is the number if disks

In the other cases, the code would follow this procedure.

1. First they would move the (n-1)- disk tower to the spare peg; The takes T(n-1) moves.
2. So, the code would the nth disk taking 1 move.
3. Finally, they would move the (n-1)- disk again, this time on top of the disk. Meaning it would take T(n-1) moves.
   Hence the recurrence formula would be T(n)= 2(T(n-1)+1)

With all that said the Big O → O(2^n)

Question # 2

A) In class, we discussed a recursive algorithm for generating the power set of a set T, i.e., the set of all possible subsets of T. In this assignment, you will develop a well-documented pseudocode that generates all possible subsets of a given set of size *n* with the following requirements: your solution must be non-recursive, and must use a stack and a queue to solve the problem. For example: if set T = {2, 4, 7, 9} then your algorithm would generate: {}, {2}, {4}, {7}, {9}, {2,4}, {2,7}, {2,9}, {4,7}, {4,9}, {7,9}, ...etc. (Note: your algorithm's output need not be in this order).

```
Algorithm: AllPossobleSubstes(T)
  Input: A set of T with n elements
  Output: All subsets of T stored in a Queue
{
  Create an empty queue Q;
  Create an empty stack S;
  Let a1, a2, …. an be all the elements in the subset of T
  s.push({}); // push the empty subset into the stack
  s.push ({a1})
  for (each element ai in T-{a1})
     {while (S is not empty)
           { x=s.pop;
             Q.enqueue(x);
             x=x ∪ {a1};
              Q.enqueue(x);
           }
        if (ai is not the last element)
          while (Qis not empty)
```

```
                    { x=Q.dequeue();
                      S.push();
                    }
          }  }
```

b) Calculate the time complexity of your algorithm using the big-Oh notation. Show all calculations.

For each element in a_i in T, takes out everything in S, put it back into S twice; once with a_i and once without a_i. So, the total time complexity is $(1+2+4+\ldots+2^n)$ times C. And C stands for the time to pop, enqueue, dequeue, and push which, they all are usually (1). Hence, when we add then together the T(n)= 2^(n+1)-1. Which would mean that O (2^n) is the o complexity.

Question # 3

Rank the following functions in non-decreasing order (≤) based on their tight big-Oh complexities and justify your ranking:

$\log \log n < \binom{n}{2} < n \log n < n/2 < 2^{\log n} < n^{1/2} < n^3 + \log n < n! < 2^n \leq 2^{2^\wedge n} < 2^n < n^n < 2^{n!}$

Written Questions (50 marks):

A) Design and submit the pseudo-code for both the versions of arithmetic calculator.

Algorithm: Calculator Stacks
Input: Expression that needs to resolved Output
Output: The answer of the expression

```
----------------------------------------------------------
public class StackCalculator
{
   public static void main(String[] args)
   {
      Scanner scan = new Scanner(System.in);
      Stack<Integer> op  = new Stack<Integer>();// Create stack arrays
      Stack<Double> op1 = new Stack<Double>()
      Stack<Integer> optmp  = new Stack<Integer>();// create temporary stacks arrays for the operators
                                                                     and operands
      Stack<Double> op1tmp = new Stack<Double>();
      print "Enter expression"       // Ask the user to enter the expression to be solved
      String input = scan.next();
      input = "0" + input;
      input = input.replaceAll("-","+-");
      String temp = "";
      for <int i = 0 to i < input.length()> //Store operands and operators in respective stacks
{
         char ch = input.charAt(i);
         if (ch == '-')
```

```
        temp = "-" + temp;
    else if (ch != '+' &&  ch != '*' && ch != '/' && ch !='^' && ch != '!' && ch != '>' && ch!='<' &&
                          ch != '==' && ch !='!=')
       temp = temp + ch;
    else
    {op1.push(Double.parseDouble(temp));
        op.push((int)ch);
        temp = "";
    }
}
op1.push(Double.parseDouble(temp));
char operators[] = {'/','*','+','^','!','>','>=','<=','<','==','!='};// Create char array of operators as per
                                                                       precedence
 for < int i = 0 to  i < 3;>
{
    boolean it = false;
    while (!op.isEmpty())
    {
        int optr = op.pop();
        double v1 = op1.pop();
        double v2 = op1.pop();
        double result = Math.pow(v1,v2);


                if <optr equals operators[i]>
           {
           // if operator matches evaluate and store in temporary stack
           if < i equal 0>
           {
              op1tmp.push(v2 / v1);
              it = true;
              break;
           }
           else if < i equal 1>
           {
              op1tmp.push(v2 * v1);
              it = true;
              break;
           }
           else if < i equal 2>
           {
              op1tmp.push(v2 + v1);
              it = true;
              break;
           }
           else if < i equal 3 >
           {
                 op1tmp.push(Math.pow(v2,v1));
                 it= true;
                 break;
           }
           else if < i equal 4 >
           {
                  op1tmp.push (v2 ! v1);
                  it = true;
              break;
```

```
            }
            else if < i equal 5 >
            {
                    op1tmp.push(v2 > v1);
                    it = true;
                    break;
            }
            else if < i equal 6 >
            {
                    op1tmp.push(v2 <= v1);
                    it = true;
                    break;
            }
            else if < I equal 7>
            {
                    op1tmp.push(v2 => v1);
                    it = true;
                    break;
            }
           else if < I equal 8 >
            {
                  op1tmp.push(v2 < v1);
                    it = true;
                    break;

            }
            else if <i equal 9 >
            {
                    op1tmp.push(v2 == v1);
                    it = true;
                    break;
            }
            else if < i equal 10 >
            {
                     op1tmp.push(v2 != v1);
                     it = true;
                     break;
                      }
            else
                      {
                     op1tmp.push(v1);
                     op1.push(v2);
                     optmp.push(optr);
            }
        } // Push back all elements from temporary stacks to main stacks
        while (!op1tmp.isEmpty())
           op1.push(op1tmp.pop());

        while (!optmp.isEmpty())
           op.push(optmp.pop());
        if (it)
           i--;
    }
          print (" Result = "+op1.pop())
}
 End If
```

Algorithm: Recursive Calculator
Input: Expression that needs to resolved
Output: The answer of the expression

```
DIVIDERS = new ArrayList<Character>
(Arrays.asList('*', '/', '-', '+', '==', '!=','<=', '>=', '>','<'));
RIGHT_DIRECTION=1;
LEFT_DIRECTION=-1;
Algorithm main
reader = new BufferedReader(new InputStreamReader(System.in));
expression = "";
// Prompt user to enter expression try expression and catch exceptions
Print ("Expression: " + expression);
expression = prepareExpression(expression);
Print ("Answer: " + calc(expression));
Algorithm calc(expression)
numbers=0;
If numbers which is index of "" is not equals to 1 {
subexp= extractExpressionFromBraces(expression, numbers);
replace ("+subexp+") in expression by calc(subexp)
return calc(expression);
}
Else If the index of * or / in expression is greater than zero {
multPos = the index of * in expression
divPos = the index of / in expression
numbers= Math.min(multPos, divPos);
If multPos is less than 0 then numbers equals divPos
Else If divPos is less than 0 then numbers equals multPos
divider = to the char at index numbers of expression
leftNum = extractNumber(expression, numbers , LEFT_DIRECTION);
rightNum = extractNumber(expression, numbers, RIGHT_DIRECTION);
expression = expression.replace(leftNum + divider + rightNum, calcShortExpr(leftNum, rightNum, divider));
return calc(expression);
} Else if index of + or - in expression is greater than 0 {
summPos = the index of + in expression
minusPos = index of - in expression
numbers = Math.min(summPos, minusPos);
If summPos is less than 0 then numbers equals minusPos
Else If minusPos is less than 0 then numbers equals summPos
divider = char at index numbers of expression
leftNum = extractNumber(expression, numbers, LEFT_DIRECTION);
rightNum = extractNumber(expression, numbers, RIGHT_DIRECTION);
expression = expression.replace(leftNum + divider + rightNum, calcShortExpr(leftNum, rightNum, divider));
 return calc(expression);
} Else If
Algorithm extractExpressionFromBraces(String expression, int pos)
braceDepth = 1;
subexp="";
For < i = pos+1 to i < expression.length() > do {
switch case is char at index i of expression {
case '(':
braceDepth++;
subexp += "(";
```

break;
case ')':
braceDepth--;
If braceDepth is not equals to 0 then subexp += ")";
break;
default:
If braceDepth is less than 0 then subexp += expression.charAt(i);
}
If braceDepth equals 0 and subexp is not equal to "" then return subexp
} return "Failure!";
Algorithm extractNumber(String expression, int pos, int direction)
resultNumber = "";
currPos = pos + direction;
If char at index currPos of expression equals to - {
resultNumber+=expression.charAt(currPos);
currPos+=direction;
}
For currPos >= 0 && currPos < expression.length() && !DIVIDERS.contains(expression.charAt(currPos)) to currPos += direction do {
resultNumber += expression.charAt(currPos);
}
If direction equals to LEFT_DIRECTION then resultNumber = new StringBuilder(resultNumber).reverse().toString();
return resultNumber;
Algorithm calcShortExpr(String leftNum, String rightNum, char divider)
switch case is divider {
case '*':
return Double.toString(Double.parseDouble(leftNum) * Double.parseDouble(rightNum));
case '/':
return Double.toString(Double.parseDouble(leftNum) / Double.parseDouble(rightNum));
case '+':
return Double.toString(Double.parseDouble(leftNum) + Double.parseDouble(rightNum));
case '-':
return Double.toString(Double.parseDouble(leftNum) - Double.parseDouble(rightNum));
case ‘==’:
return Double.toString(Double.parseDouble(leftNum) == Double.parseDouble(rightNum));
case ‘!=’:
return Double.toString(Double.parseDouble(leftNum) != Double.parseDouble(rightNum));
case ‘<=’:
return Double.toString(Double.parseDouble(leftNum) <= Double.parseDouble(rightNum));
case ‘>=’:
return Double.toString(Double.parseDouble(leftNum) >=Double.parseDouble(rightNum));
case ‘>’:
return Double.toString(Double.parseDouble(leftNum) > Double.parseDouble(rightNum));
case ‘<‘:
return Double.toString(Double.parseDouble(leftNum) < Double.parseDouble(rightNum));
default: return "0";
}
Algorithm prepareExpression(String expression)
return expression;
}**End IF**

B) Implement a Java program for the first version. In your program, you will implement and use your own array-based stack (and not the built-in Java stack) of variable size based on the doubling strategy discussed in class (refer to the code segments provided on your slides and the textbook).

SUBMITTED IN A ECLIPSE FOLDER.

C) Briefly explain the time and memory complexity of both versions of your calculator. You can write your answer in a separate file and submit it together with the other submissions.

Recursion uses you thread stack and that has a much lower limit. The recursive algorithm will add overhead sink they store recursive stacks, with that said the big-O complexity for the **recursive calculator is O(n).** However, the calculator that uses stack to calculate the expression takes less time. with that said the **stack calculator has a big-o complexity of O(n logn).**