a) Calculator Class:

```
Algorithm evalExp(expression)
Input a String mathematical expression
Output result of the expression

tokens ← expression.split(“ ”)

while (!valStk.isEmpty())
        valStk.pop()

i ← 0
while (i < tokens.length)
       z ← tokens[i]
       if (isNumber(z)) then
         valStk.push(Double.parseDouble(z))
       else if (z.equals(“)”) then
                while (!opStk.top().equals(“(“))
                        doOp()
                opStk.pop()
       else
              repeatOps(z)
              opStk.push(z)
       i++

repeatOps(“$”)
return valStk.top()

Algorithm repeatOps(op)
Input a String operator op
while (valStk.size() > 1 && prec(op) >= prec(opStk.top()))
       if (opStk.top().equals(“(“) then
         break
       doOp()

Algorithm doOp()
op ← opStk.pop()
x ← (double) valStk.pop()
y ← (double) valStk.pop()

switch (op)
    case “+”:
          valStk.push(y + x)
          break
    case “-“:
```

```
        valStk.push(y - x)
        break
    case “*“:
        valStk.push(y * x)
        break
    case “/”:
        valStk.push(y / x)
        break
    case “^“:
        valStk.push(Math.pow(y, x))
        break
    case “>“:
        if (y > x) then
          valStk.push(1.0);
        else
          valStk.push(0.0);
        break
    case “<”:
        if (y < x) then
          valStk.push(1.0);
        else
          valStk.push(0.0);
        break
    case “>=“:
        if (y >= x) then
          valStk.push(1.0);
        else
          valStk.push(0.0);
        break
    case “<=”:
        if (y <= x) then
          valStk.push(1.0);
        else
          valStk.push(0.0);
        break
    case “==“:
        if (y == x) then
          valStk.push(1.0);
        else
          valStk.push(0.0);
        break
    case “!=“:
        if (y != x) then
          valStk.push(1.0);
```

```
            else
              valStk.push(0.0);
            break

Algorithm prec(op)
Input a String operator op
Output precedence of the operator
switch (op)
    case “(“:
    case “)”:
          return 1
    case “^“:
          return 2
    case “*“:
    case “/”:
          return 3
    case “+“:
    case “-“:
          return 4
    case “>“:
    case “<”:
    case “>=“:
    case “<=”:
          return 5
    case “==“:
    case “!=“:
          return 6
    case “$“:
          return 7

return -1

Algorithm isNumber(z)
Input a String z
Output true if the string is a number, false otherwise
try
   Double.parseDouble(z)
   return true
catch (NumberFormatException e)
          return false
```

b) The time complexity of the algorithm is O(n), where n is the number of tokens in the expression. Each token is processed once, and the operations on the stacks are constant time operations, O(1). The stack's size doubles when needed, maintaining an amortized constant time for push operations. The space complexity is also O(n) due to the use of two stacks (valStk and opStk) to hold all values and operators. The space required is proportional to the number of elements in the input expression.