ST1507 DSAA ASSIGNMENT TWO (CA2)

EXPRESSION EVALUATOR & SORTER

GROUP NUMBER: 5

NAMES: IVAN TAY YUEN HENG & CHAN JUN YI

STUDENT IDS: 2335133 & 2309347

CLASS: DAAA/FT/2A/21

User Guidelines

```
$ python main.py

*****************************************************************************
* ST1507 DSAA: Expression Evaluator & Sorter                                *
* ------------------------------------------------------------------------- *
*                                                                           *
*      - Done by: IVAN TAY YUEN HENG (2335133) & CHAN JUN YI (2309347)      *
*      - Class DAAA/2A/21                                                   *
*                                                                           *
*****************************************************************************


Press Enter, to continue....
```

This programme is able to represent, print, and solve fully parenthesized mathematical expressions by means of parse trees. Besides solving individual expressions it is also be able to read a series of expressions from an input file. These expressions will then be evaluated one by one, sorted by their value and length. Subsequently the sorted results will be written back to an output file. To start the programme, navigate to the code folder in terminal and type the following:
python main.py

```
Please select your choice ('1', '2', '3', '4', '5', '6', '7'):
       1. Evaluate expression
       2. Sort expressions
       3. Minimum expression path finder (Ivan Tay)
       4. Random expression generator (Ivan Tay)
       5. Expression history (Chan Jun Yi)
       6. Simplify algebric expression (Chan Jun Yi)
       7. Exit
Enter choice:
```

You will see this as you start the program. Press Enter to continue.

A list of options will then be displayed, allowing you to select a choice between numbers 1 and 7 only. No other choices will be accepted.

1. Evaluation Expressions

Choosing option 1 will prompt the user to enter a fully parenthesized expression. The application will then calculate and print the parse tree, done in in-order traversal, followed by displaying the value that the expression evaluates to.

```
Enter choice: 1
Please enter the expression you want to evaluate:
((1/-3)+(-2**2))

Expression Tree:
   +
 /   *
1 - -*2
  3 2

Expression evaluates to:
3.6666666666666665

Press Enter, to continue....
```

```
def stackInorder(self, level, leafStack):
    if self.leftTree != None:
        self.leftTree.stackInorder(level+1, leafStack)
    leafStack.append([self.key, level])
    if self.rightTree != None:
        self.rightTree.stackInorder(level+1, leafStack)
    self.myStack = leafStack
```

Invalid expression:

```
Please enter the expression you want to evaluate:
(6+1))

Error: Invalid expression
```

```
# If all of the expression is not under 1 bracket, return False
if validation > 1:
    return False
```

If the number of brackets is unbalanced—whether there are more left brackets than right brackets or vice versa—the expression will be considered invalid, and the user will not be able to evaluate it.

```
Please enter the expression you want to evaluate:
(2+5)/2

Error: Invalid expression
```

```
Please enter the expression you want to evaluate:
(2 +- 2)

Error: Invalid expression
```

If the operations are not fully parenthesized or if a set of brackets contains two operators simultaneously, the equation will be considered invalid.

```
Please enter the expression you want to evaluate:
(3 *** 5)

Error: Invalid expression
```

```
Please enter the expression you want to evaluate:
(3 // 5)

Error: Invalid expression
```

If invalid operators are used or other operators other than (+, -, *, / , **), the equation will be considered invalid.

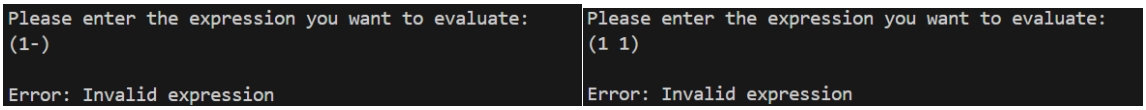
```
Please enter the expression you want to evaluate:
(3. 2 + 1)

Error: Invalid expression
```

```
Please enter the expression you want to evaluate:
(3.2.2 + 1)

Error: Invalid expression
```

If a decimal point is used incorrectly—such as having a space after it or containing two or more decimal points in a single number—the equation will be considered invalid.

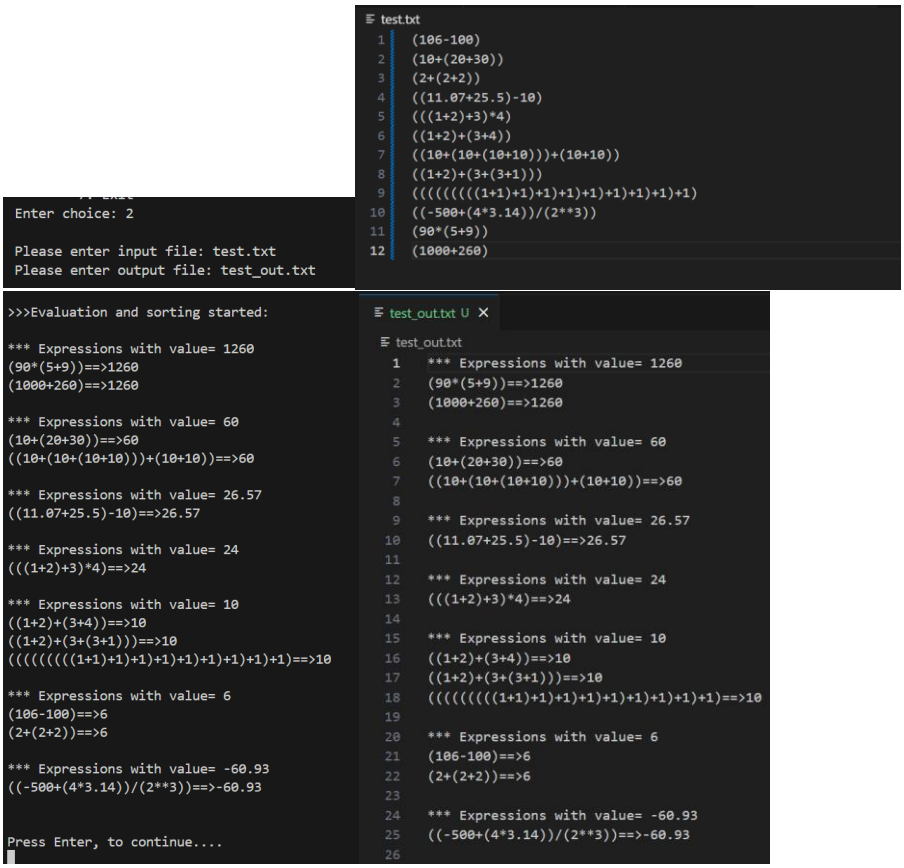
```
Please enter the expression you want to evaluate:
(1/(1-1))
Error: Division by zero
```

If the user enters an expression that results in division by zero, the equation will be considered invalid.

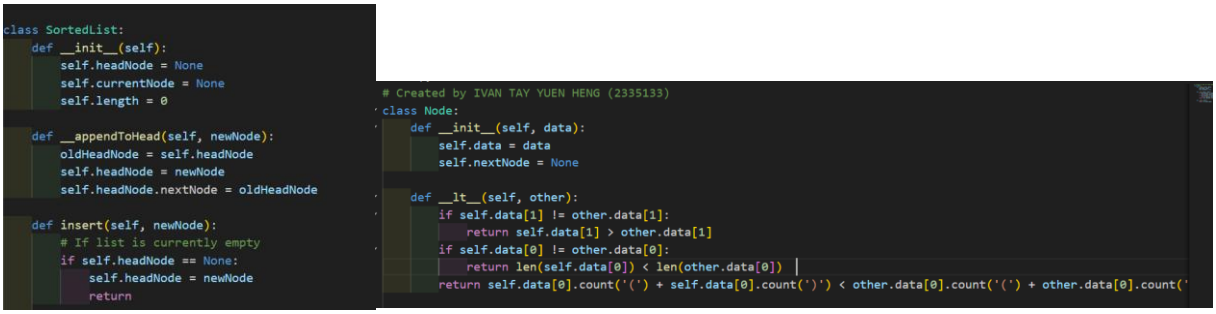


Lastly, if a bracket does not contain exactly one operator and two operands, the expression will be considered invalid.

2. Sorting Expressions



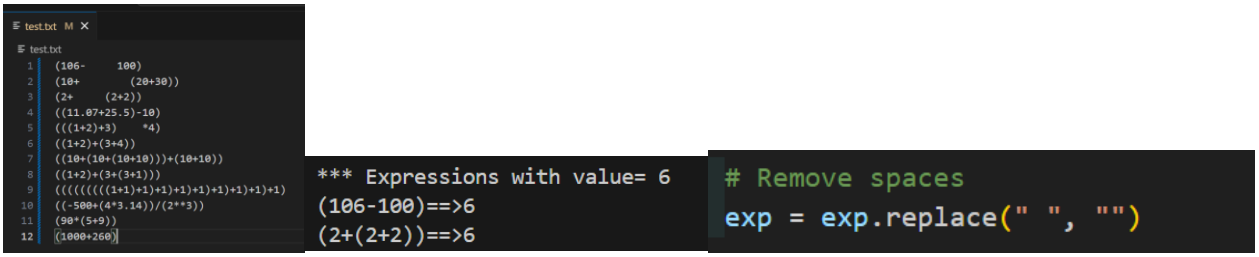
Choosing option 2 will prompt the user to enter an input file and output file. The application reads the expressions, from the input file, evaluates each expression (using a parse tree), and sorts the expression first by value (in descending order) and then by length (in ascending order). If expressions have the same values and same length they will be sorted by total number of brackets (in ascending order).



```
class SortedList:
    def __init__(self):
        self.headNode = None
        self.currentNode = None
        self.length = 0

    def __appendToHead(self, newNode):
        oldHeadNode = self.headNode
        self.headNode = newNode
        self.headNode.nextNode = oldHeadNode

    def insert(self, newNode):
        # If list is currently empty
        if self.headNode == None:
            self.headNode = newNode
            return
```

```
# Created by IVAN TAY YUEN HENG (2335133)
class Node:
    def __init__(self, data):
        self.data = data
        self.nextNode = None

    def __lt__(self, other):
        if self.data[1] != other.data[1]:
            return self.data[1] > other.data[1]
        if self.data[0] != other.data[0]:
            return len(self.data[0]) < len(other.data[0])
        return self.data[0].count('(') + self.data[0].count(')') < other.data[0].count('(') + other.data[0].count('
```

We use a sorted list to arrange the mathematical expressions, where the sorting is first based on the value (in descending order), then by length (in ascending order), and if both value and length are the same, by the total number of brackets (in ascending order), with a Node class handling the comparison logic.



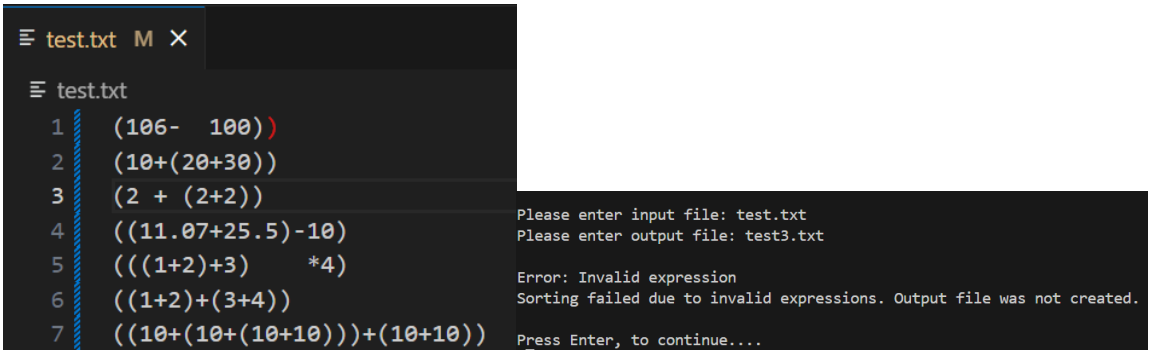
test.txt

```
(106-     100)
(10+        (20+30))
(2+     (2+2))
((11.07+25.5)-10)
(((1+2)+3)    *4)
((1+2)+(3+4))
((10+(10+(10+10)))+(10+10))
((1+2)+(3+(3+1)))
(((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1)
((-500+(4*3.14))/(2**3))
(90*(5+9))
(1000+260)
```

```
*** Expressions with value= 6
(106-100)==>6
(2+(2+2))==>6
```

```
# Remove spaces
exp = exp.replace(" ", "")
```

Even if the expressions contain spaces, the spaces will be removed before calculating their length, and the results will be displayed without any spaces.



test.txt

```
1  (106-  100))
2  (10+(20+30))
3  (2 + (2+2))
4  ((11.07+25.5)-10)
5  (((1+2)+3)    *4)
6  ((1+2)+(3+4))
7  ((10+(10+(10+10)))+(10+10))
```

```
Please enter input file: test.txt
Please enter output file: test3.txt

Error: Invalid expression
Sorting failed due to invalid expressions. Output file was not created.

Press Enter, to continue....
```

If the input file has an error or invalid expression (same as option 1). It will show an error messages and the output file will not be created.

3. **Exit**

Choosing option 7 will exit the program, showing an appreciation message to the user.

```
Please select your choice ('1', '2', '3', '4', '5', '6', '7'):
        1. Evaluate expression
        2. Sort expressions
        3. Minimum expression path finder (Ivan Tay)
        4. Random expression generator (Ivan Tay)
        5. Expression history (Chan Jun Yi)
        6. Simplify algebric expression (Chan Jun Yi)
        7. Exit
Enter choice: 7

Bye, thanks for using ST1507 DSAA: Expression Evaluator & Sorter
```

Object-Oriented Programming (OOP) approach

Encapsulation

```python
# Read input file
class ReadFile:
    def __init__(self, option):
        self.__option = option
        self.__file_content = None
        self.__file_name = None
        self.read_file()
```

```python
def get_content(self):
    return self.__file_content

def get_filename(self):
    return self.__file_name  #
```

I implemented encapsulation by using attributes like __file_content(), __file_name, and __output_file, making them inaccessible directly from outside the class. Controlled access is provided through methods such as get_content(), get_filename(), and get_output_file_name(). This ensures the internal state remains secure, allowing external code to retrieve data without the risk of unintended modifications.

Operator Overloading

```python
def __lt__(self, other):
    print(self.data[1], other.data[1])
    if self.data[1] != other.data[1]:
        return self.data[1] > other.data[1]
    if self.data[0] != other.data[0]:
        return len(self.data[0]) < len(other.data[0])
    return self.data[0].count('(') + self.data[0].count(')') < other.data[0].count('(') + other.data[0].count(')')
```

I overloaded the __lt__ method in the Node class to sort expressions based on three criteria. First, expressions are compared by their evaluated result (data[1]). If results are the same, shorter expressions (len(data[0])) take priority. If both are equal, the expression with fewer parentheses is prioritized. This ensures sorting aligns with the assignment requirements.

Polymorphism

```
class Tree:
    def __init__(self):
        self.myStack = []

    def printTree(self):
        raise NotImplementedError("Subclasses must implement this method")
```

printTree

```
def printTree(self):
    self.printExpressionTree()
```

I implemented polymorphism by defining a base class, Tree, with an abstract method that is overridden in its subclasses, BinaryTree and ExpressionTree. Each subclass provides its own unique implementation, ensuring that the method behaves differently based on the specific subclass.

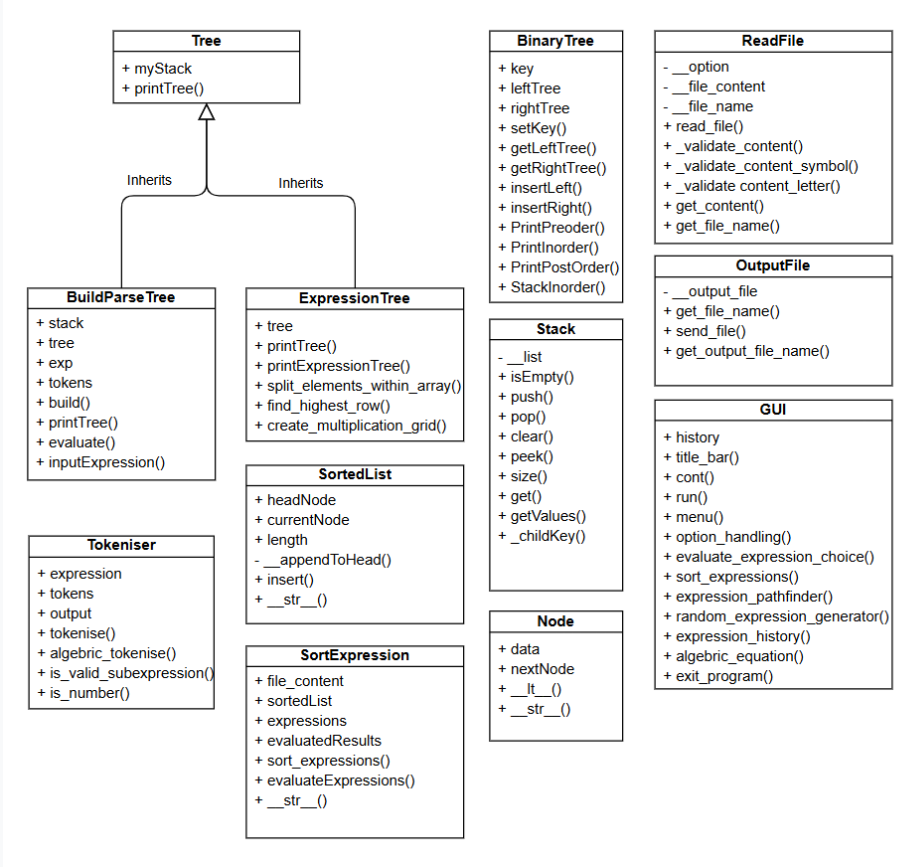
Inheritance

```
class BinaryTree(Tree):
    def __init__(self,key, leftTree = None, rightTree = None):
        super().__init__()
        self.key = key
        self.leftTree = leftTree
        self.rightTree = rightTree
```

```
class ExpressionTree(Tree):
    def __init__(self, tree):
        super().__init__()
        self.tree = tree
```

I implemented inheritance with the BinaryTree and ExpressionTree classes, both of which inherit from Tree. The Tree class defines a common attribute, myStack, and super().__init__() is used to initialize properties from the Tree class.

Class Diagram



Data structures and algorithms

Data structure	Big (O)
Binary Tree	Insert(Left or Right): O(log n), O(n) (worst case) Search: O(log n) average, O(n) (worst case) Space: O(n)
Sorted List	Insert: O(n) Search: O(n) Sorting Complexity: O(n)

Stack	Push/Pop/Peek/Get Size: O(1) Space complexity: O(n)

Binary Tree

```
def getKey(self):
    return self.key

def getLeftTree(self):
    return self.leftTree

def getRightTree(self):
    return self.rightTree

def insertLeft(self, key):
    if self.leftTree == None:
        self.leftTree = BinaryTree(key)
    else:
        t =BinaryTree(key)
        self.leftTree , t.leftTree = t, self.leftTree

def insertRight(self, key):
    if self.rightTree == None:
        self.rightTree = BinaryTree(key)
    else:
        t =BinaryTree(key)
        self.rightTree , t.rightTree = t, self.rightTree
```

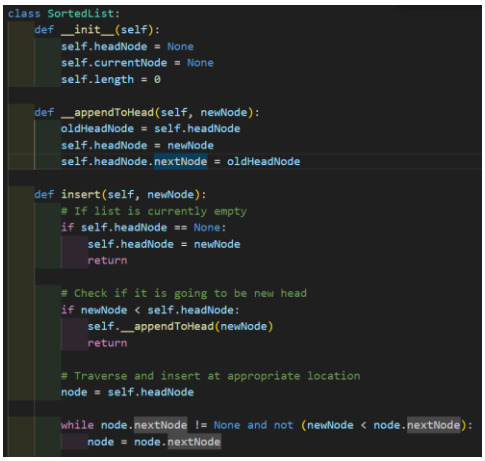
A Binary Tree enables O(log n) insertions and searches when balanced but degrades to O(n) in the worst case if unbalanced. It structures mathematical expressions hierarchically, ensuring correct operation precedence and correct evaluation

Stack

```
class Stack:
    def __init__(self):
        self.__list = []

    def isEmpty(self):
        return self.__list == []

    def push(self, item):
        self.__list.append(item)

    def pop(self):
        if self.isEmpty():
            return None
        else:
            return self.__list.pop()
```

The Stack provides O(1) time complexity for push, pop, and checking if empty, making it efficient for last-in, first-out (LIFO) operations. It helps handle nested expressions especially in creating binary tree in (option 1).

Sorted List



```python
class SortedList:
    def __init__(self):
        self.headNode = None
        self.currentNode = None
        self.length = 0

    def __appendToHead(self, newNode):
        oldHeadNode = self.headNode
        self.headNode = newNode
        self.headNode.nextNode = oldHeadNode

    def insert(self, newNode):
        # If list is currently empty
        if self.headNode == None:
            self.headNode = newNode
            return

        # Check if it is going to be new head
        if newNode < self.headNode:
            self.__appendToHead(newNode)
            return

        # Traverse and insert at appropriate location
        node = self.headNode

        while node.nextNode != None and not (newNode < node.nextNode):
            node = node.nextNode
```

Sorted Lists maintain elements in sorted order (O(n)) as they are inserted, using the Node class to compare values while storing n nodes (O(n)), ensuring expressions are correctly ordered for option 2 of your assignment.

Summary

Challenges:

This project made use of parse trees and applying object-oriented programming (OOP) principles such as encapsulation, polymorphism, and inheritance.To utilise parse trees and construct or traverse through them, we had to have a good understanding of recursion. The use of clear sorting logic and ensuring proper input validation and error handling is critical to have a successful project.

Key takeaways and learnings:

This project reinforced our understanding on binary trees, recursion, and algorithm design. Ensuring a structured and modular approach with object-oriented programming (OOP) principles helped make working in a team environment more efficient by improving code organization. ensuring that different components of the application were reusable and adaptable allowed for better task delegation, seamless integration of individual contributions, and easier debugging. This made collaboration smoother and reducing conflicts during development.

Roles and contributions

IVAN TAY YUEN HENG (2335133)

For Option 1, implement the BuildParseTree class, allowing users to construct a binary tree and validate mathematical expressions using the ExpressionTree class, with both classes inheriting from a common parent class, Tree. Develop a GUI interface to ensure user are able to input and visualise expressions.

For Option 2, create the ReadFile and OutputFile classes to handle user input and output file operations required for sorting expressions. Implement the SortExpressions class, which evaluates mathematical expressions and stores them in a SortedList. Implement Node class to sort them based on three criteria: descending order of expression value, ascending order of expression length, and ascending order of bracket count

Contribute to the group report, covering the OOP approach and data structure & algorithms with explanations and screenshots and the proper formatting of the final report layout.

The project that Ivan created: (binaryTree.py, buildParseTree.py, expressionTree.py, fileHandling.py, fileOutput.py, sortedList.py, sortedNode.py, sortExpression.py, stack.py, tokeniser.py, and tree.py)

CHAN JUN YI (2309347)

For Option 1, tokenising of expression, implement the BuildParseTree class, allowing users to construct a binary tree and planned the implementation of ExpressionTree class, with both classes inheriting from a common parent class, Tree. Develop a GUI interface to ensure user are able to input and visualise expressions.

For Option 2, validated the user input file contents required for sorting expressions.

Consolidated python codes from notebooks into python files with classes using OOP concept. Contribute to the group report, covering the user guidelines and screenshots, class diagram, summary and the proper formatting of the final report layout.

The project that Jun Yi created: (binaryTree.py, buildParseTree.py, expressionTree.py, fileHandling.py, sortedList.py, stack.py, tokeniser.py, and tree.py)

Appendix

Addition.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


from operation import BaseOperation


class AddOne(BaseOperation):
    def apply(self, num):
        return num + 1
```

algebricEquation.py

```
# Created by: CHAN JUN YI (2309347)
import re
from buildParseTree import BuildParseTree
from algebricTokeniser import AlgebricTokeniser


class AlgebricEquation(BuildParseTree):
    def __init__(self):
        super().__init__()


    def build(self):
        if self.tokens is None or self.tokens == []:
            print("\nError: Invalid expression")
            return None
        else:
            self.stack.push(self.tree)
            currentTree = self.tree


            for t in self.tokens:
                if t == '(':
```

```
                currentTree.insertLeft('?')
                self.stack.push(currentTree)
                currentTree = currentTree.getLeftTree()

            elif t in {'+', '-', '*', '/', '**'}:
                currentTree.setKey(t)
                currentTree.insertRight('?')
                self.stack.push(currentTree)
                currentTree = currentTree.getRightTree()

            elif t not in {'+', '-', '*', '/', ')', '**'}:
                if t.isdigit():  # If token is a number
                    currentTree.setKey(float(t))
                elif t.isalpha():  # If token is purely alphabetical
                        currentTree.setKey(t)  # Treat as string literal
                else:
                    try:
                        currentTree.setKey(float(t))  # Try parsing as a number
                    except ValueError:
                        currentTree.setKey(t)  # Otherwise, treat as a string
                currentTree = self.stack.pop()

            elif t == ')':
                if not self.stack.isEmpty():
                    currentTree = self.stack.pop()
        return self.tree


def make_poly(self, token):
    """
    Given a token (either a number or a variable expression) return
    its polynomial representation [dict, constant].

    A numeric token (or one that can be parsed as a float) is treated as a constant.
    A token that represents a variable (for example: "2p", "p", or "3p^2")
    is parsed into its coefficient and exponent.
    """
```

```
    # If the token is already a number (float or int) then return constant-only poly.
    if isinstance(token, (int, float)):
        return [{}, float(token)]
    if isinstance(token, str):
        # Try to interpret token as a number first.
        try:
            value = float(token)
            return [{}, value]
        except ValueError:
            # Token is not a pure number; try to parse a variable term.
            # This regex will match an optional signed number, followed by a letter,
            # optionally followed by '^' and an exponent.
            pattern = r'^([+-]?(\d+(\.\d*)?|\.\d+)?)([a-zA-Z])(?:\^(\d+))?$'
            match = re.match(pattern, token)
            if match:
                coef_str = match.group(1)
                # If no coefficient is given or only a sign is provided,
                # assume coefficient is 1 (or -1).
                if coef_str in ("", "+", "-"):
                    coef = 1.0 if coef_str != "-" else -1.0
                else:
                    coef = float(coef_str)
                # If no exponent is provided, assume power 1.
                power = int(match.group(5)) if match.group(5) else 1
                # Return a representation where the variable term is in the dictionary
                # and the constant part is zero.
                return [{power: coef}, 0]
            else:
                raise ValueError()
    raise ValueError("Unsupported token type in make_poly.")


def poly_add(self, p1, p2):
    new_dict = {}
    # Add variable parts
    for power, coeff in p1[0].items():
        new_dict[power] = coeff
```

```
    for power, coeff in p2[0].items():
        new_dict[power] = new_dict.get(power, 0) + coeff
    # Add constant parts
    new_const = p1[1] + p2[1]
    return [new_dict, new_const]

def poly_sub(self, p1, p2):
    new_dict = {}
    for power, coeff in p1[0].items():
        new_dict[power] = coeff
    for power, coeff in p2[0].items():
        new_dict[power] = new_dict.get(power, 0) - coeff
    new_const = p1[1] - p2[1]
    return [new_dict, new_const]

def poly_mul(self, p1, p2):
    new_dict = {}
    # Multiply variable parts (dictionary * dictionary)
    for pwr1, coef1 in p1[0].items():
        for pwr2, coef2 in p2[0].items():
            new_power = pwr1 + pwr2
            new_dict[new_power] = new_dict.get(new_power, 0) + coef1 * coef2
    # Multiply variable part of p1 with constant part of p2
    for pwr, coef in p1[0].items():
        new_dict[pwr] = new_dict.get(pwr, 0) + coef * p2[1]
    # Multiply variable part of p2 with constant part of p1
    for pwr, coef in p2[0].items():
        new_dict[pwr] = new_dict.get(pwr, 0) + coef * p1[1]
    new_const = p1[1] * p2[1]
    return [new_dict, new_const]

def poly_div(self, p1, p2):
    """
    Divide p1 by p2.
    Here we support division only when p2 is a constant (i.e. its dictionary is empty).
    """
```

```
    if p2[0]:
        raise ValueError("Division by a non-constant polynomial is not supported")
    if p2[1] == 0:
        raise ZeroDivisionError("Division by zero")
    new_dict = {p: coef / p2[1] for p, coef in p1[0].items()}
    new_const = p1[1] / p2[1]
    return [new_dict, new_const]


def poly_inv(self, p):
    if not p[0]:
        if p[1] == 0:
            raise ZeroDivisionError("Zero polynomial not invertible")
        return [{}, 1 / p[1]]


    if len(p[0]) == 1 and p[1] == 0:
        for q, coef in p[0].items():
            if coef == 0:
                raise ZeroDivisionError("Division by zero")
            return [{-q: 1 / coef}, 0]
    raise ValueError("Polynomial not invertible")


def poly_pow(self, p, n):
    if n < 0:
        inv = self.poly_inv(p)
        return self.poly_pow(inv, -n)
    else:
        result = [{}, 1]
        for _ in range(n):
            result = self.poly_mul(result, p)
        return result




# --- Modified evaluate method ---
def evaluate(self, node=None):


    if node is None:
```

```
        node = self.tree

    leftTree = node.getLeftTree()
    rightTree = node.getRightTree()
    op = node.getKey()

    # If leaf node then convert the token into a polynomial
    if leftTree is None and rightTree is None:
        return self.make_poly(op)

    # Recursively evaluate left and right subtrees
    left_poly = self.evaluate(leftTree)
    right_poly = self.evaluate(rightTree)

    if op == '+':
        return self.poly_add(left_poly, right_poly)
    elif op == '-':
        return self.poly_sub(left_poly, right_poly)
    elif op == '*':
        return self.poly_mul(left_poly, right_poly)
    elif op == '/':
        return self.poly_div(left_poly, right_poly)
    elif op == '**':
        # For exponentiation, the exponent must be a constant.
        if right_poly[0]:
            raise ValueError("Exponent must be a constant")
        exponent = right_poly[1]
        if not float(exponent).is_integer():
            raise ValueError("Exponent must be an integer")
        return self.poly_pow(left_poly, int(exponent))
    else:
        raise ValueError(f"Invalid operator: {op}")

def simplify(self, result):
    power, constant = result
    terms = [
```

```
    #     nothing           |          multiple of 1          |          power 1                                |              not power 1

            f"" if coeff == 0 else f"(x**{exp})" if coeff == 1 else f"({int(coeff) if coeff.is_integer() else coeff}*x)" if exp == 1 else f"(({int(coeff) if coeff.is_integer()
else coeff}*x)**{exp})"

            for exp, coeff in sorted(power.items(), reverse=True)

        ]



        if constant != 0:

            terms = terms + [str(int(constant) if isinstance(constant, float) and constant.is_integer() else constant)] #remove floating point



        def recursion(lst):

            #print(lst)

            if len(lst) == 1:

                return lst[0]

            elif not lst:

                return "0"

            elif lst[0] == "" and len(lst) == 1:

                return "0"

            elif lst[0] == "":

                return lst[-1]

            elif lst[-1] == "":

                return lst[0]

            else:

                return f"({recursion(lst[:-1])}+{lst[-1]})"



        return recursion(terms)



    def inputExpression(self):

        self.exp = input("Please enter the expression you want to evaluate:\n")

        self.tokens = AlgebricTokeniser(self.exp).tokenise()
```

algebricTokensier.py

```
# Created by: CHAN JUN YI (2309347)

from tokeniser import Tokeniser
```

```
class AlgebricTokeniser(Tokeniser):

    def __init__(self, exp):

        self.algebra = exp

        super().__init__(exp)


    def tokenise(self):

        #print(self.expression)

        if self.is_valid_expression():

            self.expression = self.algebra

            number = ""

            i = 0


            while i < len(self.expression):

                char = self.expression[i]


                if char not in "+-*/()" or char == '.':  # Build multi-digit numbers & decimals

                    number += char


                else:

                    if number:  # Store the completed number

                        self.tokens.append(number)

                        number = ""


                    # Check for '**' operator

                    if char == '*' and i + 1 < len(self.expression) and self.expression[i + 1] == '*':

                        self.tokens.append('**')

                        i += 1  # Skip the next '*'


                    elif char in "+-*/()":  # Other operators and parentheses

                        self.tokens.append(char)


                i += 1  # Move to next character


            i = 0

            while i < len(self.tokens):

                # Check if the current element is a negative sign and the previous one is an operator
```

```
            if self.tokens[i] in ['-', '+'] and i - 1 >= 0 and self.tokens[i - 1] in ['+', '-', '*', '/', '(', '**']:
                self.output.append(self.tokens[i] + self.tokens[i + 1])  # Merge '-' and the next number
                i += 2  # Skip the next element, since it's already merged
            else:
                self.output.append(self.tokens[i])
                i += 1

        output = self.output

        self.output = []
        self.tokens = []

        return output
    else:
        return None

def is_valid_expression(self):
    super().algebric_tokenise()
    return super().is_valid_expression()
```

binaryTree.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)

from tree import Tree

class BinaryTree(Tree):
    def __init__(self,key, leftTree = None, rightTree = None):
        super().__init__()
        self.key = key
        self.leftTree = leftTree
        self.rightTree = rightTree

    def setKey(self, key):
        self.key = key
```

```
def getKey(self):
    return self.key

def getLeftTree(self):
    return self.leftTree

def getRightTree(self):
    return self.rightTree

def insertLeft(self, key):
    if self.leftTree == None:
        self.leftTree = BinaryTree(key)
    else:
        t =BinaryTree(key)
        self.leftTree , t.leftTree = t, self.leftTree

def insertRight(self, key):
    if self.rightTree == None:
        self.rightTree = BinaryTree(key)
    else:
        t =BinaryTree(key)
        self.rightTree , t.rightTree = t, self.rightTree

def printPreorder(self, level):
    print( str(level*'-') + str(self.key))
    if self.leftTree != None:
        self.leftTree.printPreorder(level+1)
    if self.rightTree != None:
        self.rightTree.printPreorder(level+1)

def printInorder(self, level, leafStack):
    if self.leftTree != None:
        self.leftTree.printInorder(level+1, leafStack)
    print( str(level*'-') + str(self.key))
    leafStack.append([self.key, level])
```

```
        if self.rightTree != None:
            self.rightTree.printInorder(level+1, leafStack)
        self.myStack = leafStack

    def printPostorder(self, level):
        if self.leftTree != None:
            self.leftTree.printPostorder(level+1)
        if self.rightTree != None:
            self.rightTree.printPostorder(level+1)
        print( str(level*'-') + str(self.key))

    def stackInorder(self, level, leafStack):
        if self.leftTree != None:
            self.leftTree.stackInorder(level+1, leafStack)
        leafStack.append([self.key, level])
        if self.rightTree != None:
            self.rightTree.stackInorder(level+1, leafStack)
        self.myStack = leafStack
```

buildParseTree.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


from stack import Stack
from tokeniser import Tokeniser
from expressionTree import ExpressionTree
from binaryTree import BinaryTree


class BuildParseTree:
    def __init__(self):
        self.stack = Stack()
        self.tree = BinaryTree('?')
        self.exp = ""
        self.tokens = []
```

```
def build(self):
    #print(self.tokens)
    if self.tokens == None or self.tokens == []:
        print("\nError: Invalid expression")
        return None
    else:
        self.stack.push(self.tree)
        currentTree = self.tree

        #print("Initial Stack:")
        #print(self.stack.getValues())  # Display initial stack

        for t in self.tokens:
            # RULE 1: If token is '(' add a new node as left child
            # and descend into that node
            if t == '(':
                currentTree.insertLeft('?')
                self.stack.push(currentTree)
                currentTree = currentTree.getLeftTree()

            # RULE 2: If token is operator, set key of current node,
            # add a new right child, and move to that node
            elif t in {'+', '-', '*', '/', '**'}:
                currentTree.setKey(t)
                currentTree.insertRight('?')
                self.stack.push(currentTree)
                currentTree = currentTree.getRightTree()

            # RULE 3: If token is a number, set key of current node
            # to that number and return to parent
            elif t not in {'+', '-', '*', '/', ')', '**'}:
                try:
                    currentTree.setKey(float(t))
                    currentTree = self.stack.pop()
                except ValueError:
                    raise ValueError(f"Invalid token: {t}")
```

```
            # RULE 4: If token is ')', go back to parent node
            elif t == ')':
                if not self.stack.isEmpty():
                    currentTree = self.stack.pop()
        return self.tree


def printTree(self):
    self.tree.stackInorder(0, [])
    #print("mystack:")
    #print(mytree.tree.myStack)


    self.expressionTree = ExpressionTree(self.tree)
    self.expressionTree.printExpressionTree()


def evaluate(self, node=None):


    if node is None:
        node = self.tree
    leftTree = node.getLeftTree()
    rightTree = node.getRightTree()
    op = node.getKey()


    if leftTree is None and rightTree is None:
        return op


    left_val = self.evaluate(leftTree)
    right_val = self.evaluate(rightTree)


    # Perform operation based on operator
    if op == '+':
        return left_val + right_val
    elif op == '-':
        return left_val - right_val
    elif op == '*':
        return left_val * right_val
```

```
        elif op == '/':
            if right_val == 0:
                print("Error: Division by zero")
                return "?"
            return left_val / right_val
        elif op == '**':
            return left_val ** right_val
        else:
            raise ValueError(f"Invalid operator: {op}")


    def inputExpression(self):
        self.exp = input("Please enter the expression you want to evaluate:\n")
        self.tokens = Tokeniser(self.exp).tokenise()
        #print(self.tokens)
```

Division.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


from operation import BaseOperation
class DivideByTwo(BaseOperation):
    def apply(self, num):
        return num // 2 if num % 2 == 0 else None
```

expressionPathFinder.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


import networkx as nx
from fileOutput import OutputFile
from operation import BaseOperation
from addition import AddOne
from subtract import SubtractOne
from multiplication import MultiplyByTwo
```

```
from division import DivideByTwo

from power import Square


class NumberPathFinder():

    def __init__(self):

        self.graph = nx.Graph()  # Convert to undirected graph for MST

        self.operations = {

            "+1": AddOne(),

            "-1": SubtractOne(),

            "*2": MultiplyByTwo(),

            "/2": DivideByTwo(),

            "**2": Square()

        }

        self.expressions = []  # Store expressions

        self.__outputfile = None # Output file object


    def apply_operation(self, num, op):

        if op in self.operations:

            return self.operations[op].apply(num)

        return None


    def build_graph(self, start, max_steps=40, max_num=9999):

        """

        Build a graph connecting numbers using allowed operations.

        This is an undirected graph for MST calculation.

        """

        self.graph.clear()

        queue = [(start, 0, "")]  # Track (current number, steps, expression)

        visited = {}

        self.graph.add_node(start)


        while queue:

            current, steps, expr = queue.pop(0)


            if current in visited and visited[current] <= steps:

                continue
```

```
        visited[current] = steps

        if current > max_num:
            continue

        for operation in self.operations:
            new_num = self.apply_operation(current, operation)
            if new_num is not None and new_num >= 0 and new_num not in visited:
                new_expr = f"({expr}{operation})" if expr else f"({current}{operation})"
                self.graph.add_edge(current, new_num, weight=1)  # Add undirected edges
                queue.append((new_num, steps + 1, new_expr))

def find_mst_path(self, start, target):
    """
    Find the shortest path using Minimum Spanning Tree (MST).
    """
    if start not in self.graph or target not in self.graph:
        print(f"\nNo path found from {start} to {target}. Please select different numbers or update operations.")
        return None, 0, ""

    # Compute MST using Kruskal's algorithm
    mst = nx.minimum_spanning_tree(self.graph, algorithm='kruskal')

    if not nx.has_path(mst, start, target):
        print(f"\nNo path found from {start} to {target} in the MST.")
        return None, 0, ""

    # Find the path from start to target in the MST
    path = nx.shortest_path(mst, source=start, target=target)
    steps = len(path) - 1

    if steps == 0:
        print(f"Start and target numbers are not allowed to be the same. Please enter different numbers.")
        return None, 0, ""

    # Generate expression from path
```

```
    expression = str(path[0])
    for i in range(len(path) - 1):
        for op in self.operations:
            if self.apply_operation(path[i], op) == path[i + 1]:
                expression = f"({expression}{op})"
                break

    self.expressions.append(expression)
    return path, steps, expression

def main_menu(self):
    print("\nWelcome to Minimum Expression Path Finder")

    while True:
        print("\nPlease select your choice ('1', '2', '3', '4'):")
        print("1) Enter new start & target numbers (max: 9999)")
        print("2) Change allowed operations")
        print("3) View stored expressions")
        print("4) Exit program")

        choice = input("Select an option: ").strip()

        if choice == "1":
            start, target = self.get_numbers()
            self.build_graph(start)
            mst_path, steps, expr = self.find_mst_path(start, target)
            if mst_path:
                print(f"Shortest path from {start} to {target}: {mst_path} (Steps: {steps})")
                print(f"Expression: {expr}")

        elif choice == "2":
            self.set_operations()

        elif choice == "3":
            print("\nStored Expressions:")
            for expr in self.expressions:
```

```
            print(f"{expr}")


        # Prompt user to save expressions to a file
        save_choice = input("\nDo you want to save these expressions to a text file? (y/n): ").strip().lower()
        while save_choice not in ["y", "n"]:
            print("Invalid input! Please enter 'y' or 'n'")
            save_choice = input("Do you want to save these expressions to a text file? (y/n): ").strip().lower()


        if save_choice == "y":
            self.__outputfile = OutputFile()
            content = "\n".join(self.expressions)
            self.__outputfile.send_file(content)
            print(f"\nExpressions saved to {self.__outputfile.get_output_file_name()}.")
        else:
            print("\nSave operation cancelled.")


    elif choice == "4":
        print("\nExiting minimum expression path finder (MST Version).")
        break
    else:
        print("\nPlease choose from 1 to 4 only")


def get_numbers(self):
    while True:
        try:
            start = int(input("\nEnter the starting number: "))
            target = int(input("Enter the target number: "))
            return start, target
        except ValueError:
            print("Invalid input! Please enter whole numbers.")


def set_operations(self):
    print("\nCurrent operations:", list(self.operations.keys()))
    print("Operations allow to add or remove are (+1 -1 *2 /2 **2)")
    new_operations = input("Enter new operations separated by spaces (e.g., +1 -1 *2): ").strip().split()
```

```
        # Map of valid operations to their corresponding objects
        operation_classes = {
            "+1": AddOne(),
            "-1": SubtractOne(),
            "*2": MultiplyByTwo(),
            "/2": DivideByTwo(),
            "**2": Square()
        }

        # Validate operations
        valid_operations = {}
        for op in new_operations:
            if op in operation_classes:
                valid_operations[op] = operation_classes[op]
            else:
                print(f"Invalid operation '{op}' ignored.")

        if valid_operations:
            self.operations = valid_operations
            print("\nOperations updated successfully")
            print("Current operations:", list(self.operations.keys()))
        else:
            print("\nNo valid operations provided. Operations remain unchanged.")
            print("Current operations:", list(self.operations.keys()))
```

expressionTree.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


from tree import Tree


class ExpressionTree(Tree):
    def __init__(self, tree):
        super().__init__()
```

```
    self.tree = tree


def printTree(self):
    self.printExpressionTree()


def printExpressionTree(self):
    self.data = self.tree.myStack
    processed_array_within = self.split_elements_within_array()
    highest_row = self.find_highest_row(processed_array_within)

    self.row = highest_row + 1
    self.col = len(self.tree.myStack)

    multiplication_grid = self.create_multiplication_grid(self.row, self.col)

    for col_index, items in enumerate(processed_array_within):
        for value, row_index in items:
            multiplication_grid[row_index][col_index] = value

    grid_string = '\n'.join([''.join(row) for row in multiplication_grid])
    print(grid_string)


#for printing expression tree
def split_elements_within_array(self):
    result = []
    for item in self.data:
        value, index = item
        # Convert value to string and split into individual characters
        #removing floating point if it is an integer
        if isinstance(value, (int, float)):
            if value == int(value):
                value = int(value)
        value_str = str(value)
        split_subarray = []
        for i, char in enumerate(value_str):
            split_subarray.append([char, index + i])
```

```
            result.append(split_subarray)
        return result


    def find_highest_row(self, data):
        max_index = float('-inf')
        for subarray in data:
            for element in subarray:
                _, index = element
                if index > max_index:
                    max_index = index
        return max_index


    def create_multiplication_grid(self, rows, columns):
        grid = []
        for row in range(rows):
            grid_row = []
            for col in range(columns):
                grid_row.append(' ')
            grid.append(grid_row)
        return grid
```

fileHanlding.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


import os


# Read input file
class ReadFile:
    def __init__(self, option):
        self.__option = option
        self.__file_content = None
        self.__file_name = None
        self.read_file()
```

```
def read_file(self):
    while True:
        file_name = input("\nPlease enter input file: ")

        # Check if file exists
        if os.path.exists(file_name):
            try:
                with open(file_name, "r") as file:
                    content = file.read()

                if self._validate_content(content):
                    self.__file_content = content
                    self.__file_name = file_name
                    return

            except IOError:
                print("Error reading the file. Please resend the file again.")
        else:
            print("File does not exist. Please resend the file again.")

def _validate_content(self, content):

    for letter in content:
        if letter not in set("1234567890.+-*/() \n"):
            print("\nFile contains invalid content. Please resend the file.")
            return False
    return True

def valid_content_symbol(self, content):
    for letter in content:
        # Only * and . character are allow. However, newline or empty space should be allow too
        if letter not in ['+', '-', '*', '/', ')', '**', " ", "\n"]:
            return False
    return True

def valid_content_letter(self, content):
```

```
        # Only allow numbers. However, newline or empty space should be allow too
        for letter in content:
            if letter not in "1234567890 \n":
                return False
        return True


    def get_content(self):
        return self.__file_content # Get the file content safely


    def get_filename(self):
        return self.__file_name  # Return the file content safely
```

fileOutput.py

```
# Created by: IVAN TAY YUEN HENG (2335133)
# Read output file
class OutputFile:
    def __init__(self):


        self.__outputfile = self.get_filename()


    def get_filename(self):
        while True:
            # Ask user for output file
            output = input("Please enter output file: ")


            # Check if the output file ends with .txt
            if output.endswith(".txt"):
                return output
            else:
                print("Output file can only end with txt. Please try again.")


    def send_file(self, content):


        # Write the content to the output file
```

```
        if self.__outputfile:
            try:
                with open(self.__outputfile, "w") as file:
                    file.write(content)
            except IOError:
                print("Error creating output file.")
        else:
            print("No valid output file")


    # Get the output file namem
    def get_output_file_name(self):
        return self.__outputfile
```

GUI.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)
from buildParseTree import BuildParseTree
from fileHandling import ReadFile
from fileOutput import OutputFile
from sortExpression import SortExpressions
from expressionPathFinder import NumberPathFinder
from randomExpressionGenerator import RandomExpressionGenerator
from history import History
from algebricEquation import AlgebricEquation
from tokeniser import Tokeniser


class GUI:
    def __init__(self):
        self.history = History()
        self.title_bar()


    def title_bar(self):
        # Show the title bar
```

```
        print("""
*************************************************************************
* ST1507 DSAA: Expression Evaluator & Sorter                            *
* ----------------------------------------------------------------------*
*                                                                       *
*     - Done by: IVAN TAY YUEN HENG (2335133) & CHAN JUN YI (2309347)   *
*     - Class DAAA/2A/21                                                *
*                                                                       *
*************************************************************************
""")
        self.cont()


    def cont(self):
        input("\nPress Enter, to continue....\n")
        self.run()


    def run(self):
        while True:
            self.menu()
            option = input("Enter choice: ")
            self.option_handling(option)


    def menu(self):
        # Selection menu
        print("""Please select your choice ('1', '2', '3', '4', '5', '6', '7'):
        1. Evaluate expression
        2. Sort expressions
        3. Minimum expression path finder (Ivan Tay)
        4. Random expression generator (Ivan Tay)
        5. Expression history (Chan Jun Yi)
        6. Simplify algebric expression (Chan Jun Yi)
        7. Exit""")


    def option_handling(self, option):
        match option:
            case '1':
```

```
            self.evaluate_expression_choice()
        case '2':
            self.sort_expressions()
        case '3':
            self.expression_path_finder()
        case '4':
            self.random_expression_generator()
        case '5':
            self.expresssion_history()
        case '6':
            self.algebric_equation()
        case '7':
            self.exit_program()
        case _:
            print("\nPlease choose from 1 to 7 only")


def evaluate_expression_choice(self):
    mytree = BuildParseTree()
    mytree.inputExpression()
    mytree.build()
    result = mytree.evaluate()
    if result != "?":
        if isinstance(result, float) and result.is_integer():
            result = int(result)
        print(f"\nExpression Tree:")
        mytree.printTree()
        print(f"\nExpression evaluates to: \n{result}")
        self.history.add(mytree.tokens, result)


    self.cont()


def sort_expressions(self):
    file_content = ReadFile(1).get_content()  # Read file content
    output_file = OutputFile()  # Create an OutputFile object


    if file_content:  # Check if content is retrieved
```

```
            sorter = SortExpressions(file_content)  # Sort expressions
            success = sorter.sort_expressions()  # Sort expressions
            if success != False:
                print("\n>>>Evaluation and sorting started:\n")
                print(sorter.sortedList)
                output_file.send_file(str(sorter.sortedList))  # Write to output file
            else:
                print("Sorting failed due to invalid expressions. Output file was not created.")

            output_file.send_file(str(sorter.sortedList))  # Write to output file
        else:
            print("Error: Unable to read file content.")

        self.cont()

    def expression_path_finder(self):
        path_finder = NumberPathFinder()
        path_finder.main_menu()
        self.cont()

    def random_expression_generator(self):
        generator = RandomExpressionGenerator()
        generator.expression_sub_menu()
        self.cont()

    def expresssion_history(self):
        isHistory = self.history.showLast5()
        if isHistory:
            mytree = BuildParseTree()
            print()
            mytree.exp = self.history.editFromHistory()
            #print(mytree.exp)
            if mytree.exp != None:
                mytree.tokens = Tokeniser(mytree.exp).tokenise()
                #print(mytree.tokens)
                mytree.build()
```

```
            result = mytree.evaluate()
            if result != "?":
                if isinstance(result, float) and result.is_integer():
                    result = int(result)
                print(f"\nExpression Tree:")
                mytree.printTree()
                print(f"\nExpression evaluates to: \n{result}")
                self.history.add(mytree.tokens, result)

        self.cont()

    def algebric_equation(self):
        mytree = AlgebricEquation()
        mytree.inputExpression()
        mytree.build()
        try:
            result = mytree.evaluate()
            print(f"Simplified expression: {mytree.simplify(result)}")
        except Exception as e:
            print(e)

        self.cont()

    def exit_program(self):
        # Option 7: Exit the program.
        print("\nBye, thanks for using ST1507 DSAA: Expression Evaluator & Sorter")
        exit()

# Run the program
if __name__ == "__main__":
    gui = GUI()
```

History.py

```
# Created by: CHAN JUN YI (2309347)
from partialTokeniser import partialTokeniser


class History:
    def __init__(self):
        self.__history = []
        self.edit_tokens = []

    def add(self, tokens, result):
        self.__history.append((tokens, result))

    def showLast5(self):
        print()
        if not self.__history:
            print("\nNo history to show.")
            return False

        for i, (tokens, result) in enumerate(self.__history[-5:]):
            print(f"{min(5, len(self.__history)) - i}: {''.join(tokens)} = {result}")

        return True

    def editFromHistory(self):

        #choose from hisory
        if not self.__history:
            print("\nNo history to edit.")
            return None
        inputChoice = self.inputChoice()

        #exit history
        if inputChoice == 0:
            return

        #edit from  history
        choice = self.__history[-inputChoice]
```

```
    self.edit_tokens = self.editTokens(choice[0])
    #print(self.edit_tokens)

    #display editting
    for i in range(1, len(self.edit_tokens), 2):
        display = self.edit_tokens.copy()
        editting = "_" * len(self.edit_tokens[i])
        display[i] = editting
        print("".join(display))
        self.updateToken(i)
        print(''.join(self.edit_tokens))

    return ''.join(self.edit_tokens)

def editTokens(self, tokens):
    output = [' ']
    for token in tokens:
        #is bracket and previous token is bracket
        if token in ['(', ')'] and output[-1][-1] in ['(', ')'] or  output[-1] == '':
            output[-1] = output[-1] + token
        #is bracket and previous token is number or operator
        elif token in ['(', ')']:
            output.append(token)
        #is number and previous token is bracket
        elif token.isdigit() and output[-1][-1] in ['(', ')']:
            output.append(token)
        #is number and previous token is operator
        elif token.isdigit():
            output[-1] = output[-1] + token
        #is operator and previous token is number or operator
        elif output[-1][-1].isdigit() or output[-1][-1] in ['+', '-', '*', '/', '*']:
            output[-1] = output[-1] + token
        #is operator and previous token is bracket
        else:
            output.append(token)
    #drop first space
```

```
        output = output[1:]

        return output


    def updateToken(self, i):

            invalidFormat = True

            while invalidFormat:

                updated_tokens = input(f"Enter the updated expression this format - {' '.join(['op.' if format == None else 'no.' for format in
self.getTokenFormat(self.edit_tokens[i])])}: ")

                updated_tokens = updated_tokens.replace('(', '').replace(')', '')

                if updated_tokens != '':

                    if self.getTokenFormat(updated_tokens) == self.getTokenFormat(self.edit_tokens[i]):

                        self.edit_tokens[i] = updated_tokens

                        invalidFormat = False

                    else:

                        print("Invalid format. Please enter a valid expression.")

                else:

                    return


    def getTokenFormat(self, tokens):

        #print()

        #print(tokens)

        tokens = partialTokeniser(tokens).partial_tokenise()

        #print(tokens)

        format = []

        for i in range(len(tokens)):

            #0 if digit, else operator

            try:

                format.append(float(tokens[i])-float(tokens[i]))

            except:

                format.append(None)

        #print(format)

        return format


    def inputChoice(self):

        try:

            index = int(input("Please enter the number of the expression history you want to edit, or 0 to exit:"))
```

```
            #not in history
            if index < 0 or index > min(len(self.__history), 5):
                print("Invalid index. Please enter a number between 0 and", min(len(self.__history), 5))
                return self.inputChoice()
            return index
        except ValueError:
            print("Invalid index. Please enter a number between 0 and", min(len(self.__history), 5))
            return self.inputChoice()
```

Main.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


from GUI import GUI


if __name__ == "__main__":
    gui = GUI()
```

Multiplication.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


from operation import BaseOperation


class MultiplyByTwo(BaseOperation):
    def apply(self, num):
        return num * 2
```

Operation.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


# Base class for operations
class BaseOperation:
    def apply(self, num):
```

```
        pass
```

partialTokeniser.py

```
# Created by: Chan Jun Yi (2309347)

class partialTokeniser:

    def __init__(self, expression):

        self.expression = expression

        self.tokens = []

        self.output = []


    def partial_tokenise(self):

        number = ""

        i = 0


        while i < len(self.expression):

            char = self.expression[i]


            if char.isdigit() or char == '.':  # Build multi-digit numbers & decimals

                number += char


            else:

                if number:  # Store the completed number

                    self.tokens.append(number)

                    number = ""


                # Check for '**' operator

                if char == '*' and i + 1 < len(self.expression) and self.expression[i + 1] == '*':

                    self.tokens.append('**')

                    i += 1  # Skip the next '*'


                elif char in "+-*/":  # Other operators and parentheses

                    self.tokens.append(char)
```

```
            i += 1  # Move to next character
        if number != "":
            self.tokens.append(number)

        i = 0
        while i < len(self.tokens):
            # Check if there is a next element
            if i + 1 < len(self.tokens):
                # Check if the current element is a negative sign and the previous one is an operator
                if self.tokens[i] in ['-', '+'] and ( (i - 1 >= 0 and self.tokens[i - 1] in ['+', '-', '*', '/', '**']) or i == 0):
                    self.output.append(self.tokens[i] + self.tokens[i + 1])  # Merge '-' and the next number
                    i += 2  # Skip the next element, since it's already merged
                else:
                    self.output.append(self.tokens[i])
                    i += 1
            else:
                self.output.append(self.tokens[i])
                i += 1

        output = self.output

        self.output = []
        self.tokens = []

        return output
```

Power.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


from operation import BaseOperation
class Square(BaseOperation):
    def apply(self, num):
        return num ** 2
```

randomExpressionGenerator.py

```
# Created by: IVAN TAY YUEN HENG (2335133)

import random

from fileOutput import OutputFile  # Import file output handler

from buildParseTree import BuildParseTree  # Import expression evaluator


class RandomExpressionGenerator:

    def __init__(self):

        self.operators = ['+', '-', '*', '/', '**']

        self.correct_answers = 0

        self.total_attempts = 0

        self.current_expression = None

        self.current_answer = None

        self.current_difficulty = None

        self.content = ""  # Variable to store all generated expressions


    def generate_expression(self, difficulty):

        """Generates a new random expression based on difficulty."""

        if difficulty == 1:

            num_operators = 1

            value_range = (1, 10)

        elif difficulty == 2:

            num_operators = random.randint(2, 3)

            value_range = (1, 20)

        elif difficulty == 3:

            num_operators = random.randint(4, 5)

            value_range = (1, 100)

        else:

            raise ValueError("Invalid difficulty level.")


        while True:

            expression = ""

            for _ in range(num_operators + 1):
```

```
            num = str(random.randint(*value_range))
            if not expression:
                expression = num
            else:
                operator = random.choice(self.operators)
                expression = f'({expression} {operator} {num})'

        try:
            result = self.evaluate_expression(expression)
            if isinstance(result, (int, float)) and -9999 <= result <= 9999:
                self.current_expression = expression
                self.current_answer = result
                self.current_difficulty = difficulty
                # Append the generated expression to the content variable
                self.content += f"{expression}\n"
                return expression, result
        except OverflowError:
            continue

def evaluate_expression(self, expression):
    """Evaluates the mathematical expression."""
    try:
        result = eval(expression)
        return round(result, 2)
    except ZeroDivisionError:
        return "Undefined (Division by Zero)"
    except OverflowError:
        return "Overflow Error"

def expression_sub_menu(self):
    """Sub-menu for generating or answering an expression."""
    while True:
        print("\n===== Random Expression Generator =====")
        print(f"Score: {self.correct_answers}/{self.total_attempts} correct")

        print("\nGenerated Expression:", self.current_expression if self.current_expression else "None")
```

```
        print("\n1) Generate Expression (Easy, Medium, Hard)")
        print("2) Answer the expression")
        print("3) Save all generated expressions to file")
        print("4) Use evaluate expression")
        print("5) Return to GUI")
        choice = input("Select an option: ")

        if choice == '1':
            difficulty = self.select_difficulty()
            self.generate_expression(difficulty)
        elif choice == '2' and self.current_expression:
            self.answer_expression()
        elif choice == '3':
            self.save_expressions_to_file()
        elif choice == '4':
            self.evaluate_expression_choice()
        elif choice == '5':
            print("\nReturning to GUI...")
            return  # Exit loop, returning control to GUI
        else:
            print("Invalid choice. Please try again.")

    def select_difficulty(self):
        """Prompts the user to select a difficulty level."""
        while True:
            print("\nSelect Difficulty Level:")
            print("1. Easy")
            print("2. Medium")
            print("3. Hard")
            choice = input("Enter choice: ")
            if choice in ['1', '2', '3']:
                return int(choice)
            else:
                print("Invalid choice. Please select 1, 2, or 3.")

    def answer_expression(self):
```

```
    """Handles user answering the expression."""
    while True:
        if self.current_expression == None:
            print("No expression available to answer. Please generate an expression first.")
            return



        user_input = input("Enter your answer (2 decimal points): ")


        try:
            user_answer = round(float(user_input), 2)
            self.total_attempts += 1


            if user_answer == self.current_answer:
                self.correct_answers += 1
                print("☑ Correct!")
            else:
                print(f"❌ Incorrect! The correct answer was: {self.current_answer}")


            # Reset current expression after answering
            self.current_expression = None
            self.current_answer = None
            self.current_difficulty = None
            break


        except ValueError:
            print("Invalid input! Please enter a numerical answer.")


def save_expressions_to_file(self):
    """Saves all generated expressions to a file."""
    if not self.content:
        print("No expressions available to save.")
        return


    print("\nGenerated Expressions:")
```

```
        print(self.content)


        output_handler = OutputFile()
        output_handler.send_file(self.content)
        print(f"☑ Expressions saved to {output_handler.get_output_file_name()}")


    def evaluate_expression_choice(self):
        mytree = BuildParseTree()
        mytree.inputExpression()
        mytree.build()
        result = mytree.evaluate()
        if result != "?":
            if isinstance(result, float) and result.is_integer():
                result = int(result)
            print(f"\nExpression Tree:")
            mytree.printTree()
            print(f"\nExpression evaluates to: \n{result}")
```

sortedList.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


class SortedList:
    def __init__(self):
        self.headNode = None
        self.currentNode = None
        self.length = 0


    def __appendToHead(self, newNode):
        oldHeadNode = self.headNode
        self.headNode = newNode
        self.headNode.nextNode = oldHeadNode


    def insert(self, newNode):
```

```
        # If list is currently empty
        if self.headNode == None:
            self.headNode = newNode
            return

        # Check if it is going to be new head
        if newNode < self.headNode:
            self.__appendToHead(newNode)
            return

        # Traverse and insert at appropriate location
        node = self.headNode

        while node.nextNode != None and not (newNode < node.nextNode):
            node = node.nextNode

        newNode.nextNode = node.nextNode
        node.nextNode = newNode



    def __str__(self):
        group = {}
        node = self.headNode

        while node != None:
            value = node.data[1]
            if value not in group:
                group[value] = []
            group[value].append(node.data[0])
            node = node.nextNode

        # Output
        output = []

        for key in sorted(group.keys(), reverse=True):
            output.append(f"*** Expressions with value= {key}")
```

```
            for value in group[key]:
                output.append(f"{value}==>{key}")

            output.append("")

        return "\n".join(output)
```

sortedNode.py

```
# Created by IVAN TAY YUEN HENG (2335133)
class Node:
    def __init__(self, data):
        self.data = data
        self.nextNode = None

    def __lt__(self, other):
        if self.data[1] != other.data[1]:
            return self.data[1] > other.data[1]
        if self.data[0] != other.data[0]:
            return len(self.data[0]) < len(other.data[0])
        return self.data[0].count('(') + self.data[0].count(')') < other.data[0].count('(') + other.data[0].count(')')

    def __str__(self):
        return str(self.data)
```

sortExpression.py

```
# Created by IVAN TAY YUEN HENG (2335133)
from buildParseTree import BuildParseTree
from tokeniser import Tokeniser
from sortedList import SortedList
from sortedNode import Node


class SortExpressions():
    def __init__(self, file_content):
        self.file_content = file_content
```

```
    self.sortedList = SortedList()
    self.expressions = None
    self.evaluatedResults = None

# def split_expressions(self):
def sort_expressions(self):

    self.expressions = self.file_content.splitlines()
    self.evaluatedResults = self.evaluateExpressions()

    for expr, result in self.evaluatedResults:
        if result == '?':
            # Skip invalid expressions
            return False
        else:
            node = Node((expr, result))  # Create a Node with (expression, result)
            self.sortedList.insert(node)

    # print(self.sortedList)

def evaluateExpressions(self):
    results = []
    for exp in self.expressions:
        try:
            # Remove spaces
            exp = exp.replace(" ", "")

            # Build parse tree
            mytree = BuildParseTree()

            mytree.tokens = Tokeniser(exp).tokenise()
            mytree.build()
            # Evaluate the parse tree
            result = mytree.evaluate()

            # Change integer results to int
```

```
            if isinstance(result, float) and result.is_integer():
                result = int(result)

            # Append the expression and result as a tuple
            results.append((exp, result))

        except Exception as e:
            # Handle invalid expressions
            results.append((exp, f"Error: {e}"))
    return results

def __str__(self):
    return str(self.sortedList)
```

Stack.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


class Stack:
    def __init__(self):
        self.__list = []

    def isEmpty(self):
        return self.__list == []

    def push(self, item):
        self.__list.append(item)

    def pop(self):
        if self.isEmpty():
            return None
        else:
            return self.__list.pop()

    def clear(self):
```

```
        self.__list.clear()


    def peek(self):
        if self.isEmpty():
            return None
        else:
            return self.__list[len(self.__list)-1]


    def size(self):
        return len(self.__list)


    def get(self):
        if self.isEmpty():
            return None
        else:
            return self.__list[-1]


    def getValues(self):
        return [self._getNodeRepresentation(node) for node in self.__list]


    def _childKey(self, child):
        return child.getKey() if child is not None else "None"
```

Subtract.py

```
# Created by: IVAN TAY YUEN HENG (2335133)


from operation import BaseOperation


class SubtractOne(BaseOperation):
    def apply(self, num):
        return num - 1
```

Tokeniser.py

```
# Created by: IVAN TAY YUEN HENG (2335133) and CHAN JUN YI (2309347)


class Tokeniser:
    def __init__(self, expression):
        self.expression = expression
        self.tokens = []
        self.output = []

    def tokenise(self):
        #print(self.expression)
        if self.is_valid_expression():
            number = ""
            i = 0

            while i < len(self.expression):
                char = self.expression[i]

                if char.isdigit() or char == '.':  # Build multi-digit numbers & decimals
                    number += char

                else:
                    if number:  # Store the completed number
                        self.tokens.append(number)
                        number = ""

                    # Check for '**' operator
                    if char == '*' and i + 1 < len(self.expression) and self.expression[i + 1] == '*':
                        self.tokens.append('**')
                        i += 1  # Skip the next '*'

                    elif char in "+-*/()":  # Other operators and parentheses
                        self.tokens.append(char)

                i += 1  # Move to next character
```

```
        i = 0
        while i < len(self.tokens):
            # Check if the current element is a negative sign and the previous one is an operator
            if self.tokens[i] in ['-', '+'] and i - 1 >= 0 and self.tokens[i - 1] in ['+', '-', '*', '/', '(', '**']:
                self.output.append(self.tokens[i] + self.tokens[i + 1])  # Merge '-' and the next number
                i += 2  # Skip the next element, since it's already merged
            else:
                self.output.append(self.tokens[i])
                i += 1

        output = self.output

        self.output = []
        self.tokens = []

        return output
    else:
        return None

def is_valid_expression(self):
    stack = []  # Stack to store bracketed expressions
    i = 0
    n = len(self.expression)
    validation = 0
    while i < n:
        # Make sure everything must be under one bracket first
        if stack == []:
            validation += 1

        # If all of the expression is not under 1 bracket, return False
        if validation > 1:
            return False

        char = self.expression[i]
        if char == '(':
            stack.append([])  # Start a new bracketed expression
```

```
        elif char == ')':
            if not stack or not stack[-1]:  # Check for misplaced brackets
                return False
            sub_expr = stack.pop()  # Get the last bracketed expression
            if not self.is_valid_subexpression(sub_expr):  # Validate its contents
                return False
            if stack:
                stack[-1].append('N')  # Represent valid nested expression as 'N'
        elif char.isdigit() or char == '.':  # Check for numbers (including floating points)
            num = char
            decimal_seen = char == '.'  # Track if we see a decimal point
            while i + 1 < n and (self.expression[i + 1].isdigit() or (self.expression[i + 1] == '.' and not decimal_seen)):
                i += 1
                if self.expression[i] == '.':
                    decimal_seen = True
                num += self.expression[i]
            if num.count('.') > 1:  # More than one decimal in a number is invalid
                return False
            if stack:
                stack[-1].append(num)
        elif char in '+-*/':  # Check for operators
            # Handle exponentiation `**` as a single operator
            if char == '*' and i + 1 < n and self.expression[i + 1] == '*':
                char = '**'
                i += 1
            # Handle negative numbers
            if char == '-' and (i == 0 or self.expression[i - 1] == '(' or self.expression[i - 1] in '+-*/'):
                num = char
                while i + 1 < n and (self.expression[i + 1].isdigit() or self.expression[i + 1] == '.'):
                    i += 1
                    num += self.expression[i]
                if num.count('.') > 1:  # More than one decimal in a number is invalid
                    return False
                if stack:
                    stack[-1].append(num)
            else:
```

```
                if stack:
                    stack[-1].append(char)
            elif char == ' ':
                pass  # Ignore spaces
            else:
                return False  # Invalid character

            i += 1

        return not stack  # Stack should be empty if all brackets are properly closed

    def algebric_tokenise(self):
        #print(self.expression)
        expression = self.expression
        self.expression = ''.join(['1' if char.isalpha() else str(char) for char in expression])
        #print(self.expression)
        return self.expression

    def is_valid_subexpression(self, sub_expr):
        """
        Checks if a bracketed expression is valid:
        - Either contains exactly [operand, operator, operand]
        - OR contains a valid nested expression.
        """
        if len(sub_expr) == 3 and self.is_number(sub_expr[0]) and sub_expr[1] in ['+', '-', '*', '/', '**'] and self.is_number(sub_expr[2]):
            return True
        elif len(sub_expr) == 3 and sub_expr[0] == 'N' and sub_expr[1] in ['+', '-', '*', '/', '**'] and self.is_number(sub_expr[2]):
            return True
        elif len(sub_expr) == 3 and self.is_number(sub_expr[0]) and sub_expr[1] in ['+', '-', '*', '/', '**'] and sub_expr[2] == 'N':
            return True
        elif len(sub_expr) == 3 and sub_expr[0] == 'N' and sub_expr[1] in ['+', '-', '*', '/', '**'] and sub_expr[2] == 'N':
            return True
        return False

    def is_number(self, value):
        """ Helper function to check if a string represents a valid integer or float """
```

```
        try:
            float(value)  # Works for both integers and floating point numbers
            return True
        except ValueError:
            return False
```

Tree.py

```
# Created by: IVAN TAY YUEN HENG (2335133), CHAN JUN YI (2309347)

class Tree:
    def __init__(self):
        self.myStack = []


    def printTree(self):
        raise NotImplementedError("Subclasses must implement this method")
```