Create Task

2a. My program is named Expression Converter. It is written in Java. The purpose of the program is to take an infix-notation mathematical expression and convert it to both prefix and postfix notation, as well as solve it. This was inspired by my computer science class, where we learned to convert these expressions by hand, which can be tedious. My video shows the program converting expressions of various lengths and including various operations and numbers of different sizes.

2b. One difficulty when developing my program was figuring out how to parse the infix expression into the binary tree. At first thought, I wanted to parse through using the normal order of operations I learned in school: multiplication and division, then addition and subtraction, from left to right. Due to the nature of the binary tree, however, I eventually found that the solution involved the exact opposite. First, I had to find all the addition and subtraction operations from right to left, then I found the multiplication and division operations.

One opportunity I encountered was the ease of which I could generate the prefix and postfix expressions once the binary tree was formed. Through a simple recursive preorder and postorder traversal of the tree, I could easily generate both forms of the expression.

Both of these development points, like my entire program, were developed solely by myself.

2c.

```
private BinaryTreeNode createBinaryTree(List<String> stubs)
{
        if(stubs.size() == 1)
            return new BinaryTreeNode(stubs.get(0));

        int rootIndex = determineRootNodeIndex(stubs);
        List<String> leftStubs = stubs.subList(0, rootIndex);
        List<String> rightStubs = stubs.subList(rootIndex + 1, stubs.size());

        BinaryTreeNode rootNode = new OperatorNode(stubs.get(rootIndex));
        BinaryTreeNode leftChild = createBinaryTree(leftStubs);
        BinaryTreeNode rightChild = createBinaryTree(rightStubs);
        rootNode.setLeftChild(leftChild);
        rootNode.setRightChild(rightChild);

        return rootNode;
}
```

This algorithm is crucial to my program because it creates the binary tree of operators and operands that allow for the conversions and solutions to occur. This algorithm first finds the last operation executed in the expression and forms a root node from it. The expression is then split in half, and the algorithm is recursively called until each half is split into a single digit. Every time a split occurs, the root node receives a left and a right child- the next root operators of the left and right halves of the expression. This first algorithm that I developed also uses a second mathematical algorithm that I developed to actually determine which operator is the last to be solved- the root operator. After the root operator is found and its children assigned, recursively evaluating each operation using the operands in the two children allow for the expression's solution to be found.

```
private int determineRootNodeIndex(List<String> stubs)
{
        int indexOfPlus = stubs.lastIndexOf("+");
        int indexOfMinus = stubs.lastIndexOf("-");
        int indexOfTimes = stubs.lastIndexOf("*");
        int indexOfDivided = stubs.lastIndexOf("/");

        if(indexOfPlus == -1 && indexOfMinus == -1)
            return Math.max(indexOfTimes, indexOfDivided);
        else
            return Math.max(indexOfPlus, indexOfMinus);
}
```

This algorithm determines which operator in the expression is the root operator- the last one to be evaluated. It does this by parsing the expression in reverse order of operations, looking for the last addition and subtraction signs before looking for the last multiplication and division signs.

```
private List<String> splitExpression(String expression)
{
        List<String> stubs = new ArrayList<String>();

        for(String stub : expression.split("\\b"))
        {
            stub = stub.trim();
            if(!stub.equals("") && !stub.equals(" "))
                stubs.add(stub);
        }

        return stubs;
}
```

This third algorithm is also a dependency of the first algorithm. The first algorithm requires the expression to be broken into pieces in a list. This algorithm takes the expression and breaks it. It can handle expressions with digits all side by side or separated by spaces.

2d.

```
public class OperatorNode extends BinaryTreeNode
{
    public OperatorNode(String operator)
    {
           super(operator);
    }

    @Override
    public String getValue()
    {
           String operator = super.getValue();
           double leftChild = Double.parseDouble(getLeftChild().getValue());
           double rightChild = Double.parseDouble(getRightChild().getValue());

           switch(operator)
           {
                  case "+": return "" + (leftChild + rightChild);
                  case "-": return "" + (leftChild - rightChild);
                  case "*": return "" + (leftChild * rightChild);
                  case "/": return "" + (leftChild / rightChild);
                  default: return "0";
           }
    }
}
```

One abstraction I used was a class for the tree nodes that contained operators. In order to solve the expression, I had to recursively evaluate each operation with the operands in the left and right children. Sometimes, the children contained operators themselves that had operands. The code here creates a method (getValue) that returns the value of the operand, or the value of the result of the operation, depending on whether the node contains an operand or an operator. However, this mathematical algorithm is hidden in my OperatorNode abstraction; this way, the code that receives the expression as input can simply use the getValue method on the root node without having to worry about traversing all the nodes. This greatly simplified my program, because I could separate the code that read the expression with the code that solved it, and use this solving code multiple times for any operator in any expression.