HW06: 24 Game using binary expression tree
CSE2050 Fall 2018

1. Introduction
In this exercise, we will complete a Python class and additional functions to find all possible solutions for a given 4-card combination and print each solution.

2. Objectives
The purpose of this assignment is to give you experience in:

- Using mathematical calculations in Python.
- Implementing and using the basic operations of a binary tree.
- Practicing in-order and post-order traversal of binary trees.

***Note**: Before you start, if you are not familiar with the basic operations of a binary tree data structure and concepts of functions and classes in Python, it is recommended that you review the material covered in lecture first.*

3. Background

3.1. 24 Game
In this game, 4 cards are randomly picked from a standard deck of 52 cards. We will use the values of these 4 cards and a combination of addition, subtraction, multiplication, division, and parentheses to produce the value 24. Note that each of the 4 cards can only be used once.

For the purposes of this assignment, we will consider the following to be the face values of each card in a suit:

“A”: 1

“2”: 2

“3”: 3

“4”: 4

“5”: 5

“6”: 6

“7”: 7

“8”: 8

“9”: 9

“10”: 10

"J": 11

"Q": 12

"K": 13

For example, if the 4 cards chosen are "A 2 3 Q," we can produce the value 24 by doing the following operation:

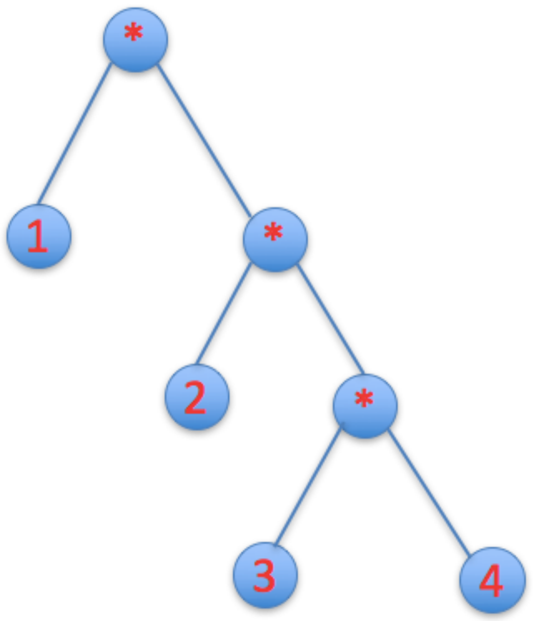
(3 – 2 + 1) * 12 = 24

Note that there are many possible solutions for a given 4-card combination. For instance the cards "A 2 3 Q" can produce 33 different solutions which produce 24. Read the following link for more information about the game: https://en.wikipedia.org/wiki/24_Game.

3.2. Binary expression tree

To solve this problem, you will also need to be familiar with **binary expression tree**, which is a kind of binary tree used to represent expressions. In this type of binary tree, each internal node corresponds to an operator and each leaf node corresponds to an operand.

For example, the expression 1*(2*(3*4)) can be represented using a binary expression tree as follows:



3.3. Fractions in Python

Note that the very first line in the skeleton code imports the smaller module 'Fraction' from module 'fractions'. A module is simply a file containing Python definitions and statements.

To use the module when you are doing division, you will use the following statement:

```
Fraction(n1, n2)
```

which will return an instance of Fraction. If you want to know the value of $\frac{n1}{n2}$, you can use the value returned by Fraction(n1, n2).

4. Assignment

In the skeleton zip file, you will find a skeleton for your .py file named *binarytree.py*. All you need to do is to complete the skeleton code based on the instructions and submit it to the Mimir system.

4.1. BNode class

Now, open the skeleton code. In this assignment, you will be using a binary tree to evaluate an expression. To do this, a BNode class has already been created for you. What you need to do is implement each of the methods within the BNode class.

Each node in the binary tree has three attributes – element, left child, and right child. The element holds an operator or an operand. The left and right child point to the respective left and right child nodes if they exist for that node. Each internal node is a parent node. The addition of left and right child nodes to the parent node is performed using the addleft() and addright() methods of the BNode class.

4.2. evaluate() method

The evaluate() method is within the BNode class. This method will take in a node of the binary tree and recursively call evaluate method to evaluate it.

In this method, if the element is an operation (i.e. is either '+', '-', '*', or '/') you will call the evaluate method for the left and right child nodes. Now, perform the operation on the result and return the value. If the operation is division and the denominator is 0, you can simply return 0 to the method and terminate the evaluation, because this means that it will not evaluate to 24 anyway.

If the element is a card value, then simply return that value. This is the base case.

Once the method finishes running, the final value returned will be the result of evaluating the expression.

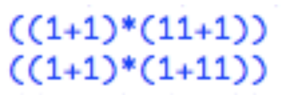
4.3. display() method

This method will display all the values of the nodes in the binary tree in infix notation. It's implementation is similar to the evaluate() method. This method will take in a node of the binary tree and recursively call display method to display all the child nodes of the input node.

In this method, if the element is an operation (i.e. is either '+', '-', '*', or '/') you will call the display method for the left and right child nodes and return the expression. Make sure to include parentheses while displaying an expression.

The base case is when the element is a card value. In this case, convert the element into a string and return that value.

The display() method should produce an output as follows:



```
((1+1)*(11+1))
((1+1)*(1+11))
```

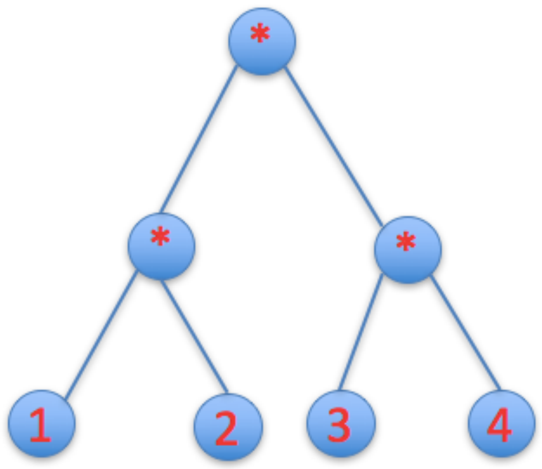
4.4. evaluatefive() Function

This function will find all valid forms of representing an expression using a binary expression tree for each 4 card combination and evaluate them to see if they are equal to 24. You will call your evaluate() method inside of this function to evaluate the expression. The function takes a list of operations and a list of cards as argument.

There are five different possible forms of representing an expression using a binary expression tree for 4 cards, where 'C' denotes a card value and 'X' denotes an operation:

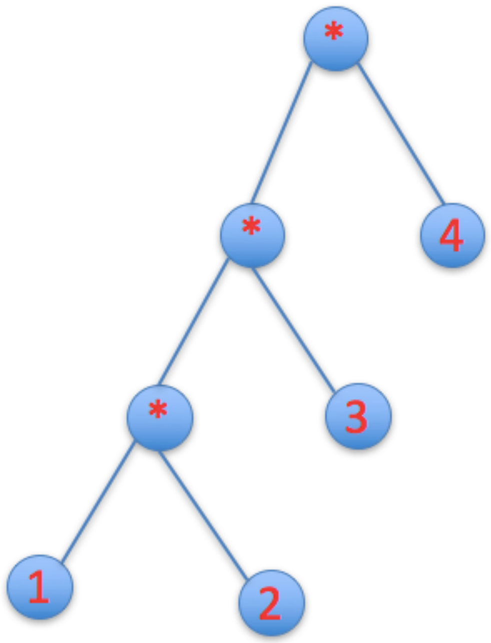
1. CCXCCXX

   Example: (1*2)*(3*4)



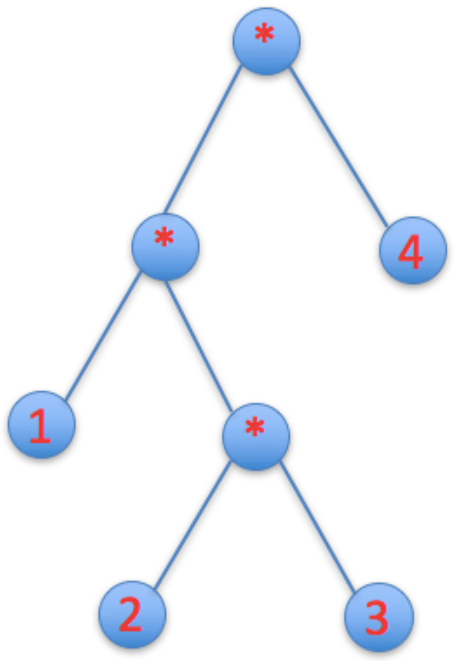
2. CCXCXCX

   Example: ((1*2)*3)*4



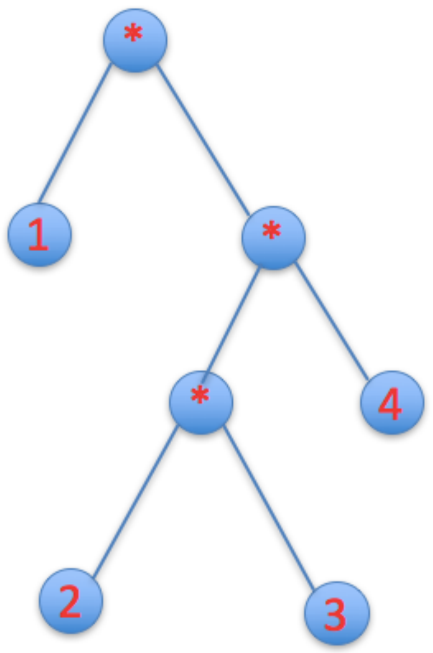
3. CCCXXCX

   Example: (1*(2*3))*4)



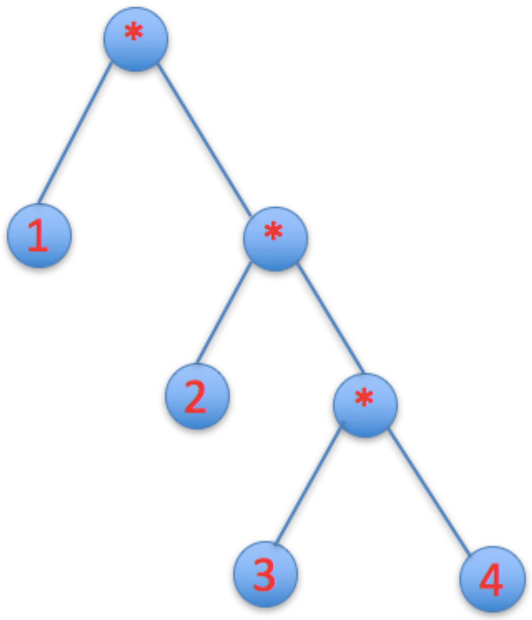
4. CCCXCXX

   Example: 1*((2*3)*4)



5. CCCCXXX

   Example: 1*(2*(3*4))



Every valid possible expression will take one of these five forms. Note that when you have 4 values, it is only possible to have 3 operations in the expression for it to be a valid expression.

You will need to implement the methods of the class BNode to represent each form as a binary tree. Once you have generated all of the five trees, evaluate them, and if an expression evaluates to 24, display that tree and add it to the results set.

4.5. Permutation

The following piece of code identifies all the different permutations of the values provided in the input. This ensures that we run through all the possible arrangements of operands.

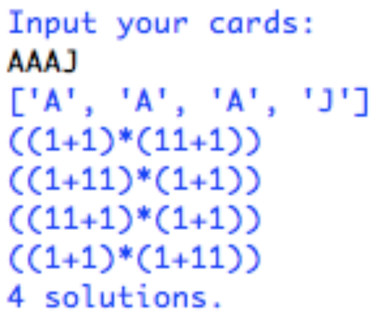
```
for L in list(itertools.permutations([0, 1, 2, 3], 4)):

    for i in range(4):
        s1[i] = s[L[i]]
```

4.6. Running your program

Once you have written the display method in 4.3 and both evaluation functions in 4.2 and 4.4, to run your program you will input the 4 cards as a single string, with no spaces. For example, if the cards selected are A, A, A, and J, the input you will type in is "AAAJ".

The output should look like this:



```
Input your cards:
AAAJ
['A', 'A', 'A', 'J']
((1+1)*(11+1))
((1+11)*(1+1))
((11+1)*(1+1))
((1+1)*(1+11))
4 solutions.
```

Note that the first list displayed after the "AAAJ" input is just a list containing the card values themselves. All following lists printed out are some combination of four cards and three operations which form a valid 24 Game solution.

This part of the program is already done for you; see the skeleton code for comments on how the code works.