CS2006 - Practical 2

Requirements:

The practical required a program with an additional set of operators implemented. The program must be able to handle input of values from the user, and according to the description of twisted integers, produce an output depending on the operation requested.

Program Functionality:

The basic functionality set in the requirements has been fully completed:

- The implementation of twisted integers has been extended to support addition and multiplication as set in the requirements. The user can set variables a and b to values using *"a=twistedInt(x,n)"* and can print the results of an operation using *"print(a+b)"* which will print the result.

```
C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2>py
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from TwistedInt import TwistedInt
>>> a=TwistedInt(3,5)
>>> b=TwistedInt(2,5)
>>> print(a+b)
<0:5>
>>> print(a*b)
<1:5>
>>>
```

- If the value for n is different for both a and b, the program rejects the input, raising an appropriate exception for the operation.

```
C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2>py
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from TwistedInt import TwistedInt
>>> a=TwistedInt(3,4)
>>> b=TwistedInt(3,5)
>>> print(a+b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2\TwistedInt.py", line 40, in __add__
    raise Exception("Invalid add operation")
Exception: Invalid add operation
>>>
```

Additionally, most of the additional requirements set have been completed as well:

- A function for finding all values of x, for a given n, that results in *"$x \otimes x = 1$"*. By iterating through every x value between 1 and n (as x cannot be greater than n), then returning each value that equals 1, a list of values is produced.

See *FindProductOf1.py*

```
C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2>py FindProductOf1.py
Enter the lower bound of n(at least 2): 2
Enter the upper bound of n: 15
When n = 2, x that satisfies x * x = <1,2> are:<1:2>
When n = 3, x that satisfies x * x = <1,3> are:None
When n = 4, x that satisfies x * x = <1,4> are:None
When n = 5, x that satisfies x * x = <1,5> are:None
When n = 6, x that satisfies x * x = <1,6> are:None
When n = 7, x that satisfies x * x = <1,7> are:<2:7>
When n = 8, x that satisfies x * x = <1,8> are:None
When n = 9, x that satisfies x * x = <1,9> are:None
When n = 10, x that satisfies x * x = <1,10> are:None
When n = 11, x that satisfies x * x = <1,11> are:None
When n = 12, x that satisfies x * x = <1,12> are:None
When n = 13, x that satisfies x * x = <1,13> are:None
When n = 14, x that satisfies x * x = <1,14> are:<3:14>
When n = 15, x that satisfies x * x = <1,15> are:None
```

- A function for checking the stated properties for operations has been implemented. The user inputs values for n, x, y and z into the function and the program will test all 5 rules for the given inputs and print whether or not the rules hold true or not.
  See *TestRules.py*

```
C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2>py TestRules.py
Enter an integer for n: 20
Enter an integer for x: 3
Enter an integer for y: 4
Enter an integer for z: 5
When x = <3:20>, y = <4:20>, z = <5:20>:
x + y = y + x is: True
x * y = y * x is: True
(x + y) + z = x + (y + z) is: True
(x * y) * z = x * (y * z)) is: True
(x + y) * z = (x * z) + (y * z)) is: False
```

- A class called *TwistedIntegers* which returns a list of *TwistedInts*. The constructor takes an integer *n* from user and generates a list of *TwistedInt*s from <0,n> to <n-1,n> (where n is positive) or from <n+1> to <0,n> (where n is negative). Also, the class supports an overwritten *print()* method and a *Size()* method which returns the size of the list.
  See *TwistedIntegers.py*

```
>>> from TwistedIntegers import TwistedIntegers
>>> a=TwistedIntegers(5)
>>> print(a)
[<0:5><1:5><2:5><3:5><4:5>]
>>> a.Size()
5
```

- An iterator for *TwistedIntegers* called *IteratorOfTwistedIntegers*. The iterator takes an instance of *TwistedIntegers* (with size *n*) and iterates the list. On each iteration, it returns a *TwistedInt* from the start of the list to the end of the list.
  See *IteratorOfTwistedIntegers.py*

```
>>> from TwistedIntegers import TwistedIntegers
>>> from IteratorOfTwistedIntegers import IteratorOfTwistedIntegers
>>> for i in IteratorOfTwistedIntegers(TwistedIntegers(10)):
...     print(i)
...
<0:10>
<1:10>
<2:10>
<3:10>
<4:10>
<5:10>
<6:10>
<7:10>
<8:10>
<9:10>
```

- Two functions which find τ & ε for a given *n* such that τ ⊕ x = x & ε ⊗ x = x. These functions use the iterator to iterate the list in *TwistedIntegers(n)* and prints out all items that satisfy the condition.
  From the result, it can be seen that only <0,n> are in τ & ε. Since only (0+x) mod n = x and (x+x) mod n = x, and x does not satisfy the condition to build a TwistedInt. So the only element in τ & ε is <0,n>.
  User can run *FindTauAndEpsilon.py* and enter an arbitrary integer to find τ & ε for the given n.

```
C:\Users\dwwoo\Documents\!University\CS2006\CS2006-P2>py FindTauAndEpsilon.py
Enter an integer: 8
For n = 8:
When x = 0, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 1, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 2, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 3, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 4, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 5, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 6, Tau contains:<0:8>, Epsilon contains: <0:8>;
When x = 7, Tau contains:<0:8>, Epsilon contains: <0:8>;
```

- A class called *TwistedMatrix* which supports the *TwistedInt* matrix.
  The matrix is initialized with *row*col* size matrix filled by 'None's. User needs to fill the matrix either by array assigning (e.g. matrix[0][0] = TwistedInt(1,2)) or by *AddElement()* method (which takes *row, col ,value* and *n* to generate a *TwistedInt(value,n)* at *matrix[row][col]*).

```
>>> from TwistedMatrix import TwistedMatrix
>>> a=TwistedMatrix(2,2)
>>> a.AddElement(0,0,0,4)
>>> a.AddElement(0,1,1,4)
>>> a.AddElement(1,0,2,4)
>>> a.AddElement(1,1,3,4)
>>> print(a)
[ <0:4> <1:4> ]
[ <2:4> <3:4> ]
A 2*2 TwistedInt matrix
```

  Two *TwistedMatrix* with the same size can be added together by using '+'. Elements in the same position will be added together (using *TwistedInt* addition). If one of the

elements is *None* or one of the elements have a different *n*-value, the result will be *'Undefined'*.

Two *TwistedMatrix* can also be multiplied together by using '*' (e.g. A * B). The multiplication follows 'Row-by-Column' rule, if the number of columns of A is not equal to the number of rows of B, an exception will be raised. Corresponding elements will be multiplied together and added to generate the result matrix, if one the elements in the calculation is *None* or there is an element with a different *n,* the corresponding position will be *'Undefined'*.

See *TwistedMatrix.py*.

- Testing, for the last part, we implemented a selection of tests, for TwistedInt and TwistedMatrix. These ran through around 30 calculations or tests, from basic requirements to more advanced requirements. While it would possibly have been better to write these tests at the beginning, we instead tried to complete the requirements in ascending order of difficulty, in case we were unable to complete any, at least we'd have a solid practical to fall back on. *TwistedIntTests.py* and *TwistedMatrixTest.py* can both be called by the *TestAll.py* file for ease. This outputs the name of each test as it begins, and then whether or not the test was passed. For the tests which initially failed, we improved upon them, and added expected values. For tests involving floating point numbers, we rounded the result to 3 decimal places more than needed so as to eliminate rounding errors, but not round the number too prematurely. This adding of tests while on the surface seems superfluous, greatly helped with implementing floating point support and complex number support.

Additional Extensions:

- While the specification did not require it, nor did it mention it, we found the program and maths could cope quite neatly with additional types of numbers. To begin with, the background referenced a calculation using negative numbers:

  $2 \otimes (-3) = 2 - 3 + 2 \cdot (-3) = -7,$

- So we viewed this as a suggestion to add possible negative numbers. This meant we had to alter the requirements from $0 \leq val < n$ to $|val| < |n|$. Other than this change, the program ran fine, and worked as expected.
- After this, we chose to implement floating point numbers, as again, the maths should remain fine, as modulo decimals still functions. Again, the maths didn't have to change, and new results were fine.
- Finally, we decided to implement complex numbers. To begin with they worked fine, and required no extra coding, since even though the comparative operators are undefined for complex numbers, we were already using abs, which returned the size of the complex number. The only problem occurred with modulo, since this was un-supported for complex numbers. This required looking into how to actually

calculate modulo for complex numbers. We decided on implementing a function similar to what Wolfram Alpha implements. This functions in a slightly complex fashion, for further details look at the actual code. This took two iterations to get right, but upon finishing, again, the rest of the calculations function fine with it.

Provenance:

The code submitted in the practical is all of our own work. No large segments of code have been sourced from online, everything was written by us.