220005149, 220017161, 220010065		Tutor: Olexandr Konovalov		6 March 2024

CS2006 Python 1 Practical

Overview:

For this practical, we were asked to develop a Python module representing a new data type, IntricateInteger. This class needed to adhere to the definition of multiplication given in the specification - that is, $x \otimes y = (x+y+\alpha(\mathrm{lcm}(x,y))) \bmod n$, with the necessary restrictions on each variable. We implemented this successfully, as well as all additional requirements and several extensions, including optimising memory usage and making and presenting performance comparisons for a range of values of n and alpha. The inclusion of the additional requirements allows the user to find various properties in certain IntricateIntegers, such as commutativity and associativity and allows them to iterate over an expected set of IntricateIntegers. Our final program has also been tested extensively using the unittest library. As such, we can provide a conclusive and efficient solution to the problem set by the requirements.

What is Complete:

1. All basic requirements
2. All easy additional requirements
3. All medium additional requirements
4. All hard additional requirements
5. All very hard additional requirements
6. Unit testing of the entire code base
7. Additional performance analysis

Functionality:

The implementation of our practical contains all of the basic functionality from the specification along with several of the suggested additional requirements. Furthermore, we added several ease-of-use functionalities to increase the useability of the program. To complete the basic requirements of the specification several steps were taken. We took the provided starter code from the specification and altered the three provided methods to meet the requirements. For the initializer function, four conditional statements were added at the start to ensure that the values provided for obj, n and alpha are positive and within the range of n, i.e. zero to n - 1 inclusive. Should any of the provided values not match these requirements, an exception is raised with a helpful message provided. Lastly, all of the values are set to their correct self. We decided to reduce the object of the integer by a modulo of the z-range rather than outright banning the value. This enables more flexibility in the code and ease of use for the programmer when testing values. For the str or print function, the program simply creates a new string object with the object, z-range value and alpha printed in the specification format. Lastly, the multiplication function was altered with a conditional statement. Should the z-range value for both provided integers not match, a new exception is raised with a helpful message. Otherwise, the multiplication is calculated using the given binary operation provided in the requirements. While writing this functionality we ensured to use higher-order functions instead of recursion. While this not only simplified the process it resulted in a cleaner and more efficient code base. At this point in the project, we had completed all of the basic requirements in the specification. We knew this was not enough to satisfy a strong implementation so we began work on the suggested additional requirements.

Starting with the easy requirements we added a check for peculiar property and commutative intricate multiplication. Both of these were fairly straightforward as they only required the addition of for loops to cycle through all of the values for a given z-range. At each value, a conditional statement checks that the intricate integer with the values at this iteration match the specific property being tested. For peculiar it was an Intricate Integer times itself must equal the same result as the Intricate Integer itself. For commutative it was an Intricate Integer times another Intricate Integer with a value from 0 to z-range. For this specific implementation, a dictionary was added. By implementing this, we have optimised the code to avoid redundant calculations. Both of these implementations were put inside of their respective functions which can be called in the Python interpreter. Furthermore, additional functions were created as test cases to see if all the values for z-range one to 50 and alpha zero to z-range are peculiar and commutative. When running these functions both return true.

After this, we began work on implementing the medium requirements. The first step was to add a new function to check whether a specific z-range and alpha value have associativity for x, y, and z values in the set of z-range. For each instance, the program checks if the calculation is already in a dictionary. If so the result is pulled. Otherwise, the value is calculated. Should associativity not follow for x, y, and z, then False is returned. Additionally, another function was created as test cases to see if all the values for z-range one to 20 and alpha zero to z-range are associative. If false is returned for any value, the value is added to a list. Finally, all of the values are printed to show which z-range and alpha do not hold for associativity. The other two medium requirements involved writing a new class, IntricateIntegers. This class is very similar to IntricateInteger but it does not take an object value for input. The rest of this class is fairly similar to the previous IntricateInteger. The last medium requirement was to implement IntricateIntegersIterator. This is a new class type, iterator, which iterates over an instance of the previously defined IntricateIntegers class. All of the previous requirements, one through three, were also tested using this new version of iterator. All of the results are the same when running and comparing the two outcomes. The main difference between the regular implementation and iterative is the time difference for calculations. Appropriate tests have been created to show the difference in time between the regular and iterative implementation. The results show that iterative is significantly more efficient and runs the tests faster.

Along with this, we added the hard requirements. The first of these requirements was intricate_roots_of_one. This new function checks for every value in the z-range if an IntricateInteger times itself equals one. Should this be true, the value of the object is appended to a valid list. At the end of the function, the list is returned. Additionally, another function was created, roots_test, which cycles through every value of z-range from 1 to 25 and alpha, 0 to z-range. A valid list appends all of the values returned from the previous declared function. Then the results are returned. Lastly, for this requirement, another function, minimum_roots_test, was defined. This function looks at the conjecture provided in the specification. This function cycles through all values of z-range one to 25, and every odd alpha value in that z-range. Then if the greatest common denominator of n and alpha is not equal to one and the intricate root of one for these values does not equal one, the items are added to a list. The list is returned immediately, preventing future calculation, thereby providing the minimum counter example.

The final requirement involved taking a set of IntricateInteger objects ('generators') and returning its span - a set of all possible products that could be obtained by multiplying the generators however many times. We initially decided to use a method involving finding the powers of each generator, since this (usually) eventually resulted in a sequence of IntricateIntegers for each generator (i.e. 1 = 1, 1^2= 1 x 1 = 4, 1^3 = 1 x 1 x 1 = 4 x 1 = 1..., for instance, with no particular values) which could be used to find every possible combination from simply the unique elements of each sequence. However, since we realise this method relies on the associativity of the integers in question, and we could not guarantee this for certain values of n, we took a different approach. Ultimately, we made use of the itertools library's combinations function, recursively calling it with an ever-expanding (to a point) set of possible products. We include the necessary comparisons to check if any new products have been added since the last pass through the set - if so, the obtained span is returned, if not the function is called again.

The implementation of the above also meant we needed to display a set of simply the values of each Integer, not the references to the objects. Therefore, we have also included a variety of helper methods which enable this and other necessary parts of the functionality of this last requirement.

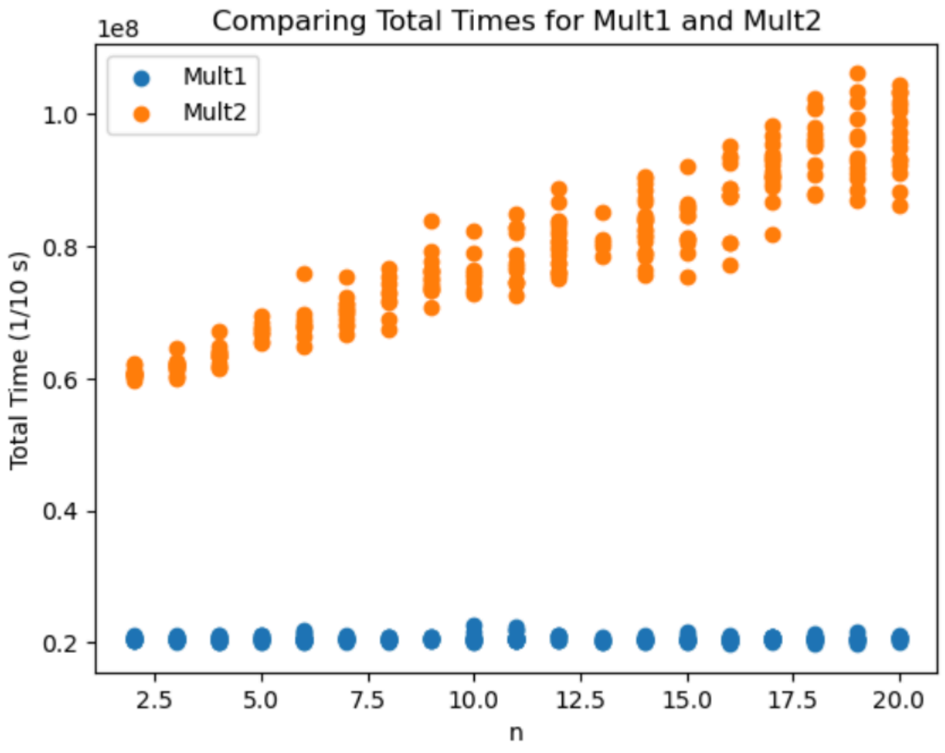
We also considered using a combined approach: checking for the associative properties of the given values of n and alpha, then proceeding with either the recursive combinations method or the sequences of powers method depending on the result. However, given the programmatically intensive nature of the has_associative_intricate_multiplication method, we ultimately decided that it would be best for performance purposes to use combinations in all cases.

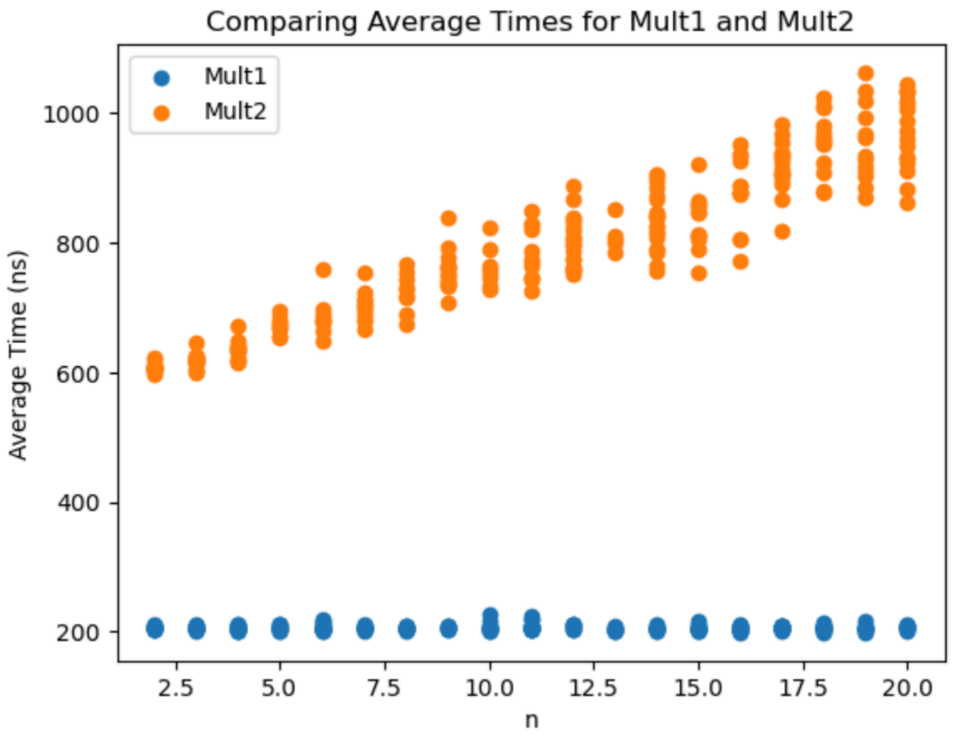
The last requirement added to our implementation is unit testing and coverage checks on the code base. Testing.py is a unit testing class which when run will compile all of the code and run through all of the necessary tests to ensure the functionality of Intricate Integers and its functions. There are several functions/tests declared which run through edge cases, default examples, and documented tests. The coverage class was also added to this project to check how much of the code base is covered by tests. The specific instructions on installing and running coverage are contained in the README file. By looking at the coverage report, the program is 95% covered by the unit testing added. This is near-perfect coverage with test cases for every single example and counter example necessary.

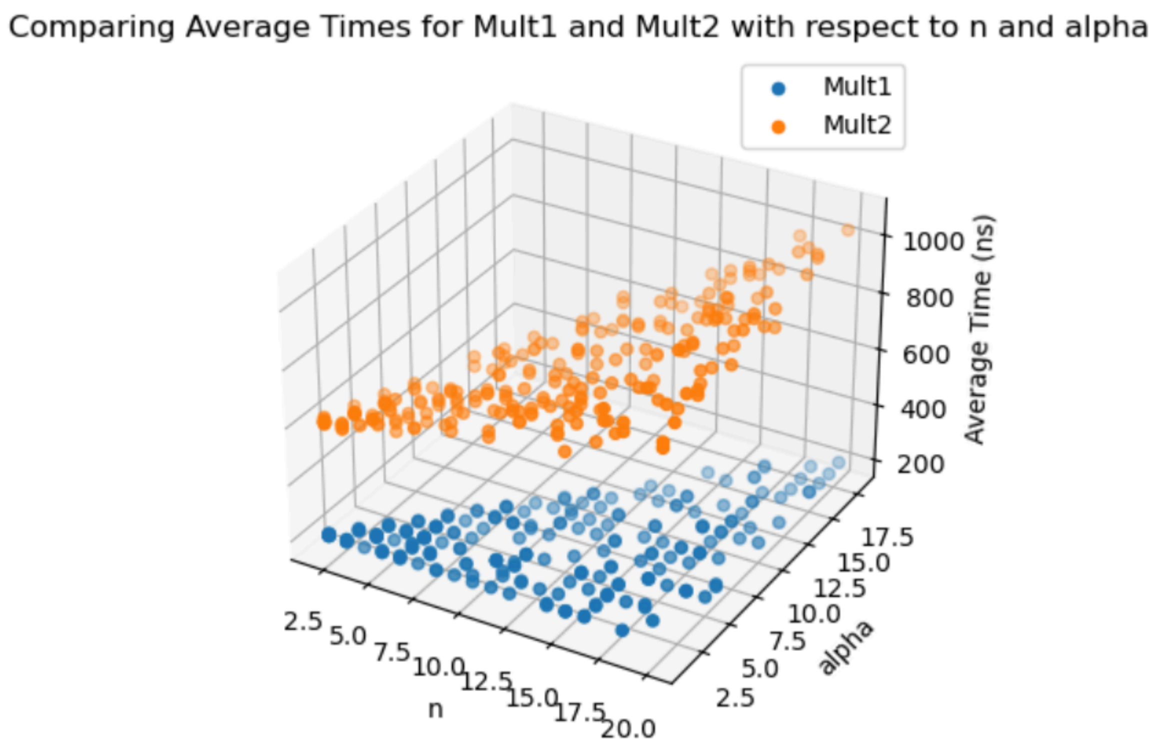
Testing and Performance Analysis:

As suggested by the specification, our solution has been thoroughly tested using a range of unit tests, the output of which is shown below in the examples section.

Additionally, we researched ways to improve efficiency where possible, such as parallelisation for checking associativity to distribute the workload across multiple processors or threads since it was so programmatically intensive (which we ended up not including because associativity checks are inherently sequential and dependent on the results of previous computations so parallelization wouldn't provide any speedup and could even introduce unnecessary overhead), and memoisation - storing previous calculations to use later by adding a dictionary to the IntricateInteger class. We wanted to test this new mult method against the old one so created a set of 250 valid multiplications with randomly generated combinations of x, y, n and alpha and ran both multiply functions against them (with 100,000 repetitions of the code snippet to balance ensuring maximal accuracy and not compromising processing time), investigating their performance using the timeit module and writing the results to a CSV file to be analysed using nanoseconds instead of seconds for accuracy purposes. Here are our results (where Mult1 is the newer version of the original Mult2 - found as __mult__ in the code):


Comparing Total Times for Mult1 and Mult2
1e8
Mult1
Mult2
Total Time (1/10 s)
0.2
0.4
0.6
0.8
1.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
n


Comparing Average Times for Mult1 and Mult2
Mult1
Mult2
Average Time (ns)
200
400
600
800
1000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
n



Please note that the original CSV data and the Jupyter Notebook that produces these graphs are available in the directory src/Performance_Testing and code to run these tests is in the src directory.

Issues:

All of the easy, medium, hard and very hard requirements were fully implemented. We did not run into any major issues in our submission. The time to run all of the tests takes a significant measure, but it does not seem to cause any specific issues. When we first implemented coverage, we got around 70%. To combat this issue, we went back and created unit tests for all of the edge cases and instances which were currently not covered in the code. The last minor issue we encountered was in commutative and associative. Both of the functions for these requirements originally did not have any checks to see if the same values were being calculated multiple times. We fixed this by adding dictionaries to both of these functions to prevent redundant calculations between the cycles. The only other stand-out issue was that for larger values of n, despite our best efforts to optimise memory usage, the program would sometimes crash due to a lack of memory when running the generator-span methods. This, however, given the recursive nature of the implementation of this requirement, was essentially unavoidable, and almost justifiable, since we are only required to test for associativity up to and including n=20 - the generator-span methods can handle the same range. These were the only problems that came up during implementation that required stepping back to rework.

Examples:

```
[redacted]@pc8-062-l:.../Documents/CS2006/Python1/src $ python3 Testing.py

Testing Intricate Integer Values:

.Testing Intricate Integers Iterator:

['<0 mod 3 | 2 >', '<1 mod 3 | 2 >', '<2 mod 3 | 2 >']

.Testing Associative Iterator:

1 0

2 0

2 1

3 0

3 2

4 0

4 2

4 3

5 0
```

```
6 0

6 3

7 0

8 0

8 4

9 0

10 0

10 5

11 0

12 0

12 6

13 0

14 0

14 7

15 0

16 0

16 8

17 0

18 0

18 9
```

```
19 0

20 0

20 10

.Testing Commutative Iterator:

.Testing Peculiar Iterator:

.Testing Edge Case of Invalid Alpha:

.Testing Multiplication Using Different Values of N and Alpha:

.Testing Multiplication Example from Specification:

.Testing Multiplication Using Other Values:

.Testing Creating Intricate Integer with Negative Numbers:

.Testing Intricate Integer Creation:

.Testing Creating Intricate Integer with String Values:

.Testing Edge Case of Valid Alpha:

.Testing Associative Regular vs Iterative:

1 0

2 0

2 1

3 0

3 2

4 0
```

4 2

4 3

5 0

6 0

6 3

7 0

8 0

8 4

9 0

10 0

10 5

11 0

12 0

12 6

13 0

14 0

14 7

15 0

16 0

16 8

```
17 0

18 0

18 9

19 0

20 0

20 10

Regular Tests Completed in 0.9089614960248582

1 0

2 0

2 1

3 0

3 2

4 0

4 2

4 3

5 0

6 0

6 3

7 0

8 0
```

```
8 4

9 0

10 0

10 5

11 0

12 0

12 6

13 0

14 0

14 7

15 0

16 0

16 8

17 0

18 0

18 9

19 0

20 0

20 10

Iterative Tests Completed in 0.5552182249957696
```

```
Difference in tests 0.35374327102908865

.Testing Commutative Regular vs Iterative:

Regular Tests Completed in 10.029846703982912

Iterative Tests Completed in 8.961108061019331

Difference in tests 1.0687386429635808

.Testing Peculiar Regular vs Iterative:

Regular Tests Completed in 0.011229048017412424

Iterative Tests Completed in 0.013243823021184653

Difference in tests 0.0020147750037722229

.Testing Generator Span Algorithm:

{1, 2, 3, 5}

.Testing Associative Multiplication:

1 0

2 0

2 1

3 0

3 2

4 0

4 2

4 3
```

5 0

6 0

6 3

7 0

8 0

8 4

9 0

10 0

10 5

11 0

12 0

12 6

13 0

14 0

14 7

15 0

16 0

16 8

17 0

18 0

```
18 9

19 0

20 0

20 10

.Testing Commutative Multiplication:

.Testing Peculiar Property:

.Testing Intricate Roots:

[[0, 126], [1, 199]]

.Testing Intricate Roots Minimum Value:

[6, 3]

.

----------------------------------------------------------------------

Ran 22 tests in 41.087s


OK

[redacted]@pc8-062-l:.../Documents/CS2006/Python1/src $
```

<u>Reference Material</u>:

For this practical nearly all of the code was written by ourselves or modified in the provided source files. IntricateInteger.py contains the IntricateInteger class which is a modification of the starter code provided in the specification. The rest of this file's methods were written by us. IntricateIntegers.py was heavily modified to implement all of the additional specifications. Only slight inspiration was taken from the Python lectures. Testing.py is entirely original code which was written by us to test all the various methods in the aforementioned files. We did not source code from anywhere other than lectures, studres examples and the provided starter code. Several of the program's functions required looking up on websites like Python docs and Geeks for Geeks, which provide documentation of Python coding and packages.

Conclusion:

Upon the completion of this project, we have provided a specification-accurate implementation. All of the requirements were implemented, as well as various personalised extensions. The program has been well and rigorously tested for a range of possible values of n and alpha, and every aspect of the implementation works as expected, saving unavoidable errors and potential crashes caused by memory overflow for unreasonably large values of n. All the code has been commented and documented and is easily readable for inspection. Coverage has been generated and a report states that nearly all of our code has been tested. As a result, the holistic solution we have provided is very successful in meeting the requirements of the specification.

Version Control Repo:

via GitHub at: https://github.com/FChamb/Python1

Word Count: 2275