GSJ: Volume 12, Issue 1, January 2024, Online: ISSN 2320-9186

www.globalscientificjournal.com

IMPLEMENTATION OF ALGORITHMS FOR $\mathbb{Z}_n$ RINGS IN PYTHON PROGRAMMING LANGUAGE.

Otieno Francis Odhiambo[a], Okumu Otieno Kevin[b], Njuguna Edward[c]

a. Department of Mathematics and Physical Sciences, Maasai Mara University, P.O Box 861-20500, Narok, Kenya. Email: Francisotieno@gmail.com
b. Department of Mathematics and Physical Sciences, Maasai Mara University, P.O Box 861-20500, Narok, Kenya. Email: kevinotieno15@gmail.com
c. Department of Mathematics and Physical Sciences, Maasai Mara University, P.O Box 861-20500, Narok, Kenya. Email: edwardnjuguna@gmail.com

Corresponding author email: kevinotieno15@gmail.com

Abstract: This research focuses on implementing algorithms for $\mathbb{Z}_n$ rings in Python programming language. $\mathbb{Z}_n$ Rings are important structures in abstract algebra, and the implementation of algorithms for these structures is essential in many areas of mathematics and computer science. The research includes the implementation of basic operations such as addition, subtraction, and multiplication in $\mathbb{Z}_n$ rings, as well as more advanced algorithms such as modulo exponentiation and the extended Euclidean algorithm. The implementation is done using object-oriented programming principles to ensure code reusability and maintainability. This research aims to provide a useful resource for researchers and students in mathematics and computer science who are interested in working with $\mathbb{Z}_n$ rings. In this research, it was found out that the ***mod_inv*** function is more efficient and reliable than the ***mod_mult_inverse*** function. It uses the extended Euclidean algorithm to find the inverse of $\boldsymbol{a}$ modulo $\boldsymbol{n}$, which is faster than checking all possible values of **b**. additionally, it works correctly for all values of $\boldsymbol{a}$ and $\boldsymbol{n}$**.** Based on the results and discussions presented, it was evident that the implementation of algorithms for $\mathbb{Z}_n$ rings in Python programming language is a viable approach. The findings have highlighted the potential of using this implementation for various mathematical applications. It is recommended that further research to be conducted to explore the full capabilities of this implementation and its possible applications in real-world scenarios.

Keywords: $\mathbb{Z}_n$ rings, Algorithms, Algebra systems, Python programming

1. Introduction

Rings are a fundamental mathematical concept that has a wide range of applications in many fields such as computer science, cryptography, and coding theory. One of the most important types of

rings is the ring of integers modulo n, commonly known as $\mathbb{Z}_n$ ring. The $\mathbb{Z}_n$ ring is a finite ring that consists of integers from 0 to n-1 and it is closed under the operations of addition, subtraction, and multiplication.

The implementation of algorithms for $\mathbb{Z}_n$ rings is a crucial task in many applications, such as coding theory, cryptography, and computer algebra systems. In coding theory, for example, the $\mathbb{Z}_n$ ring is used to construct error-correcting codes that can detect and correct errors in digital communication systems. In cryptography, the $\mathbb{Z}_n$ ring is used in public-key cryptosystems such as RSA and ElGamal. In computer algebra systems, the $\mathbb{Z}_n$ ring is used to perform symbolic computations in finite fields.

The Python programming language is a popular choice for implementing mathematical algorithms due to its readability and the availability of a wide range of libraries for numerical computation and symbolic mathematics. However, the efficiency of the implementation is crucial for the performance of the algorithms in large-scale applications. While there are several libraries available for implementing $\mathbb{Z}_n$ rings in Python, such as SymPy, NumPy, and SageMath, there is limited research on the efficiency of these implementations.

This research probed the implementation of algorithms for $\mathbb{Z}_n$ rings in the Python programming language. The study evaluated the performance of the implementation using a set of test cases, including basic and complex operations. The results were compared to a reference implementation, implemented in another programming language known to have an efficient implementation of the same operations.

This research provide insights into the efficiency of the implementation of algorithms for $\mathbb{Z}_n$ rings in the Python programming language, and also contribute to the knowledge of efficient implementation of mathematical algorithms in general, it will be useful for researchers, developers, and practitioners working on applications that uses $\mathbb{Z}_n$ rings.

1.1. Mathematical Concepts and Algorithms for $\mathbb{Z}_n$ Rings

Figures such as real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$, and rational numbers $\mathbb{Q}$ are objects that are commonly used in mathematical operations, these objects are known as algebraic structures whose classifications as groups, rings, and, fields only differ per axioms of the structures. With the examples, $\mathbb{R}, \mathbb{Z}\ and\ \mathbb{C}$ it is known from a general knowledge that they are infinite sets, but for the interest of the research, we will look into an operation that gives a finite set that is the **modulo arithmetic.** This implies that algebraic structures such as rings are formed by sets together with

operations which is not necessarily the usual addition or multiplication, giving us the definition of a ring as: - a set **R** together with binary operations e.g. $\oplus \,\&\, \odot$ such that it is closed under the under the two operations, there exist associativity, Identity, inverse elements and the distributive law [8].

Definition of $\mathbb{Z}_n$ Rings: A $\mathbb{Z}_n$ ring is a set that consists of integers from 0 to n-1 that is, $\mathbb{Z}_n = \{0, 1, \ldots \ldots n-1\}$ and it is closed under the operations of addition, subtraction, and multiplication. The basic properties of $\mathbb{Z}_n$ rings include closure, associativity, and distributivity.

Basic operations: The basic operations in $\mathbb{Z}_n$ rings include modulo addition/subtraction, and modulo multiplication. The algorithm for addition (and subtraction) (see 5,6,11) in $\mathbb{Z}_n$ ring is [5]

$$(a+b)\ \boldsymbol{mod}\ n = \begin{cases} (a+b) & if\ (a+b) < n \\ (a+b-n) & if\ (a+b) \geq n \end{cases} \qquad \forall\ a, b \in \mathbb{Z}_n\ \ldots\ldots\ldots\ (Algo\ \mathbf{1}) \quad \mathbf{(1)}$$

While the algorithm for modulo multiplication is

$$(a \times b)\ \boldsymbol{mod}\ n = \begin{cases} (a \times b) & if\ (a \times b) < n \\ (a \times b) - ([(a \times b)//n] \times n) & if\ (a \times b) \geq n \end{cases} \qquad \forall\ a, b \in \mathbb{Z}_n \ldots . (Algo\ \mathbf{2})$$

(2)

Complex operations: In addition to the basic operations, $\mathbb{Z}_n$ rings also support more complex operations such as modulo inversion, power calculation/exponentiation, and polynomial operations. The algorithm for modulo inversion in $\mathbb{Z}_n$ ring is $(a^{-1})\ \boldsymbol{mod}\ n$ whereby $(a^{-1} \times a) = \mathbf{1}\ \boldsymbol{mod}\ n$. $a^{-1} \in \mathbb{Z}_n \forall a \in \mathbb{Z}_n$ can be computed using Extended Euclidean Algorithm [2] whereby a^{-1} will only exist if the $\gcd(a, n) = 1$

The exponentiation algorithm is $(a^b)\boldsymbol{mod}\ n$. And for polynomial operations, one can use the algorithm of polynomial operations in $\mathbb{Z}_n$ ring, $\forall\ a, b\ \in\ \mathbb{Z}_n$

1.2 Algorithms and Complexity analysis

The time and space complexity of the algorithms for basic and complex operations in $\mathbb{Z}_n$ rings are discussed in this section. For a wholly ritualistic discussion of Algorithms and their Complexity, one needs to get into a realm of theoretical computer science. However, this can be avoided since the notion of "the hardness of computational problems and efficiency" gives us enough meaningful discussion for this research.

Admissibly algorithm is a *Turing Machine*. For a clear definition, algorithms are finite steps that take inputs from non-negative integers and produce an output of the required results after finite steps. An integer ***n*** in a computer is represented by a string of *bits* and therefore each step in an algorithm is a *binary operation.* The "size" of the input is really important which mostly makes us approximate the time and space we require to get the output of the algorithm. For example; suppose $n \in \mathbb{Z}_{\geq 0}$ the length of n is defined as: $Length\ (n) := \log_2(n+2)$, while for $m \in \mathbb{Z}_{<0}$ $Length(m) := 1 + length(|m|)$ approximates the number of *bits* required to write *n, m* into binary respectively. And the length "size" of the input is the sum of the lengths of integers that compose it. That is, if the inputs for an algorithm are $m, n \in \mathbb{Z}$ then $Length\ (input) = length(m) + length(n)$ furthermore, from the length of the input, we can always know the number of steps required to get the output of an algorithm (*Running time/Time complexity*). For example, the time complexity of modulo inversion algorithm is O (log n) and the space complexity is O [1,11].

1.3 Implementation of Algorithms for $\mathbb{Z}_n$ Rings in Programming Languages

Several implementations of $\mathbb{Z}_n$ rings have been proposed in different programming languages such as C++, Java, and Matlab. These implementations mainly focus on basic operations such as addition, subtraction, and multiplication, as well as more complex operations such as modulo inversion and power calculation.

Experiments were conducted to compare the performance of the existing implementations of $\mathbb{Z}_n$ rings in different programming languages. These experiments were performed on a set of test cases, including basic and complex operations. The results showed that the performance of the implementations in C++ and Java were similar, with execution times of around *50ms for* basic operations and *200ms* for complex operations. However, the performance of the Matlab implementation was slower, with execution times of around *150ms* for basic operations and *600ms* for complex operations [15].

The existing implementations of $\mathbb{Z}_n$ rings in different programming languages have some limitations that affect their performance. For example, some implementations might not be suitable for large moduli, or they might not support certain operations i.e getting the answer 2^{10} is simple and easy to find, but solving 2^{100000} is very cumbersome such that the programs implemented to solve it are not capable of giving the output since it is an overflow of the ***int-bit*** of the programming language used. Additionally, the use of certain libraries in some languages may not be optimized for large-scale operations [1, 4,17].

1.4 Python Libraries for $\mathbb{Z}_n$ Rings

There are several popular Python libraries for implementing $\mathbb{Z}_n$ rings such as *SymPy, NumPy, and SageMath*. SymPy is a Python library for symbolic mathematics, which supports a wide range of algebraic operations. NumPy is a Python library for numerical computation, which supports various mathematical operations on arrays. SageMath is a Python library for mathematical computation, which supports a wide range of mathematical operations including algebraic operations and symbolic computations [3,8].

We conducted experiments to compare the performance of the existing Python libraries for $\mathbb{Z}_n$ rings. The experiments were performed on a set of test cases, including basic and complex operations. The results showed that the performance of the SymPy and SageMath libraries was similar, with execution times of around *100ms* for basic operations and *300ms* for complex operations. However, the performance of the NumPy library was slower, with execution times of around 150ms for basic operations and *400ms* for complex operations.

2. Methods

2.1 Algorithms and Data Structures

The algorithms for $\mathbb{Z}_n$ rings were implemented in **Python 3.11** using the built-in *modulo operator (%)* and custom modulo functions. The "mod" operator in Python allows for easy calculation of the residue of a division operation, which is a fundamental operation in modulo arithmetic. Additionally, custom modulo functions were implemented for more complex operations such as modulo inversion and power calculations [11,16].

The implementation of the $\mathbb{Z}_n$ ring was based on the mathematical concept of **modulo arithmetic**, which is the arithmetic of integers where only the remainder upon division by a fixed integer is considered. Modulo arithmetic operations such as addition, subtraction, and multiplication were implemented using the standard arithmetic operators in Python, with the result being taken modulo n using the built-in "mod" operator.

For more complex operations such as modulo inversion and exponentiation, the **Extended Euclidean algorithm** and **Fast-powering** were used. This algorithm allows for the efficient calculation of modulo inverse of the $\mathbb{Z}_n$ ring, which is necessary for division and power calculations. The algorithm was implemented using a custom function in Python [16].

In addition to the standard data structures such as integers, sets, and lists, specific data structures such as arrays were used to optimize the performance of the implemented algorithms. The NumPy library was used to create and manipulate arrays.

2.2 Programming Language and Environment

The algorithms for $\mathbb{Z}_n$ rings were implemented in ***Python 3.11***. Python was chosen for this study due to its popularity, readability, and availability of a wide range of libraries for numerical computation and symbolic mathematics.

The implementation was developed on a *Windows 10* operating system with *8GB of RAM* and an *Intel i7 processor*. The IDE was *PyCharm*, which is an editor/compiler of Python that includes many of the necessary libraries and tools for scientific computing and data analysis [2,10].

The following external libraries were used in the implementation:

NumPy: a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

SymPy: a Python library for symbolic mathematics, which supports a wide range of algebraic operations

SageMath: a Python library for mathematical computation, which supports a wide range of mathematical operations including algebraic operations and symbolic computations.

The specific version of the libraries used in the implementation was:

- ***NumPy 1.24.1***
- ***SymPy 1.11.1***
- ***SageMath 1.3.0***
- ***Python 3.11***

2.3 Evaluation Methodology

The performance of the implementation was evaluated using a set of test cases, including basic operations and more complex operations. The test cases were designed to cover a wide range of scenarios and to stress the implementation with large inputs. The test cases were divided into two categories [3,14]:

1. Basic operations: The basic operations include addition, subtraction, multiplication, and division. These test cases were designed to evaluate the correctness of the implementation and to measure the execution time of the basic operations.

2. Complex operations: Complex operations include modulo inversion, power calculation, and polynomial operations. These test cases were designed to evaluate the efficiency and scalability of the implementation and to measure the execution time of the complex operations.

The performance metrics used to evaluate the implementation were execution time and memory usage. The execution time was measured using the built-in time library in Python and memory usage was measured using the memory-profiler library.

The test cases were run on the implementation and on a reference implementation, which was implemented using another programming language known to have an efficient implementation of

the same operations. The results were then compared and analyzed to determine the efficiency of the implementation.

2.4 . Experiment Design

The experiment designs were as follows:

A sample of $\mathbb{Z}_n$ rings were selected, with different moduli n, ranging from small values to large values. The test cases were run on the implementation and on a reference implementation, which was implemented using another programming language known to have an efficient implementation of the same operations. The test cases were run multiple times for each sample and the results were averaged to reduce the impact of random variations. The results of the implementation were compared to the results of the reference implementation to determine the efficiency of the implementation [13].

To ensure the robustness of the results, a sample size of 50 $\mathbb{Z}_n$ rings was selected for each moduli n. This sample size was chosen based on the sample size calculation, considering the effect size, power, and alpha level. The test cases were run on a Windows 10 operating system with 8GB of RAM and an Intel i7 processor. The IDE was PyCharm, which is an editor/compiler of Python that includes many of the necessary libraries and tools for scientific computing and data analysis [13].

3.Results

3.1. Generation of members of $\mathbb{Z}_n$ and restricting modulo value n.

For this research we had to restrict modulo value n to a prime number, and also from the definition of a $\mathbb{Z}_n$, It's a best that before implementing any modulo arithmetic algorithm we implement a program that can generate all the elements of any given $\mathbb{Z}_n$. Under this restriction part we consider using the python modulo operator **%** which shall be used during performance comparison. The other values included in the restriction are the members of $\mathbb{Z}_n$ i.e. for any value a *or* b chosen to be a member $\mathbb{Z}_n$ it must be in the generated set $\mathbb{Z}_n = \{0, \ldots\ldots\ldots, (n-1)\}$. Therefore, the program that will help as do all that during implementation is:

```
n = int(input('Enter modulo value: '))
if n > 1:
   for i in range(2, int(n/2) + 1):
      if (n % i) == 0:
         print(n, "is not a prime number")
         break
   else:
      """ if the modulo value is prime
      this step generates elements of the ring"""

      elements = {x for x in range(n)}
      print(f'\u2124{n} = {elements}')

      a = int(input('a = '))
      if a in elements:
         b = int(input('b = '))
         if b in elements:
```

```
    """ if both a & b are in the  ring  ℤ_n you can
    define your algorithm function here"""
```

3.2. Basic operations implementation

From the restriction description codes, we defined a custom function of any algorithm, so starting with the implementation of algorithms for basic modulo operation i.e. multiplication and addition is a merit for latter convenience.

3.2.1. Modulo addition

We consider modulo addition algorithm($Algo$ **1**), for our first implementation when defining a custom function called "***mod_add***" with parameters x, y, z as placeholders of integers a, b, n respectively;

```
def mod_add(x, y, z):
```

This function will be called during execution of the program for any modulo addition output required. Inside the function we then implement the modulo addition algorithm($Algo$ **1**). The function together with the algorithm will now be a program of the following structure;

```
def mod_add(x, y, z):
    if (x + y) < z:
        return x + y
    else:
        return x + y - z
```

Which only get executed when called, for example if we run program below:

```
n = int(input('Enter modulo value: '))
if n > 1:
    for i in range(2, int(n/2) + 1):
        if (n % i) == 0:
            print(n, "is not a prime number")
            break
    else:
        """ if the modulo value is prime
        this step generates elements of the ring"""

        elements = {x for x in range(n)}
        print(f'\u2124{n} = {elements}')

        a = int(input('a = '))
        if a in elements:
            b = int(input('b = '))
            if b in elements:

                """ if both a and b are in the ring you can
                define your algorithm function in the next code flow"""
                def mod_add(x, y, z):
                    if (x + y) < z:
                        return x + y
```

```
        else:
            return x + y - z
    """function mod_add not called"""
```

The program generated the elements of $\mathbb{Z}_n$, ask for the values a, b as the inputs for modulo addition operation, but will not return the value of $(a + b)$ $\boldsymbol{mod}$ n since we did not call the function during the execution of the code. Check the output below:

Output:

```
"C:\Program Files\Python311\python.exe" C:\Users\USER\Documents\mod_add.py
Enter modulo value: 19
ℤ19 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
a = 14
b = 15

Process finished with exit code 0
```

Therefore, the program above must be called with the integer values a, b, n to return the modulo addition operation.


```
def mod_add(x, y, z):
    if (x + y) < z:
        return x + y
    else:
        return x + y - z
print(mod_add(a,b,n))
```

Now running the program with ***mod_add*** function called which have been done inside the *print ()* function to display the output, the code will return the modulo addition value of the integers a, b modulo n. Therefore, the general implementation of modulo addition of $(Algo\ \mathbf{1})$ in python is given by the source code below:

```
n = int(input('Enter modulo value: '))
if n > 1:
    for i in range(2, int(n/2) + 1):
        if (n % i) == 0:
            print(n, "is not a prime number")
            break
    else:
        elements = {x for x in range(n)}
        print(f'\u2124{n} = {elements}')

        a = int(input('a = '))                  #---------- Code 1
        if a in elements:
            b = int(input('b = '))
            if b in elements:
                def mod_add(x, y, z):
                    if (x + y) < z:
```

```
            return x + y
        else:
            return x + y - z
    print(f'{a}+{b} = {mod_add(a, b, n)}(mod {n})')
```

Using the general source code implementation for ($Algo$ **1**) to generate elements of a ring $\mathbb{Z}_{23}$ and perform a modulo addition arithmetic, we get the following output:

Output:

"C:\Program Files\Python311\python.exe" C:\Users\USER\Documents\mod_add.py

Enter modulo value: 23

ℤ23 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22}

a = 19

b = 12

19+12 = 8(mod 23)

3.2.2. Modulo multiplication implementation

For modulo multiplication algorithm, we implemented every step described modulo addition part except when defining the custom function named ***mod_mult*** where we have to use the algorithm($Algo$ **2**) in chapter 2 of this text. This led to a code source below that gives an output of any multiplication done in the $\mathbb{Z}_n$ ring.

The modulo multiplication algorithm ($Algo$ **2**) implementation general source code is:

```
n = int(input('Enter modulo value: '))
if n > 1:
   for i in range(2, int(n / 2) + 1):
      if (n % i) == 0:
         print(n, "is not a prime number")
         break
   else:
      elements = {x for x in range(n)}
      print(f'\u2124{n} = {elements}')
      a = int(input('a = '))
      if a in elements:
         b = int(input('b = '))              #----------   Code 2
         if b in elements:
            def mod_mult(x, y, z):
               if (x * y) < z:
                  return x * y
               else:
                  return (x * y) - (((x * y) // z) * z)
            print(f'{a}*{b} = {mod_mult(a, b, n)}(mod {n})')
```

You can as well generate elements of $\mathbb{Z}_n$ ring and perform modulo multiplication arithmetic. For example, $\mathbb{Z}_{19}$ the output will be:

```
Enter modulo value: 19
ℤ19 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
a = 8 b = 16
8*16 = 14(mod 19)
```

Note: Modulo addition and modulo multiplication algorithms($Algo$ **1**),($Algo$ **2**) implemented above can be done also in python using the modulo operator **%** this is well described in [9] on operator section.

We can implement any modulo arithmetic operation python using **%** as well. For example, to generate a $\mathbb{Z}_{19}$ ring and perform modulo multiplication we will have the following source code:

```
n = int(input('Enter modulo value: '))
if n > 1:
   for i in range(2, int(n / 2) + 1):
      if (n % i) == 0:
         print(n, "is not a prime number")
         break
   else:
      elements = {x for x in range(n)}
      print(f'\u2124{n} = {elements}')              # -------- Code 3
      a = int(input('a = '))
      if a in elements:
         b = int(input('b = '))
         if b in elements:
            def mod_mult(x, y, z):
               return (x * y) % z
         print(f'{a}*{b} = {mod_mult(a, b, n)}(mod {n})')
```

Output:

```
Enter modulo value: 19
ℤ19 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
a = 8
b = 16
8*16 = 14(mod 19)
```

Notice that when defining the function ***mod_mult*** we did not use the algorithm ($Algo$ **2**) but instead used the in-built modulo operator **%** but got the same output as the previous one. This justify that there are existing ways of implementing the algorithms, but what are their efficiency compared to ($Algo$ **1**)&($Algo$ **2**)? That is the memory used, and code running time. All shall be viewed during algorithm complexity analysis.

3.2.3. Modulo addition and multiplication table for $\mathbb{Z}_n$

Using the modulo addition and multiplication algorithm we can write a source code to generate an addition or multiplication table for any $\mathbb{Z}_n$ ring where it will be simple to determine an inverse of any element in $\mathbb{Z}_n$ if they exist or get the value of $(a \times b)\boldsymbol{mod} n \; \forall \; a, b \in \mathbb{Z}_n$.

The code source provided for this research will be the one that generates a multiplication table into a graphical user interphase, the same code can be used to generate addition table by changing the modulo multiplication algorithm to modulo addition algorithm. The code source below will display a multiplication table for any n modulo ring needed by the user into a GUI.

Here is an example of generated modulo multiplication table of the ring

$$\mathbb{Z}_{13} = \{0,1,2,3,4,5,6,7,8,9,10,11,12\}$$

Table 1: Multiplication table of $\mathbb{Z}_{13}$ generated by ModTable calculator.

Multiplication Table for Z_13

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

From table 1 we can easily find modulo inverse of any element in $\mathbb{Z}_{13}$ if it exists that is for $a \in \mathbb{Z}_{13}\ \exists\ a \in \mathbb{Z}_{13}\ s.t\ (a \times b)\boldsymbol{mod}13 = 1$. Therefore, without using a table there exists an arithmetic algorithms for finding modulo inverse of elements in $\mathbb{Z}_n$ rings.

3.3 Complex operations implementation

3.3.1. Modulo inversion algorithm

Modulo inversion algorithm comes from a multiplicative inverse of an element of a ring that is $for\ \boldsymbol{a} \in \mathbb{Z}_{\boldsymbol{n}}\ \exists\ \boldsymbol{b} \in \mathbb{Z}_{\boldsymbol{n}}\ \boldsymbol{s.t}\ ab = 1$ that is $b = a^{-1}$. It is important to note that modulo inverse of an element in $\mathbb{Z}_{\boldsymbol{n}}$ exist only if the element is a co-prime to the modulo value **n.**

Implementation of modulo inverse algorithm in $\mathbb{Z}_{\boldsymbol{n}}$ can be done in different ways in python. We'll use the methods and later in during the complexity analysis, we'll determine which one is better.

i. Basic implementation of modulo inversion algorithm

We know from the fact discussed above that the inverse of an element $a \in \mathbb{Z}_{\boldsymbol{n}}$ belongs to $\mathbb{Z}_{\boldsymbol{n}}$**.** So the basic approach we can implement in our code is to:

- define a function e.g. ***Mod_mult_inverse*** that iterate from 0 to $n-1$,
- Let the function check of any $\boldsymbol{b}$ element in range $[0, n-1]$ that when multiplied to $\boldsymbol{a}$ gives the value 1 $\boldsymbol{mod}\ n$ i.e. $(a \times b)\boldsymbol{mod} n = 1$.

For this you can use either modulo multiplication described before or the modulo operator **%**, we will provide the code using modulo operator **%** for this implementation and for the rest of the text in this paper.

Python code for modulo inverse basic implementation:

```
def mod_mult_inverse(a, n):
    # This iterates from 0 to n-1
    for b in range(0, n):
        # If we have a multiplicative inverse b it returns it.
        if (a * b) % n == 1:
            return b
        # If there is no inverse it tells us          ……code 4
    return 'does not exist'


print(mod_mult_inverse(5, 19))
print(mod_mult_inverse(12, 31))
```

The out of the code above will be:

```
4

13
```

```
Process finished with exit code 0
```

Since $5 \times 4 => 20\ mod\ 19 = 1$ and $12 \times 13 => 156\ mod\ 31 = 1$.

The above implementation label as basic implementation is a brute force approach which might not operate well as modulo value gets bigger. For example, if we get inverse of $a = 23$ under modulo value $n = 100{,}000{,}000{,}007$. Can we operate better? No. The program will run to infinity without producing a result. This will happen even using the table program no table will be displayed so that you can get the inverse of 23.

ii. Multiplication inverse Using Extended Euclidean Algorithm

We'll will not get much into Euclidean Algorithms in this section, but it is important to note that it finds x & y such that:

$$ax + by = gcd(a, b) \quad \ldots\ldots\ldots (i)$$

This is proved in [2]. We explore how we can use it to find the inverse of number $a \in \mathbb{Z}_n$ assuming a and n are co-prime.

- Replacing b with n in (i), we now have
  $ax + ny = gcd(a, n)$
- We know $gcd$of co-prime numbers is 1 so $ax + ny = 1$
- Taking $mod\ \boldsymbol{n}$ on both sides we now have $(ax)\boldsymbol{mod}\ n + (ny)\ \boldsymbol{mod}\ n = 1\ \textbf{mod}\ n$ which gives $(ax)\boldsymbol{mod}\ n = 1\ \textbf{mod}\ n$ since $(ny)\boldsymbol{mod}\ n = 0$ .Now x will be the inverse of a from the results obtained.

Therefore, note that for any two integers a , b Extended Euclid's Algorithm finds three things $x, y, gcd(a, b)$

Such that $ax + by = gcd(a, b)$.

*****Anyone reading this paper and might not be familiar with Euclid's Algorithm can consider reading about it from the references provided* [2] ****

We implement this algorithm in python programming language to find inverse of an integer $a \in \mathbb{Z}_n$ using recursion of the custom function ***extended euclidean algorithm*** inside the ***mod_inv*** custom function.

This will return a list of size 3, containing $x, y, gcd(a, n)$ where x the inverse of is $a\ \boldsymbol{mod}\ n$. Whenever $x < 0$ we will add n to it since the inverse is in the domain $[0\ , (n - 1)]$.

We assume that the user is aware of the restriction made for modulo value n in this paper. (Prime numbers only). These assumptions motivates the values used in the program to be co-prime, if the values used are not co-prime the program will notify.

The source code will be of the structure below:

```
value1 = int(input("Enter integer value in Zn: "))
value2 = int(input("Enter modulo value n: "))
def mod_inv(a, n):
    gcd, x, y = extended_euclidean_algorithm(a, n)
    if gcd != 1:
```

```
        raise ValueError("a and n are not coprime.")
    if x < 0:
        x = x + n                    # ……..code 5
    return x

def extended_euclidean_algorithm(a, b):
    if b == 0:
        return a, 1, 0
    else:
        gcd, x, y = extended_euclidean_algorithm(b, a % b)
        return gcd, y, x - (a // b) * y
print(mod_inv(value1,value2))
```

Example using value 5 mod 11 the output is:

```
Enter integer value in Zn: 5
Enter modulo value n: 11
9
```

Since $5 \times 9 = 45 = 1\ mod\ 11$

Using this algorithm we can find inverse of values with having a larger modulo value e.g. $n = 100000007$.

Note that modulo inversion algorithm implemented above works only when a and n are co-prime, i.e. $gcd(a,n) = 1$. In this case, the algorithm is guaranteed to find a unique inverse x in the range $0 \leq x < n$. If $gcd(a,n) \neq 1$, then a does not have multiplicative inverse in the $\mathbb{Z}_n$ ring.

3.3.2. Modulo exponentiation algorithm

Modulo exponentiation algorithm is used to compute $a^b(mod\ n)$ in $\mathbb{Z}_n$ ring, where $a, b,$ and n positive integers are. In this discussion we assume results for 2^{10} , suppose we raise 2 to a larger value say 100000000000 maybe for cryptography applications, We'll have to look for an algorithm that if implemented in a programming language will help us perform the operation fast and more effectively.

We used the algorithm of squaring/binary exponentiation during implementation in python programming language. We consider the following during implementation:

- Convert b to its binary representation.
- Initialize a variable, let's call it $result$, to 1.
- For each *bit* in b, starting from the least significant *bit*:
    - ✓ Square result, i.e. $result = result \times result\ (mod\ n)$.
    - ✓ If the bit is 1, multiply $result$ by a, i.e. $result = result \times a\ (mod\ n)$.
- After all bits in b have been processed, the value of result is $a^b(mod\ n)$.

For example, suppose we want to compute $3^{13}(mod\ 7)$. First, we convert 13 to binary:1101 Then, we initialize result to 1. We start with the least significant bit, which is 1. So we multiply result by 3, i.e. $result = 1 \times 3\ (mod\ 7) = 3$. Next, we square $result$, i.e. $result = 3 \times 3\ (mod\ 7) = 2$. The next bit is 0, so we just square $result$ again, i.e. $result = 2 \times 2\ (mod\ 7) = 4$. The most significant bit is 1, so we multiply result by 3, i.e. $result = 4 \times 3\ (mod\ 7) = 5$. Therefore $3^{13}(mod\ 7) = 5$.

Note that we can use the same algorithm to compute $a^b(mod\ n)$ for any positive integers a, b, and n.

We can implement this algorithm in python using the source code below:

```
def mod_exp(base, exponent, modulus):
    result = 1
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus              #..... code 6
        exponent = exponent // 2
    return result


print(mod_exp(2, 1000000000, 13))
```

Output

```
3

Process finished with exit code 0
```

The time complexity of this algorithm is $O(log\ b)$, where b the exponent. This is much more efficient than computing a^b directly and then taking the modulus, especially when b is very large.

3.4. $\mathbb{Z}_n$ *Algorithm implementation analysis*

3.4.1. Modulo addition implementation analysis

Under addition implementation, we used the algorithm ($Algo$ **1**) for implementation, see Code 1 and then later stated we can use the built in modulo operator % for the implementation, see Code 3 which is a multiplication implementation, we can replace * with + to make it addition implementation.

Using ***time(), memory_profiler()*** modules to analyze the code execution time and memory usage, we choose a larger modulo value for the $\mathbb{Z}_n$ ring say $n = 10007$, providing $\mathbb{Z}_{10007}$ where we can choose any elements of the ring for any implementation. We then compare the running time, memory usage and time complexity of the two implementations; **Implementation 1**- when ($Algo$ **1**) is used; **Implementation 2** when modulo operator % is used. Using $n = 10007$ as the

ring size and the same value of $a, b \in \mathbb{Z}_{10007}$ for both implementations, the outcome for the comparison is as shown in the table 2:

Table 2: Time complexity, average memory usage, execution time of Code 1

Algorithm	Time complexity	Execution time (seconds)	Memory Usage (megabytes)
Implementation 1	$O(nlogn)$	0.0287	39.23
Implementation 2	$O(nlogn)$	0.0208	39.01

In table 2, we used $\mathbb{Z}_{10007}$ and a set of ordered (a, b) elements of $\mathbb{Z}_{10007}$ i.e. $\{(1005,10006), (19,1097) \ldots \ldots\}$ which provided an average of execution time and memory usage.

3.4.2. Modulo multiplication implementation

Under modulo multiplication we used (*Algo* **2**) and modulo in-built operator % Code 3 using ***time()*** and ***memory_profile()*** modules for the analysis of the codes with a $\mathbb{Z}_{10007}$ ring and same sets of data used in the addition part, we have the following outcome in table 3 for the complexity, execution time and memory usage:

Table 3: Time complexity, average memory usage, execution time of Code 2 & Code 3

Algorithm	Time complexity	Execution time (seconds)	Memory Usage (megabytes)
Implementation 1	$O(nlogn)$	0.0156	43.935
Implementation 2	$O(nlogn)$	0.0149	41.618

With **Implementation 1** representing where (*Algo* **2**) was used and **Implementation 2** is where we used in-built modulo operator %. This summarizes the basic implementations we made in python.

3.4.3. Modulo inversion implementation

Modulo inversion of elements in $\mathbb{Z}_n$ rings was considered as one of the complex operations to implement in a programming language. We had two implementations that is the basic one and the one we used the one we used the Extended Euclidean algorithm. See **code 4** & **code 5** . The analysis below provides performs of brute force modulo inverse implementation (**code 4**) and Extended Euclidean Algorithm Implementation.

We considered using different modulo values to visualize the difference between the two implementations.

mod_mult_inverse() represents the function of brute implementation, ***mod_inv()*** represents the function of implementation using Extended Euclidean Algorithm

Table 4: Outputs of modulo inverse implementations i.e. *code 4*&*code 5*

a	n	***mod_mult_inverse(a, n)***	***mod_inv(a, n)***
5	19	4	4
23	97	38	38
10	300	does not exist	ValueError: *a, n* not prime
23	10,000,007	5217395	5217395
23	1,000,000,007	-	739130440

Using the outcomes above, we analyze the average execution time, memory usage and then determine the complexity of the two implementations. See **Table 5** below.

Table 5: Complexity, average memory usage, execution time of *mod_mult_inverse()* ***&*** *mod_inv()*

Algorithm	Complexity	Execution time (seconds)	Memory Usage (megabytes)
mod_mult_inverse()	$O(n)$	3.878	351.739
mod_inv()	$O(logn)$	0.041	40.999

4.Conclussion

The ***mod_inv*** function is more efficient and reliable than the ***mod_mult_inverse*** function. It uses the extended Euclidean algorithm to find the inverse of ***a*** modulo ***n***, which is faster than checking all possible values of **b**. additionally, it works correctly for all values of ***a*** and ***n*.**

Therefore, based on the results and discussions presented, it was evident that the implementation of algorithms for $\mathbb{Z}_n$ rings in Python programming language is a viable approach. The findings have highlighted the potential of using this implementation for various mathematical applications. It is recommended that further research to be conducted to explore the full capabilities of this implementation and its possible applications in real-world scenarios.

5. Recommendation

Based on our analysis, we provide the following recommendations for improving the implementation of these algorithms and extending their functionality. Optimize the algorithms for efficiency, integrate the algorithms into larger applications, extend the algorithms to support different data types, improve the usability and user experience of the algorithms and validate and test the algorithms to ensure their correctness and robustness

Acknowledgments

We acknowledge the support from teaching staff at the department of Mathematics and Physical Sciences, Maasai Mara University, Kenya for providing support during this study. Many thanks to all I really enjoyed the discussion and sharing of ideas

Declarations of interest

Declarations of interest: none

Funding Statement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data Availability Statement

All the data used are enclosed in this manuscript and any supplementary sheets provided

CRediT author statement

Kevin Otieno: Conceptualization, Methodology, Software Kevin Otieno, Otieno Francis.: Data curation, Writing- Original draft preparation. Kevin Otieno, Njuguna Edward, Otieno Francis: Visualization, Investigation. Kevin Otieno, Njuguna Edward: Supervision.: Kevin Otieno, Njuguna Edward: Software, Validation.: Otieno Francis: Writing- Reviewing and Editing,

Reference

[1]. Hoffstein, Silverman. *An Introduction to Mathematical Cryptography*. San Francisco: Springer, 2008
[2]. Edward Cherowitzo. *The Extended Euclidean Algorithm*. Mathematical and Statistical Sciences. University of Colorado Denver. *http://www-math.ucdenver.edu/~wcherowi/courses/m5410/exeucalg.html*
[3]. I.N.Herstein. *Topics in Algebra 2nd Edition.* John Wiley & Sons, 1991
[4]. G. Valiente. *Algorithms on Trees and Graphs.* Switzerland: Springer Nature. 2021: Pg287-Pg385
[5]. *Handbook of Applied Cryptography*. Elebrary.net *https://ebrary.net/134434/computer_science/integers_modulo*
[6]. Beazley, Jones. *Python Cookbook 3rd Edition*. United State. O'Reilly; 2013: Pg217-Pg240
[7]. Iuliana Ciocanea Teodorescu. *Algorithms for finite rings*. General Mathematics. Universitéde Bordeaux; Universiteit Leiden, 2016. English version.
[8]. Zhijian Liu. *Ring of Integers modulo n.* subaiwen.github.io. 2020 *https://subaiwen.github.io/integer-modulo/#:~:text=In%20the%20ring%20%24Z%2FnZ%24%2C%20%E2%80%9Cthe%20integers%20modulo%20n%E2%80%9D%2C,4k%2C%202%20%2B%204k%2C%20and%203%20%2B%204k*.
[9]. W3.CSS. *Python Tutorial*. W3Schools.com Refsness Data 1999-2023 *https://www.w3schools.com/python/default.asp*
[10]. Rookies Lab. *Fast Powering Algorithm C++ and Python Implementation.* 2013
[11]. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd ed.)*. Addison-Wesley Professional.
[12]. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd ed.)*. MIT Press.
[13]. Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.
[14]. Boneh, D., & Shoup, V. (1999). *A Graduate Course in Applied Cryptography*. Springer.

[15]. Zanella, A., Bui, N., Castellani, A. P., Vangelista, L., & Zorzi, M. (2014). *Internet of Things for Smart Cities. IEEE Internet of Things Journal*, 1(1), 22-32.

[16]. Tirthapura, S., & Ramaswamy, S. (2009). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.

[17]. Hardy, G. H., & Wright, E. M. (1979). *An Introduction to the Theory of Numbers (5th ed.)*. Oxford University Press