2WF90 Software Assignment 2

Daua Karajeanes (1619675)
Atilla Rzazade (1552848)
Gergana Valkona (1676385)
Valentina Marinova (1665154)

Eindhoven University of Technology
The Netherlands
27/10/2022

Contents

1 Introduction

This document serves as the documentation of the second software assignment of the course Algebra for Security. For this assignment we have been asked to construct software that can perform certain polynomial and finite field arithmetic. The program reads a file that contains exercises, and then it outputs the answers in an answers file. These files are structured similarly to the instructions provided by the course.

All the functions work in their specified field. For the polynomial arithmetic, these functions act on the $\mathbb{Z}/p\mathbb{Z}$, where p is a prime. The inputs and outputs are given in form of an array, where the position represents the degree of the X variable at that position e.g $[1, 1, 1] = 1 \cdot X^0 + 1 \cdot X^1 + 1 \cdot X^2$. We have programmed functions for basic polynomial arithmetic such as: addition, subtraction, multiplication, long division. Some extra functions such as: extended euclidean algorithm, irreducibly check and irreducible polynomial generator, have also been programmed.

The finite field part of the assignment, we have programmed functions such that these will act accordingly in the finite field given by $\mathbb{Z}/p\mathbb{Z}[X]/h(x)$, where p is a prime and h is an irreducible polynomial. The inputs and outputs are given in form of an array, where the position represents the degree of the X variable at that position e.g $[1, 1, 1] = 1 \cdot X^0 + 1 \cdot X^1 + 1 \cdot X^2$. We have programmed functions for basic finite field arithmetic such as: addition, subtraction, multiplication, division. Some extra functions such as: inverse, primitivity check and primitive polynomial generator, have also been programmed.

2 Software

2.1 Specifications

The software is written in 3.10.7 (version of Python 3). All of the used libraries are part of the explicitly permitted libraries in the assignment description.

2.2 Purpose

The purpose of the software is to deserialize input exercise data from a file, perform the operations indicated by this data, and finally serialize and write the answer to a file. The types of operations the software is able to solve are Polynomial and Finite Field Arithmetic. More specifically, regarding Polynomial arithmetic the software is capable of performing the following:

1. Addition
2. Subtraction
3. Multiplication
4. Long Division
5. Extended Euclidean Algorithm
6. Irreducibility Check
7. Irreducible Element Generation

For Finite Field arithmetic it is capable of performing the following:

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Inversion
6. Primitivity Check
7. Primitive Element Generation

2.3 Approach to solving the stated problem

The following sections provide a comprehensive explanation of the approach the software takes to solve each of the considered operations.

2.3.1 Addition

Explanation:
The algorithm takes as an input integer modulus and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f)$, $deg(g) \in [-1, 256]$. From the description of the task is known that $p \in [2, 509]$. First, the algorithm checks, which polynomial has a greater degree and copies. A copy of the higher degree polynomial is used to fill in the answer array. After that a for-loop of iterations that equals the degree of the smaller polynomial is entered, where every element of the same degree of the two polynomials are added together. Following this calculation, a while-loop is used to ensure that the answer belongs to the given integer modulus. The output of the algorithm is an array that represents $f + g$ according to the description of the assignment.

2.3.2 Subtraction

Explanation:
The algorithm takes as an input integer modulus and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f)$, $deg(g) \in [-1, 256]$. From the description of the task is known that $p \in [2, 509]$. First, the algorithm checks, which polynomial has a greater degree and copies. A copy of the higher degree polynomial is used to fill in the answer array. After that a for-loop of iterations that equals the degree of the smaller polynomial is entered, where every element of the same degree of the two polynomials are subtracted. Following this calculation, a while-loop is used to ensure that the answer belongs to the given integer modulus. The output of the algorithm is an array that represents $f - g$ according to the description of the assignment.

2.3.3 Multiplication

Explanation:
The algorithm takes as an input integer modulus and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f)$, $deg(g) \in [-1, 128]$. From the description of the task is known that $p \in [2, 509]$. The function creates an array p, where $p = f * g$, degree of polynomial p is $degree(f) + degree(g)$. Then the function iterates through each index i of f multiplying it with each index j of g. $f[i]$ denotes the coefficient of the monomial X^i (same for j of g), the product $f[i] * g[j]$ is the coefficient of X^{i+j}. The product is added to the index $i + j$ of the product polynomial array p. After iterating through all the indices, the array goes through normalization based on the given modulus m. The normalized array is then returned by the function.

2.3.4 Long Division

Explanation:
The algorithm takes as an input integer modulus and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f)$, $deg(g) \in [-1, 256]$. From the description of the task is known that $p \in [2, 509]$. The function uses algorithm[2.2.2] mentioned in the algebra script given to us. Firstly, the remainder r is set to be f and quotient array q is initiliazed. Before the while loop starts, the function stores $lc(r)$, which is at the last index of r, same for $lc(g)$. For $lc(g)$, the function also determines the inverse. A while loop iterates with the condition that $degree(r) \geq degree(g)$ and r is not a zero polynomial. The formula in the book states that $q = q + lc(r) * lc(b)^{-1} * X^{degree(r)-degree(g)}$. Integer i is defined to be $degree(r) - degree(b)$, where $degree(r)$ denotes the degree of r for that iteration, for each iteration of the while loop. After that, q is calculated as previously mentioned. r for next iteration is calculated as $r = r - lc(r) * lc(b)^{-1} * X^{degree(r)-degree(g)} * g$. The values previously mentioned are used to calculate the new r and the loop continues until it does not meet one of the conditions. After the while loop terminates, the resulting q, r are returned (they are cleared from the leading zeroes). The algorithm is developed with the help of Algorithm 2.2.2 [Long Division] from the book [1].

2.3.5 Extended Euclidean Algorithm

Explanation:
The algorithm takes as an input integer modulus and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f)$, $deg(g) \in [-1, 256]$. From the description of the task is known that $p \in [2, 509]$. The function determines x, y and $\gcd(f, g)$, such that $x \cdot f + y \cdot g = \gcd(f, g)$. In other to compute the above, we have use recursion. As the base case we have when the $\deg(f)$ ¡ $\deg(g)$, we know that $x \cdot f + y \cdot g = \gcd(f, g)$ will equal $x = 0, y = 1$ and gcd $= g$. We then calculate r, which equals the remainder of g/f. we then call the extended euclidean algorithm on r and f, and variables gcd, $x1$, $y1$, contain the values of the gcd, x and y. We calculate q, which is the quotient of the division g/f. Then polynomail multiplication is used to calculate the product of q and $x1$. We then calculate x by using polynomial subtraction, to find the difference of $y1$ and $q * x1$. Lastly y takes the value of $x1$, and the gcd, x and y are returned.

2.3.6 Irreducibility Check

Explanation:
The algorithm takes as an input integer modulus and a polynomial f, where $f \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and $deg(f) \in [1, 5]$. From the description of the task is known that $p \in [2, 13]$. The function determines whether the given polynomial is irreducible or not. To do so, it uses Eisenstein's Irreducibility Criterion. The criterion is modified due to the ranges of p and $deg(f)$. The (modified) criterion is as follows. If $f = a_n * x^n + a_{n-1} * x^{n-1} + ... + a_0 * x^0$ is

irreducible, then there exists prime p such that: 1) p does not divide a_n, 2) p divides all other coefficients (i.e. $a_{n-1}, a_{n-2}, ..., a_0$), 3) p^2 does not divide a_0. Since we have a range for modulus numbers, we find all the primes in range $[2, 13]$, which is an array $= 2, 3, 5, 7, 11, 13$. A for loop is used to check for each prime whether the conditions are met. The loop terminates once it finds the right prime p. Returns "True" if it manages to find such p.

2.3.7 Irreducible Element Generation

Explanation:
The algorithm takes as an input integer modulus and am integer n, where $n \in [1, 5]$ and $deg(f) = n$. From the description of the task is known that $p \in [2, 13]$ and f is irreducible. To find such f, the function creates random coefficients in range $[2, 13]$ and index them in an array (length of array is $n + 1$). After that, the result array is checked for irreducibility, where Irreducibility Check (previously mentioned) function is used. If the result array is not irreducible, then it is added inside another array which holds all the previously generated reducible polynomials. This array is initialized before the while loop that continuously generates and checks the array for the conditions.

2.3.8 Addition

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 256]$. From the description of the task is known that $p \in [2, 509]$. First, the polynomial arithmetic addition algorithm is used to calculate the summation of the two polynomials. After that the result of the calculation together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. Lastly, the algorithm returns an array that is cleared of leading zeros, corresponding to the finite field addition of $f + g$.

2.3.9 Subtraction

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 256]$. From the description of the task is known that $p \in [2, 509]$. First, the polynomial arithmetic subtraction algorithm is used to calculate the difference of the two polynomials. After that the result of the calculation together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. Lastly, the algorithm returns an array that is cleared of leading zeros, corresponding to the finite field subtraction of $f - g$.

2.3.10 Multiplication

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 128]$. From the description of the task is known that $p \in [2, 509]$. First, the polynomial arithmetic multiplication algorithm is used to calculate the product of the two polynomials. After that the result of the calculation together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. Lastly, the algorithm returns an array that is cleared of leading zeros, corresponding to the finite field multiplication of $f \cdot g$.

2.3.11 Division

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and two polynomials f and g, where $f, g \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 256]$. From the description of the task is known that $p \in [2, 509]$. In order to calculate f/g.To calculate the division we use polynomial long division to calculate the quotient and remainder of f/g.After that the results, for the quotient and remainder of the calculation, together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. Lastly, the algorithm returns an array that is cleared of leading zeros, corresponding to the finite field division of f/g

2.3.12 Inversion

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and one polynomials f where $f \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 256]$. From the description of the task is known that $p \in [2, 509]$. To produce this algorithm, we use algorithm 4.1.5 [Finding Inverses] from the book. We first compute the extended euclidean algorithm for the polynomials f, h. If the $\gcd(f, h) = 1$, we know that the inverse of f exists and we can proceed with the algorithm, otherwise, we return NULL. if an inverse does exist, this inverse will be the x, where $x \cdot f + y \cdot h = 1$. After that the result, x, of the calculation together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. Lastly, the algorithm returns an array that is cleared of leading zeros, corresponding to the finite field inversion of f The algorithm is developed with the help of Algorithm 4.1.5 [Finding Inverse] from the book [2].

2.3.13 Primitivity Check

Explanation:
The algorithm takes as an input integer modulus, irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$ and a polynomial f, where $f \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]/(\mathbb{H})$ and $deg(h) \in [2, 6]$. From the description of the task is known that $p \in [2, 13]$. Three separate functions were defined before the primitivity check itself. The first function is a decomposition function that decomposes a number into its prime dividers. The next one checks if a number is prime, and the last function check if a polynomial f equals just to 1. The primitiity check function starts by determining the order of the field by raising the integer modulus to the power of the polynomial's degree and subtracting 1 from this result. If the order is a prime number, then f is indeed primitive. Then three variables are introduced: *check* that stores the boolean value whether f is primitive or not, *powers* that stores the powers to which the polynomial should be raised, and *results* that stores the results of the raised polynomials. Then, the algorithm starts by getting the prime divisors of the order only once. A for loop is entered, in order to calculate the desired powers to which f will be raised; this is done so by dividing the *order* over each prime divider. These values are stored in the array *powers*. With the help of the function $finite_f ield_m ultiplication$ the values of the raised polynomial to the different powers are calculated, They are stored in the array *results*. The formally defined function $is_o ne$ checks if the array *results* contains any polynomials that equal to 1. If that is the case, then f is not primitive. However, if there is not any 1 in *results*, f is indeed primitive. The algorithm is developed with the help of Algorithm 5.1.11 [Check primitivity] from the book [3].

2.3.14 Primitive Element Generation

Explanation:
The algorithm takes as an input integer modulus and irreducible polynomial (polynomial modulus) $H \in \mathbb{Z}_{\mathbb{P}}[\mathbb{X}]$, where $deg(h) \in [1, 6]$. From the description of the task is known that $p \in [2, 13]$. First, a separate function (random element in F) is defined that generates random elements from the field F. There an array of the same size as the irreducible polynomial is generated. The array together with the irreducible polynomial and the integer modulus are used in the polynomial division to compute the desired answer. The answer is cleared of leading zeros and returned back to the primitive element generation algorithm. After that the result is checked if it is a primitive by a while loop. The while loop is only entered if the randomly generated elements are not primitive. If the while loop is entered new random elements are generated. Lastly, the algorithm returns an array that contains primitive elements, which satisfy the given conditions. The algorithm is developed based on Algorithm 5.1.12 [Primitive element] from the book [3].

2.4 Limitations

The software has the following limitations:

1. The code is written in Python 3, therefore, it can only execute in this programming language.
2. The software can only read json files, so any other types are not supported.
3. The software does not support modules that are not primes and are outside of the range [2, 13].
4. The output file might be a bit unreadable for larger input files.

3 Examples

3.1 Polynomial Arithmetic

3.1.1 Addition

```python
def polynomial_arithmetic_additon(mod, f, g):
    min_length = min(len(g), len(f))
    # copying the longer array into the answer array
    if len(g) < len(f):
        a = [i for i in f]
    else:
        a = [i for i in g]

    #fills the answer array with the addition of the terms of the two polynomials
    for i in range(min_length):
        if (i < min_length):
            a[i] = g[i] + f[i]
            #makes sure that the end result is in the given mod
            while (a[i] >= mod):
                a[i] = a[i] - mod

    return a

# for testing:
user_file = open('exercise15 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(polynomial_arithmetic_additon(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['g']))
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
[30, 68, 2, 59, 47, 94, 90, 42, 19, 99, 69, 16, 59, 105, 80, 65, 16, 16, 48, 16, 7, 5, 68, 83, 6, 43, 44, 7, 15, 56, 59, 53, 74, 93, 107, 108, 38, 104, 17, 81,
74, 73, 33, 90, 85, 99, 52, 106, 96, 35, 104, 96, 66, 13, 58, 50, 65, 106, 50, 62, 54, 59, 46, 26, 90, 52, 70, 85, 101, 65, 39, 22, 78, 106, 58, 24, 89, 29, 64,
 92, 7, 84, 9, 39, 94, 74, 90, 51, 76, 49, 11, 98, 15, 62, 14, 103, 32, 39, 40, 13, 88, 30, 24, 77, 23, 6, 106, 15, 80, 63, 62, 66, 37, 98, 102, 33, 26, 25, 21,
 48, 47, 57, 4, 99, 54, 3, 92, 65, 40, 45, 12, 9, 44, 9, 70, 33, 102, 22, 106, 7, 86, 24, 23, 22, 9, 80, 60, 89, 98, 28, 104, 48, 32, 96, 85, 43, 71, 75, 38, 13
, 70, 101, 69, 37, 28, 69, 85, 66, 56, 103, 86, 6, 66, 11, 81, 72, 49, 36, 81, 52, 13, 13, 69, 106, 66, 9, 90, 33, 98, 25, 51, 30, 10, 99, 101, 101, 6, 18, 68,
40, 39, 54, 37, 28, 20, 41, 60, 79, 57, 38, 86, 77, 27, 81, 61, 7, 54, 91, 106, 1, 78, 58, 14, 84, 27, 104, 79, 28, 95, 73, 97, 15, 81, 13, 74, 48, 42]
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.1.2 Subtraction

```python
def polynomial_arithmetic_subtraction(mod, f, g):
    # copying the longer array into the answer array
    min_length = min(len(g), len(f))
    if len(g) < len(f):
        a = [i for i in f]
    else:
        a = [i for i in g]

    #fills the answer array with the subtraction of the terms of the two polynomials
    for i in range(min_length):
        if (i < min_length):
            a[i] = f[i] - g[i]
            #makes sure that the end result is in the given mod
            while (a[i] < 0):
                a[i] = a[i] + mod

    return a


# for testing:
user_file = open('exercise3 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(polynomial_arithmetic_subtraction(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['g']))
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
[14, 13, 16, 4, 2, 8, 15, 13, 14, 14, 14, 12, 6, 3, 12, 11, 6, 6, 12, 16, 9, 14, 16, 3, 6, 16, 7, 12, 1, 16, 14, 6, 9, 11, 16, 4, 11, 4, 2, 16, 1, 13, 1, 12, 14
, 14, 5, 12, 9, 1, 11, 6, 8, 6, 8, 8, 8, 11, 2, 1, 7, 0, 6, 10, 13, 14, 5, 11, 16, 12, 11, 15, 4, 9, 3, 13, 5, 10, 15, 4, 7, 12, 15, 15, 6, 3, 8, 7, 4, 4, 5, 11
, 7, 9, 12, 5, 13, 3, 14, 0, 2, 9, 16, 3, 14, 13, 2, 9, 5, 5, 14, 4, 6, 4, 11, 8, 1, 16, 16, 2, 3, 2, 11, 2, 16, 14, 9, 12, 10, 16, 12, 9, 3, 16, 14, 5, 4, 4, 2
, 2, 12, 10, 16, 7, 3, 0, 1, 16, 13, 13, 0, 12, 6, 15, 6, 16, 5, 15, 3, 10, 15, 12, 0, 5, 6, 4, 1, 13, 15, 5, 9, 15, 10, 14, 1, 6, 6, 11, 16, 8, 8, 8, 6, 14, 12
, 16, 3, 6, 16, 5, 15, 2, 14, 9, 4, 9, 1, 6, 1, 2, 7, 5, 4, 2, 3, 8, 4, 15, 15, 1, 15, 1, 11, 14, 2, 8, 12, 8, 15, 5, 5, 5, 8, 3, 4, 15, 11, 4, 3, 5, 15, 8, 6,
6, 7, 7, 6, 4]
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.1.3 Long Division

```
    inv_lcg = mod_inv(g[-1],m)
    r, q = f, [ 0 ] * (max(get_degree(f),get_degree(g))+1)

    while  get_degree(r) >= get_degree(g) and (get_degree(r) != -1):
        i = get_degree(r) - get_degree(g)
        c = (r[-1] * inv_lcg) % m
        t = [ 0 ] * (i+1)
        t[-1] = c
        q[i] += (c)
        tg = polynomial_multiplication(t, g, m)
        r = clean_array(polynomial_arithmetic_subtraction(m,r,tg))

    return clean_array(q),r

# for testing:
user_file = open('exercise2 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(polynomial_division(parsed_json['f'], parsed_json['g'], parsed_json['integer_modulus']))
```

PROBLEMS 1   OUTPUT   TERMINAL   GITLENS   JUPYTER   DEBUG CONSOLE

```
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
([3, 3, 7, 3, 10, 2, 9, 10, 10, 9, 5, 5, 6, 1, 1, 6, 7, 9, 4, 5, 9, 9, 4], [1, 6, 10, 3, 9, 4, 8, 10, 5, 5, 1, 3, 3, 3, 3, 6, 8, 8, 7, 4, 4, 2, 10, 6, 9, 9, 2,
9, 0, 2, 2, 7, 2, 10, 10, 10, 1, 5, 3, 0, 8, 9, 2, 2, 6, 5, 4, 2, 3, 5, 3, 2, 5, 5, 8, 10, 5, 8, 9, 2, 5, 8, 1, 9, 10, 8, 4, 10, 3, 5, 4, 10, 10, 7, 10, 10, 2,
6, 5, 2, 5, 7, 0, 2, 9, 7, 8, 10, 3, 6, 8, 3, 10, 7, 10, 10, 10, 8, 2, 5, 7, 4, 0, 10, 10, 4, 5, 3, 1, 6, 4, 4, 6, 2, 9, 9, 7, 6, 10, 8, 3, 0, 6, 3, 3, 0, 3, 2,
 6, 5])
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.1.4 Extended Euclidean Algorithm

```
# finds x,y and d, where d = gcd(f,g) and xf + yg = d (all values are mod m)
def polynomial_extended_euclidian(f,g,m):

    if get_degree(f) == -1:
        return g, [0], [1]

    r = polynomial_division(g,f,m)[1]
    gcd,x1,y1 = polynomial_extended_euclidian(r,f,m)

    q = polynomial_division(g,f,m)[0]
    qx1 = polynomial_multiplication(q,x1,m)

    x = polynomial_arithmetic_subtraction(m,y1,qx1)
    y = x1

    return gcd,x,y

# for testing:
user_file = open('exercise16 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(polynomial_extended_euclidian(parsed_json['f'], parsed_json['g'], parsed_json['integer_modulus']))
```

PROBLEMS 1   OUTPUT   TERMINAL   GITLENS   JUPYTER   DEBUG CONSOLE

```
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
([66, 207, 66, 115, 86, 77, 245, 148, 256, 212, 184, 211, 231, 250, 194, 95, 129, 108, 194, 173, 130, 22, 98, 34, 239, 128, 235, 197, 5, 72, 70, 11, 75, 252, 53
, 251, 200, 211, 46, 31, 107, 69, 205, 248, 218, 98, 144, 64, 260, 157, 42, 214, 174, 219, 166, 121, 110, 148, 125, 83, 246, 190, 3, 7, 171, 98, 98, 176, 162, 1
92, 132, 103, 39, 89, 111, 248, 53, 264, 27, 267, 0, 61, 108, 165, 100, 240, 252, 60, 259, 173, 260, 87, 22, 260, 231, 58, 62, 103, 35, 200, 218, 183, 134, 209,
 74, 186, 253, 250, 222, 198, 11, 111, 262, 108, 222, 164, 245, 240, 78, 93, 241, 90, 2, 7, 217, 227, 113, 203, 66, 226, 153, 121, 68, 63], [1], [0])
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.1.5 Irreducibility Check

```python
    # all linear polynomials are irreducible
    if get_degree(f) == 1:
        return True

    # all polynomials (!= 0) without a constant is reducible
    if f[0] == 0:
        return True

    # all polynomials = c + x^n are irreducible
    if f[0] != 0 and [f[i] == 0 for i in range(1,len(f)-1)]:
        return True


    # implement Eisenstein criteron

    return True

# for testing:
user_file = open('exercise16 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(polynomial_irreducibility_check(parsed_json['f'], parsed_json['integer_modulus']))
```

```
PROBLEMS 1   OUTPUT   TERMINAL   GITLENS   JUPYTER   DEBUG CONSOLE

True
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.1.6 Irreducible Element Generation

```python
def irreducible_polynomial_generator(n,p):
    # create an array (len=n) of random coeff in range(0,p)
    polynomial = element_gen(n,p)
    reducibles = [[]]

    while (not polynomial_irreducibility_check(polynomial,p)) or (polynomial in reducibles):
        if (polynomial not in reducibles):
            reducibles.append(polynomial)
        polynomial = element_gen(n,p)

    # degree(f) = n & f is irreducible
    return polynomial

# for testing:
user_file = open('exercise4 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(irreducible_polynomial_generator(parsed_json['integer_modulus'], parsed_json['degree']))
```

```
PROBLEMS 1   OUTPUT   TERMINAL   GITLENS   JUPYTER   DEBUG CONSOLE

ents/GitHub/Assignment-2---Algebra-for-security/solve.py
[3, 2, 2, 3, 1, 1]
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.2 Finite Field arithmetic

3.2.1 Addition

```python
def finite_field_addition(mod, f, g, p_mod):
    a = polynomial_arithmetic_additon(mod, f, g)
    q, r = polynomial_division(a, p_mod, mod)

    return clean_array(r)

# for testing:
user_file = open('exercise10 (4).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(finite_field_addition(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['g'], parsed_json['polynomial_modulus']))
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
Traceback (most recent call last):
  File "c:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security\solve.py", line 304, in <module>
    print(finite_field_addition(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['g'], parsed_json['polynomial_modulus']))
KeyError: 'polynomial_modulus'
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security> & C:/Users/valko/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/valko/Docum
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
[45, 81, 102, 116, 200, 135, 47, 119, 323, 115, 264, 318, 132, 176, 62, 288, 198, 115, 251, 326, 272, 204, 130, 249, 203]
```

3.2.2 Subtraction

```python
def finite_field_subtraction(mod, f, g, p_mod):
    a = polynomial_arithmetic_subtraction(mod, f, g)
    q, r = polynomial_division(a, p_mod, mod)

    return clean_array(r)

# for testing:
user_file = open('exercise1 (3).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security> & C:/Users/valko/AppData
ents/GitHub/Assignment-2---Algebra-for-security/solve.py
[50, 263, 27, 120]
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.2.3 Division

```python
def finite_field_division(mod, f, g, p_mod):

    q , r = polynomial_division(f, g, mod)

    r_q, q =  polynomial_division(q, p_mod, mod)

    q_r , r =  polynomial_division(r, p_mod, mod)

    return clear_array(q) , clear_array(r)


# for testing:
user_file = open('exercise8 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(finite_field_division(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['g'], parsed_json['polynomial_modulus']))
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
([150, 177, 246, 72, 240, 161, 221, 158, 135, 77, 152, 223, 59, 31, 3, 44, 160, 290, 32, 113, 43, 99, 86, 228, 192, 69, 151, 171, 2, 12, 151, 130, 271, 184, 197
, 202, 162, 288, 256, 100, 17, 28, 239, 286, 29, 42, 59, 279, 24, 58, 58, 281, 99, 202, 1], [233, 185, 280, 88, 97, 279, 139, 195, 80, 229, 126, 38, 142, 8, 240
, 140, 156, 88, 112, 60, 254, 199, 98, 159, 77, 62, 80, 38, 21, 183, 91, 8, 143, 142, 70, 262, 217, 281, 188, 95, 157, 24, 204, 30, 77, 164, 71, 215, 24, 53, 20
2, 261, 1, 222, 174, 234, 61, 165, 214, 278, 271, 68, 109, 32, 208, 143, 4, 54, 36, 174, 186, 99, 144, 78, 131, 18, 20, 82, 173, 217, 143, 149, 46, 195, 193, 20
0, 266, 171, 160, 92, 191, 127, 46, 239, 236, 289, 116, 213, 47, 38, 140, 246, 45, 54, 176, 50, 238, 163, 119, 82, 134, 39, 195, 162, 31, 121, 168, 87, 267, 181
, 52, 112, 281, 47, 175, 121, 117, 98, 135, 31, 289, 98, 123, 210, 19, 21, 147, 290, 214, 128, 108, 149, 133, 151, 286, 89, 229, 259, 140, 146, 177, 81, 29, 227
, 79, 219, 13, 240, 173, 195, 35, 74, 59, 268, 120, 6, 289, 244, 226, 35, 177, 30, 47, 224, 176, 153, 273, 94, 136, 201])
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.2.4 Inversion

```python
def finite_field_inversion(mod, f, p_mod):
    d, x, y = polynomial_extended_euclidian(f,p_mod, mod)
    q_f, r_f = polynomial_division(x,p_mod,mod)

    if len(d) == 1:
        return r_f
    else:
        return None

# for testing:
user_file = open('exercise5 (2).json', 'r')
file_contents = user_file.read()
user_file.close()
parsed_json = json.loads(file_contents)

print(finite_field_inversion(parsed_json['integer_modulus'], parsed_json['f'], parsed_json['polynomial_modulus']))
```

PROBLEMS 1 OUTPUT TERMINAL GITLENS JUPYTER DEBUG CONSOLE

```
[92, 126, 1, 186, 116, 178, 89, 129, 162, 101, 177, 174, 75, 14, 33, 196, 20, 201, 193, 203, 24, 138, 28, 68, 122, 209, 193, 182, 197, 33, 185, 148, 83, 204, 38
, 7, 149, 164, 112, 178, 82, 61, 73, 70, 1, 12, 102, 87, 178, 64, 70, 51, 201, 89, 54, 40, 194, 147, 158, 10, 200, 58, 102, 28, 160, 19, 90, 95, 52, 169, 34, 21
, 39, 51, 209, 111, 84, 40, 168, 34, 195, 9, 115, 175, 96, 60, 0, 140, 170, 148, 70, 178, 153, 208, 83, 80, 76, 151, 144, 199, 188, 116, 191, 144, 180, 26, 173,
 21, 62, 152, 94, 40, 93, 24, 157, 176, 137, 103, 132, 66, 57, 138, 61, 142, 72, 58, 209, 152, 23, 195, 19, 175, 53, 143, 58, 178, 86, 19, 209, 5, 73, 128, 14,
182, 202, 106, 144, 125, 169, 114, 179, 55, 137, 0, 153, 31, 149, 100, 43, 95, 41, 79, 26, 108, 84, 61, 191, 76, 131, 151, 137, 181, 117, 72, 182, 130, 23, 78,
143, 180, 207, 193, 96, 144, 141, 133, 40, 35, 198, 27, 123, 77, 207, 189, 131, 152, 205, 146, 135, 60, 48]
PS C:\Users\valko\Documents\GitHub\Assignment-2---Algebra-for-security>
```

3.2.5 Primitivity Check

```python

def finite_field_primitivity_check(mod, f, p_mod):
    ord = (mod**get_degree(f)) - 1

    if is_prime(ord):
        return True

    check = True
    powers = []
    results = []

    ord_dec = set(decomposition(ord)) # get the prime divisors only once
    for dec in ord_dec:
        powers.append(ord // dec)
    for power in powers:
        total = f
        for i in range (power-1):
            total = finite_field_multiplication(mod, f, total, p_mod)
        results.append(total)
    for result in results:
        if is_one(result):
            check = False

    return check
print(finite_field_primitivity_check(7, [0,1], [5,2,1]))

```

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

```
curity> & C:/Users/20211233/AppData/Local/Programs/Python/Python310/python.exe c:/Us
ers/20211233/Documents/GitHub/Assignment-2---Algebra-for-security/solve.py
True
PS C:\Users\20211233\Documents\GitHub\Assignment-2---Algebra-for-security>
```

powershe
Python

3.2.6 Primitive Element Generation

```python

def primitive_element_generation(mod, p_mod):
    a = random_element_in_F(mod, p_mod)
    while not finite_field_primitivity_check(mod, a, p_mod):
        a = random_element_in_F(mod, p_mod)

    return a
print(primitive_element_generation(5, [4,1,2,1]))
```

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

```
ssignment-2---Algebra-for-security/solve.py
[2, 3, 3]
PS C:\Users\20211233\Documents\GitHub\Assignment-2---Algebra-for-security>
```

4 Conclusion

Our group managed to create a software that is able to perform 7 different operations in Polynomial Arithmetic, and another 7 ones in Finite Field Arithmetic. We managed to come up with algorithms for the different 14 tasks. In general, this assignment has not only improved our programming skills, but our team working skills as well. Compared to the first software assignment, we had better time management, since we started earlier than the deadline.

5 Contribution

5.1 Atilla Rzazade (1552848)

I mainly worked on the polynomial arithmetic, namely: multiplication, division, the extended euclidean algorithm, irreducible check and irreducible polynomial generator. I also contributed to the documentation on the explanation of the algorithms above.

5.2 Gergana Valkona (1676385)

I worked together with Valentina on Addition and Subtraction for both Polynomial and Finite field arithmetic. Also, we worked on the Primitivity Check and Primitive Element Generation from Finite Field arithmetic. After we were finished, we worked on the explanation of the algorithms that we wrote in the documentation and provided the examples in the documentation.

5.3 Valentina Marinova (1665154)

I worked together with Gergana on Addition and Subtraction for both Polynomial and Finite field arithmetic. Also, we worked on the Primitivity Check and Primitive Element Generation from Finite Field arithmetic. After we were finished, we worked on the explanation of the algorithms that we wrote in the documentation and provided the examples in the documentation.

5.4 Daua Karajeanes (1619675)

I mainly worked on the finite field multiplication, division an. I also contributed to the documentation, explaining each of the above algorithms. I also worked on some complementary functions of the software.

References

[1] Hülsing, A. *Algebra for Security*, pp. 21, October 2021.

[2] Hülsing, A. *Algebra for Security*, pp. 44, October 2021.

[3] Hülsing, A. *Algebra for Security*, pp. 54, October 2021.

hihihi, please do not be too strict, we tried our best! :(