2WF90 Algebra for security

Finite Fields programming assignment

Bogdans Afonins [redacted]

October 26, 2017

Contents

1 Preface

The following document describes the aspects of the algorithms presented in the provided code. To be more precise, how various objects are constructed and the principles behind each arithmetic operation. We also present the auxiliary functions that are used in the algorithms. Moreover, in this document you can find not only contents about the programming part of the assignment such as limitations or installation manual but also information about work distribution among the group.

Despite the fact, that the actual program is written in the Java programming language, we do not stick to the syntax of the language within document. Instead, every code sample provided here should be considered as a pseudocode and not as Java or any other existing programming language. The pseudocode follows the basic rules of programming languages in general, it has well-known primitive operations, data structures, statements and control sequences, like arrays, lists, basic operations, `if-else` statements, `for`, `while` loops and so on. We will give comments on non-trivial parts of the provided samples in this document when necessary.

All the references indicated in the following text are collected in the end of the document and can be found in the last section of the document, see 6.

2 Setup

The program is written in the Java programming language, so we assume that the reviewer has `JRE 1.8` installed on his computer, see [1]. We provide the sequence of steps that has to executed in a terminal of your choice to compile the source code and run the program.

```
> java -jar 2WF90.jar
```

3 Input

Since it was not specified how the program should process any inputs, we define the way the program handles different portions of inputs in this section.

Once the program has started, the user sees a list of options and their brief description. For example, here is the limited part of what the user sees:

```
...
Type '5' to multiply two polynomials. (1st * 2nd)
Type '6' to multiply a polynomials with.
Type '7' to execute the long division algorithm on
          two polynomials. (1st / 2nd)
Type '8' to calculate the GCD of two polynomials.
Type '9' to calculate the extended GCD of two
          polynomials. (x * 1st + y * 2nd = gcd(1st, 2nd))
   ...
```

The user is able to control the flow of the program by issuing it commands that are listed on the screen. From the provided sample, one can see that if the user types `5`, then the multiplcation procedure of two polynomials is executed and the user will be asked for additional information specific to the chosen option. In order to run through one complete phase of the program execution, we continue presenting the output of the program in case the user types `5`. The users input is colored in red, the rest is the output from the program.

```
Operation: 5
Input modulus p, where 1 < p < 100 = 7
Input a degree of the polynomial d >= 0 = 3
Input polynomial as a sequence of coefficients = 3 2 0 1
Your input polynomial: 3X^3+2X^2+1
Input a degree of the polynomial d >= 0 = 1
Input polynomial as a sequence of coefficients = 1 1
Your input polynomial: 1X^1+1
Result = 3X^4+5X^3+2X^2+1X^1+1
```

First, the chosen operation is shown. Then the modulus is asked. In order to multiply two polynomials, the program asks the user to input those, namely the program will ask for the degree and a sequence of coeffcients that represent this polynomial. As you can see, a the sequence 3 2 0 1 results in $X^3 + X^2 + 1$. So, i^{th} numer in an input sequence defines a coefficient of the term in the resulting polynomial having degree

equal to $degree - i$. The second polynomial is defined as 1 1 resulting in $X + 1$. Note, that we assume that the user explicitly specifies what terms should not be a part of the resulting polynomial by assigning 0 as the coefficient for that term. We do not advise assigning 0 to the highest order term, such input will not affect correctness of the program, but might less readable output (that is still correct, but not formatted properly).

We assume that the user follows the rules described in the assigment devoted to input values. Namely,

- A modulus must be prime in the range $(1, 100)$ and it must be an integer.
- Fractions cannot appear in a polynomial.
- The degree of a polynomial must be ≥ 0.
- When a command to decide whether two polynomials in the `mod p` setting are equal modulo a third polynomial, we assume the first two polynomials are congruent. The program itself does not verify this.
- A bounding polynomial for a field must be irreducible. The program itself does not verify this.

Note, that the assumptions mentioned above are not limitations of the program, but are advises to improve users experience while using the program.

4 Program

This section is indented to describe the main details and features of the program.

4.1 Objects representation

4.1.1 Integers

Interegs `mod p` are the simplest structures in the whole program. Each integer is represent by its value `mod p`. Integers support a set of basic operations like addition, sutraction, multiplication and division. Any time we perform any of these operations, the result is always `mod p`. Also, we can represent any integer in the `mod p` world in a different, but equal form. For example, $6 \equiv -1 \bmod 7$, hence 6 and -1 are equivalent and can be used interchangeably in the program. What is more, we can get the inverse of an integer.

4.1.2 Polynomials

Polynomials are repsented as arrays containing terms of the polynomial (array *terms*). For example, if terms = [1, 0, 0, 3], then the polynomial `f(X)` has the following form: $3X^3 + 1$ So, $terms[0]$ contains the constant factor, and each $terms[i]$, for $0 < i \leq degree(f(X))$ holds a coefficient for the term. If $terms[i] = 0$, then $0 * X^i$.

We can add, subtract, multiply polynomials, perform long division, find the gcd and extended gcd of two polynomials. check for congruence. Once a polynomial is created it is automatically reduced according to the value of p, so we always deal with reduced polynomials.

4.1.3 Finite Fields

A finite field $\mathbb{Z}/p\mathbb{Z}/_{(f(X))}$ is represented as a structure containing an array of polynomials within this field, prime p and a bounding, irreducible polynomial. One can produce multiplication/addition tables of the field, add and multiply polynomials within the field, look for the inverse of a polynomial.

4.2 Subroutines

In this section we describe the helper functions that are referenced throughout the document and will have an important role further in the text. These functions are vital and provide convenient approaches to deal with our representation of different types.

As mentioned before, the program does not support fractions, so for division of modular integers we use does not use a straightforward algorithm. When two integers must be divided, we use the inverse of the second integer and simply multiply the first integer and the inverse of the second integer, $\frac{a}{b} \pmod p = a * b^{-1} \pmod p$. The way to find the inverse of an integer is described in the lecture notes of the course, see [3] and if we relate to the inverse of an integer - this algorithm is used.

Data: integers a, b (mod p)
Result: returns the result of division a by b
1 **return** $a \cdot b^{-1}$;

Algorithm 1: divide

Data: integer x, prime p
Result: returns an integer $a = x \pmod p$ such that $a \geq 0$
1 $n \leftarrow x$;
2 **if** $n \geq 0$ **then**
3 | **return** $n \ mod \ p$);
4 **end**
5 $n \leftarrow -n$;
6 **if** $n < p$ **then**
7 | **return** $p - n$;
8 **end**
9 **if** $n > p$ **then**
10 | **return** $p - (n \ mod \ p)$;
11 **end**
12 **return** 0;

Algorithm 2: getPos

Data: integer $x \pmod p$
Result: returns number mod modulus ($x \leq 0$)
1 **return** $-(p - x.getPos())$;

Algorithm 3: getNeg

4.3 Arithmetic with polynomials

4.3.1 Summation

Summation of two polynomials is one of the basic operations we had to implement in terms of this assignment. The idea behind the algorithm is simple - add up all the corresponding terms under the modulo.

In lines $1 - 3$ we, firstly, find out what will be the *degree* of the resulting polynomial. That is at most the maximum degree of the two, since modulus might reduce the leading term of the result. Then the lengths of representations of both polynomials are being extended to $degree + 1$ (remember that the number of *terms* for polynomial of $degree = n$ is equal to $n + 1$). We do so, since the result will be stored into 'original' polynomial (x in pseudocode) and in case the length of the 'second' polynomial is less than $max(degree(x), degree(y)) + 1$ (missing terms of the 'second' polynomial have be filled with zeros in order to be added up with the terms of the 'original' polynomial).

In lines $4 - 6$ there is a loop that sums up the corresponding terms of the two (under the modulo) and after that the result of the summation is returned.

Data: polynomials a and b, prime p
Result: $a + b \in \mathbb{F}_p[X]$
1 $n\_max \leftarrow max(degree(a), degree(b))$;
2 $a.length \leftarrow n\_max + 1$;
3 $b.length \leftarrow n\_max + 1$;
4 **for** $i \leftarrow 0$ ***to*** $n\_max + 1$ **do**
5 | $a.terms[i] \leftarrow a.terms[i] + b.terms[i] \pmod p$;
6 **end**
7 **return** a;

Algorithm 4: sum

4.3.2 Difference

Mention that for `difference` the very same method implemented for summation is used. Simillary, the idea behind the algorithm is simple - subtract all the corresponding terms of the second polynomial from the original under the modulo.

In lines $1-3$ we, firstly, find out what will be the *degree* of the resulting polynomial. That is at most the maximum *degree* of the two, since modulus might reduce the leading term of the result or the subtraction might result in lower degree. Then the lengths of representations of both polynomials are being extended to $degree + 1$ (remember that the number of *terms* for polynomial of $degree = n$ is equal to $n + 1$). We do so, since the result will be stored into 'original' polynomial (x in pseudocode) and in case the length of the second polynomial is less than $max(degree(a), degree(b)) + 1$ (missing terms of the second polynomial have to be filled with zeros in order to be subtracted from the terms of the original polynomial).

In lines $4-6$ there is a loop that subtracts the terms of the second polynomial from the corresponding ones of the original polynomial (under the modulo) and after that the result of the summation is returned.

Data: polynomials a and b, prime p
Result: $a - b \in \mathbb{F}_p[X]$
1 $n\_max \leftarrow max(degree(a), degree(b))$;
2 $a.length \leftarrow n\_max + 1$;
3 $b.length \leftarrow n\_max + 1$;
4 **for** $i \leftarrow 0$ ***to*** $n\_max + 1$ **do**
5 | $a.terms[i] \leftarrow a.terms[i] - b.terms[i] \pmod p$;
6 **end**
7 **return** a;

Algorithm 5: difference

4.3.3 Scalar Multiple

For the `scalar multiple` a really simple algorithm consisting of one loop only is used. The idea is that every term of the polynomial has to be multiplied with the given multiple (under the modulo). For this needs a loop that iterates exactly $degree + 1$ times (line 1 gives exactly this number of iterations) suits perfectly. After this loop halts and the result is returned.

Data: scalar multiple s, polynomial a, prime p
Result: $a \cdot s \in \mathbb{F}_p[X]$
1 **for** $i \leftarrow 0$ ***to*** $a.length$ **do**
2 | $a.terms[i] \leftarrow a.terms[i] \cdot s \pmod p$;
3 **end**
4 **return** a;

Algorithm 6: scalarMultiple

4.3.4 Product

The algorithm used for `product` of two polynomials is more complex rather than the ones for `summation` or `scalar multiple`. However, the idea is really close to the one behind the primary school method of multiplication [5]. Basically, multiply each term of the original polynomial with each term of the second polynomial while summarazing the corresponding resulting terms (all arithmetic operations on terms are being executed under the modulo).

First of all, to execute these operations of terms the lenghts of both polynomials have to be adjusted. As you can see on lines $1-3$, we again set both to $max(degree(a), degree(b))$ (further, this number will referenced as $n_{max} + 1$). Then, in lines $4-7$ we create a storage polynomial *result* of length $(n_{max} + 1) \cdot 2$ where every term is equal to zero(*result* also collects the modulo m of the original polynomial). This length is explained by the fact that degree of the result can be at most $n_{max} + n_{max}$, but then its length has to be at least $n_{max} \cdot 2 + 1$. However, if you take a look at lines $8-12$, you will se that the length of the result has to be even and in order to secure this the length of the 'storage' polynomial is exactly $n_{max} \cdot 2 + 2$ which is equal to $(n_{max} + 1) \cdot 2$.

As it was already mentioned the method behind this operation is 'multiply each term of the original polynomial with each term of the second polynomial while summarazing the corresponding resulting terms' which exactly is being done in lines $8-12$ (to collect the result the storage polynomial *result* is used). Then, the obtained result is returned.

Data: polynomials a, b, prime p
Result: $a \cdot b \in \mathbb{F}_p[X]$
1 $n_{max} \leftarrow max(degree(a), degree(b))$;
2 $a.length \leftarrow n_{max} + 1$;
3 $b.length \leftarrow n_{max} + 1$;
4 $length \leftarrow (n_{max} + 1) \cdot 2$;
5 $result.length \leftarrow length$;
6 **for** $i \leftarrow 0$ ***to*** $n_{max} + 1$ **do**
7 | **for** $j \leftarrow 0$ ***to*** $n_{max} + 1$ **do**
8 | | $result.terms[i + j] \leftarrow result.terms[i + j] + a.terms[i] \cdot b.terms[j] \pmod p$;
9 | **end**
10 **end**
11 **return** $result$;

Algorithm 7: product

4.3.5 Division, GCD, Extended GCD

We would like to point out that three algorithms used in the program will not be described here in the detail, namely the long division algorithm, the Extended Euclidian algorithm, and the Euclidian algorithm. The reason for this is that we follow exactly the same steps as it is described in the course notes of the course. Since we do not provide the exact implementation in the Java programming language, but more abstract notation leaving all the Java specific syntax constructs, in our opinion there is no need for copying exactly the same procedures. For the long division algorithm, see [2], for the Extended Euclidian algorithm, see [4] and for the Euclidian algorithm, see [4]. Any time we refer or reason about any procedures that use one of the mentioned algorithms - you can refer to the references listed in this paragraph.

4.3.6 Congruence

The goal of this operation is determine whether polynomials a and b are congruent mod p to a polynomial c. To check that, it suffices to verify whether $a - b$ is divisble by c (under the modulo). Note that on line 1 the operation `difference` is meant under sign '$-$'.

Data: polynomials a, b, c, such that $a \equiv b \pmod p$
Result: *true* if $c \equiv b \pmod p$ and hence $c \equiv a \pmod p$, $false$ otherwise
1 $d \leftarrow a - b$;
2 $q, r \leftarrow longDivision(d, c)$;
3 **return** $r = 0$;

Algorithm 8: Congruence

4.4 Arithmetic in a finite field

This section descibes all operations implemented to perform arithmetic in a finite field.

4.5 Polynomials in a field

In order to deal with a field we have to generate all polynomials of that field. This might seem as a non-trivial task, mainly because we have to keep track of several characteristics of a field. First, its degree. None of polynomials in a field can have degree greater that the degree of the bounding polynomial. Also, polynomials of all permutations of coefficients must be generated. This process can be described recursively. We first show how to generate all possible polynomials of a certain degree mod p.

First, we explain parameters that are passed along the execution of the function. We pass the degree d of polynomials to be generated, a term t to consider in a specific recursive step, the bounding prime p, a value c to be set as a coefficient to t^{th} term, a list ℓ to which the generated polynomials are appended, an array of terms ts that is built in a recursive step.

We define the base case on lines $1 - 5$. Once the t^{th} coefficient in ts is set - the process for this specific ts should terminate in the next recursive step. Since we implement t before each recursive call, the algorithm must terminate if the initial values are defined properly. Namely, the most important arguments are $t \leftarrow 0$, $c \leftarrow 0$, $ts.length = d + 1$. A for loop starting on the line 6 goes through all availale integers in the mod p world, while a for loop on the line 7 fill ts and once it is finished we recurse to the next index of ts and $c + 1$.

Data: d, t, p, c, ℓ, ts
Result: a permutation of all polynomials of degree d mod p
1 **if** $t = degree + 1$ **then**
2 | $p \leftarrow$ polynomial out of ts (mod p);
3 | $\ell.add(p)$;
4 | **return**;
5 **end**
6 **for** $i \leftarrow c$ ***to*** p **do**
7 | **for** $j \leftarrow$ ***to*** $degree + 1$ **do**
8 | | $ts[j] \leftarrow i$;
9 | **end**
10 | $generateOfDegree(d, t + 1, p, c + 1, \ell, ts)$;
11 **end**

Algorithm 9: generateOfDegree

In order to use advantage of the `generateOfDegree` function we have to define one addition helper function that performs the inital call and collects polynomials of all degrees $< maxDegree$. `generatePolynomials` is responsible for that. On the line 1 we define an empty list that is passed to the `generateOfDegree` function and holds all generated polynomials. On the lines $2 - 5$ the for loop goes through all possible degrees for potential polynomials.

Data: degree of the bounding polynomial in a field $maxDegree$, prime p
Result: an array of polynomials within a field
1 $result \leftarrow \{\}$;
2 **for** $i \leftarrow 0$ ***to*** $maxDegree$ **do**
3 | $ts.length \leftarrow i + 1$;
4 | $generateOfDegree(i, 0, p, 0, result, ts)$;
5 **end**
6 **return** $result$;

Algorithm 10: generatePolynomials

4.5.1 Addition Table

Once we have generated all polynomials in a field, producing the addition table is trivial. We simply go through all elements and sum them. The result of each operation is stored in the resulting matrix, such that $matrix[i][j]$ is the result of adding $f(X)_i, f(X)_j \in \mathbb{Z}/p\mathbb{Z}_{(z(X))}$. Note, when when add two polynomials in a field, we use the `add` procedure defined in the field structure, not the one descried before to sum two polynomials mod p, see 13.

Data: a finite field f
Result: a 2d matrix $table$, such that $table[i][j]$ is a sum of i^{th} and j^{th} polynomials in f
1 $length \leftarrow f.elements.length$;
2 $table.length \leftarrow length \times length$;
3 **for** $i \leftarrow 0$ ***to*** $length$ **do**
4 | **for** $j \leftarrow 0$ ***to*** $length$ **do**
5 | | $table[i][j] \leftarrow sum(f.elements[i], f.elements[j], f)$;
6 | **end**
7 **end**
8 **return** $table$;

Algorithm 11: produceAdditionTable

4.5.2 Multiplication Table

The idea mentioned previously for producing addition tables applies here as well. With only one difference, $matrix[i][j]$ is the result of multiplying $f(X)_i, f(X)_j \in \mathbb{Z}/p\mathbb{Z}_{(z(X))}$. Again, the `product` procedure is defined in the field, see 14.

Data: a finite field f
Result: a 2d matrix *table*, such that $table[i][j]$ is a product of i^{th} and j^{th} polynomials in f

```
length ← x.elements.length;
for i ← 0 to length do
    for j ← 0 to length do
        table[i][j] ← product(f.elements[i], f.elements[j], f);
    end
end
return table;
```

Algorithm 12: produceMultiplicationTable

4.5.3 Summation

Data: polynomials a, b, a finite field f, prime p
Result: $a + b \in \mathbb{Z}/p\mathbb{Z}/_{f(X)}$

```
sum ← sum(a, b, p);
qr ← longDivision(sum, f(X));
return r;
```

Algorithm 13: summation

4.5.4 Product

Data: a - first polynomial, b - second polynomial, finite field x
Result: $a \cdot b \in \mathbb{Z}/p\mathbb{Z}/_{f(X)}$

```
product ← product(a, b, p);
qr ← longDivision(product, f(X));
return r;
```

Algorithm 14: product

4.5.5 Quotient

Data: polynomials a, b, a finite field f
Result: return $a * b^{-1}$

```
b^-1 ← b.inverse();
if exists(b^-1) and b^-1 ≠ 0 then
    return the inverse of b does not exist;
else
    return product(a, b, f);
end
```

Algorithm 15: Quotient

4.5.6 Irreducibility

Data: polynomial a, a finite field f
Result: *true* if a is irreducible in $\in \mathbb{Z}/p\mathbb{Z}/_{f(X)}$

```
if degree(a) ≤ 0 then
    return true;
end
for i ← 0 to f.elements.length do
    x ← f.elements[i];
    if degree(x) = 0 then
        skip;
    end
    q, r ← longDivision(a, f(X) if r = 0 and degree(a) ≠ degree(x) then
        return false;
    end
end
return false;
```

Algorithm 16: isIrreducible

Generation of irreducible polynomials in a field follows the procedure described in the course notes, see [6]. We simply use the default implementation of pseudorandom number generation of a programming language.

5 Limitations

One can point out that our implementation of the `generateOfDegree` function is not efficient. With this implementation generating a set of polynomials of very big degree is very time consuming, not because the amount of polynomials grows exponentially, but because recursive functions can result in stack overflow. The function can be changed without using recursion, just loops.

In the beginnning of this document we presented a list of assumptions that the program implements, see 3. We could have added more error handling to the program to make the interraction process smoother and easier to the user. Unfortunately, it is not done and the user is supposed to follow the rules, otherwise the program can behave unexpectedly.

6 Contribution

References

[1] Oracle. *http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html*, Java Runtime Environment 1.8

[2] Hans Cuypers and Hans Sterk. *Course notes Finite Fields*, 2WF90 - Algebra for Security, page 5, 2017, TU/e

[3] Hans Cuypers and Hans Sterk. Benne de Weger. *Part 1 - Algorithmic Number Theory*, 2WF90 - Algebra for Security, page 18, version 0.65, 2017, TU/e

[4] Hans Cuypers and Hans Sterk. *Course notes Finite Fields*, 2WF90 - Algebra for Security, page 7, 2017, TU/e

[5] Benne de Weger. *Part 1 - Algorithmic Number Theory*, 2WF90 - Algebra for Security, page 4, version 0.65, 2017, TU/e

[6] Hans Cuypers and Hans Sterk. *Course notes Finite Fields*, 2WF90 - Algebra for Security, page 31, 2017, TU/e