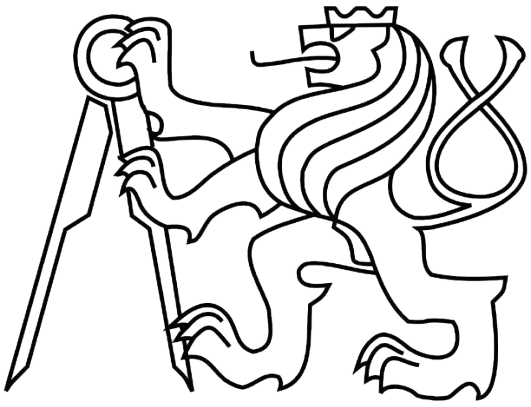
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Implementation of Goldwasser-Kilian primality test on elliptic curves

Marat Gimadiev

Supervisor: Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D.

Study Program: Open Informatics

Field of Study: Software Engineering

May, 24, 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Gimadiev** Jméno: **Marat** Osobní číslo: **474703**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Implementation of Goldwasser-Kilian primality test on elliptic curves

Název diplomové práce anglicky:

Implementation of Goldwasser-Kilian primality test on elliptic curves

Pokyny pro vypracování:

Goldwasser-Kilian algorithm uses elliptic curves to reduce the question of primality of P to the question of primality of Q < P. Iterations of Goldwasser-Kilian scheme generate certificate of primality, which can be used by everyone who wants to make sure in primality of number P.
The diploma's goal is to improve the Goldvasser-Kilian algorithm.
To achieve this goal, it is necessary to solve the following tasks:
1) Modify the Shanks-Mestre algorithm, which is used to find the cardinality of elliptic curve
2) Compare the two implementations of the algorithm with the ""standard"" algorithm and with the ""modified"" Shanks-Mestre

Seznam doporučené literatury:

1) S. Goldwasser, J. Kilian., Allmost all primes can be quickly certified
2) Crandall R. E., Pomerance C. B., Prime Numbers: A Computational Perspective
3) C. Ritzenthaler, Point counting on elliptic curves
4) S. Ishmukhametov, Methods of factoring natural numbers
5) M. Rabin, Probabilistic algorithm for testing primality

Jméno a pracoviště vedoucí(ho) diplomové práce:

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D., katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.04.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, May, 2018

Abstract

Due to the development in information security and cryptography fields, in particular public-key cryptography, the importance of searching for large prime numbers has increased. Solovay and Strassen, in 1977, and Miller and Rabin, in 1980, were developed **probabilistic primality tests, using ideas of Fermat's theorem.** Despite the fact that the answer of these tests is not always true, these algorithms had been used in public-key cryptography to construct a set of cryptosystems such as RSA, the Diffie-Hellman key-agreement protocol, El-Gamal scheme.

The Goldwasser-Kilian primality test uses elliptic curves in order to reduce the question of the question of primality of to the question of primality . Iterations of Goldwasser-Kilian scheme generate certificate of primality, which can be used by everyone who wants to make sure in primality of number .

Key words: Prime numbers, primality tests, ellitptic curves, ECPP, cryptography

AKNOWLEDGMENTS

I would like to thank to my supervisor from KFU side, Prof. Ishmukhametov Shamil Talgatovich for his help before and during this work.

A special thanks to my family and to my friends for supporting me, during the education.

Contents

List of Figures

List of Tables.

1. Introduction

Primality tests play an important role in number theory, since they are connected with problem of number factorization. The very first algorithm for primality of the number was invented by Eratosthenes in order to effectively generate a set of primes from 1 to in time

.

Due to the development in information security and cryptography fields, in particular public-key cryptography, the importance of searching for large prime numbers has increased. Solovay and Strassen, in 1977, and Miller and Rabin, in 1980, were developed **probabilistic primality tests, using ideas of Fermat's theorem.** Despite the fact that the answer of these tests is not always true, these algorithms had been used in public-key cryptography to construct a set of cryptosystems such as RSA, the Diffie-Hellman key-agreement protocol, El-Gamal scheme.

The Goldwasser-Kilian primality test uses elliptic curves in order to reduce the question of the question of primality of to the question of primality . Iterations of Goldwasser-Kilian scheme generate certificate of primality, which can be used by everyone who wants to make sure in primality of number .

The goal of this master's thesis is to make an attempt to improve Goldwasser-Kilian algorithm. To achieve this goal, it is necessary to solve the following tasks:

- Examine theoretical background starting with elementary terms of number theory
- Overview common algorithms for primality test

- Consider theoretical material about of elliptic curves, elliptic arithmetic and algorithms for calculating the order of elliptic curves
- Implement Goldwasser-Kilian algorithm using Shanks-Mestre algorithm
- Modify Shanks-Mestre algorithm to make an attempt to improve performance of Goldwasser-Kilian algorithm
- Compare and analyze standard and modified versions of algorithms

2. Prime numbers

Natural number is called prime if it has exactly two factors: 1 and itself. A number, that is not prime, is called composite number. In **number theory, prime numbers are considered as "building blocks"** for all numbers. All real numbers can be represented as rational numbers. All rational numbers as the ratio of integers. All integers are positive or negative natural numbers. All natural numbers greater than 1 are products of prime numbers.

The fact that every natural number greater than 1 is a product of primes is explained by the Basic theorem of Arithmetic.

Theorem 1. Basic theorem of arithmetic. Every positive integer greater than 1 can be written as a product of prime numbers and there is only one way to write it: , where and – primes.

Prime numbers have been studied by Euclid, a Greek mathematician (300 BC). He showed that, for any finite number there are more than **prime numbers. This fact can be restated as follows: "There** are infinitely many prime **numbers".**

Theorem 2. Set of primes is infinite.

Indeed, if we take any finite set of prime numbers, we can always find another prime number that does not yet belong to a given set.

3. Recognizing primes.

There are simple signs of divisibility for some small numbers which are based on the form of the decimal notation of the number. For example, the number is even if the last digit of it is divisible by 2. The number is divisible by 3 or 9 if the sum of the digits of a given number is also divisible by 3 or 9 respectively. The number is divisible by 5 if it ends on 0 or 5.

There are other signs of divisibility, but they are more cumbersome, since we use the decimal form of writing number in everyday life. The divisibility features described above can be used in primality tests for cutting off known composite numbers.

3.1. Trial division

The trial division method is the simplest method for primality testing. It based on the sequential division of number by all integers from 2 to $\sqrt{}$.

Algorithm 1. Trial division algorithm.

```
int trivial_div(int n) {
    for (int f = 2; f < sqrt(n); ++f) {
        if (n % f == 0) {
            return f;
        }
    }
    return 0;
}
```

Source code. c++ 1. Trial division method

The complexity of the method can be estimated $\sqrt{}$. This algorithm has an exponential estimate with respect to the length of the input number, so this method is not suitable for testing large numbers.

3.2. Eratosthenes sieve

The Eratosthenes sieve is a method **that allows you to "sift" number**s from 1 to the input number , and produce as a result an array of primes that do not exceed .

```
void sieveOfEratosthenes(int n)
{
    bool prime[n+1];
    memset(prime, true, sizeof(prime));

    for (int p=2; p*p<=n; p++)
    {
        if (prime[p] == true)
        {
            // Update all multiples of p
            for (int i=p*2; i<=n; i += p)
                prime[i] = false;
        }
    }
    for (int p=2; p<=n; p++)
       if (prime[p])
          cout << p << " ";
}
```

Source code. c++ 2. Eratosthenes sieve

Example of output of the algorithm for input

2 3 5 7 11 13 17 19 23 29

3.3. Miller-Rabin probabilistic primality test

Miller-Rabin and Solovay-Strassen tests are probabilistic and are based on the idea of using the definition of pseudo-primality. The **answer "composite" is always true, but the answer "probable prime"** means that the candidate-number can be composite.

Algorithm 2. Miller-Rabin test.

Initialization: odd integer, which must be checked; k is the number of rounds.

1) Represent in the form , where is odd. (for example, by using sequential division of by).

2) initialize

3) Cycle A: while :

a. Choose a random integer

b. By using modular exponentiation algorithm calculate

.

c. If or , then, go to the next iteration of cycle A

d. Cycle B: repeat times

i.

ii.if then return is composite

iii.if , then go to next iteration of cycle A.

e. return is composite

4) return is probable prime.

```
typedef long long llong;
inline bool millerRabin(const llong n)
{
	if (n == 2 || n == 3)
		return false;
	llong m = n % 2;

	if (m == 0)
		return true;

	const auto n1 = n - 1;
	llong r = n1;
	llong s = 0;

	m = r % 2;

	while (m == 0)
	{
		r = r / 2;
		m = r % 2;
		s++;
	}

	const auto n2 = n - 2;


	std::random_device rd;
	std::mt19937 mt(rd());
	const std::uniform_real_distribution<double> dis(0.0,
1.0);

	for (auto i = 1; i <= 20; i++)
	{
		const llong a = 2 + (dis(mt))*(n2 - 2);
		auto y = powMod(a, r, n);
		if (y != 1 && y != n1)
		{
			llong j = 1;
```

```
                        while (j <= s && y != n1)
                        {
                                y = powMod(y, 2, n);
                                if (y == 1)
                                        return true;

                                j++;
                        }

                        if (y != n1)
                                return true;
                }
        }

        return false;
}
```

Source code. c++ 3. Miller-Rabin test.

The probability of a mistake in Miller-Rabin test is – .

3.4. Solovay-Strassen probabilistic primality test

Algorithm 3. Solovay-Strassen test.

1) odd

2) Choose random and for choose a random .

3) for calculate:

a) —

b)

4) if for any , return is composite

5) otherwise return is probable prime.

```
bool solovoyStrassenAlgorithm(long long p, int rounds)
{
    if (p < 2)
        return false;
    if (p != 2 && p % 2 == 0)
        return false;

    for (int i = 0; i < rounds; ++i)
    {
        long long ai = rand() % (p - 1) + 1;
        long long ci = (p + calcJac (a, p)) % p;
        long long bi = modulo(a, (p - 1) / 2, p);

        if (!ci || bi != ci)
            return false;
    }
    return true;
```

```
}
```
Source code. c++ 4. Solovay-Strassen test

The probability of a mistake in Solovay-Strassen test is –

4. Introduction to elliptic curves.

Consider a cubic polynomial in over the field of real numbers :

(1)

Suppose that the right-hand side of the equation has distinct roots. Then the graph of this curve is called an elliptic curve.

Definition 1. An elliptic curve is the set of points satisfying equation (1), where

(2),

in addition, a special point – a point at infinity is added to the set of points. The form (1) is called the Weierstrass normal form for elliptic curves. Condition (2) is necessary to exclude singular curves.

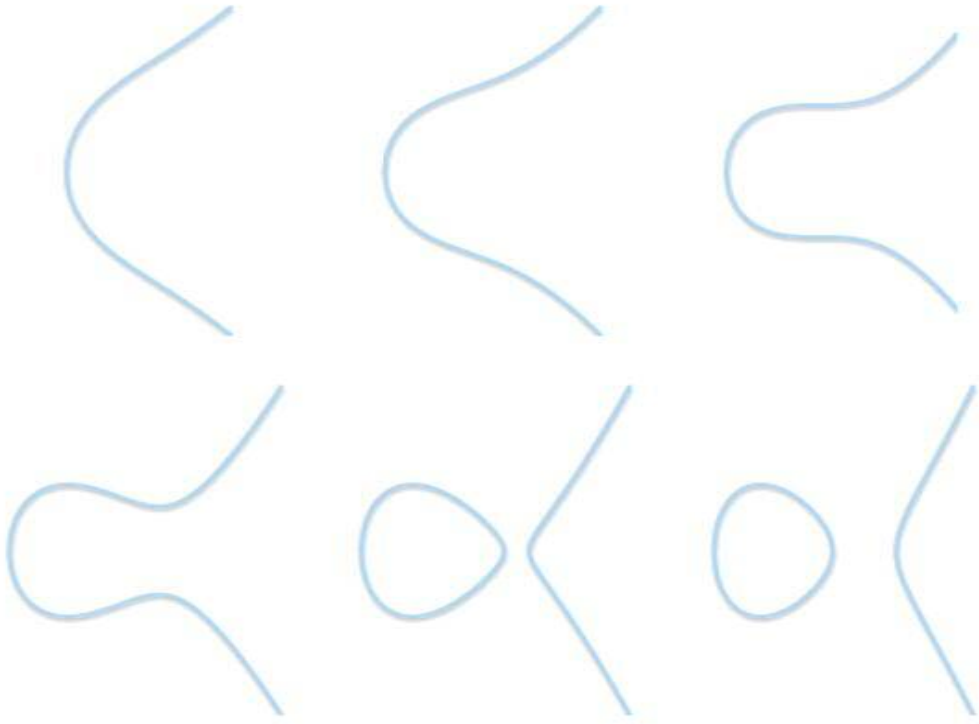


Figure 1. Example of elliptic curves.

Figure 2. Singular curves.

Left: curve with cusp , right: curve with self-intersection

On the set of points we can define the binary operation addition and denote by the symbol +. In order for the set to be an additive abelian group, the addition operation must correspond to four properties:

1) Closure. If , then .
2) Associativity
3) There is exactly one element 0:
4) Each element has an inverse:

On elliptic curves "zero" for points is a point at infinity , and the inverse point to the point is the point .

Let's define the operation addition. It is known, that

, .

The points are not equal to .

If then the line intersecting both points have slope:

$$\frac{\quad}{\quad}$$

Otherwise:

The intersection of this line with curve is a third point

:

Hence:

In addition, we define scalar multiplication:

The simplest algorithm for calculating scalar multiplication is a binary algorithm, which is very similar to the fast exponentiation algorithm.

Algorithm 4. Binary algorithm.

In order to calculate it is necessary to represent in binary form:

1)
2) for
a)
b) if ( ) then
c)
3) return Q

Pseudocode 1. Binary algorithm

For our further actions, we need to define elliptic curves over finite fields . The definition given above can be represented in this form:

Elliptic curve over finite fields become:

where is the same point at infinity, – parameters of the curve.

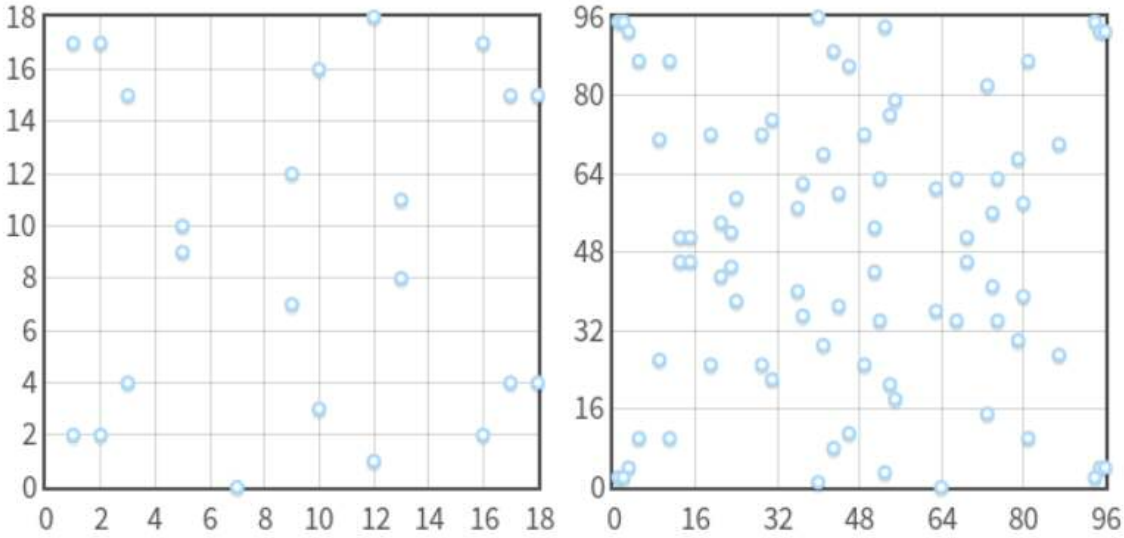


Figure 3. Elliptic curves over finite fields.

Left: ***, right:***

Formulas for calculating the sum of two points on the curve will look like:

where .

if , then

Otherwise, if , then

Remark 1. Elliptic curves over finite fields has a finite number of points.

Definition 2. The order of the elliptic curve is a number of points on it.

Theorem 3. (Hasse) The order of satisfies

Let's look at the example of calculating multiples of point . We will perform all calculations on the curve (mod 23).

We can continue calculation further, but we can already see that points are repeated cyclically. We can rewrite for any number k:

This is sufficient to prove that the set of multiples of point is a cyclic subgroup.

Definition 3. Cyclic subgroup – subgroup, which elements are repeating cyclically.

Point is a generator or a base point of cyclic subgroup. All points of the elliptic curve form a cyclic group.

Consider the question: if we know , then, what is , such that ? This is known problem is called the discrete logarithm problem in the group of points of an elliptic curve (ECDLP). This **problem is considered "difficult" because there is no known** algorithm with polynomial time that can be run on a classical computer. This problem is analogous for discrete logarithm problem, except we need to use modular exponentiation instead of scalar multiplication. ECDLP was used to build cryptosystems such as El-Gamal scheme for elliptic curves, ECDH (variant of Diffie-Hellman key-agreement protocol on elliptic curves), ECDSA – Elliptic Curve Digital Signature Algorithm.

5. Goldwasser-Kilian primality test

Goldwasser-Kilian algorithm is an algorithm for (dis)proving primality of number with expected polynomial complexity , where – absolute constant, that is assumed for all and is rigorously proved for the "majority" of primes n.

The algorithm is recursive. The idea is very simple: it is necessary to find suitable curve which order is large enough and has a large "probable prime" factor and carry out recursion with subsequent verification of the primality of these factors. During the recursion probable prime numbers become less and so continues until the number, which has to be checked, becomes so small that its primality can be proved by looking through all possible potential factors. The proof of primality at the last step justifies all previous steps and establishes a certificate of primality of the original number.

Algorithm is based on ECPP-theorem.

Theorem 4. Let – be an integer coprime with 6, let – be a pseudocurve, and be positive integers with . Assume that there exists a point , such that we can carry out the curve operations for , to find

And for every prime , dividing , we can carry out the curve operation to obtain

—

Then for every prime , dividing , we have

Moreover, if – , then is prime.

Algorithm 5. Goldwasser-Kilian primality test. Given a nonsquare integer , strongly suspected of being prime (in particular,) and presumably has already passed a probabilistic primality test), this algorithm attempts to reduce the question of primality of to that of a smaller number . The algorithm returns either the assertion " is a composite", or the assertion "if is prime then is prime", where is integer smaller than .

1) Choose a pseudocurve over . Choose a random pair , such that GCD

2) Assess curve order. Calculate the integer , that would be

3) Attempt to factor. Attempt to factor , where , and is probable prime and , but if this cannot be done according to some time-limit criterion go to step 1.

4) Choose point on curve. Choose random , such that has – .

5) Point operations. Calculate the multiple — .

If , then go to step 4.

Compute .

If , then return "**n** is **prime", otherwise return "if** is prime, then **is prime**".

Example.

Input. .

Iteration #1.

Step 1. Chosen curve .

Step 2. Order of curve .

Step 3. m can be presented as .

Step 4. Chosen point on curve:

Step 5. ,

Iteration #2.

Step 1. Chosen curve

Step 2. Order of curve

Step 3. m can be presented as

Step 4. Chosen point on curve:

Step 5. , .

Iteration #3.

Step 1. Chosen curve

Step 2. Order of curve

Step 3. m can be presented as

Step 4. Chosen point on curve:

Step 5. , .

is prime (can be proved by using trial division method)

Decision: number 10009 is prime.

The result of the algorithm can be considered as a certificate of primality. Each iteration is a chain consisting of consecutive records, showing the entire process of the recursion. This view is convenient, because it can be used by all who wish to be convinced in primality of the number. The peculiarity of the certificate is that we can begin the verification from any step of the iteration, and

many proofs of primality can be verified only by running them again from the very beginning.

6. Counting points on elliptic curve

Finding the order of the group is a nontrivial problem. However, Hesse's theorem asserts that for the curve , the order lies in the interval

– –

For example, according to the Hesse's theorem, for curve , the order lies in the interval [192, 262]. We can easily calculate points on a given curve and verify that the order of the curve 240 actually lies in the indicated interval.

6.1. Naive algorithm

Consider the most basic algorithm for counting points on an elliptic curve. Although the algorithm is called "naive" it is not based on sequential counting of points one by one, there are some theoretical calculations that can be applied. Of course, there are, better algorithms, which we will consider later, however they are much more complicated.

Definition 4. Let is integer, and is prime and . Legendre symbol – is defined as

1. – , if
2. – , if is a quadratic residue (
3. – , if is a quadratic nonresidue (

Consider curve . If is prime, then the order of the curve:

In equation ——— means the Legendre symbol. For each quadratic residue , i.e. — , there are two points on the curve: and , then they need to be added to .

Similarly, for quadratic nonresidue — , there are no points on the curve, so we need to subtract them from . For , where — , there is only on point. Adding of 1 to in this formula means adding a point at infinity.

6.2. Shanks-Mestre algorithm

For small numbers , less than 1000, we can explicitly calculate the **order of the curve using the naïve method. But** without using any optimizations, the complexity of the algorithm is estimated There is a more faster method is to select the point on curve and find all multiples . The idea belongs to two scientists D. Shanks and J. Mestre. Baby-step, giant-step algorithm was successfully used for solving discrete logarithm problem in finite fields. Using Mestre's theorem and adapting Shanks algorithm, we can calculate the number of points on an elliptic curve.

The asymptotic complexity of the algorithm . The adaptation of the Shanks algorithm for elliptic curves is based on the Hasse theorem (theorem 3). From this estimate it follows that the exact order of the group can be given in the form:

where — — .

Let , where

. Then for any point :

Assume that we found multiple of , such that

. According to Lagrange's theorem [12], we know that the order of the point always divides the order of the group. This means that we have found the order of the point or its multiple. This number is not necessarily the order of the curve. It may be that more than one zero multiple is found. However, if the selected point has order greater than , the algorithm will find a single multiple in the required interval, and this will be the true order of the curve.

The auxiliary function ind( ) – returns the index : .

The auxiliary function shanks( ).

1)

2)

3) Return the intersection

Algorithm 6. Shanks-Mestre algorithm for counting points on an elliptic curve

1) Check magnitude of .

if return

2) Initialize Shanks search

Find quadratic nonresidue .

Compute

Compute

3) Mestre cycle

Choose random

Compute ———

if continue. if , then , otherwise

.

Determine the starting point

Find Shanks intersection:

if , go to Mestre cycle

Set .

Find

Select the sign in the expression , so that

.

The first step of the algorithm that checks the value of the input number is not artificial. There is a possible situation where the existence of a singleton set of coincidences is not guaranteed.

6.3. Schoof algorithm

We have considered algorithms with the asymptotic complexity and . However, these algorithms are not suitable for numbers whose value exceeds Let's turn to the algorithm for counting points on a curve, which was invented by R. Schoof. The algorithm has a polynomial complexity , where – absolute constant.

The algorithm breaks down the order of the curve for a large number of small prime numbers , such, that and then restores it by using the Chinese remainder theorem.

Schoof algorithm begins with the consideration of the special case . According to the group order property, the order of the group is the smallest common multiple for the order of the elements

of the group. Thus, in order that the order of the group be even, it is necessary that at least one of the element of the group be of order 2. For the remaining cases, for which , algorithm uses the generalized method. According to the Frobenius endomorphism [5], we know that for . Another theorem related to the Frobenius endomorphism asserts that if the order of the curve , then

for any point on curve. This means that the Frobenius endomorphism satisfies the quadratic equation with the trace (the sum of the roots of the polynomial equals to .
Now consider only those points such that , where is any positive number. This set of points is denoted by and it is a subgroup , and maps into itself. Thus:

The relation above forms the bases of Schoof algorithm. The algorithm calculates all possible values of , by trial and error, until the value satisfying (6) is found. Division polynomials can be used for this.
Definition 5. We associate the division polynomials
, with elliptic curve , defined as follows:

while all further cases are given by

moreover, any occurrence of powers of greater than the first power has to be reduced in according to the relation .

Properties of division polynomials.

1) Division polynomial is, for odd, a polynomial in alone, while for even it is times a polynomial in alone.

2) For odd and not a multiple of degree of division polynomial is

3) For even ad not multiple of degree of division polynomial is

4) If point , then

Algorithm 7. Schoof algorithm.

Input: Elliptic curve .

Output: Number of points of curve over .

1) Choose a set of primes .

2) For each

a. if

i.if GCD(, then , otherwise

b. , compute

.

if , than compute

for :

if x-coordinates of and match, then

if y- coordinates of and also match, then , otherwise

3) Using the Chinese remainder theorem, we compute (mod M) from the equations , where M is the product of primes from the set S. return .

7. Implementation

In 1985, Lenstra first used elliptic curves for the factorization problem. After that, Shafi Goldwasser and Joe Kilian invented an algorithm for (dis)proving of primality of input number by using groups of rational points of elliptic curves over finite fields.
The main difficulty of the algorithm is to find the order of the elliptic curve. The recommended algorithm is Schoof algorithm. However, Atkin and Morain said that the problem with the Goldwasser-Kilian algorithm is that Schoof algorithm is almost impossible to implement [9], but these comments were given before Elkies and Atkin improved algorithm (SEA – Schoof-Elkies-Atkin algorithm).
In this implementation, we will use Shanks-Mestre algorithm for counting number of points on a chosen elliptic curve.
Another difficulty of the algorithm is the factorization of a number. It is necessary to check whether the number satisfies the condition that allows us to apply theorem 4, that is, is not prime, but its largest factor is . To solve this problem, we will use -Pollard algorithm.
Algorithm 8. -Pollard algorithm.

Input: – number which must to be factorized.

Output: prime factor of .

1) If return 2
2) Choose randomly , c from the interval
3)
4) while ( )
a.

b.

c.

d.

5) return

C++ programming language was used for the implementation of a program.

The results of the program are shown on Table 1, as well as in Figure 4.

The experiments were carried out on notebook with characteristics:

- CPU Intel Core i7-4720HQ, 2.60GHz
- Number of cores: 4
- Ram 8Gb

Test/candidate	811411	5534923	10999171	14963257	2147483647
1	1,33839	7,58743	2,07004	11,4965	32,6888
2	0,918449	3,23017	22,856	9,58335	2,55353
3	0,357055	2,6443	7,12046	2,4462	7,17768
4	3,21164	6,08891	10,0108	11,4965	36,7073
5	1,24545	1,00427	0,59699	28,7999	32,4111
6	3,534007	4,36304	32,0766	146,1648	4,78126
7	3,51043	6,141	9,46836	13,74154	10,21217

8	0,690446	22,7775	13,5144	0,641521	53,8173
9	2,42724	6,40522	17,5129	19,1225	31,1572
10	1,73031	63,5953	6,2996	23,8668	31,1572
11	0,3906	0,768119	8,4669	112,586	7,827
12	1,73037	3,308	16,0672	15,235	64,64188

Table 1. Goldwasser-Kilian algorithm. Elapsed time of "standard" version

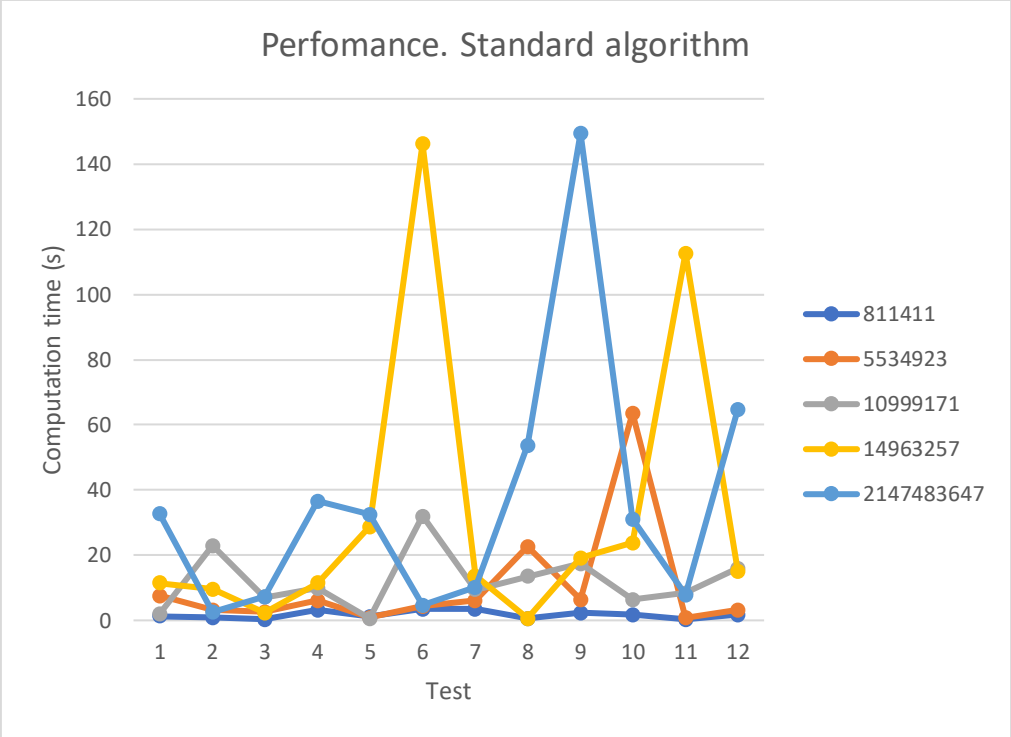


Figure 4. Goldwasser-Kilian algorithm. Elapsed time of "standard" version

8. Modification

As mentioned above, Shanks-Mestre algorithm was used to compute number of points on elliptic curve, which as also known as Baby-step, giant-step algorithm. Let us consider this algorithm more deeply.

We need to select the point on curve and find its multiples . In the original algorithm it is proposed to search the decomposition of the found in the form: , where . The parameter characterizes baby steps, and – giant steps.

The idea of improving the algorithm is to reduce the number of giant and baby steps by introducing additional type of steps. Thus, we will search the decomposition , where the parameter is bounded above by the estimate .

Results of the implementation are shown on table 2, as well as on figure 5.

Test/candidate	811411	5534923	10999171	14963257	2147483647
1	2,37931	19,4358	8,42428	30,28923	54,3154
2	2,40717	0,742953	7,74485	53,6943	20,4387
3	4,32552	7,29208	19,5627	4,694	52,5802
4	0,215344	9,6911	108,16	29,3654	35,1708
5	3,53591	0,690255	5,40241	78,7961	11,6296

6	1,67964	11,3545	14,6696	52,5802	55,8416
7	0,381833	5,03348	43,1689	77,99638	18,9097
8	2,53952	20,9282	26,58887	24,257	43,4507
9	2,23821	22,8213	93,2156	6,07863	149,434
10	0,103133	17,2679	36,0286	123,868	17,349
11	0,39796	6,33635	93,2156	72,5859	27,6867
12	2,00012	3,308	36,0286	68,2663	58,3657

Table 2. Goldwasser-Kilian algorithm. Elapsed time of "modified" version

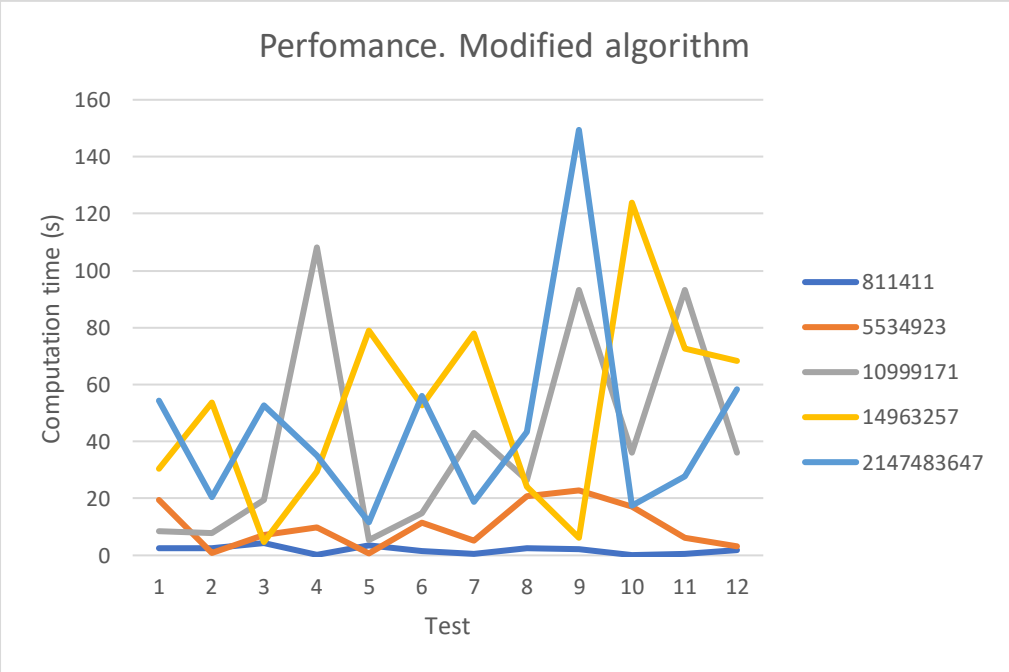


Figure 5. Goldwasser-Kilian algorithm. Elapsed time of "modified" version

9. Comparison

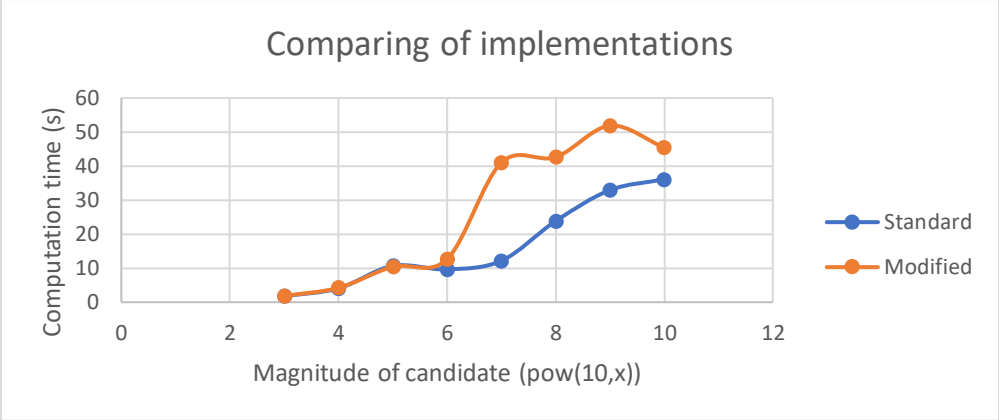


Figure 6. Comparison of implementations

Figure 6 shows a comparative graph of two implementations, depending on the magnitude of the candidate input number. We can see that our modification of the algorithm led to an increase in the computation time of algorithm.

Let's carry out the profiling of two implementations using *valgrind* tool, to find a bottleneck of the algorithm.

100.00	0.00	(0)	0x0000000000000ea0	ld-2.26.so
99.96	0.00	1	_start	source
99.96	0.00	1	(below main)	libc-2.26.so: libc-start.
99.96	0.00	1	main	source: Source.cpp, ch
99.96	1.45	1	goldwasserKilian(long l...	source: GoldwasserKil
80.70	80.70	1 199 253	multiply(long long, long...	source: EllipticCurve.h
8.35	0.60	40 403	squareRootModPrime(l...	source: ModularArithn
7.68	4.76	47 660	randomNumberInRang...	source: ModularArithn
6.51	4.04	40 399	randomNumberInRang...	source: ModularArithn
3.78	3.78	88 390	double std::generate_c...	source: random.tcc
2.05	0.72	564 093	void std::vector<long lo...	source: vector.tcc, stl_
1.18	0.55	740 460	<cycle 1>	ld-2.26.so

Figure 7. Profiler output. "Standard" version algorithm

100.00	0.00	(0)	0x0000000000000ea0	ld-2.26.so
99.97	0.00	1	_start	source
99.97	0.00	1	(below main)	libc-2.26.so: libc-start.c
99.96	0.00	1	main	source: Source.cpp, ch
99.96	1.30	1	goldwasserKilian(long l...	source: GoldwasserKil
66.08	66.08	1 043 508	multiply(long long, long...	source: EllipticCurve.h
17.39	1.22	102 399	squareRootModPrime(l...	source: ModularArithn
16.09	9.98	120 952	randomNumberInRang...	source: ModularArithn
13.62	8.45	102 412	randomNumberInRang...	source: ModularArithn
7.89	7.89	223 752	double std::generate_c...	source: random.tcc
1.87	0.56	774 425	<cycle 1>	ld-2.26.so

Figure 8. Profiler output. "Modified" version of algorithm

As we can see on figures 7-8, scalar multiplication requires huge computation time; reducing the number of steps did not lead to improvements of performance of the algorithm.

10. Conclusion

The results which were **obtained during the master's thesis:**

- Theoretical background about elementary terms of number theory was covered
- Common primality test was examined at 3rd chapter
- Theoretical background about elliptic curves, elliptic curve arithmetic was considered at 4th chapter
- Implemented Goldwasser-**Kilian algorithm with "standard"** version of Shanks-Mestre algorithm
- A modification of Shanks-Mestre algorithm was invented and implemented to make an attempt to improve the algorithm
- Both implementations were compared and profiled.

11. Bibliography

1) Sh. T. Ishmukhametov, R.G. Rubtsova, Mathematical foundations of information security, 2012

2) Sh. T. Ishmukhametov, R. H. Latypov R.G. Rubtsova, E.L. Stolov, Introduction to number theory and coding theory, 2014

3) Sh. T. Ishmukhametov, Methods of factorization of natural numbers, 2011

4) S. Goldwasser, J. Kilian, Almost all primes can be quickly certified

5) R. Crandall, C. Pomerance, Prime numbers. A computational perspective, 2001

6) C. Ritzenthaler, Point counting on elliptic curves

7) M. Rabin, Probabilistic algorithm for testing primality

8) **John J. McGee, René Schoof's Algorithm for Determining** the Order of the Group of Points on an Elliptic Curve over a Finite Field, 2006

9) Atkin, A.O.L., Morain, F., Elliptic Curves and Primality Proving

10) http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/

11) S. Sundriyal, Counting points on elliptic curve over Zp, 2008

12) https://en.wikipedia.org/wiki/Lagrange's_theorem_%28group_theory%29

13) O. Uzunukol, Atkin's ECPP (Ellitpic curve primality proving) algorithm, 2004

12. Appendix (code c++)

ModularArithmetic.h

```
#ifndef MODULARTARITHMETIC_H
#define MODULARTARITHMETIC_H
#include <vector>
#include <algorithm>
#include <ctime>
#include <random>
#include <map>
#include <limits.h>

using namespace std;
typedef long long llong;
typedef long double dlong;
typedef std::pair<llong, llong> pll;


inline llong addMod(const llong a, const llong b, const llong p)
{
	const llong res = (a + b) % p;
	return res;
}

inline llong subMod(const llong a, const llong b, const llong p)
{
	const llong res = (a - b) % p;

	return res < 0 ? res + p : res;
}

inline llong mulMod(const llong a, const llong b, const llong p)
{
	const llong res = a * b%p;
	return res;
}

inline llong randomNumberInRange(const llong left, const llong right)
{
	std::random_device rd;
	std::mt19937 mt(rd());
	std::uniform_real_distribution<> dis(0.0, 1.0);

	return left + (dis(mt))*(right - left);
}

inline llong gcd(const llong a, const llong b)
{
	if (a == 0)
		return b;
	return gcd(b%a, a);
}

inline llong weierstrass(const llong x, const llong a, const llong b,
const llong p)
{
	return (((x * x % p) * x) % p + (a * x) % p + b) % p;
}
```

```
inline llong lcm(const llong a, const llong b)
{
	return a * b / gcd(a, b);
}

inline llong pollardRho(const llong n)
{
	if (n % 2 == 0)
		return 2;

	llong x = randomNumberInRange(1, n);
	const llong c = randomNumberInRange(1, n);
	llong y = x;
	llong g = 1;

	while (g == 1)
	{
		x = ((x*x) % n + c) % n;
		y = ((y*y) % n + c) % n;
		y = ((y*y) % n + c) % n;
		g = gcd(x > y ? x - y : y - x, n);
	}
	return g;
}

inline llong powMod(const llong x, llong y, const llong z)
{
	llong t = 1;
	llong a = x % z;
	while (y > 0)
	{
		if (y % 2 != 0) {
			t = t * a%z;
		}
		y >>= 1;
		a = a * a%z;
	}
	return t;
}

inline bool millerRabin(const llong n)
{
	if (n == 2 || n == 3)
		return false;
	llong m = n % 2;

	if (m == 0)
		return true;

	const auto n1 = n - 1;
	llong r = n1;
	llong s = 0;

	m = r % 2;

	while (m == 0)
	{
		r = r / 2;
		m = r % 2;
		s++;
	}
```

```
	const auto n2 = n - 2;


	std::random_device rd;
	std::mt19937 mt(rd());
	std::uniform_real_distribution<> dis(0.0, 1.0);

	for (auto i = 1; i <= 20; i++)
	{
		const llong a = 2 + (dis(mt))*(n2 - 2);
		auto y = powMod(a, r, n);
		if (y != 1 && y != n1)
		{
			llong j = 1;

			while (j <= s && y != n1)
			{
				y = powMod(y, 2, n);
				if (y == 1)
					return true;

				j++;
			}

			if (y != n1)
				return true;
		}
	}

	return false;
}

inline void extendedEuclideanAlgorithm(llong a, llong b, llong &x, llong
&y, llong &g)
{
	if (b == 0)
	{
		g = a;
		x = 1;
		y = 0;
		return;
	}
	llong x1 = 0, x2 = 1, y1 = 1, y2 = 0;

	while (b > 0)
	{
		const llong q = a / b;
		const llong r = a - q * b;
		x = x2 - q * x1;
		y = y2 - q * y1;
		a = b;
		b = r;
		x2 = x1;
		x1 = x;
		y2 = y1;
		y1 = y;
	}

	g = a;
	x = x2;
	y = y2;
}
```

```
inline llong inverse(const llong a, const llong p)
{
	llong g, x, y;

	extendedEuclideanAlgorithm(a, p, x, y, g);

	return g == 1 ? x : 0;
}

inline llong divMod(const llong a, const llong b, const llong p)
{
	const auto res = mulMod((a % p), inverse(b, p), p);
	return res < 0 ? res + p : res;
}

inline void factorize(llong n, std::vector<llong> &factors)
{
	if (std::find(factors.begin(), factors.end(), n) !=
factors.end())
		return;

	if (n == 1)
		return;

	if (!millerRabin(n)) {
		factors.emplace_back(n);
		return;
	}

	const llong divisor = pollardRho(n);
	factorize(divisor, factors);
	factorize(n / divisor, factors);
}

inline void factorize2(llong n, std::vector<llong> &factors) {
	if (n == 1)
		return;

	if (!millerRabin(n)) {
		factors.emplace_back(n);
		return;
	}

	const auto divisor = pollardRho(n);
	factorize2(divisor, factors);
	factorize2(n / divisor, factors);
}

inline int jacobi(const llong a, const llong n)
{
	if (a == 0)
		return 0;

	if (a == 1)
		return 1;

	auto e = 0;
	auto a1 = a;
	auto quotient = a1 / 2;
	auto remainder = a1 % 2;
```

```
	while (remainder == 0)
	{
		e++;
		a1 = quotient;
		quotient = a1 / 2;
		remainder = a1 % 2;
	}

	llong s = 0;

	if (e % 2 == 0)
		s = 1;

	else
	{
		const auto mod8 = n % 8;

		if (mod8 == 1 || mod8 == 7)
			s = +1;

		if (mod8 == 3 || mod8 == 5)
			s = -1;
	}

	const auto mod4 = n % 4;
	const auto a14 = a1 % 4;

	if (mod4 == 3 && a14 == 3)
		s = -s;

	const auto n1 = n % a1;

	return s * jacobi(n1, a1);
}

inline llong squareRootModPrime(const llong a, const llong p)
{
	llong e = 0;
	llong bp, q = p - 1, n;
	const auto p1 = p - 1;

	// p - 1 = 2 ^ e * q with q odd

	auto m = q % 2;

	while (m == 0)
	{
		q = q / 2;
		m = q % 2;
		e++;
	}

	// find generator

	auto jac = 0;

	do
	{
		n = randomNumberInRange(1, p1);
		jac = jacobi(n, p);
	} while (jac == -1);
```

```
	const auto z = powMod(n, q, p);
	auto y = z;
	auto r = e;
	auto x = powMod(a, (q - 1) / 2, p);
	auto b = (a*a%p)*x%p;
	x = a * x%p;

	while (true)
	{
		if (b == 1 || b == p1)
			return x;

		llong s = 1;

		do
		{
			bp = powMod(b, pow(2, s), p);
			s++;
		} while (bp != 1 && bp != p1 && s < r);

		if (s == r)
			return 0;

		const llong t = powMod(y, pow(2, r - s - 1), p);
		y = t * t%p;
		x = x * t%p;
		b = b * y%p;
		r = s;
	}
}

inline llong sqrt(const llong n)
{
	if (n == 0)
		return 0;

	const auto r = sqrt(n >> 2);
	const auto r2 = r << 1;
	const auto s = r2 + 1;

	if (n < s*s)
		return r2;

	return s;
}

inline int legandre(llong a, llong b, llong p, llong x)
{
	auto i = 0;

	const auto y = weierstrass(x, a, b, p);
	const auto l = powMod(y, (p - 1) / 2, p);

	if (l == p - 1)
		i = -1;
	else i = l;

	return i;
}
#endif // MODULARTARITHMETIC_H
```

EllipticCurve.h

```
#ifndef EllipticCurce_H
#define EllipticCurce_H
#include "ModularArithmetic.h"

class EllipticCurve {
public:
	llong a, b, p;
	EllipticCurve()
	{
		a = 0;
		b = 0;
		p = 0;
	}
	EllipticCurve(const llong a, const llong b, const llong p) :
a(a), b(b), p(p) {}
};

class ECPoint {
public:
	llong x, y;
	ECPoint()
	{
		x = 0;
		y = 0;
	}
	ECPoint(const llong x, const llong y) :x(x), y(y) {}
	bool isPointO() const
	{
		return x == 0 && y == 1;
	}
	bool operator==(ECPoint &rhs) const
	{
		return this->x == rhs.x;// && this->y == rhs.y;
	}
	ECPoint negate(const llong p) const
	{
		ECPoint Q;
		Q.x = this->x;
		Q.y = subMod(p, this->y, p);//(p - this->y) % p;
		if (Q.y < 0)
			Q.y += p;

		return Q;
	}
};

inline bool add(const llong a, const llong p, ECPoint P, ECPoint Q,
ECPoint &R)
{
	llong lambda;

	if (P.isPointO() && Q.isPointO())
	{
		R = ECPoint();
		R.x = 0;
		R.y = 1;
		return true;
	}

	const llong ps = subMod(p, Q.y, p);
```

```
	if (P.x == Q.x && P.y == ps)
	{
		R = ECPoint();
		R.x = 0;
		R.y = 1;
		return true;
	}
	if (P.isPointO())
	{
		R = Q;
		return true;
	}
	if (Q.isPointO())
	{
		R = P;
		return true;
	}
	if (P.x != Q.x)
	{
		int i = subMod(Q.x, P.x, p);
		const auto y = subMod(Q.y, P.y, p);
		if (i < 0)
			i += p;
		i = inverse(i, p);
		if (i == 0)
			return false;
		lambda = mulMod(i, y, p);
	}
	else
	{
		const llong y2 = mulMod(2, P.y, p);
		auto i = inverse(y2, p);
		if (i < 0)
			i += p;
		if (i == 0)
			return false;
		llong x3 = mulMod(3, P.x, p);
		x3 = mulMod(x3, P.x, p);
		x3 = addMod(x3, a, p);
		lambda = mulMod(x3, i, p);
	}

	const auto lambda2 = mulMod(lambda, lambda, p);
	const auto delta1 = addMod(P.x, Q.x, p);
	R.x = subMod(lambda2, delta1, p);
	auto delta2 = subMod(P.x, R.x, p);
	delta2 = mulMod(lambda, delta2, p);
	R.y = subMod(delta2, P.y, p);

	if (R.x < 0)
		R.x += p;

	if (R.y < 0)
		R.y += p;

	return true;
}

inline bool multiply(const llong a, const llong n, const llong p, const
ECPoint P, ECPoint &R)
{
	llong k = n;
```

```
        ECPoint S = P;

        R = ECPoint();
        R.x = 0;
        R.y = 1;

        while (k != 0)
        {
                const llong m = k % 2;

                if (m == 1)
                        if (!add(a, p, R, S, R))
                                return false;

                k = k / 2;

                if (k != 0)
                        if (!add(a, p, S, S, S))
                                return false;
        }
        return true;
}

inline llong delta(const llong a, const llong b, const llong p)
{
        const auto a3 = (a*a%p)*a%p;
        const auto b2 = b * b%p;
        auto delta = (-16 * (4 * a3 + 27 * b2)) % p;

        if (delta < 0)
                delta += p;

        return delta;
}

#endif // EllipticCurce_H
```

ShanksMestreAlgorithm.h

```
#ifndef SHANKSMESTREALGORITHM_H
#define SHANKSMESTREALGORITHM_H
#include "EllipticCurve.h"


using namespace std;

inline ECPoint choosePointOnCurve(const EllipticCurve curve)
{
        auto stop = false;
        auto x = 0, y = 0;
        while (!stop)
        {

                x = randomNumberInRange(1, curve.p);
                const auto z = weierstrass(x, curve.a, curve.b, curve.p);

                if (z != 0)
                {
                        y = squareRootModPrime(z, curve.p);
                        const auto y2 = mulMod(y, y, curve.p);
```

```
                        if (y != 0 && z == y2)
                                stop = true;
                }
        }
        return { x,y };
}

class ShankMestreAlgorithm
{
        llong q, w, leftBound, rightBound, g = 1;
        EllipticCurve curve;
        void initialize(const llong a, const llong b, const llong p)
        {
                curve = EllipticCurve(a, b, p);

                q = pow(curve.p, 1.0 / 2);

                if (q * q < curve.p)
                        q++;

                leftBound = curve.p + 1 - 2 * q;
                rightBound = curve.p + 1 + 2 * q;

                w = pow(q, 1.0 / 2);

                const auto pForth = pow(w, 4);

                if (pForth < curve.p)
                        w++;
                else if (pForth > curve.p)
                        w--;
                const long double sqrt2 = pow(2, 1.0 / 2);
                w *= sqrt2;
                w = ceil(w);
        }

        llong order(const llong sigma, llong a, const llong ea, llong t,
const ECPoint P) const
        {
                const llong p1 = curve.p + 1;
                ECPoint Q = ECPoint();

                t %= 2 * q;

                if (t < 0)
                        t += 2 * q;

                auto m = p1 + t;
                auto n = p1 + sigma * t;

                if (n >= leftBound || n <= rightBound)
                {
                        if (multiply(ea, m, curve.p, P, Q))
                                if (Q.isPointO())
                                        return n;
                }

                m = p1 - t;
                n = p1 - sigma * t;

                if (n >= leftBound || n <= rightBound)
```

```
		{
			if (multiply(ea, m, curve.p, P, Q))
				if (Q.isPointO())
					return n;
		}

		return -1;
	}

	bool shanks(llong &betta, llong &gamma, const ECPoint P) const
	{
		vector<llong> A;
		vector<llong> B;

		vector<pll> S;

		// baby steps

		betta = 0;
		gamma = 0;

		for (llong i = 0; i < w; i++)
		{
			const auto m = curve.p + 1 + i;
			ECPoint temp{};

			multiply(curve.a, m, curve.p, P, temp);

			if (temp.x != 0)
				A.emplace_back(temp.x);
		}

		// giant steps
		for (llong j = 0; j < w; j++)
		{
			const auto m = mulMod(j, w, curve.p);
			ECPoint temp{};

			multiply(curve.a, m, curve.p, P, temp);

			if (temp.x != 0)
				B.emplace_back(temp.x);
		}

		// sort
		sort(A.begin(), A.end());
		sort(B.begin(), B.end());



		// intersect
		vector<llong> intersection;
		set_intersection(A.begin(), A.end(), B.begin(), B.end(),
back_inserter(intersection));

		if (intersection.size() != 1)
			return false;

		betta = find(A.begin(), A.end(), intersection[0]) -
A.begin();
		gamma = find(B.begin(), B.end(), intersection[0]) -
B.begin();
```

```
		return true;
	}

	llong algorithmException() const
	{
		llong cardinality = curve.p + 1;
		for (auto i = 0; i < curve.p; ++i)
		{
			cardinality += legandre(curve.a, curve.b, curve.p,
i);
		}
		return cardinality;
	}

	llong shanksMestre(llong &iterations)
	{
		if (curve.p <= 229)
			return algorithmException();

		// shanks initializer
		while (g < curve.p) {
			if (legandre(curve.a, curve.b, curve.p, g) == -1)
				break;
			g++;
		}

		if (g >= curve.p)
			return -1;

		const auto g2 = mulMod(g, g, curve.p);
		const auto g3 = mulMod(g2, g, curve.p);
		const auto c = mulMod(g2, curve.a, curve.p);
		const auto d = mulMod(g3, curve.b, curve.p);

		llong betta = 0, gamma = 0;

		ECPoint P;

		llong sigma, ea = 0, y = 0;
		auto m = -1;
		while (m == -1 && iterations < 600) {
			int i = 0;
			// mestre cycle
			while (i < 32)
			{
				auto x = randomNumberInRange(0, curve.p - 1);
				sigma = legandre(curve.a, curve.b, curve.p,
x);

				if (sigma == 1)
				{
					ea = curve.a;
					y = weierstrass(x, curve.a, curve.b,
curve.p);
				}
				else if (sigma == -1)
				{
					ea = c;
					x = mulMod(x, g, curve.p);
					y = weierstrass(x, c, d, curve.p);
				}
```

```
                        P.x = x;
                        const auto squareRoot = squareRootModPrime(y,
curve.p);

                        if (squareRoot != 0 && mulMod(squareRoot,
squareRoot, curve.p) == y)
                        {
                                P.y = squareRoot;

                                if (P.y != 0)
                                {
                                        if (shanks(betta, gamma, P))
                                                break;
                                }
                        }
                        i++;
                }

                auto t = betta + gamma * w;
                m = order(sigma, curve.a, ea, t, P);

                if (m != -1)
                        return m;

                t = betta - gamma * w;
                m = order(sigma, curve.a, ea, t, P);

                if (m != -1)
                        return m;

                t = -betta + gamma * w;
                m = order(sigma, curve.a, ea, t, P);

                if (m != -1)
                        return m;
                t = -betta - gamma * w;
                m = order(sigma, curve.a, ea, t, P);

                if (m != -1)
                        return m;

                iterations++;
        }
        // failed, try again
        return -1;
    }
public:
    llong cardinality(const llong a, const llong b, const llong p)
    {

        const llong curvea = a;
        const llong curveb = b;
        const llong curvep = p;

        if (delta(curvea, curveb, curvep) == 0)
                return -1;

        initialize(curvea, curveb, curvep);

        llong iterations = 1;
        const auto cardinality = shanksMestre(iterations);
```

```
                return cardinality;
        }
};
#endif // SHANKSMESTREALGORITHM_H
```

ShanksMestreModifiedAlgorithm.h

```
#ifndef SHANKSMESTREMODIFIEDALGORITHM_H
#define SHANKSMESTREMODIFIEDALGORITHM_H
#include "EllipticCurve.h"

using namespace std;

class ShankMestreModifiedAlgorithm
{
        llong q, w, w2, leftBound, rightBound, g = 1;
        EllipticCurve curve;
        void initialize(const llong a, const llong b, const llong p)
        {
                curve = EllipticCurve(a, b, p);

                q = pow(curve.p, 1.0 / 2);

                if (q * q < curve.p)
                        q++;

                leftBound = curve.p + 1 - 2 * q;
                rightBound = curve.p + 1 + 2 * q;

                w = pow(q, 1.0 / 3);

                const auto pForth = pow(w, 6);

                if (pForth < curve.p)
                        w++;
                else if (pForth > curve.p)
                        w--;
                w2 = pow(w, 2);
        }

        llong order(const llong sigma, llong a, const llong ea, llong t,
const ECPoint P) const
        {
                const llong p1 = curve.p + 1;
                ECPoint Q = ECPoint();

                t %= 2 * q;

                if (t < 0)
                        t += 2 * q;

                auto m = p1 + t;
                auto n = p1 + sigma * t;

                if (n >= leftBound || n <= rightBound)
                {
                        if (multiply(ea, m, curve.p, P, Q))
                                if (Q.isPointO())
                                        return n;
                }

                m = p1 - t;
```

```
                n = p1 - sigma * t;

                if (n >= leftBound || n <= rightBound)
                {
                        if (multiply(ea, m, curve.p, P, Q))
                                if (Q.isPointO())
                                        return n;
                }

                return -1;
        }

        bool shanks(llong &betta, llong &gamma, llong &alfa, const
ECPoint P) const
        {
                vector<llong> A;
                vector<llong> B;
                vector<llong> C;

                vector<pll> S;

                // baby steps

                betta = 0;
                gamma = 0;
                alfa = 0;

                for (llong i = 0; i < w; i++)
                {
                        const auto m = curve.p + 1 + i;
                        ECPoint temp{};

                        multiply(curve.a, m, curve.p, P, temp);

                        if (temp.x != 0)
                                A.emplace_back(temp.x);
                }

                // giant steps
                for (llong j = 0; j <= w; j++)
                {
                        const auto m = safeMul(j, w, curve.p);
                        ECPoint temp{};

                        multiply(curve.a, m, curve.p, P, temp);

                        if (temp.x != 0)
                                B.emplace_back(temp.x);
                }

                for (llong k = 0; k <= w; k++)
                {
                        const auto m = safeMul(k, w2, curve.p);
                        ECPoint temp{};

                        multiply(curve.a, m, curve.p, P, temp);

                        if (temp.x != 0)
                                C.emplace_back(temp.x);
                }

                // sort
```

```
		sort(A.begin(), A.end());
		sort(B.begin(), B.end());
		sort(C.begin(), C.end());



		// intersect
		vector<llong> intersection1;
		vector<llong> intersection2;

		set_intersection(A.begin(), A.end(), B.begin(), B.end(),
back_inserter(intersection1));
		if (intersection1.empty())
			return false;
		set_intersection(intersection1.begin(),
intersection1.end(), C.begin(), C.end(), back_inserter(intersection2));

		if (intersection2.size() != 1)
			return false;

		betta = find(A.begin(), A.end(), intersection2[0]) -
A.begin();
		gamma = find(B.begin(), B.end(), intersection2[0]) -
B.begin();
		alfa = find(C.begin(), C.end(), intersection2[0]) -
C.begin();
		return true;
	}

	llong algorithmException() const
	{
		llong cardinality = curve.p + 1;
		for (auto i = 0; i < curve.p; ++i)
		{
			cardinality += legandre(curve.a, curve.b, curve.p,
i);
		}
		return cardinality;
	}

	llong shanksMestre(llong &iterations)
	{
		if (curve.p <= 229)
			return algorithmException();

		// shanks initializer
		while (g < curve.p) {
			if (legandre(curve.a, curve.b, curve.p, g) == -1)
				break;
			g++;
		}

		if (g >= curve.p)
			return -1;

		const auto g2 = mulMod(g, g, curve.p);
		const auto g3 = mulMod(g2, g, curve.p);
		const auto c = mulMod(g2, curve.a, curve.p);
		const auto d = mulMod(g3, curve.b, curve.p);

		llong betta = 0, gamma = 0, alfa = 0;
```

```
		ECPoint P;

		llong sigma, ea = 0, y = 0;
		auto m = -1;
		while (m == -1 && iterations < 600) {
			int i = 0;
			// mestre cycle
			while (i < 32)
			{
				auto x = randomNumberInRange(0, curve.p - 1);
				sigma = legandre(curve.a, curve.b, curve.p,
x);

				if (sigma == 1)
				{
					ea = curve.a;
					y = weierstrass(x, curve.a, curve.b,
curve.p);
				}
				else if (sigma == -1)
				{
					ea = c;
					x = safeMul(x, g, curve.p);
					y = weierstrass(x, c, d, curve.p);
				}
				else continue;

				P.x = x;
				const auto squareRoot = squareRootModPrime(y,
curve.p);

				if (squareRoot != 0 && mulMod(squareRoot,
squareRoot, curve.p) == y)
				{
					P.y = squareRoot;

					if (P.y != 0)
					{
						if (shanks(betta, gamma, alfa,
P))
							break;
					}
				}
				i++;
			}

			auto t = safeAdd(safeAdd(safeMul(alfa, w2, 2 * q),
betta, 2 * q), safeMul(gamma, w, 2 * q), 2 * q);

			m = order(sigma, curve.a, ea, t, P);

			if (m != -1)
				return m;

			t = safeAdd(safeAdd(safeMul(alfa, w2, 2 * q),
betta, 2 * q), safeMul(-gamma, w, 2 * q), 2 * q);

			m = order(sigma, curve.a, ea, t, P);

			if (m != -1)
				return m;
```

```
                        t = safeAdd(safeAdd(safeMul(alfa, w2, 2 * q), -
betta, 2 * q), safeMul(gamma, w, 2 * q), 2 * q);

                        m = order(sigma, curve.a, ea, t, P);

                        if (m != -1)
                                return m;

                        t = safeAdd(safeAdd(safeMul(alfa, w2, 2 * q), -
betta, 2 * q), safeMul(-gamma, w, 2 * q), 2 * q);

                        m = order(sigma, curve.a, ea, t, P);

                        if (m != -1)
                                return m;

                        iterations++;
                }
                // failed, try again
                return -1;
        }
public:
        llong cardinality(const llong a, const llong b, const llong p)
        {

                const llong curvea = a;
                const llong curveb = b;
                const llong curvep = p;
                if (delta(curvea, curveb, curvep) == 0)
                        return -1;

                initialize(curvea, curveb, curvep);

                llong iterations = 1;
                const auto cardinality = shanksMestre(iterations);

                return cardinality;
        }
};
#endif // SHANKSMESTREMODIFIEDALGORITHM_H
```

GoldwasserKilianAlgorithm.h

```
#ifndef GOLDWASSERKILIANALGORITHM_H
#define GOLDWASSERKILIANALGORITHM_H
#include "ShanksMestreAlgorithm.h"
#include "ShanksMestreModifiedAlgorithm.h"
#include <iostream>
#include "Schoof.h"

inline EllipticCurve chooseCurve(const llong p)
{
        auto deltaCurve = 0;

        llong a = 0, b = 0;

        while (deltaCurve == 0) {
                a = randomNumberInRange(0, p - 1);
                b = randomNumberInRange(0, p - 1);

                deltaCurve = delta(a, b, p);
```

```
	}
	const EllipticCurve curve(a, b, p);
	return curve;
}

inline llong calculateLowerForFactor(llong p)
{
	llong factorBound = pow(p, 1.0 / 4);
	factorBound += 1;
	factorBound = pow(factorBound, 2);
	return factorBound;
}


inline llong goldwasserKilian(llong p, bool modified)
{
	auto composite = false;
	const auto upper = 1000;
	ShankMestreAlgorithm sm;
	ShankMestreModifiedAlgorithm smm;
	//SchoofsAlgorithm sa;
	ECPoint U, V, P;
	EllipticCurve curve;
	vector<llong> factors;
	llong maxFactor;
	llong factorBound = calculateLowerForFactor(p);
	llong m = -1;

	int i = 0, j = 0, k = 0;

preparationStep:
	composite = millerRabin(p);
	if (composite)
		goto finalStep;
	if (p <= upper)
	{
		for (int f = 2; f < sqrt(p); ++f)
		{
			if (p % f == 0)
			{
				cout << "factor = " << f << endl;
				return 0;
			}
		}
		cout << p << " is prime" << endl;
		return 1;
	}


chooseCurve:
	k++;
	curve = chooseCurve(p);
	//sa.setCurve(curve.a, curve.b, curve.p);
	m = !modified ? sm.cardinality(curve.a, curve.b, curve.p) :
smm.cardinality(curve.a, curve.b, curve.p);
	if (m < 0 && k == 32)
		goto shanksMestreAlgoFailed;
	if (!millerRabin(m))
		goto chooseCurve;
	factors.clear();
	factorize2(m, factors);
	sort(factors.begin(), factors.end());
```

```
	maxFactor = factors[factors.size() - 1];
	if (maxFactor < factorBound || millerRabin(maxFactor))
		goto chooseCurve;

pointOperations:
	P = choosePointOnCurve(curve);

	if (!multiply(curve.a, m / maxFactor, p, P, U))
		goto finalStep;

	if (U.isPointO())
		goto pointOperations;

	if (!multiply(curve.a, maxFactor, p, P, V))
		goto finalStep;

	if (V.isPointO())
		goto pointOperations;
	cout << "Iteration #" << i + 1 << endl;
	cout << "p[" << i << "] = " << p << endl;
	cout << "E: " << "x3 + " << curve.a << "x + " << curve.b << endl;
	cout << "#E = " << m << endl;
	cout << "Bound for factor: " << factorBound << endl;
	cout << "Factor = " << maxFactor << endl;
	cout << "P = (" << P.x << ", " << P.y << ")\n";
	cout << "U = (" << U.x << ", " << U.y << ")\n";
	cout << "V = (" << V.x << ", " << V.y << ")\n\n";
	cout << "test next number " << maxFactor <<"\n\n";
	p = maxFactor;
	factorBound = calculateLowerForFactor(maxFactor);
	++i;
	++j;
	goto preparationStep;
shanksMestreAlgoFailed:
	if (j == 0)
		return -2;
finalStep:
	if (j == 0)
	{
		if (millerRabin(p))
			return 0;
		return -1;
	}
	--j;
	goto preparationStep;
}
#endif // GOLDWASSERKILIANALGORITHM_H
```

Source.cpp

```
#include "GoldwasserKilianAlgorithm.h"
#include "Schoof.h"
#include <cstdlib>
#include <chrono>
using namespace std::chrono;

int main()
{
	const llong p = 811411;
```

```
	const high_resolution_clock::time_point start =
high_resolution_clock::now();
	const llong result = goldwasserKilian(p, false);
	const double totalDuration =
duration_cast<duration<double>>(high_resolution_clock::now() -
start).count();
	cout << "\ncomputational time: " << totalDuration << " s" <<
endl;
	cout << "\nDesicion:\n";
	if (result == 1)
		cout << "\np = " << p << " is prime" << endl;
	if (result == 0)
		cout << "\np = " << p << " is composite" << endl;
	if (result == -1)
		cout << "Goldwasser Kilian algorithm failed\n";
	if (result == -2)
		cout << "Shanks-Mestre algorithm failed\n";
	cout << "----------------------------" << endl;

	return 0;
}
```