Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science and Engineering

Master's Thesis

Analysis and investigation of the convergence of the Bezout coefficients search algorithm.

Alisher Zhumaniezov

Supervisor: Ing. Karel Frajtak, PhD

Study Program: Open Informatics

Field of Study: Software Engineering

May 24, 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Zhumaniezov**	Jméno: **Alisher**	Osobní číslo: **474706**
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:

Analysis and investigation of the convergence of the Bezout coefficients search algorithm

Název diplomové práce anglicky:

Analysis and investigation of the convergence of the Bezout coefficients search algorithm

Pokyny pro vypracování:

Bezout´s equation is a representation of the greatest common divisor d of two integers A and B as a linear combination Ax+By = d, where x,y are integers called Bezout´s coeficients. Usually Bezout´s coeffcients are counted using the extended version of the classical Euclidian Algorithm.
We elaborate a new algorithm for counting Bezout´s coefficients on the base of the k-ary gcd Algorithm. This can have numerous applications in the Number Theory and Cryptography, for example, for counting multiplicative inverse elements in modular arithmetic.

Seznam doporučené literatury:

1) Sorenson J. ""Two fast GCD Algorithms"", Journal of Algorithms, 1994.
2) Ishmuhametov Sh., Rubtsova R. ""About approximate k-ary algorithm computing GCD, Lobachevskii Journal of Mathematics, Kazan, Russia, 2016
3) Ishmuhametov Sh. ""Finding Bezout´s coeficients for k-ary algorithm of computing GCD"", Lobachevskii Journal of Mathematics, Kazan, Russia, 2016
4) Chor B., Goldreich O. An improved Parallel Algorithm for integer GCD, Algorithmica, 5, 1?10 (1990)
5) Ishmuhametov Sh., Rubtsova R. ""A parallel computation of the GCD of natural numbers"", Lobachevskii Journal of Mathematics, Kazan, Russia, 2017

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Frajták, 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**

Ing. Karel Frajták, 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

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor from KFU side, Prof. Shamil Ishmukhametov for his invaluable help in the work, for useful ideas and precise approach during the whole process. Thanks a lot to Ekaterina Turilova for the great support during all work. Finally, thanks a lot to my family for their unconditioned support and faith in me.

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.

Kazan, May, 2018

ABSTRACT

In the age of modern technology, time is a very valuable resource. Therefore, an important indicator of the program's work is its computational speed.

This essay will describe the optimization of Bezout coefficients search algorithm by introduction different optimization schemes.

Bezout’s equation is a representation of the greatest common divisor of two integers and as a linear combination , where are integers called Bezout’s coefficients. Usually Bezout’s coefficients are counted using the extended version of the classical Euclidian Algorithm.

Keywords: Euclidean algorithm, extended Euclidean algorithm, k–ary algorithm for computing GCD, calculation of inverse elements modulo.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Actuality of the problem: Cryptography is a science engaged in the development of cryptosystems, that is, by researching mathematical methods for transformation information in order to protect systems from unauthorized access.

In the age of modern technology, time is a very valuable resource. Therefore, an important indicator of the program's work is its computational speed. The process of increasing the speed goes in the following directions:

1. Increase the productivity of equipment. Achieved by using more powerful processors. The main difficulty in this approach is the great difficulty in manufacturing such processors. Another difficulty is the renewal of equipment on all machines, which entails additional costs.
2. Parallelizing the program. The distribution of common work between different threads allows you to reduce the total running time of the program. However, the Bezout coefficients search algorithm is iterative, and therefore not subject to parallelization.
3. Reduction of the asymptotic complexity of the algorithm used. This approach will be used in this work. It consists in reducing the number of elementary operations used by the algorithm.

The Bezout relation is the representation of the greatest common divisor of integers in the form of their linear combination with integer coefficients [14]. For the case of two integers and and their — is the greatest common divisor, this relation is written in the following form [22]:

(1)

Moreover, the coefficients and are called the Bezout coefficients.

Finding the Bezout coefficients can be used to solve the following problems:

1. The solution of linear Diophantine equations in the form [33]:

   (2)

2. The solution of first-degree comparisons in the form [36]:

   (3)

3. The search for an inverse element in a field — is a particular case of equation (3):

   (4)

4. It is the basis for some cryptographic algorithms with a public key, such as RSA. [24]

Euclid's algorithm is an effective algorithm for finding the greatest common divisor of two integers [25]. This algorithm is iterative and is defined by:

(5)

Extended Euclidean algorithm is a modification of the Euclidean algorithm, which allows to find Bézout coefficients [10]. The main idea is to represent the resulting remainder at each iteration in the form of a linear combination of the original numbers and .

The main idea of optimizing the Bezout coefficients search algorithm will be to change the choice of a new number, which should lead to a faster convergence of the algorithm and, accordingly, to a decrease in the number of iterations.

The main disadvantage of the approach described above is an increasing of the load per iteration, which can lead to a decrease in the speed of the complete algorithm. Therefore, we must also optimally find new values for the next iteration.

Object of investigation: cryptography and algorithms of number theory.

Subject of investigation: an algorithm for finding the Bezout coefficients, its algorithmic complexity and schemes for its optimization.

The goal of the master's thesis: the investigation of optimization schemes for the Bezout coefficients search algorithm, their algorithmic complexity and their implementation.

To achieve this goal, the following problems are formulated and solved in the work:

1. An analysis of the Bezout coefficients search algorithm.
2. Analysis of existing optimization schemes for the basic Euclidean algorithm.
3. Construction of a new algorithm for finding Bezout coefficients based on the studied schemes.

Scope and structure of work: The master's thesis consists of an introduction, three chapters, a conclusion, a list of used literature and an annex in the form of a listing. The volume of the main text is 44 pages of typewritten text.

In the first chapter, a brief analysis of the subject area — the basic and extended Euclidean algorithms — was carried out. The following optimization schemes for the basic Euclidean algorithm are considered: binary, k–ary, and approximating. An approximate number of iterations is also calculated for all algorithms.

In the second chapter, modified extended Euclidean algorithms are considered. For the modification, the following schemes were used: k–ary, approximating and approximating with parallelization. All these schemes were used for analysis.

The third chapter describes the program, the libraries used, and the results of the experiments.

Each chapter ends with brief conclusions, and all work with conclusion.

CHAPTER 1. THEORETICAL ASPECTS OF THE OPTIMIZATION SCHEMES OF THE BASIC EUCLIDEAN ALGORITHM

1.1. Euclidean algorithms

1.1.1. Description of the algorithm

Euclid's algorithm is an effective algorithm for finding the greatest common divisor of two integers [25]. The algorithm is iterative. The next iteration is determined by the relation (5). Thus, the computation for the pair is replaced in the next step by the calculation for . Iterations are performed until . The result of the calculations will be equal to the value at the last iteration.

Thus we obtain the following steps of the algorithm:

(6)

1.1.2. Computational complexity of the algorithm

Denote by — number of iterations for the algorithm on the numbers and . According to [23] and the relation (6) the following recurrence formula holds:

(7)

The worst case for convergence is a pair of neighboring Fibonacci numbers and . In this case, the number of iterations is equal to [23]. Since there is a Bin formula for the Fibonacci number [38]:

(8)

Where the is a golden ratio. [27]

In this case, the maximum number of iterations can be reduced to the form:

(9)

To calculate the average number of iterations, we use the following formula:

(10)

Where the is a Euler's totient function.

Since the [26], it can be seen that a simple sum over all will be greatly noisy and the smoothness is broken, therefore the sum is considered only for coprime pairs. According to [34] this function is represented in the form:

$$— \qquad (11)$$

Where the is an infinitesimal, and is a **Porter's constant**. [41]

1.1.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 252 and 195. The result is shown in the table 1.1:

252	195	57
195	57	24
57	24	9
24	9	6
9	6	3
6	3	0
3	0	—

Table 1.1. The result of the work for 252 and 195(Euclidean).

Table 1.1 shows that the result is .

- As an additional example of the operation of the algorithm, consider the pair 242 and 146. The result is shown in the table 1.2:

242	146	96
146	96	50
96	50	46
50	46	4
46	4	2
4	2	0
2	0	—

Table 1.2. The result of the work for 242 and 146(Euclidean).

Table 1.2 shows that the result is .

1.2. Extended Euclidean algorithm

1.2.1. Description of the algorithm

Extended Euclidean algorithm is a modification of the Euclidean algorithm, which allows to find Bézout coefficients [10]. The main idea is to represent the resulting remainder at each iteration in the form of a linear combination of the original numbers and [10]:

(12)

However, in practice, a «direct» approach is rarely used. Therefore, the «reverse» approach to the algorithm is more common. [2]

At the first stage, the basic Euclidean algorithm is run, but in addition to the remainder an incomplete quotient is also calculated .

In the second step, for the last iteration, the values of the Bezout coefficients are assumed to be equal to and .

In the third stage, all steps of the Euclidean algorithm are gone back and for each iteration the Bezout coefficients are recalculated using the following formula [2]:

(13)

At the end of the algorithm, the values and will be a pair of Bezout coefficients for and .

1.2.2. Computational complexity of the algorithm

Since in the "direct" approach, we only go through Euclid's algorithm once, then the number of iterations remains the same: function (9) for the worst case at neighboring Fibonacci numbers and ; function defined in (11) for the average number of iterations.

In the "reverse" approach, there are two runs according to the Euclidean algorithm: the direct one for computing reminders and incomplete quotients; and the inverse for the calculation of the Bezout coefficients directly. Therefore, the number of iterations is doubled: the maximum number of iterations at neighboring Fibonacci numbers and :

(14)

Where the is a golden ratio. [27]

The average number of iterations is defined by formula:

(15)

Where the is an infinitesimal, and **is a Porter's constant.** [41]

1.2.3. Example of the work of the algorithm

- As an example of the operation of the algorithm under the "direct" approach, we consider the pair 245 and 227. The result is shown in the table 1.3:

245	227	18	1	1	–1
227	18	11	12	–12	13
18	11	7	1	13	–14
11	7	4	1	–25	27
7	4	3	1	38	–41
4	3	1	1	–63	68
3	1	0	3	—	—

Table 1.3. The result of the work for 245 and 227(Extended Euclidean).

Table 1.3 shows that Bezout coefficients are equal –63 and 68.

- As an example of the operation of the algorithm under the "reverse" approach, we consider the same pair 245 and 227. The result is shown in the table 1.4:

245	227	18	1	–63	68
227	18	11	12	5	–63
18	11	7	1	–3	5
11	7	4	1	2	–3
7	4	3	1	–1	2
4	3	1	1	1	–1
3	1	0	3	0	1

1	0	—	—	1	0

Table 1.4. The result of the work for 245 and 227(Extended Euclidean).

Table 1.4 shows that Bezout coefficients are equal –63 and 68.

1.3. Binary Euclidean algorithm

1.3.1. Description of the algorithm

The binary Euclidean algorithm was first published in [32]. The main idea of optimization was getting rid of the heavy operation of finding the remainder and replacing it with easier subtraction and division by 2. At each iteration, the next step for is determined by the following rule [32]:

1.
2. If and are both even, then – –
3. if is even and is odd, then –
4. If is odd and is even, then – (16)
5. If and are both odd and , then ——
6. If and are both odd and , then ——

1.3.2. Computational complexity of the algorithm

Note that at each non–finite step, at least one number decreases by a factor of 2. Thus we obtain a limitation on the number of iterations for any pair and from above [40]:

(17)

Then the average number of iterations for any will be limited from above:

—— (18)

The worst case is a pair of the form and . Then the number of iterations . Thus, we obtain an estimation for the maximum number of iterations :

—— (19)

Note that the number of iterations of the binary algorithm is more than original, but they are easier, so acceleration is achieved.

1.3.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 193 and 215. The result is shown in the table 1.5:

		Additional factor
193	215	–
193	11	–
91	11	–
40	11	–
20	11	–
10	11	–
5	11	–
5	3	–
1	3	–
1	1	

Table 1.5. The result of the work for 193 and 215(Binary).

Table 1.5 shows that the result is .

- As an additional example of the operation of the algorithm, consider the pair 228 and 224. The result is shown in the table 1.6:

		Additional factor
228	224	+
114	112	+
57	56	–
57	28	–
57	14	–
57	7	–
25	7	–
9	7	–
1	7	–
1	3	–
1	1	

Table 1.6. The result of the work for 228 and 224(Binary).

Table 1.6 shows that the result is .

1.4. K-ary Euclidean algorithm

1.4.1. Description of the algorithm

The k–ary Euclidean algorithm was first published in [29] and [30]. Then in [35] and [21] improvements were proposed. This algorithm is iterative, like the original algorithm. The main difference is to find a new pair for the next step. For this, we use the theorem proved in [35] and [21]:

Theorem 1. Let are natural numbers and is a small positive number that is coprime to and . Then there are integers and , satisfying the relation – such that:

(20)

Thus, it follows directly from the theorem that the number:

(21)

Is natural. In case that it is necessary to divide the number by . Then the transition to the next iteration has the form:

(22)

Note that when the algorithm works, additional multipliers may appear [3]. To eliminate them, you must start recursively , where is the result obtained at the output of the k–ary algorithm.

1.4.2. Computational complexity of the algorithm

We construct an upper bound for :

– (23)

From this estimate it follows that at each step the greater of the numbers decreases at least – times [19]. Thus, we obtain an estimation for the number of iterations for any pair and :

– – (24)

Thus, we obtain a upper bound for the average number of iterations for any [19]:

(25)

1.4.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 203 and 157. The result is shown in the table 1.7:

203	157	2	2	45
157	45	1	−1	7
45	7	3	−1	1
7	1	2	2	1
1	1	1	−1	0
1	0	—	—	—

Table 1.7. The result of the work for 203 and 157(K–ary).

Table 1.7 shows that the result is .

- As an additional example of the operation of the algorithm, consider the pair 207 and 171. The result is shown in the table 1.8:

207	171	1	3	45
171	45	2	2	27
45	27	2	2	9
27	9	1	−3	0
9	0	—	—	—

Table 1.8. The result of the work for 207 and 171(K–ary).

Table 1.8 shows that the result is .

1.5. Approximating Euclidean algorithm

1.5.1. Description of the algorithm

In article [19] optimization of the k–ary algorithm was presented. The main idea is the replacement of the algorithm for finding the coefficients and for the next iteration. For this is taken from the interval and is taken close to .

At first we introduce the following notation:

(26)

(27)

(28)

(29)

Then the formula for is [19]:

(30)

Now we find the formula for [19]:

$$— \qquad — \tag{31}$$

Note that is arbitrary, so for a minimum we take [19]:

(32)

Then the final formula for [19]:

(33)

Thus it is necessary to find such a , to minimize . To find them, consider the formula (31):

$$– \tag{34}$$

We note that the original fraction – has an arbitrary form, so we must find an approximation — – on condition . Then formula (34) takes the form:

$$– \qquad – \tag{35}$$

Then the minimum, which is equal to 0, will be achieved with and . Now we obtain the final formulas for the transition:

(36)

1.5.2. Computational complexity of the algorithm

In article [20] the proof of the following theorem was presented:

Theorem 2. There is an integer , such that:

$$— \tag{37}$$

And the search for such can be produced by an algorithm with complexity .

From the theorem it follows directly that it is possible in an acceptable time to find such a pair of coefficients and , that:

(38)

That is, at each step there is a decrease in one number, at least in times. Thus we obtain a estimation on the number of iterations for any and from above [19]:

(39)

Then the average number of iterations for any will be limited from above:

(40)

1.5.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 485 and 321. The result is shown in the table 1.9:

485	321	–14	22	17
321	17	9	–169	1
17	1	15	–255	0
1	0	—	—	—

Table 1.9. The result of the work for 485 and 321(Approximating).

Table 1.9. The result of the work for and .

Table 1.9 shows that the result is .

- As an additional example of the operation of the algorithm, consider the pair 339 and 315. The result is shown in the table 1.10:

339	315	2	–2	3
315	3	15	–1575	0

3	0	—	—	—

Table 1.10. The result of the work for 339 and 315(Approximating).

Table 10 shows that the result is

1.6. Farey sequence

1.6.1. Description of the algorithm

To search for the approximation of an arbitrary fraction – we use the Farey sequence. At the first step we find the segment on which our fraction lies:

— — - - - — — (41)

We denote this segment by — — . Now try to improve the approximation. Consider the median of our segment:

— —— (42)

Then from the subsegments — — and — — we choose one in which our fraction lies. Denote it by — — . We perform this operation on a new segment until the new denominator of the median exceeds . We obtain a segment — — . As an approximation, we return the near to end point of segment.

1.6.2. Computational complexity of the algorithm

Note that the resulting algorithms will form a chain in the Stern–Brocot tree [39]. Search for fractions – in this tree will be produced no more than steps [39]. From this we obtain the time complexity of the maximum number of steps :

(43)

To estimate the average number of steps, note that the selection of two subsegment options correspond to two disjoint subtrees. That is, we obtain a binary search for the Stern–Brocot tree. Since the total number of Farey fractions with a denominator not exceeding is equal , then the average number of steps [12]:

(44)

1.6.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the number and module . The result is shown in the table 1.11:

1	2	2	3	3	5
1	2	3	5	4	7
1	2	4	7	5	9
1	2	5	9	6	11
1	2	6	11	7	13
1	2	7	13	8	15
1	2	8	15	9	17

Table 1.11. The result of the work for 0.5218487889069591(Farey).

Table 1.11 shows that the approximation of —

- As an additional example of the operation of the algorithm, consider the number and module . The result is shown in the table 1.12:

1	3	1	2	2	5
2	5	1	2	3	7
3	7	1	2	4	9
4	9	1	2	5	11
5	11	1	2	6	13
5	11	6	13	11	24

Table 1.12. The result of the work for 0.4581481154474917(Farey).

Table 1.12 shows that the approximation of —

1.7. Conclusions on the chapter 1

All schemes of optimization of the Euclidean algorithm were analyzed in detail. In the second chapter, each considered scheme will be modified for use in the extended Euclidean algorithm.

Also, the original and advanced Euclidean algorithms were considered. They will act as a reference for comparing the remaining schemes in the course of our investigation.

For each algorithm considered, examples of work are given and an estimate of the number of iterations is given.

CHAPTER 2. METHODS OF IMPLEMENTATION OF OPTIMIZATION SCHEMES OF THE ALGORITHM OF SEARCH OF THE BEZOUT COEFFICIENTS

2.1. Binary extended Euclidean algorithm

2.1.1. Description of the algorithm

As the first optimization, consider the binary Euclidean algorithm. Its extended version was presented in [23]. In the algorithm we will use the "reverse" approach. For each variant of the step, we give our version of the calculation of the Bezout coefficients:

1.1. If , then

1.2. If , then

2.

3.1. If is even, then

3.2. If is odd, then

4.1. If is even, then (45)

4.2. If is odd, then

5.1. If is even, then

5.2. If is odd, then

6.1. If is even, then

6.2. If is odd, then

2.1.2. Computational complexity of the algorithm

Since we use the "reverse" approach, the number of steps will be twice as large as in the usual version. Then the average number of iterations for any will be limited from above:

$$\text{—} \quad (46)$$

The worst case is a pair of the form and . Thus, we obtain an estimation for the maximum number of iterations :

$$\text{—} \quad (47)$$

Note that the acceleration of the program is achieved due to simpler operations for the machine, but the number of iterations increases. This is the first reason why this method does not suit us.

The second reason why this method does not suit us is the lack of internal parameters. This leads to a rigid algorithm, and we can't influence its course of work.

2.1.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 238 and 220. The result is shown in the table 2.1:

238	220	49	–53
119	110	49	–53
119	55	49	–106
32	55	43	–25
16	55	31	–9
8	55	7	–1
4	55	14	–1
2	55	28	–1
1	55	1	0
1	27	1	0
1	13	1	0
1	6	1	0
1	3	1	0
1	1	1	0

Table 2.1. The result of the work for 238 and 220(Extended binary).

Table 2.1 shows that Bezout coefficients are equal 49 and –53.

- As an additional example of the operation of the algorithm, consider the pair 160 and 135. The result is shown in the table 2.2:

160	135	38	–45
80	135	76	–45
40	135	17	–5

20	135	34	–5
10	135	68	–5
5	135	1	0
5	65	1	0
5	30	1	0
5	15	1	0
5	5	1	0

Table 2.2. The result of the work for 160 and 135(Extended binary).

Table 2.2 shows that Bezout coefficients are equal 38 and –45.

2.2. K-ary extended Euclidean algorithm

2.2.1. Description of the algorithm

The following optimization of the extended Euclidean algorithm. Was proposed in the [18]. In this algorithm, too, we will use the "reverse" approach. To construct the transition formula, we present the Bezout equation:

(48)

Now we substitute in this equation the transition formulas (21) and (22):

(49)

Now we group the terms:

(50)

Thus, we have explicitly obtained the transition formula for the Bezout coefficients [18]:

(51)

However, in this formula there is one drawback – the non–integer division into . Since it is not guaranteed that the result of dividing will always be integer, this move our calculations into the field of real numbers, which is undesirable for us. Therefore, to stay in the field of integers, we introduce auxiliary variables and [18]. They are defined as follows:

(52)

Then the recurrence relation for them will be written in the form [18]:

(53)

We note that the Bezout equality itself takes the form:

(54)

Since we can't divide by , because we leave the field of integers, we can replace it by an integer division modulo . We obtain the intermediate coefficients and by formula:

(55)

Then the Bezout equation takes the form:

(56)

Hence we obtain the final formula for the Bezout coefficients:

(57)

2.2.2. Exceptions of the algorithm

Despite the resulting formula, it will be true only under very strong assumptions. Therefore, for a complete algorithm, it is necessary to analyze how to operate in exceptional situations:

- For good convergence, it is necessary that the new number replace the larger of the old numbers. This means maintaining the condition for all the resulting pairs. If , then our condition is preserved. However, when The condition will be broken therefore, instead of calculating for the pair we shall count for the pair . Then the recurrence relation (53) takes the form:

(58)

- To be right (21), it is necessary that . If this is not the case, then formula (21) is transformed into the form:

(59)

Where the is the number by which we divide to satisfy the condition. Then the recurrence relation (53) takes the form:

(60)

In this case, equality (54) takes the form:

(61)

Where the is the product of all the additional numbers. In general, the form of this number is complicated for further work. However, in two cases it has a convenient form:

1. — is the prime number. Then .
2. — is the power of two. Then .

2.2.3. Computational complexity of the algorithm

Since we use the "reverse" approach, the number of steps will be twice as large as in the usual version. Then the average number of iterations for any will be limited from above:

$$—\qquad(62)$$

2.2.4. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 209 and 183. The result is shown in the table 2.3:

0	209	183	—	—	—	—	—	—	102	8	–7	8
1	209	183	2	2	49	493568	903168	1	—	—	—	—
2	183	49	2	2	29	25600	246784	1	—	—	—	—
3	49	29	3	1	11	13824	12800	1	—	—	—	—
4	29	11	2	2	5	512	4608	1	—	—	—	—
5	11	5	1	1	1	256	256	1	—	—	—	—
6	5	1	3	1	1	0	256	1	—	—	—	—
7	1	1	1	–1	0	0	16	1	—	—	—	—
8	1	0	—	—	—	1	0	1	—	—	—	—

Table 2.3. The result of the work for 209 and 183(Extended k–ary).

Table 2.3 shows that Bezout coefficients are equal –7 and 8.

- As an additional example of the operation of the algorithm, consider the pair 255 and 147. The result is shown in the table 2.4:

0	255	147	—	—	—	—	—	—	159	229	–132	229
1	255	147	3	1	57	–768	12032	1	—	—	—	—
2	147	57	3	–1	3	768	–256	8	—	—	—	—
3	57	3	1	–3	3	0	256	1	—	—	—	—
4	3	3	1	–1	0	0	16	1	—	—	—	—
5	3	0	—	—	—	1	0	1	—	—	—	—

Table 2.4. The result of the work for 255 and 147(Extended k–ary).

Table 2.4 shows that Bezout coefficients are equal –132 and 229.

2.3. The inverse element of a given number

2.3.1. Description of the algorithm

As can be seen in formula (55), the inverse element of a fixed number is used for an arbitrary modulus [18]. The search for the reverse element is a long operation so we **need to optimize for our case. Since the module is arbitrary, we can't use simple** pre–calculations.

Since the number is defined in advance, we can calculate in advance the inverse elements modulo . Then we can only express the desired inverse element through the module .

At first we introduce the notation:

(63)

Now, denoting by , we obtain the equality:

(64)

Adding a comparison modulo on both sides, we get:

(65)

Whence we find the formula for :

(66)

Substituting (66) into expression (64), we obtain:

(67)

Now we find the expression for :

(68)

Now adding the modulo on both sides, we get:

(69)

2.3.2. Computational complexity of the algorithm

Since the algorithm uses only simple arithmetic operations: 1 multiplication, 1 subtraction, 1 division and 1 module taking. Then the total complexity will be [37]:

(70)

This time is acceptable for us.

2.3.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the module and number :

- As an additional example of the operation of the algorithm, consider the module and number :

2.4. Approximating extended Euclidean algorithm

2.4.1. Description of the algorithm

The basic steps of the algorithm are the same as for the k–ary extended algorithm. The only difference is the use of a different approach for choosing the coefficients and . For this, an algorithm using Farey's sequence is used, as in the usual approximating algorithm.

2.4.2. Computational complexity of the algorithm

Since we use the "reverse" approach, the number of steps will be twice as large as in the usual version. Then the average number of iterations for any will be limited from above:

(71)

2.4.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 435 and 331. The result is shown in the table 2.5:

0	435	331	—	—	—	—	—	—	285	46	–35	46
1	435	331	–15	23	17	–70320	92464	4	—	—	—	—
2	331	17	–15	293	1	–240	4688	1	—	—	—	—
3	17	1	1	–17	0	0	16	1	—	—	—	—
4	1	0	—	—	—	1	0	1	—	—	—	—

Table 2.5. The result of the work for 435 and 331(Extended approximating).

Table 2.5 shows that Bezout coefficients are equal –35 and 46.

- As an additional example of the operation of the algorithm, consider the pair 391 and 357. The result is shown in the table 2.6:

0	391	357	—	—	—	—	—	—	268	172	–157	172
1	391	357	–13	15	17	–208	240	1	—	—	—	—
2	357	17	1	–21	0	0	16	1	—	—	—	—
3	17	0	—	—	—	1	0	1	—	—	—	—

Table 2.6. The result of the work for 391 and 357(Extended approximating).

Table 2.6 shows that Bezout coefficients are equal –157 and 172.

2.5. Parallel approximating extended Euclidean algorithm

2.5.1. Description of the algorithm

When using Farey sequence, we first obtain an approximation segment, from which we then select one of the end points. Hence the second approach follows, when we take both end points [15]. Then the transition to (22) is transformed into:

(72)

Then the transition to (53) is transformed into:

(73)

And if there are additional factors and introduce and additions to :

(74)

Then instead of (60) we obtain the following transition:

(75)

All other formulas keep unchanged.

For parallelization, the portion of the algorithm that computes and . Each of them can be calculated independently.

2.5.2. Computational complexity of the algorithm

Since the replacement occurs for both numbers and , one of which satisfies (38) and the other is less than the original ones, then the number of iterations for any pair and is limited from above:

(76)

Then the average number of iterations for any under the basic algorithm will be limited from above:

— (77)

And for the extended algorithm:

— (78)

2.5.3. Example of the work of the algorithm

- As an example of the work of the algorithm, consider the pair 431 and 287. The result is shown in the table 2.7:

0	431	287	—	—	—	—	—	—	—	—	—	296	428	–285	428
1	431	287	–15	31	19	1	–1	9	–209	321	8	—	—	—	—
2	19	9	7	–13	1	9	–19	0	7	–13	1	—	—	—	—
3	1	0	—	—	—	—	—	—	1	0	1	—	—	—	—

Table 2.7. The result of the work for 431 and 287(Extended parallel).

Table 2.7 shows that Bezout coefficients are equal –285 and 428.

- As an additional example of the operation of the algorithm, consider the pair 511 and 385. The result is shown in the table 20:

0	511	385	—	—	—	—	—	—	—	—	—	27	150	–113	150
1	511	385	–13	19	21	7	–9	7	1049	–1383	2	—	—	—	—
2	21	7	–15	61	7	1	–3	0	–15	61	1	—	—	—	—
3	1	0	—	—	—	—	—	—	1	0	1	—	—	—	—

Table 2.8. The result of the work for 511 and 385(Extended parallel).

Table 20 shows that Bezout coefficients are equal –113 and 150.

2.6. Conclusions on the chapter 2

All optimization schemes of the extended Euclidean algorithm, which is used to find the Bezout coefficients, were analyzed in detail. To implement and conduct the experiments, the following were selected:

- K–ary extended Euclidean algorithm.
- Approximating extended Euclidean algorithm.
- Parallel extended Euclidean algorithm.

For each algorithm considered, examples of work are given and an estimation of the number of iterations is given.

CHAPTER 3. SOFTWARE IMPLEMENTATION OF OPTIMIZATION SCHEMES OF THE ALGORITHM OF SEARCH OF THE BEZOUT COEFFICIENTS

3.1. How the program works

Python 3.6.4 was used as the programming language. Of all the development environments, IDLE Python was selected.

The experiments were performed on a 4–bit module, 8–bit module and a 16–bit module for a prime module and power of 2. As values, pairs of 512–bit numbers were taken. For the experiments, all the realized schemes were launched in three stages.

Stage one — run 100 repetitions of the implementation without counting the steps. Then followed the calculation of the time of work.

Stage two — the launch of the implementation with the calculation of steps. Getting the number of steps as a result of launching.

Stage three — the calculation of the average time of work for one iteration. Save all received values in a file.

The experiments were carried out on the computer Acer TravelMate P2 P259–MG–39WS.

Characteristics:

- CPU Intel Core i3 6006U.
- Number of Cores: 2.
- CPU frequency: 2 GHz.
- RAM size: 6 Gb.

3.2. Prime number

3.2.1. 4-bit module

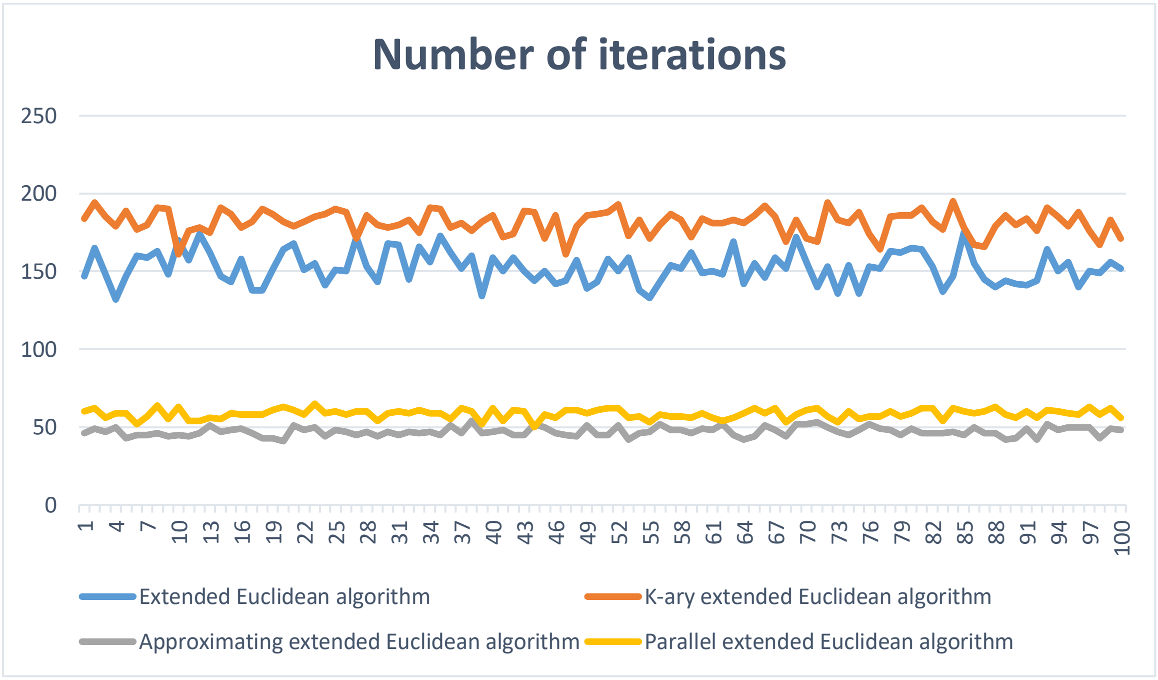

Figure 3.1. Comparison of the number of steps with the original algorithm (prime, 4-bit).

k-ary: as seen from the graph above, the value of the module is too small, so the convergence is low and the number of steps becomes larger than with the original algorithm.

Approximating: As can be seen from the graph above, the number of iterations has decreased approximately 3 times compared to the original algorithm. Among all the schemes considered, this showed the best result.

Parallel: As can be seen from the graph above, the number of iterations has decreased approximately 3 times compared to the original algorithm. However, the result was somewhat worse than the approximating one.

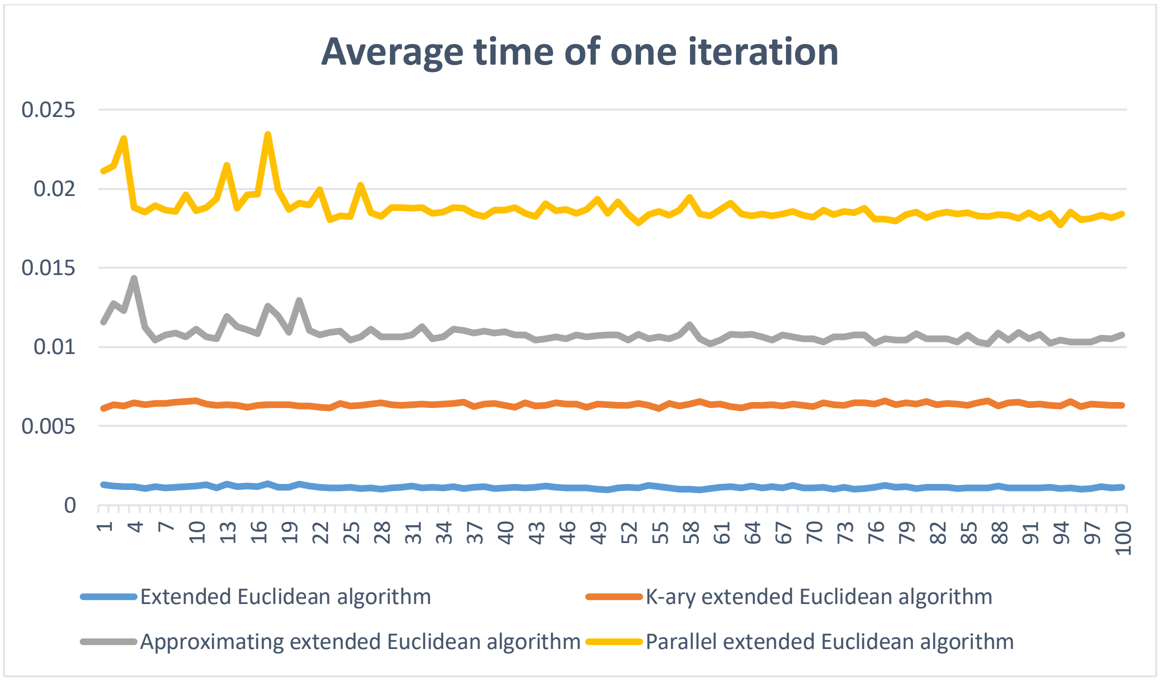

Figure 3.2. Comparison of the running time of the step with the original algorithm (prime, 4-bit).

k-ary: As can be seen from the graph above, the amount of work on one iteration increased significantly in comparison with the original algorithm. This reinforces the lag from the original algorithm, so this module size does not fit.

Approximating: as can be seen from the graph above, the amount of work on one iteration increased approximately 10 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm.

Parallel: As can be seen from the graph above, the amount of work on one iteration increased approximately 20 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm. This also leads to a lag in comparison with the approximating algorithm.

3.2.2. 8-bit module

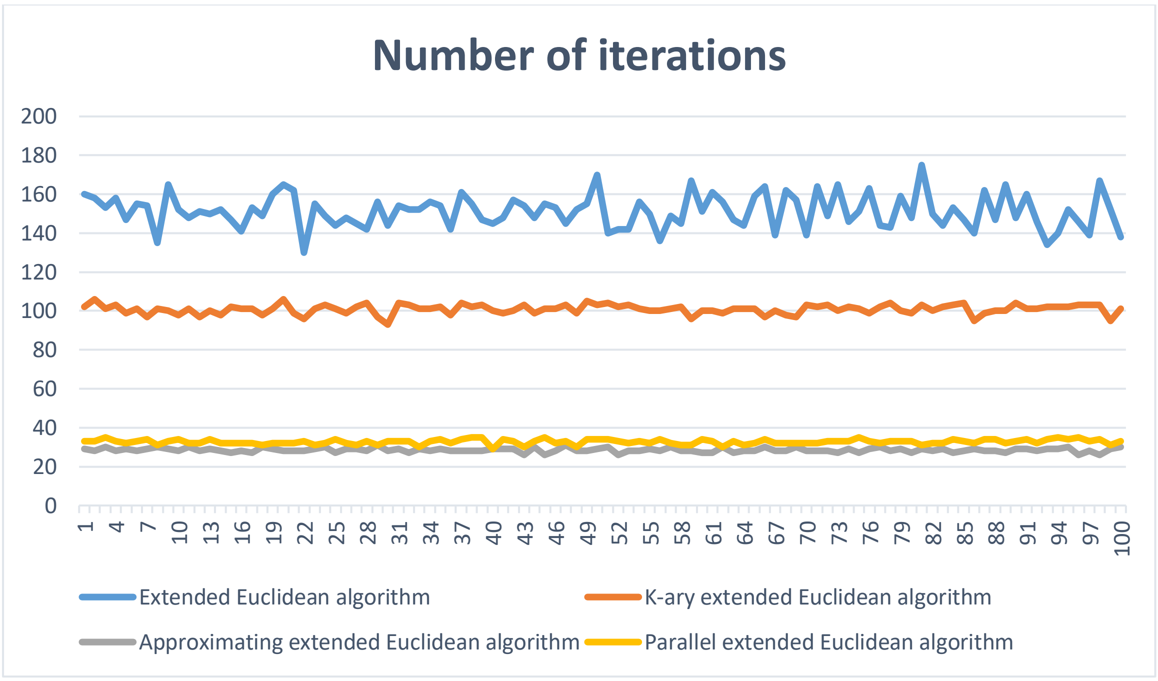

Figure 3.3. Comparison of the number of steps with the original algorithm (prime, 8-bit).

k-ary: As can be seen from the graph above, the number of steps became approximately 1.5 times less compared to the original algorithm. However, of all the algorithms considered, this is the worst result.

Approximating: As can be seen from the graph above, the number of iterations has decreased by approximately 5 times in comparison with the original algorithm. Among all the schemes considered, this showed the best result.

Parallel: As can be seen from the graph above, the number of iterations has decreased by approximately 5 times in comparison with the original algorithm. However, the result was somewhat worse than the approximating one.

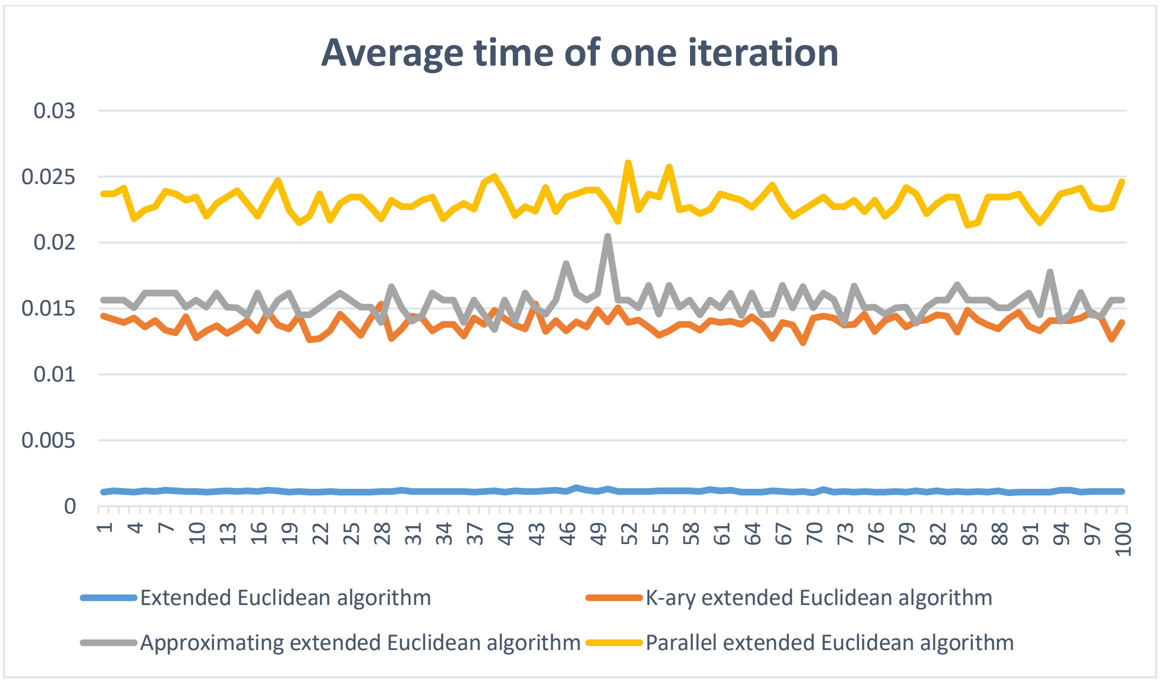

Figure 3.4. Comparison of the running time of the step with the original algorithm (prime, 8-bit).

k-ary: As can be seen from the graph above, the amount of work on one iteration increased significantly in comparison with the original algorithm. This leads to a time lag in comparison with the original algorithm. It is also clear that the running time at the step is very close to the approximating algorithm.

Approximating: as can be seen from the graph above, the amount of work on one iteration increased approximately 15 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm.

Parallel: As can be seen from the graph above, the amount of work on one iteration increased approximately 20 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm. This also leads to a lag in comparison with the approximating algorithm.

3.2.3. 16-bit module

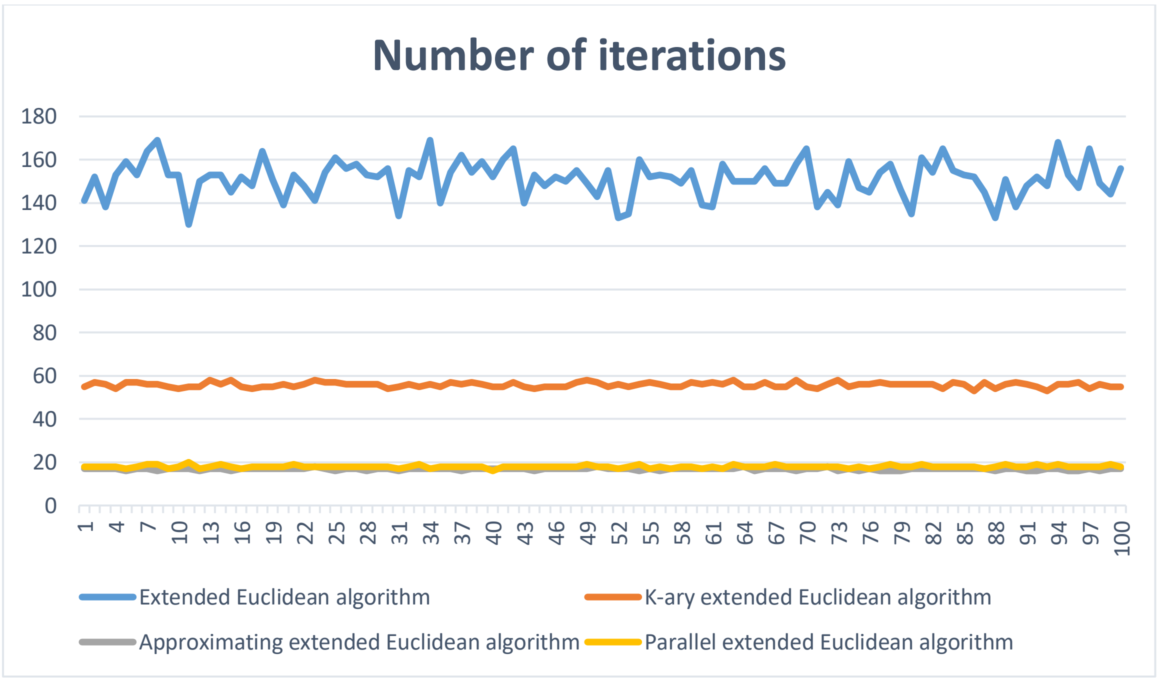

Figure 3.5. Comparison of the number of steps with the original algorithm (prime, 16–bit).

k–ary: As can be seen from the graph above, the number of steps has become more than 2 times less compared to the original algorithm. However, of all the algorithms considered, this is the worst result.

Approximating: As can be seen from the graph above, the number of iterations has decreased by about 7 times compared to the original algorithm. The result was close to parallel.

Parallel: As can be seen from the graph above, the number of iterations has decreased by about 7 times compared to the original algorithm. The result was close to approximating.

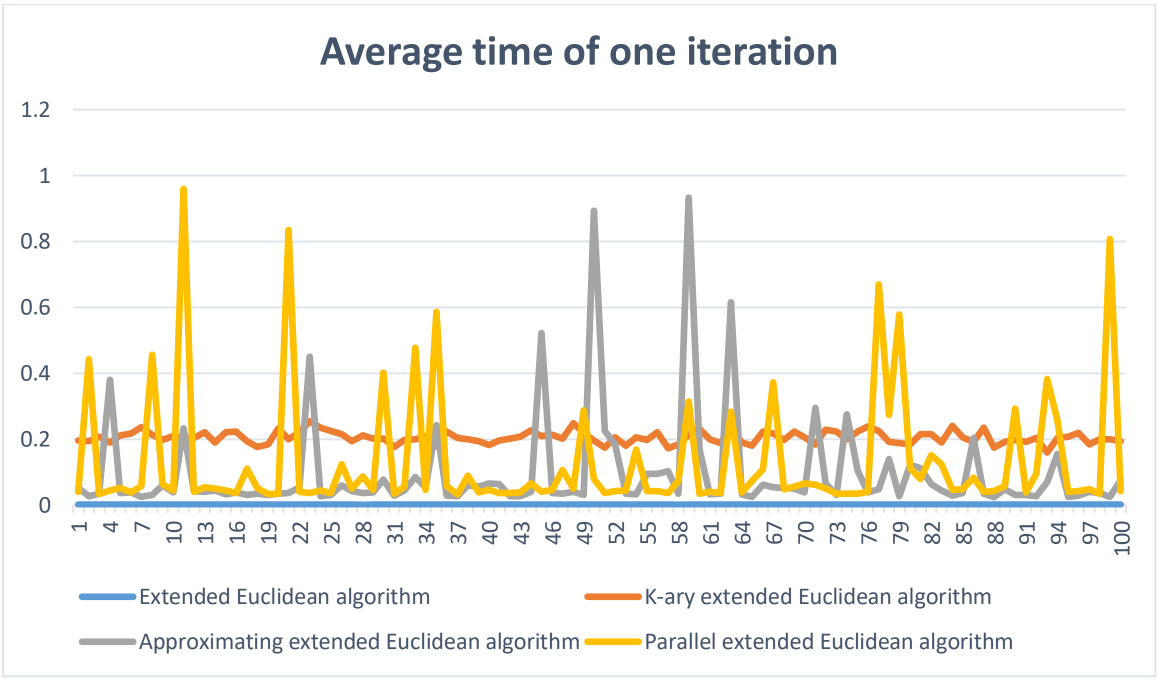

Figure 3.6. Comparison of the running time of the step with the original algorithm (prime, 16–bit).

k–ary: As can be seen from the graph above, the amount of work on one iteration increased about 20 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm.

Approximating: as seen from the graph above, the amount of work on one iteration has an extremely strong variance. Because of this, it becomes difficult to evaluate the attitude to the original algorithm. It also becomes difficult to compare with other schemes.

Parallel: As can be seen from the graph above, the amount of work on one iteration has an extremely strong variance. Because of this, it becomes difficult to evaluate the attitude to the original algorithm. It also becomes difficult to compare with other schemes.

3.3. Power of 2

3.3.1. 4-bit module

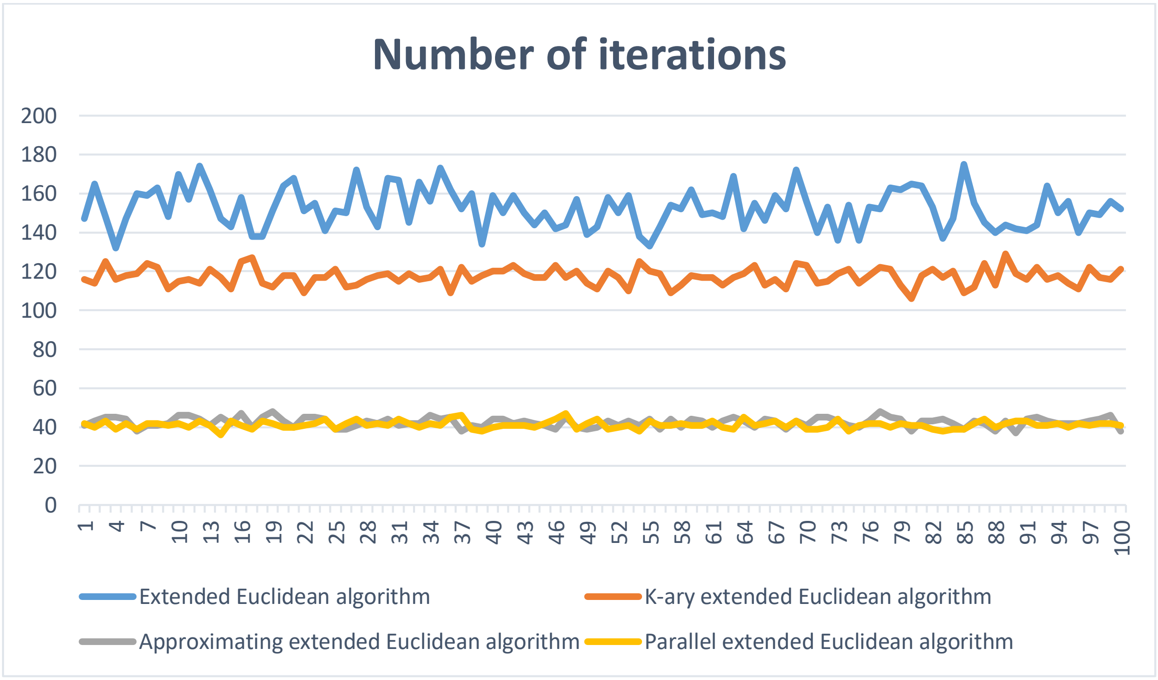

Figure 3.7. Comparison of the number of steps with the original algorithm (binary, 4-bit).

k-ary: As can be seen from the graph above, the number of steps became slightly less than 1.5 times less compared to the original algorithm. However, of all the algorithms considered, this is the worst result.

Approximating: as can be seen from the graph above, the number of iterations has decreased more than 3 times in comparison with the original algorithm. The result was close to parallel.

Parallel: As can be seen from the graph above, the number of iterations has decreased more than 3 times compared to the original algorithm. The result was close to approximating.

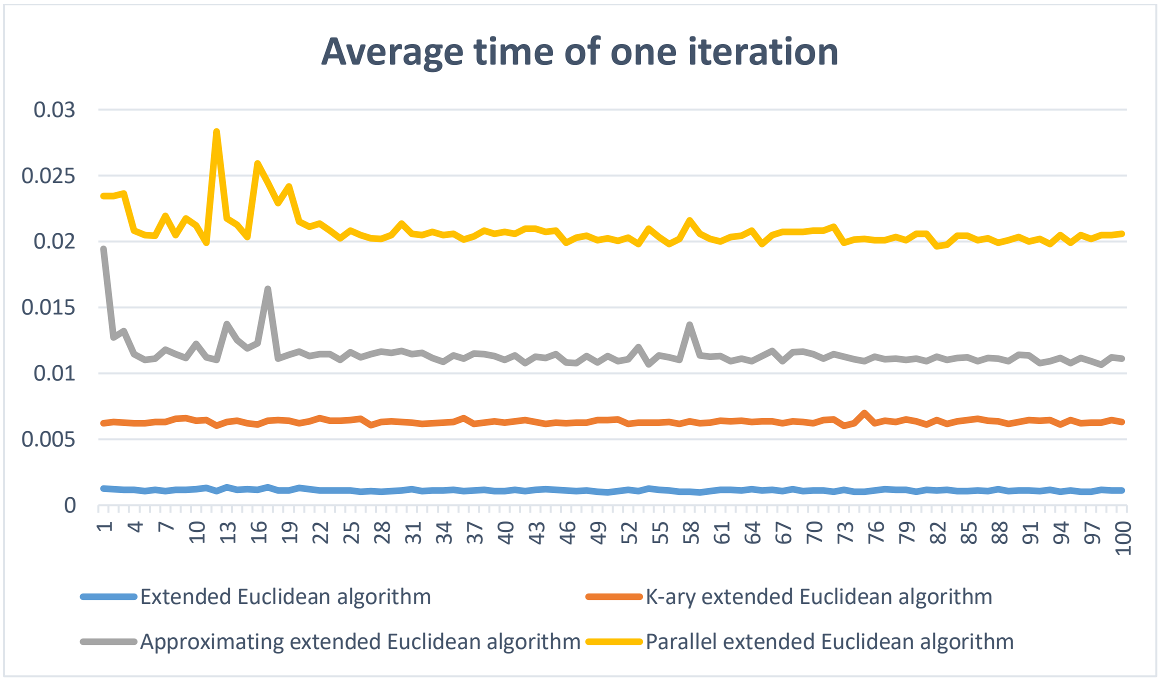

Figure 3.8. Comparison of the running time of the step with the original algorithm (binary, 4-bit).

k-ary: as seen from the graph above, the amount of work on one iteration increased about 6 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm. It is also clear that the running time at the step is very close to the approximating algorithm.

Approximating: as can be seen from the graph above, the amount of work on one iteration increased approximately 10 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm.

Parallel: As can be seen from the graph above, the amount of work on one iteration increased approximately 20 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm. This also leads to a lag in comparison with the approximating algorithm.

3.3.2. 8-bit module

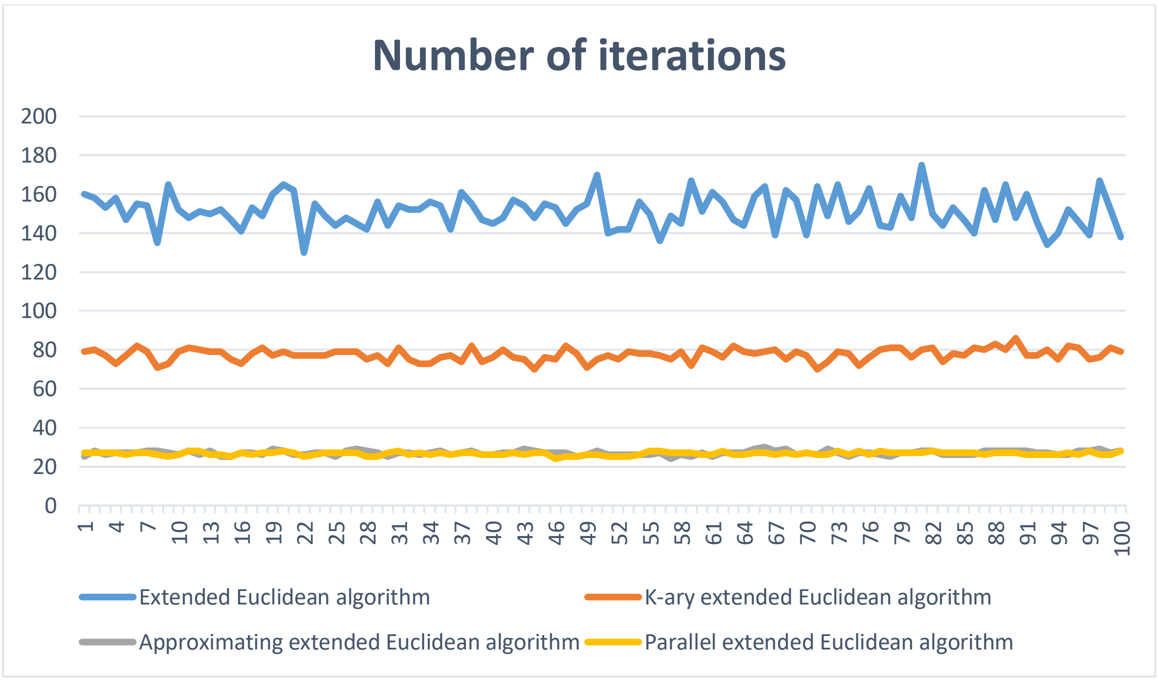

Figure 3.9. Comparison of the number of steps with the original algorithm (binary, 8-bit).

k-ary: As can be seen from the graph above, the number of steps became approximately 2 times less compared to the original algorithm. However, of all the algorithms considered, this is the worst result.

Approximating: As can be seen from the graph above, the number of iterations has decreased by approximately 6 times in comparison with the original algorithm. The result was close to parallel.

Parallel: As can be seen from the graph above, the number of iterations has decreased by approximately 6 times in comparison with the original algorithm. The result was close to approximating.

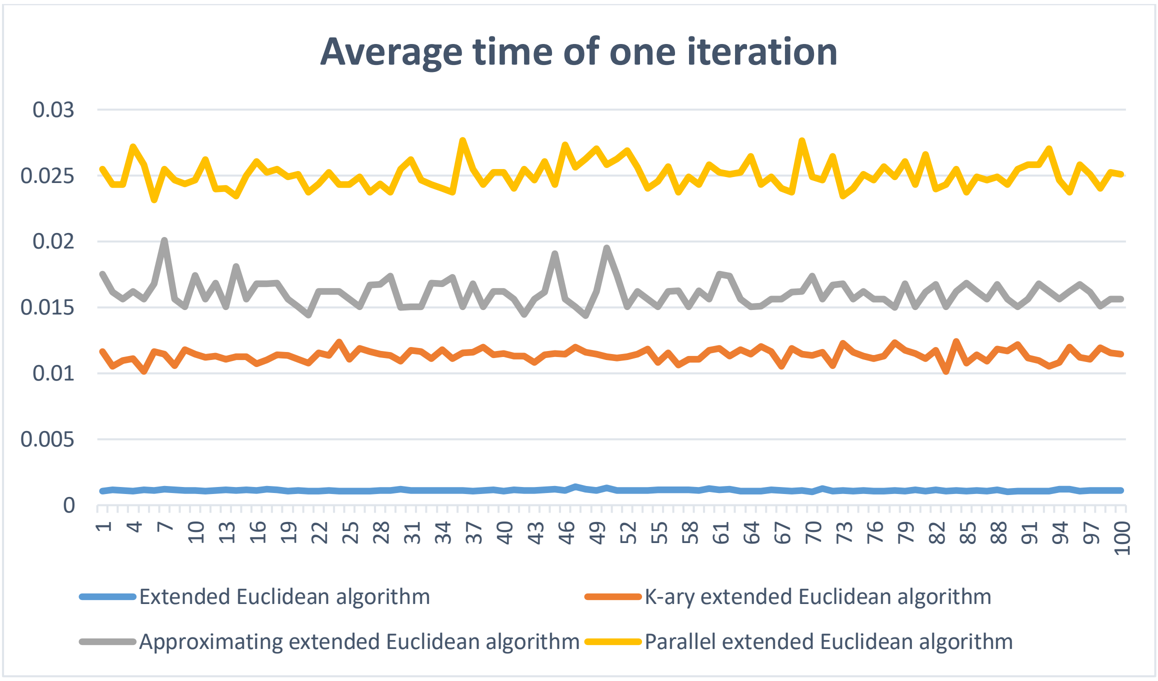

Figure 3.10. Comparison of the running time of the step with the original algorithm (binary, 8–bit).

k–ary: As can be seen from the graph above, the amount of work on one iteration increased by about 10 times in comparison with the original algorithm. This leads to a time lag in comparison with the original algorithm. It is also clear that the running time at the step is very close to the approximating algorithm.

Approximating: as can be seen from the graph above, the amount of work on one iteration increased approximately 15 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm.

Parallel: As can be seen from the graph above, the amount of work on one iteration increased approximately 20 times compared to the original algorithm. This leads to a time lag in comparison with the original algorithm. This also leads to a lag in comparison with the approximating algorithm.

3.3.3. 16-bit module

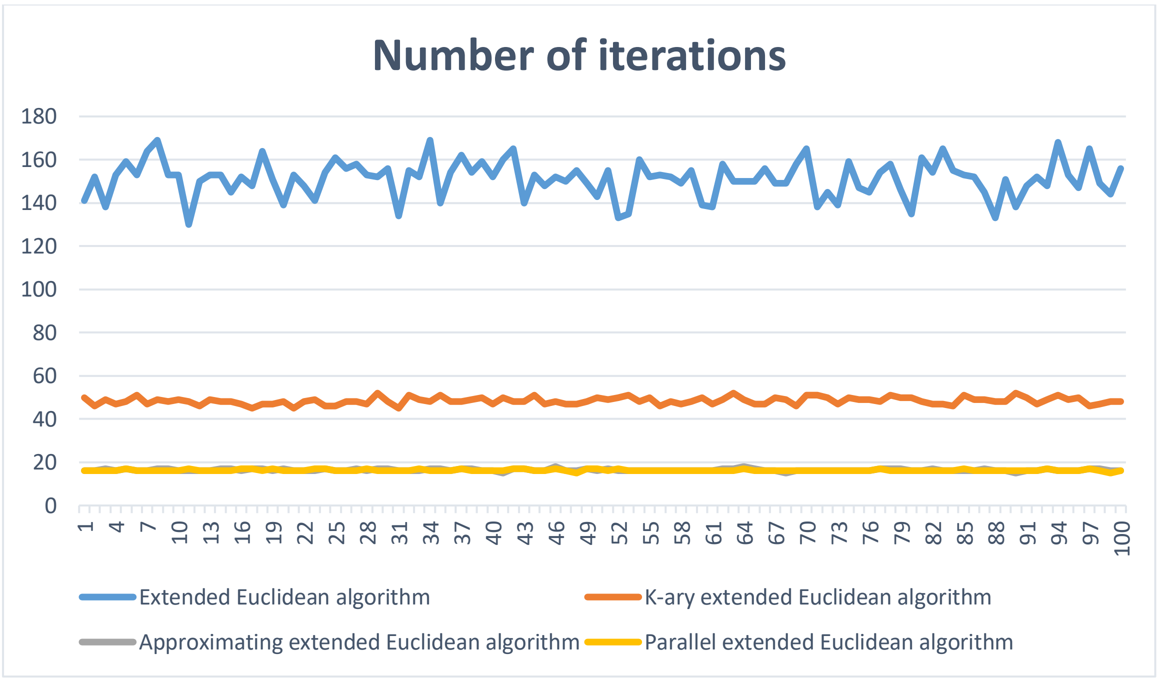

Figure 3.11. Comparison of the number of steps with the original algorithm (binary, 16–bit).

k–ary: As can be seen from the graph above, the number of steps became approximately 3 times less compared to the original algorithm. However, of all the algorithms considered, this is the worst result.

Approximating: As can be seen from the graph above, the number of iterations has decreased by about 7 times compared to the original algorithm. The result was close to parallel.

Parallel: As can be seen from the graph above, the number of iterations has decreased by about 7 times compared to the original algorithm. The result was close to approximating.

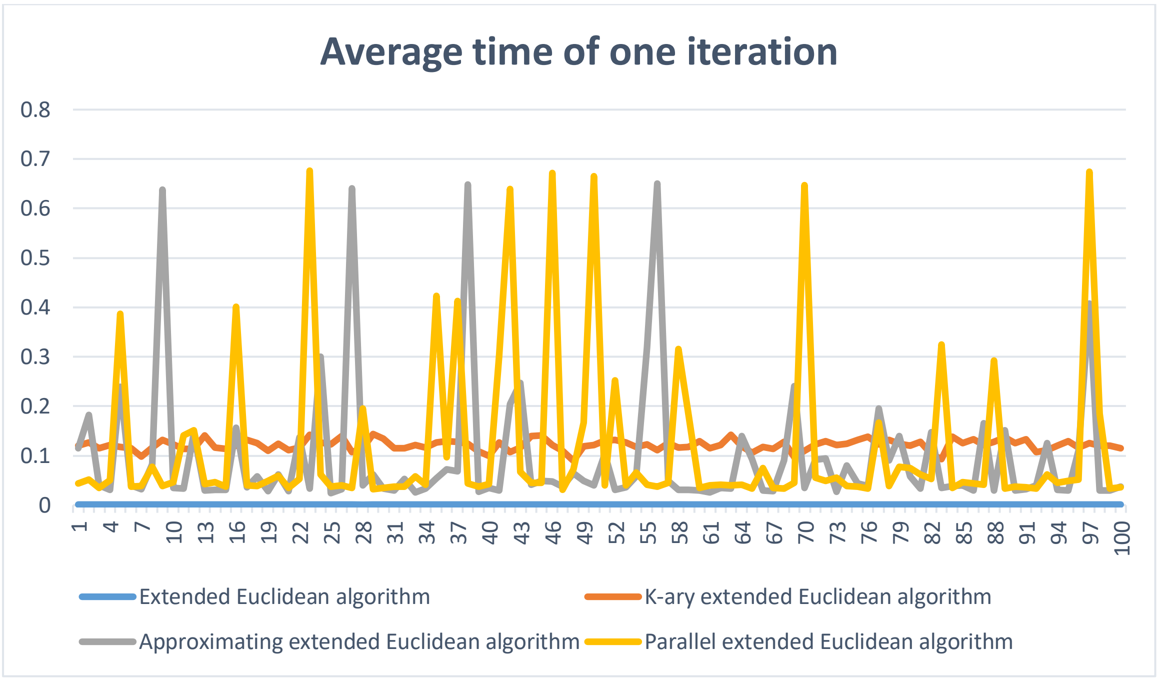

Figure 3.12. Comparison of the running time of the step with the original algorithm (binary, 16–bit).

k–ary: as seen from the graph above, the amount of work on one iteration has greatly increased in comparison with the original algorithm. This leads to a time lag in comparison with the original algorithm.

Approximating: as seen from the graph above, the amount of work on one iteration has an extremely strong variance. Because of this, it becomes difficult to evaluate the attitude to the original algorithm. It also becomes difficult to compare with other schemes.

Parallel: As can be seen from the graph above, the amount of work on one iteration has an extremely strong variance. Because of this, it becomes difficult to evaluate the attitude to the original algorithm. It also becomes difficult to compare with other schemes.

3.4. Conclusions on the chapter 3

Based on the results of the work, a program has been written that is attached to the application. The whole program is divided into two sections.

The first section of the program is the Testing.py file. It contains the code for all considered algorithms that are not involved in this study. This section sets the task of preliminary testing of all the considered schemes before use in further work.

The second section of the program is all other files. Participate directly in this work. They are divided into three parts:

- Finding the coefficients , and . It is represented by one file Coefficients.py. Contains all algorithms for finding new coefficients – direct search, use of the predefined inverse elements, an approximate variant and a parallel approximating variant. Implemented as options for an arbitrary number, and for the case with bit arithmetic.
- A directly extended Euclidean algorithm with all the schemes considered. It is represented by files: GCD.py, k_GCD.py, bin_k_GCD.py, paral_k_GCD.py and bin_paral_k_GCD.py. All the schemes studied were implemented using the search for coefficients from the first part and an additionally extended algorithm without comparison schemes. Implemented as an option for a simple module, and for a power of 2.
- Use of ready-to-use Bezout coefficient search functions. It is represented by 2 files: Mastership_test.py, created for testing the received algorithms; Mastership.py directly performing experiments.

The Mastership.py file contains two main functions: Gen(), which has an input number of generated pairs, generates pairs of random numbers during the operation and writes them to a file; Test(), which measures the time performance on the generated pairs and writes the results to files.

CONCLUSION

The theoretical and practical questions of the Bezout coefficients search are considered. Developed: a program containing the implementation of all the considered optimization schemes for the extended Euclidean algorithm. Also, in the dissertation, to evaluate the efficiency of optimization of the extended Euclidean algorithm, experiments were performed and the results are shown.

As a result of the work, the following results were obtained:

- The analysis of ordinary and extended Euclidean algorithms is carried out. The schemes of algorithms are analyzed and their convergence is estimated. Also possible practical applications of algorithms are shown.
- The analysis of existing methods of optimization of the Euclidean algorithm is carried out. The chosen ones were: binary, k-ary, and approximating algorithms. The schemes of algorithms are analyzed and their convergence is estimated.
- The analysis of the binary extended Euclidean algorithm is carried out. The scheme of the algorithm is analyzed and its convergence is estimated. Also, arguments were given why this algorithm is not considered in this work.
- The analysis of optimization methods of the Euclidean algorithm is carried out. The chosen ones were: k-ary, approximating algorithms. Their modifications for the extended Euclidean algorithm are presented. Possible exceptional situations are considered and methods for their solution are analyzed. The schemes of algorithms are analyzed and their convergence is estimated.
- The analysis of the parallel extended Euclidean algorithm is carried out. The scheme of the algorithm is analyzed and its convergence is estimated.
- Implemented all the analyzed schemes of optimization of the Euclidean algorithm. The correctness of the work of the received implementations was tested in practice.
- Implemented all the analyzed optimization schemes for the extended Euclidean algorithm. The correctness of the work of the received implementations was tested in practice.
- Implemented a program for conducting experiments. The time performance of the algorithms obtained during the program are stored in files.

- The analysis of the received characteristics is carried out. Graphical display of results.

In the course of this work, the following conclusions were made on the schemes considered:

- K-ary — showed the worst result among all the considered schemes. At small values of the module, the number of steps turned out to be greater than that of the original one, at large there was a significant gain. The amount of work on one iteration with a small module is less than that of other schemes, but more than the original algorithm. This leads to a total loss in time for the entire algorithm in comparison with the original.
- Approximating — showed the best result among all the considered schemes. The number of steps is several times less than the original algorithm. For large values of the module, the number of steps coincides in parallel with the number of steps. The amount of work on one iteration with a small module is less than that of a parallel module, but more than the original algorithm. However, for large values, the dispersion becomes too large, which complicates the analysis. One of the reasons may be the nonuniform convergence of the algorithm on Farey's fractions.
- Parallel — the number of steps is several times less than the original algorithm. For large values of the module, it coincides with the number of steps with the approximating step. The amount of work on one iteration with a small module is the largest in comparison with other algorithms. However, for large values, the dispersion becomes too large, which complicates the analysis. One of the reasons may be the nonuniform convergence of the algorithm on Farey's fractions.

The further direction of the study may be the reduction of the work time spent on one iteration. Among the possible approaches: the disclosure of internal function calls, the use of a lower-level language, the use of a faster algorithm for selecting coefficients.

ACRONYMS

1. — greatest common divisor of number and .
2. — comparability of numbers and by modulo . and give the same remainder upon division by .
3. — remainder of dividing by .
4. — incomplete quotient of division by .

BIBLIOGRAPHY

1. Akhavi, A., Vallee, B. Average Bit-Complexity of Euclidean Algorithms, In: Proceedings ICALP'00, Lecture Notes Computer Science 1853, 373–387 (2000).
2. Akritas A. Elements of Computer Algebra with applications: translation from English. (Russian) — Moscow, Mir, 1994.
3. Brent, R. Twenty years' analysis of the Binary Euclidean Algorithm, Millenial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in honour of A. Hoare, Palgrave, NY, 4153 (2000).
4. Brocot, A.: Calcul des rouages par approximation, nouvelle methode. Revue Chronom´etrique 6 (1860), 186–194.
5. Chor, B., Goldreich, O. An improved Parallel Algorithm for integer GCD, Algorithmica, 5, 1–10 (1990).
6. Crandell R., Pomerans P. Simple numbers: cryptographic and computational aspects (Russian) (URRS, Moscow, 2011)
7. Dixon, J. The Number of Steps of the Euclidian Algorithm, Journal of Number Theory, 1, 414–422 (1970).
8. Dolgov D. A. Comparative Analysis of Module Reduction Calculation Algorithms. Research Journal of Applied Sciences, 2015, 10, 442–446.
9. Dolgov D. A. GCD calculation in the search task of pseudoprime and strong pseudoprime numbers, Lobachevskii Journal of Mathematics, 2016, Volume 37, Issue 6.
10. Egorov D. F. Elements of number theory. (Russian) — Moscow— Petrograd: Gosizdat, 1923.
11. Farey, J.: On a Curious Property of Vulgar Fractions. London, Edinburgh and Dublin Phil. Mag. 47, 385, 1816
12. Forisek M. "Approximating Rational Numbers by Fractions". Bratislava, Slovakia, 2007.
13. Hardy, G.H., Wright, E.M. An Introduction to the Theory of Numbers, Oxford, Calrendon Press, 4-th ed (1959).
14. Hasse H. «Vorlesungen Über Zahlentheorie», Berlin, 1950
15. Ishmuhametov S.T. A parallel computation of the GCD of natural numbers, Kazan Federal University, Kazan, Russia, 2016
16. Ishmukhametov S. T., Rubtsova R. G. A fast algorithm for counting GCD of natural numbers, Proc. of intern.conf. Algebra, Analyses and Geometry, Kazan, Kazan Federal University (2016)
17. Ishmukhametov S., Mubarakov B., Mochalov A. Euclidian Algorithm for Recurrent Sequences, Applied Discrete Mathematics and Heuristic Algorithms, International Scientific Journal, Samara, 1, №2, 57–62, 2015.
18. Ishmukhametov S.T. Calculation of the Bezout coefficients for the k-ary algorithm for finding the GCD. (Russian) Kazan Federal University, Kazan, Russia, 2016.
19. Ishmukhametov S.T. Rubtsova R.G. On the approximating k-arn algorithm for calculating GCD. (Russian) Kazan Federal University, Kazan, Russia, 2016.
20. Ishmukhametov S.T. Approximating k-ary algorithm for finding the GCD. (Russian) Lobachevskii Journal of Mathematics, Kazan, Russia, 2016.
21. Jebelean T. A. Generalization of the Binary GCD Algorithm, Proc. Of Intern.Symp.on Symb.and Algebr. Comp.(ISSAC'93), 111–116 (1993)

22. Jones, G. A., Jones, J. M. §1.2. Bezout's Identity // Elementary Number Theory. — Berlin: Springer–Verlag, 1998.
23. Knuth D. E. The Art of Computer Programming. — 3. — Addison–Wesley, 1997. — T. 2: Seminumerical Algorithms. — ISBN 0–201–89684–2.
24. Menezes A., van Oorschot P., Vanstone S. Handbook of Applied Cryptography. CRC Press, 1996.
25. Mordukhai–Boltovsky D. D. Euclid's Elements T. 2: translation from the Greek. and comm. ed. Vygodsky M. Y. and Veselovsky I.N. (Russian) — GITTL, 1949.
26. Ore O. Number Theory and Its History. — McGraw–Hill, 1948.
27. Savin A. The Phidias number –the golden ratio (Russian) // "Quantum": Popular science physics and mathematics journal. — 1997. — № 6.
28. Schohage, A. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In: European Computer Algebra Conference, 144, 315 (1982).
29. Sorenson J. The k–ary GCD Algorithm, Universitet of Wisconsin–Madison, Tecn.Report, 1–20 (1990)
30. Sorenson J. Two fast GCD Algorithms, J.Alg. 16, №1, 110–144 (1994)
31. Sorenson. J. An analysis of the generalized binary GCD algorithm, in: A. van der Poorten, A. Stein (Eds.), High Primes and Misdemeanors, Lectures in Honour of Hugh Cowie Williams, Banff, Alberta, Canada, AMS Math. Review, 2005, 41 (2005h: 11279), 254–258
32. Stein, J. (1967), "Computational problems associated with Racah algebra", Journal of Computational Physics, 1 (3): 397–405, doi:10.1016/0021–9991(67)90047–2, ISSN 0021–9991
33. Sushkevich A. K. Number theory. Elementary course. (Russian) – Kharkov: Publishing house of Kharkov University, 1954.
34. Tonkov T. On the average length of finite continued fractions // Acta Arithmetica. — 1974. — T. 26.
35. Weber K. The accelerated integer GCD algorithm, ACM Trans.Math.Software, 21, №1, 1–12 (1995)
36. https://brilliant.org/wiki/bezouts–identity/ [accessed Oct 2016]
37. http://cmcstuff.esyr.org/vmkbotva–r15/3%20%D0%BA%D1%83%D1%80%D1%81/6%20%D0%A1%D0%B5%D0%BC%D0%B5%D1%81%D1%82%D1%80/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F%20%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C%20%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2/abramov10_2005.pdf [accessed Jan 2017]
38. http://e–maxx.ru/algo/fibonacci_numbers [accessed Oct 2016]
39. http://e–maxx.ru/algo/stern_brocot_farey [accessed Nov 2017]
40. http://www.itlab.unn.ru/Uploads/coaChapter09.pdf [accessed Nov 2016]
41. http://mathworld.wolfram.com/PortersConstant.html [accessed Nov 2016]

Testing.py

```
import random
import Coefficients
import bin_k_GCD
#import bin_paral_k_GCD
#from Coefficients import FindBin as find
from Coefficients import FindApproxBin as find
from random import randint as rnd
from bin_k_GCD import k_extended_euclide as k_gcd
#from bin_paral_k_GCD import k_extended_euclide as k_gcd
from Coefficients import Farey

pow_p = 4
p = (1 << pow_p)
Inv = [0] * p
Inv[1] = 1
for i in range(3, p, 2):
   Inv[i] = pow(i, (p >> 1) - 1, p)
setattr(Coefficients, 'Bin_Inv', Inv)

def bin_ext_gcd(a, b):
   if a == 0:
      print(a, b, 0, 1)
      return [0, 1]
   if b == 0:
      print(a, b, 1, 0)
      return [1, 0]
   if a == b:
      print(a, b, 1, 0)
      return [1, 0]
   if (a % 2 == 0) & (b % 2 == 0):
      x, y = bin_ext_gcd(a // 2, b // 2)
      print(a, b, x, y)
      return [x, y]
   if a % 2 == 0:
      x, y = bin_ext_gcd(a // 2, b)
      if x % 2 == 0:
         x, y = x // 2, y
      else:
         x, y = (x + b) // 2, y - a // 2
      print(a, b, x, y)
      return [x, y]
   if b % 2 == 0:
      x, y = bin_ext_gcd(a, b // 2)
      if y % 2 == 0:
         x, y = x, y // 2
      else:
         x, y = x - b // 2, (y + a) // 2
      print(a, b, x, y)
      return [x, y]
   if a > b:
        x, y = bin_ext_gcd((a - b) // 2, b)
        if x % 2 == 0:
            x, y = x // 2, y - x // 2
        else:
            x, y = (x + b) // 2, y - (a - b) // 2 - (x + b) // 2
        print(a, b, x, y)
        return [x, y]
    if a < b:
        x, y = bin_ext_gcd(a, (b - a) // 2)
        if y % 2 == 0:
            x, y = x - y // 2, y // 2
        else:
            x, y = x - (b - a) // 2 - (y + a) // 2, (y + a) // 2
        print(a, b, x, y)
        return [x, y]

def bin_gcd(a, b):
    print(a, b)
    if a == 0:
        return b
    if b == 0:
        return a
    if a == b:
        return a
    if (a % 2 == 0) & (b % 2 == 0):
        return 2 * bin_gcd(a // 2, b // 2)
    if a % 2 == 0:
        return bin_gcd(a // 2, b)
    if b % 2 == 0:
        return bin_gcd(a, b // 2)
    if a > b:
        return bin_gcd((a - b) // 2, b)
    if a < b:
        return bin_gcd(a, (b - a) // 2)

def gcd(a, b):
    print(a, b)
    if b != 0:
        return gcd(b, a % b)
    return a

List = [[0, 1]]

def bezu(a, b):
    global List
    if b == 0:
        return
    if(len(List) == 1):
        print(a, b, 1, -(a // b))
        List.append([1, -(a // b)])
        bezu(b, a % b)
        return
    x1, y1 = List[-2]
    x2, y2 = List[-1]
    x = x1 - x2 * (a // b)
    y = y1 - y2 * (a // b)
    List.append([x, y])
    print(a, b, a // b, x, y)
    bezu(b, a % b)

def bezu1(a, b):
    if b == 0:
        print(a, b, '-', 1, 0)
        return [1, 0]
    x, y = bezu1(b, a % b)
    print(a, b, a // b, y, x - y * (a // b))
    return [y, x - y * (a // b)]

def k_gcd1(a, b):
    if b == 0:
        return a
    x, y, c = find(a, b, p)
    if c:
        while c & 1 == 0:
            c>>= 1
    print(a, b, x, y, c, ':')
    if b > c:
        return k_gcd(b, c)
    else:
        return k_gcd(c, b)

def Inverse(a):
    return ((1 - a * Inv[a % p]) // p) % a

a, b = [rnd(128, 256) * 2 + 1, rnd(128, 256) * 2 + 1]
191, 157
#[rnd(64, 128) * 2 + 1, rnd(64, 128) * 2 + 1]
if a < b:
    t = b
    b = a
    a = t
a, b = 391, 357
#[rnd(128, 256), rnd(128, 256)]

x, y = k_gcd(a, b)
print(a * x + b * y)

#r = random.random()
#t = Farey(r, 16)
#print(r, t)
```

Coefficients.py

```
from multiprocessing import Pool

def Crutch(A, x, k):
    C = A[0] * x[0] + A[1] * x[1]
    pow_d = 0
    C//= k
    if C != 0:
        while C % k == 0:
            C//= k
            pow_d+= 1
    if C >= 0:
        return [x[0], x[1], C, pow_d]
    return [-x[0], -x[1], -C, pow_d]

def CrutchBin(A, x, k):
    pow_k = k.bit_length() - 1
    C = A[0] * x[0] + A[1] * x[1]
    pow_d = 0
    C>>= pow_k
    if C != 0:
        while (C & (k - 1)) == 0:
            C>>= pow_k
            pow_d+= pow_k
        while C & 1 == 0:
            pow_d+= 1
            C>>= 1
    if C >= 0:
        return [x[0], x[1], C, pow_d]
    return [-x[0], -x[1], -C, pow_d]

def ParalCrutch(t):
    print(t)
    m, n = t[0]
    print(m, n)
    x = n
    s = -m - t[1] * n
    y = s * t[3] - t[4] * x
    return Crutch(t[2], [x, y], t[3])

def ParalCrutchBin(t):
    m, n = t[0]
    x = n
    s = -m - t[1] * n
    y = (s << (t[3].bit_length() - 1)) - t[4] * x
    return CrutchBin(t[2], [x, y], t[3])

def FindNaive(A, B, k):
    x = 1
    while x * x <= k:
        y = 1
        while y * y <= k:
            if (A * x + B * y) % k == 0:
                return Crutch([A, B], [x, y], k)
            if (A * x - B * y) % k == 0:
                return Crutch([A, B], [x, -y], k)
            y+= 1
        x+= 1
```

```
def Find(A, B, k):
   x = 1
   while x * x <= k:
      y = -((A * x * Inv[B % k]) % k)
      if y * y <= k:
         return Crutch([A, B], [x, y], k)
      y+= k
      if y * y <= k:
         return Crutch([A, B], [x, y], k)
      x+= 1

def FindNaiveBin(A, B, k):
   x = 1
   while x * x <= k:
      y = 1
      while y * y <= k:
         if ((A * x + B * y) & (k - 1)) == 0:
            return CrutchBin([A, B], [x, y], k)
         if ((A * x - B * y) & (k - 1)) == 0:
            return CrutchBin([A, B], [x, -y], k)
         y+= 1
      x+= 1

def FindBin(A, B, k):
   x = 1
   while x * x <= k:
      y = -((A * x * Bin_Inv[B & (k - 1)]) & (k - 1))
      if y * y <= k:
         return CrutchBin([A, B], [x, y], k)
      y+= k
      if y * y <= k:
         return CrutchBin([A, B], [x, y], k)
      x+= 1

def Farey(r, k):
   if r * k >= k - 0.5:
      return [1, 1]
   if r * k >= k - 1:
      return [k - 2, k - 1]
   if r * k <= 0.5:
      return [0, 1]
   if r * k <= 1:
      return [1, k - 1]
   if r < 0.5:
      m2 = 1
      n2 = int(1 / r)
      m1 = 1
      n1 = n2 + 1
   else:
      n1 = int(1 / (1 - r))
      m1 = n1 - 1
      n2 = n1 + 1
```

```
        m2 = n1
    while n1 + n2 < k:
        m = m1 + m2
        n = n1 + n2
        if r * n <= m:
            m2 = m
            n2 = n
        else:
            m1 = m
            n1 = n
    if (m1 / n1 + m2 / n2) >= 2 * r:
        return [m1, n1]
    return [m2, n2]

def FindApprox(A, B, k):
    q = (A * Inv[B % k]) % k
    u = (A - q * B)
    v = B * k
    s1 = u // v
    u-= s1 * v
    if u < 0:
        u+= v
        s1-= 1
    m, n = Farey(u / v, k)
    x = n
    s = -m - s1 * n
    y = s * k - q * x
    return Crutch([A, B], [x, y], k)

def FindApproxBin(A, B, k):
    pow_k = k.bit_length() - 1
    q = ((A * Bin_Inv[B & (k - 1)]) & (k - 1))
    u = ((A - q * B) >> pow_k)
    v = B
    s1 = u // v
    u-= s1 * v
    if u < 0:
        u+= v
        s1-= 1
    m, n = Farey(u / v, k)
    x = n
    s = -m - s1 * n
    y = (s << pow_k) - q * x
    return CrutchBin([A, B], [x, y], k)

def FareyPar(r, k):
    if r * k >= k - 1:
        return [[k - 2, k - 1], [1, 1]]
    if r * k <= 1:
        return [[0, 1], [1, k - 1]]
    if r < 0.5:
        m2 = 1
        n2 = int(1 / r)
        m1 = 1
        n1 = n2 + 1
    else:
        n1 = int(1 / (1 - r))
        m1 = n1 - 1
        n2 = n1 + 1
        m2 = n1
    while n1 + n2 < k:
        m = m1 + m2
        n = n1 + n2
        if r * n <= m:
            m2 = m
            n2 = n
        else:
            m1 = m
            n1 = n
    return [[m1, n1], [m2, n2]]

def FindApproxPar(A, B, k):
    q = (A * Inv[B % k]) % k
    u = (A - q * B)
    v = B * k
    s1 = u // v
    u-= s1 * v
    if u < 0:
        u+= v
        s1-= 1
    r = u / v
    res = FareyPar(u / v, k)
    res1 = [[res[0], s1, [A, B], k, q], [res[1], s1, [A, B], k, q]]
    #pool = Pool(2)
    #ans = pool.map(ParalCrutch, res1)
    #pool.close()
    #pool.join()
    ans = list(map(ParalCrutch, res1))
    return ans

def FindApproxParBin(A, B, k):
    pow_k = k.bit_length() - 1
    q = ((A * Bin_Inv[B & (k - 1)]) & (k - 1))
    u = ((A - q * B) >> pow_k)
    v = B
    s1 = u // v
    u-= s1 * v
    if u < 0:
        u+= v
        s1-= 1
    res = FareyPar(u / v, k)
    res1 = [[res[0], s1, [A, B], k, q], [res[1], s1, [A, B], k, q]]
    #pool = Pool(2)
    #ans = pool.map(ParalCrutchBin, res1)
    #pool.close()
    #pool.join()
    ans = list(map(ParalCrutchBin, res1))
    return ans
```

GCD.py

```
def extended_euclide(A, B):
    if B == 0:
        return [1, 0]
    u, v = extended_euclide(B, A % B)
    return [v, u - v * (A // B)]

def extended_euclide_with_count(A, B):
    if B == 0:
        return [1, 0, 1]
    u, v, s = extended_euclide_with_count(B, A % B)
    return [v, u - v * (A // B), s + 1]
```

k_GCD.py

```
import Coefficients
import fractions
#from Coefficients import FindNaive as find
#from Coefficients import Find as find
from Coefficients import FindApprox as find

MOD = 0
p = 251
Inv = [0] * p
Inv[1] = 1
for i in range(2, p):
    Inv[i] = -(p // i) * Inv[p % i]
    Inv[i] = (Inv[i] + p) % p
setattr(Coefficients, 'Inv', Inv)

def k_extended_euclide_iter(A, B):
    if B == 0:
        return [1, 0]
    x, y, C, pow_d = find(A, B, p)
    d = pow(p, pow_d)
    if B > C:
        u, v = k_extended_euclide_iter(B, C)
        return [v * x, u * p * d + v * y]
    else:
        u, v = k_extended_euclide_iter(C, B)
        return [u * x, v * p * d + u * y]


def k_extended_euclide_iter_with_count(A, B):
    global MOD
    if B == 0:
        return [1, 0, 1, 0]
    if A % B == 0:
        return [0, 1, 1, 0]
    x, y, C, pow_d = find(A, B, p)
    d = pow(p, pow_d, MOD)
    if B > C:
        u, v, s, d1 = k_extended_euclide_iter_with_count(B, C)
        u1 = (v * x) % MOD
        v1 = (u * p * d + v * y) % MOD
    else:
        u, v, s, d1 = k_extended_euclide_iter_with_count(C, B)
        u1 = (u * x) % MOD
        v1 = (v * p * d + u * y) % MOD
    return [u1, v1, s + 1, d1 + pow_d]

def k_extended_euclide(A, B):
    global MOD
    MOD = A
    if (A % p == 0) | (B % p == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - Inv[A % p] * A) // p
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= n - 1
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y]

def k_extended_euclide_with_count(A, B):
    global MOD
    MOD = A
    if (A % p == 0) | (B % p == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - Inv[A % p] * A) // p
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= n - 1
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y, n + 1]
```

bin_k_GCD.py

```
import Coefficients
import fractions
#from Coefficients import FindNaiveBin as find
#from Coefficients import FindBin as find
from Coefficients import FindApproxBin as find

MOD = 0
pow_p = 8
p = (1 << pow_p)
Inv = [0] * p
Inv[1] = 1
for i in range(3, p, 2):
    Inv[i] = pow(i, (p >> 1) - 1, p)
setattr(Coefficients, 'Bin_Inv', Inv)
```

```
def k_extended_euclide_iter(A, B):
   if B == 0:
      return [1, 0]
   x, y, C = find(A, B, p)
   C>>= pow_p
   pow_d = 0
   if C:
      while C & 1 == 0:
         C>>= 1
         pow_d+= 1
   if B > C:
      u, v = k_extended_euclide_iter(B, C)
      return [v * x, (u << (pow_d + pow_p)) + v * y]
   else:
      u, v = k_extended_euclide_iter(C, B)
      return [u * x, (v << (pow_d + pow_p)) + u * y]


def k_extended_euclide_iter_with_count(A, B):
   global MOD
   if B == 0:
      return [1, 0, 1, 0]
   if A % B == 0:
      return [0, 1, 1, 0]
   x, y, C, pow_d = find(A, B, p)
   if B > C:
      u, v, s, d1 = k_extended_euclide_iter_with_count(B, C)
      u1 = (v * x) % MOD
      v1 = (((u << pow_d) << pow_p) + v * y) % MOD
   else:
      u, v, s, d1 = k_extended_euclide_iter_with_count(C, B)
      u1 = (u * x) % MOD
      v1 = (((v << pow_d) << pow_p) + u * y) % MOD
   return [u1, v1, s + 1, d1 + pow_d]

def k_extended_euclide(A, B):
   global MOD
   MOD = A
   if (A & 1 == 0) | (B & 1 == 0):
      raise NameError("Incorrect input. Change it.")
   a = (1 - A) >> 1
   x, y, n, d = k_extended_euclide_iter_with_count(A, B)
   d+= (n - 1) * pow_p
   a = pow(a, d, A)
   x = (x * a) % A
   y = (y * a) % A
   t = (x * A + y * B) // A
   return [x - t, y]

def k_extended_euclide_with_count(A, B):
   global MOD
   MOD = A
```

```
    if (A & 1 == 0) | (B & 1 == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - A) >> 1
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= (n - 1) * pow_p
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y, n + 1]
```

paral_k_GCD.py

```
import Coefficients
import fractions
from Coefficients import FindApproxPar as find

MOD = 0
p = 251
Inv = [0] * p
Inv[1] = 1
for i in range(2, p):
    Inv[i] = -(p // i) * Inv[p % i]
    Inv[i] = (Inv[i] + p) % p
setattr(Coefficients, 'Inv', Inv)

def k_extended_euclide_iter(A, B):
    if B == 0:
        return [1, 0]
    x = [res[0][0], res[1][0]]
    y = [res[0][1], res[1][1]]
    C = [res[0][2] // p, res[1][2] // p]
    d = [1, 1]
    for i in range(2):
        if C[i]:
            while C[i] % p == 0:
                d[i]*= p
                C[i]//= p
    d1 = max(d)
    for i in range(2):
        d[i] = d1 // d[i]
    u, v = k_extended_euclide_iter(C[0], C[1])
    return [x[0] * u * d[0] + x[1] * v * d[1], y[0] * u * d[0] + y[1] * v * d[1]]

def k_extended_euclide_iter_with_count(A, B):
    global MOD
    if B == 0:
        return [1, 0, 1, 0]
    if A % B == 0:
        return [0, 1, 1, 0]
    res = find(A, B, p)
    x = [res[0][0], res[1][0]]
    y = [res[0][1], res[1][1]]
    C = [res[0][2], res[1][2]]
    pow_d = [res[0][3], res[1][3]]
    d = [pow(p, pow_d[0]), pow(p, pow_d[1])]
    d0 = max(d)
    for i in range(2):
        d[i] = d0 // d[i]
    u, v, s, d1 = k_extended_euclide_iter_with_count(C[0], C[1])
    u1 = (x[0] * u * d[0] + x[1] * v * d[1]) % MOD
    v1 = (y[0] * u * d[0] + y[1] * v * d[1]) % MOD
    return [u1, v1, s + 1, d1 + max(pow_d)]

def k_extended_euclide(A, B):
    global MOD
    MOD = A
    if (A % p == 0) | (B % p == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - Inv[A % p] * A) // p
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= n - 1
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y]

def k_extended_euclide_with_count(A, B):
    global MOD
    MOD = A
    if (A % p == 0) | (B % p == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - Inv[A % p] * A) // p
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= n - 1
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y, n + 1]
```

bin_paral_k_GCD.py

```
import Coefficients
import fractions
from Coefficients import FindApproxParBin as find

MOD = 0
pow_p = 8
p = (1 << pow_p)
Inv = [0] * p
Inv[1] = 1
for i in range(3, p, 2):
    Inv[i] = pow(i, (p >> 1) - 1, p)
setattr(Coefficients, 'Bin_Inv', Inv)

def k_extended_euclide_iter(A, B):
    if B == 0:
        return [1, 0]
    x = [res[0][0], res[1][0]]
    y = [res[0][1], res[1][1]]
    C = [res[0][2] >> pow_p, res[1][2] >> pow_p]
    pow_d = [0, 0]
    for i in range(2):
        if C[i]:
            while C[i] & 1 == 0:
                pow_d[i]+= 1
                C[i]>>= 1
    pow_d1 = max(pow_d)
    for i in range(2):
        pow_d[i] = pow_d1 - pow_d[i]
    u, v = k_extended_euclide_iter(C[0], C[1])
    return [((x[0] * u) << pow_d[0]) + ((x[1] * v) << pow_d[1]), ((y[0] * u) << pow_d[0]) + ((y[1] * v) <<
pow_d[1])]

def k_extended_euclide_iter_with_count(A, B):
    global MOD
    if B == 0:
        return [1, 0, 1, 0]
    if A % B == 0:
        return [0, 1, 1, 0]
    res = find(A, B, p)
    x = [res[0][0], res[1][0]]
    y = [res[0][1], res[1][1]]
    C = [res[0][2], res[1][2]]
    pow_d = [res[0][3], res[1][3]]
    if C[0] < C[1]:
        t = C[0]
        C[0] = C[1]
        C[1] = t
        t = x[0]
        x[0] = x[1]
        x[1] = t
        t = y[0]
        y[0] = y[1]
        y[1] = t
        t = pow_d[0]
        pow_d[0] = pow_d[1]
        pow_d[1] = t
    pow_d1 = max(pow_d)
    for i in range(2):
        pow_d[i] = pow_d1 - pow_d[i]
    u, v, s, d1 = k_extended_euclide_iter_with_count(C[0], C[1])
    u1 = (((x[0] * u) << pow_d[0]) + ((x[1] * v) << pow_d[1])) % MOD
    v1 = (((y[0] * u) << pow_d[0]) + ((y[1] * v) << pow_d[1])) % MOD
    return [u1, v1, s + 1, d1 + pow_d1]

def k_extended_euclide(A, B):
    global MOD
    MOD = A
    if (A & 1 == 0) | (B & 1 == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - A) >> 1
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= (n - 1) * pow_p
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y]

def k_extended_euclide_with_count(A, B):
    global MOD
    MOD = A
    if (A & 1 == 0) | (B & 1 == 0):
        raise NameError("Incorrect input. Change it.")
    a = (1 - A) >> 1
    x, y, n, d = k_extended_euclide_iter_with_count(A, B)
    d+= (n - 1) * pow_p
    a = pow(a, d, A)
    x = (x * a) % A
    y = (y * a) % A
    t = (x * A + y * B) // A
    return [x - t, y, n + 1]
```

Mastership_test.py

```
import random
import datetime
import GCD
import k_GCD
import bin_k_GCD
import paral_k_GCD
import bin_paral_k_GCD
from random import randint as rnd
from GCD import extended_euclide as gcd
from GCD import extended_euclide_with_count as gcd_cnt
from k_GCD import k_extended_euclide as k_gcd
from k_GCD import k_extended_euclide_with_count as k_gcd_cnt
from bin_k_GCD import k_extended_euclide as bin_k_gcd
from bin_k_GCD import k_extended_euclide_with_count as bin_k_gcd_cnt
from paral_k_GCD import k_extended_euclide as paral_k_gcd
from paral_k_GCD import k_extended_euclide_with_count as paral_k_gcd_cnt
from bin_paral_k_GCD import k_extended_euclide as bin_paral_k_gcd
from bin_paral_k_GCD import k_extended_euclide_with_count as bin_paral_k_gcd_cnt

N = 1<<16#1 << 256
M = 1000
a, b = [rnd(N // 2, N), rnd(N // 2, N)]
#a, b = [3234243321, 543543234]
while (a % 13 == 0) | (b % 13 == 0) | ((a & 1) == 0) | ((b & 1) == 0):
    a, b = [rnd(N // 2, N), rnd(N // 2, N)]
#a, b = 5641, 2484
a, b = 65269, 38957
```

```
#a, b = 550065, 492073
#a, b = 700177, 438587
#a, b =
103496410105685614694258950134719037771415278288559602065182004096950215434067,
67914742837363768444204967225748869296138108950035401011314871117989811132657#??
??
if a < b:
   a, b = b, a
print(a, b)

dt = datetime.datetime.now()
for i in range(M):
   t = gcd(a, b)
dt1 = datetime.datetime.now() - dt
print(t, dt1)
print(t[0] * a + b * t[1])

t = gcd_cnt(a, b)
print(t)

dt = datetime.datetime.now()
for i in range(M):
   t = k_gcd(a, b)
dt1 = datetime.datetime.now() - dt
print(t, dt1)
print(t[0] * a + b * t[1])

t = k_gcd_cnt(a, b)
print(t)

dt = datetime.datetime.now()
for i in range(M):
   t = bin_k_gcd(a, b)
dt1 = datetime.datetime.now() - dt
print(t, dt1)
print(t[0] * a + b * t[1])

t = bin_k_gcd_cnt(a, b)
print(t)

dt = datetime.datetime.now()
for i in range(M):
   t = paral_k_gcd(a, b)
dt1 = datetime.datetime.now() - dt
print(t, dt1)
print(t[0] * a + b * t[1])

t = paral_k_gcd_cnt(a, b)
print(t)

dt = datetime.datetime.now()
for i in range(M):
   t = bin_paral_k_gcd(a, b)
```

```
dt1 = datetime.datetime.now() - dt
print(t, dt1)
print(t[0] * a + b * t[1])

t = bin_paral_k_gcd_cnt(a, b)
print(t)
```

Mastership.py

```
import random
import datetime
import GCD
import k_GCD
import bin_k_GCD
import paral_k_GCD
import bin_paral_k_GCD
from random import randint as rnd
from GCD import extended_euclide as gcd
from GCD import extended_euclide_with_count as gcd_cnt
from k_GCD import k_extended_euclide as k_gcd
from k_GCD import k_extended_euclide_with_count as k_gcd_cnt
from bin_k_GCD import k_extended_euclide as bin_k_gcd
from bin_k_GCD import k_extended_euclide_with_count as bin_k_gcd_cnt
from paral_k_GCD import k_extended_euclide as paral_k_gcd
from paral_k_GCD import k_extended_euclide_with_count as paral_k_gcd_cnt
from bin_paral_k_GCD import k_extended_euclide as bin_paral_k_gcd
from bin_paral_k_GCD import k_extended_euclide_with_count as bin_paral_k_gcd_cnt

N = 1 << 256
M = 1000
Func = [[gcd, gcd_cnt], [k_gcd, k_gcd_cnt], [bin_k_gcd, bin_k_gcd_cnt],
        [paral_k_gcd, paral_k_gcd_cnt], [bin_paral_k_gcd, bin_paral_k_gcd_cnt]]
#Func = [[k_gcd, k_gcd_cnt], [bin_k_gcd, bin_k_gcd_cnt]]
S = "k_8/"
S1 = ["GCD/", "Approximate_GCD/", "Parallel_approximate_GCD/"]
#S1 = ["k_GCD/"]
S2 = ["step.txt", "time.txt", "tps.txt"]
N1, N2 = [N >> 2, N >> 1]

def Gen(num):
   f = open("k_4/input.txt", 'w')
   for i in range(num):
      a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
      while (a % 13 == 0) | (b % 13 == 0):
         a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
      f.write(str(a) + ',' + str(b) + '\n')
   f.close()
   f = open("k_8/input.txt", 'w')
   for i in range(num):
      a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
      while (a % 251 == 0) | (b % 251 == 0):
         a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
      f.write(str(a) + ',' + str(b) + '\n')
   f.close()
```

```
    f = open("k_16/input.txt", 'w')
    for i in range(num):
        a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
        while (a % 65521 == 0) | (b % 65521 == 0):
            a, b = [2 * rnd(N1, N2) + 1, 2 * rnd(N1, N2) + 1]
        f.write(str(a) + ',' + str(b) + '\n')
    f.close()

def Cnt(a, b, files, funcs):
    dt = datetime.datetime.now()
    for i in range(M):
        t = funcs[0](a, b)
    dt1 = datetime.datetime.now() - dt
    files[1].write(str(dt1.total_seconds()) + '\n')
    t = funcs[1](a, b)
    files[0].write(str(t[2]) + '\n')
    files[2].write(str(dt1.total_seconds() / t[2]) + '\n')

def Test():
    f = open(S + "input.txt")
    f1 = []
    for s1 in S1:
        f2 = []
        for s2 in S2:
            f2.append(open(S + s1 + s2, 'w'))
        f1.append(f2)
        if s1 != "GCD/":
            f2 = []
            for s2 in S2:
                f2.append(open(S + s1 + "bin_" + s2, 'w'))
            f1.append(f2)

    cnt = 0
    for d in f.readlines():
        a, b = map(int, d.split(','))
        if len(Func) != len(f1):
            print("ERROR!!!!")
            return
        for i in range(len(f1)):
            Cnt(a, b, f1[i], Func[i])
        cnt+= 1
        print("Progress:", cnt, '/', 100)

    for f2 in f1:
        for t in f2:
            t.close()

#Gen(100)
Test()
```