НАЦІОНАЛЬНА АКАДЕМІЯ НАУК УКРАЇНИ
ІНСТИТУТ ПРОБЛЕМ МОДЕЛЮВАННЯ
В ЕНЕРГЕТИЦІ ІМ. Г.Є. ПУХОВА

Кваліфікаційна наукова
праця на правах рукопису

МІСЬКО Віталій Миколайович

УДК 511:003.26.09

ДИСЕРТАЦІЯ

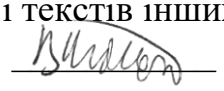
ОБЧИСЛЮВАЛЬНІ МЕТОДИ НА ОСНОВІ КВАДРАТИЧНОГО РЕШЕТА ПРИ КРИПТОАНАЛІЗІ RSA АЛГОРИТМУ АПАРАТНО-ПРОГРАМНИМИ ЗАСОБАМИ

Спеціальність 01.05.02 – математичне моделювання та обчислювальні методи

Галузь знань – технічні науки

Подається на здобуття наукового ступеня кандидата технічних наук

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

 В.М. Місько

Науковий керівник:
Винничук Степан Дмитрович,
доктор технічних наук,
старший науковий співробітник

Київ – 2019

АНОТАЦІЯ

Місько В.М. Обчислювальні методи на основі квадратичного решета при криптоаналізі RSA алгоритму апаратно-програмними засобами. - Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня кандидата технічних наук (доктора філософії) за спеціальністю 01.05.02 «Математичне моделювання та обчислювальні методи» (технічні науки). - Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, м. Київ, 2019.

Зміст анотації.

Сучасне суспільство все більше стає інформаційно-обумовленим. Завдання забезпечення захисту інформації при її обробці в інформаційних, телекомунікаційних та інформаційно-телекомунікаційних системах на даний час є одним з найбільш пріоритетних в багатьох країнах світу і, в тому числі, в Україні.

Обов'язковою умовою обробки інформації з обмеженим доступом в національних інформаційно-телекомунікаційних системах є застосування засобів технічного та/або криптографічного захисту інформації, які допущено до експлуатації. Рішення про допуск приймається за результатами тематичних досліджень, одним з елементів яких є оцінка криптографічної стійкості криптоалгоритмів, що використовуються в об’єктах досліджень.

У системах криптографічного захисту інформації сьогодні широко використовуються асиметричні криптоалгоритми, серед яких особливою популярністю користується алгоритм RSA. Результати досліджень щодо методів його криптоаналізу представлені в роботах авторів Song Y. Yan, Kocker P.C., Shamir A., D. Genkin, B. Weger Д., Sounak G. and Goutam P., Brown D., Авдошин С.М., Горбенко І.Д. та багатьох інших. Аналіз публікацій показує, що відомі приклади компрометації RSA алгоритму пов’язані з певними його реалізаціями, а в загальному випадку не ефективніші за задачу факторизації. Тому при дослідженні стійкості системи шифрування RSA значна увага приділяється вирішенню задачі факторизації її криптомодуля, для чого застосовуються кращі серед алгоритмів факторизації та використовуються останні технічні досягнення.

Серед багатьох методів факторизації метод квадратичного решета (QS – quadratic sieve) займає друге місце у списку найшвидших алгоритмів, поступаючись тільки методу решета числового поля, а для чисел розміром до 110 десяткових знаків і досі є найкращим. Відносна простота алгоритму сприяла виникненню багатьох його модифікацій. При цьому відмічається що основна проблема це - складність пошуку B-гладких чисел. Число B-гладких є відносно більшим при $X = X_0 = [\sqrt{N} + 1]$ та швидко зменшується при відхиленнях X від X_0. Тому кращою модифікацією QS вважається метод множинного квадратичного решета (MPQS – multiple polynomial quadratic sieve) в якому B-гладкі шукають серед остач $y_{a,b}(X) = (aX + b)^2 - N$, де a, b – спеціально підібрані цілі числа. В порівнянні з методом QS при a > 1 для поліномів $y_{a,b}(X)$ в a раз зменшується кількість можливих значень пробних X при тому ж радіусі просіювання. В той же час не досліджувалося використання поліномів виду $y_k(X) = X^2 - kN$, для яких при більшості значень k кількість пробних X в інтервалі просіювання залишається такою ж, як і для методу QS.

Тому можна очікувати, що при використанні поліномів $y_k(X)$ та однакового інтервалу просіювання при різних k зменшиться час роботи алгоритму з пошуку достатньої кількості B-гладких та будуть виявлені умови, при яких можливою стане факторизація чисел порядку до 2^{1024}, що важливо при вирішенні завдань криптоаналізу RSA алгоритму.

На час факторизації суттєво впливає розвиток технічних засобів. Проте методи QS та MPQS вимагають значного обсягу пам’яті для своєї реалізації що унеможливлює використання сучасних апаратних засобів (включаючи графічні карти). В зв’язку із чим виникає протиріччя між технічними можливостями сучасних апаратних засобів та існуючими способами алгоритмічної реалізації таких методів факторизації.

Тому необхідною є адаптація методу факторизації до можливостей апаратних реалізацій стосовно способів обробки даних при обмеженнях на обсяг доступної пам’яті та використання стандартних типів даних. У зв’язку з цим актуальною є задача удосконалення існуючих або розробка нових сучасних обчислювальних методів факторизації на основі методів QS та MPQS, які можна використовувати в

сучасних апаратно програмних комплексах, що забезпечить зниження обчислювальної складності в порівнянні з уже існуючими методами QS та MPQS при вирішенні завдань криптоаналізу RSA алгоритму.

У зв'язку з цим актуальною є задача удосконалення існуючих або розробка нових сучасних обчислювальних методів факторизації на основі методів QS та MPQS, які можна використовувати в сучасних апаратно програмних комплексах, що забезпечить зниження обчислювальної складності в порівнянні з уже існуючими методами QS та MPQS при вирішенні завдань криптоаналізу RSA алгоритму, що має важливу наукову та практичну спрямованість при удосконаленні існуючих і створенні перспективних апаратно-програмних засобів криптоаналізу асиметричних криптоалгоритмів RSA.

На основі проведених досліджень були отримані такі наукові результати:

1. Вперше розроблено метод множинного квадратичного k-решета (MQkS), в якому застосовується новий поліном $y_k(X) = X^2 - kN$ (k - натуральне число), який при більшості своїх варіантів забезпечує пошук B-гладких серед всіх пробних значень в єдиному інтервалі просіювання, в якому на відміну від методів QS та MPQS:
   - Використовуються загальна факторна база та поточна база (для кожного з поліномів).
   - Розмір інтервалу просіювання адаптовано під використання нового поліному.
   - Виконується попереднє просіювання пробних X.
   - При просіюванні пробних X, пошук дільників остач $y_k(X)$, показник степеня яких може перевищувати одиницю, здійснюється для обмеженої кількості простих чисел з поточної факторної бази, за рахунок чого можливе врахування обмежень на обсяг пам'яті та доступні стандартні типи даних апаратних засобів.

Встановлено, що існує діапазон значень параметрів для визначеної множини чисел порядку 10^m, де m=20÷32, отримано значення коефіцієнту $C < 1$ в оцінці складності методу MQkS виду $O\left(exp\left(C\sqrt{ln\,N\,lnln\,N}\right)\right)$. У відомих оцінках обчислювальної

складності методів методів QS та MPQS коефіцієнт $C \geq 1$. Для аналізованої множини чисел N порядку 10^m, де m=9÷32 встановлено також, що в порівнянні з методом QS кількість пробних X, на основі яких шукають B-гладкі, в 6 та більше разів перевищує їх кількість для аналогічного числа пробних в методі QS та зменшується час пошуку B-гладких.

2. Запропоновано метод діагоналізації матриці «на ходу» та метод визначення достатньої кількості B - гладких чисел, що в окремих випадках може забезпечити розкладання криптомодуля N на множники раніше ніж будуть знайдені B-гладкі остачі у кількості більших ніж розмір факторної бази (незалежно від величини числа N). Даний результат може бути використаний для методів MQkS, QS, та MPQS.

3. За рахунок встановлення існування серед остач $y_k(X)$ таких, що $y_k(X) = y1(X) \cdot y2(X)$, де $y1(X)$ є добутком простих чисел з факторної бази, а $y2(X)$ – квадратом цілого числа, які названо умовно B-гладкими, запропоновано модифікацію методів MQkS, QS, та MPQS – умовно B-гладкі. Показано, що існують випадки, коли на основі застосування запропонованої модифікації можливе скорочення часу отримання достатньої кількості B-гладких, хоча спосіб виявлення умовно B-гладких є дуже затратним в обчислювальному плані та необхідні подальші дослідження стосовно способів їх отримання.

4. Запропоновано способи реалізації методу MQkS на апаратно-програмних засобах, які включають кластери та графічні процесори, в якому враховуються обмеження на стандартні типи даних та обсяг доступної пам’яті, а виконання арифметичних операцій з багаторозрядними числами, замінюються операціями з числами типу long (чи long long) та double.

Наукова цінність основних положень дисертаційного дослідження полягає у розробці методу множинного квадратичного k-решета в якому забезпечується зменшення розміру факторної бази та інтервалу просіювання. Організація роботи з великими числами на основі їх подання коефіцієнтами розкладання за деякою основою та розроблений метод роботи з матрицею дозволили реалізувати програму

для системи розподілених паралельних обчислень з відео картами, де мають місце обмеження на обсяг пам’яті та стандартні типи даних.

Практична цінність роботи полягає в тому, що розроблений обчислювальний метод MQkS та його модифікації дозволяють підвищувати швидкодію апаратно-програмних засобів, що використовуються при проведенні тематичних досліджень асиметричних криптоалгоритмів, за рахунок використання поліномів $X^2 - kN$, та методів діагоналізації матриці на ходу, або визначення достатньої кількості B-гладких а також врахування обмежень апаратно-програмних засобів за рахунок вибору значень параметрів методу MQkS, та способів представлення великих чисел через стандартні типи int64.

Ключові слова. цілочисельна факторизація, метод квадратичного решета, множинне решето, багаторозрядні числа, MQkS, RSA.

ANNOTATION

Misko Vitalii. The Computational methods based on the algorithm of Quadratic Sieve factorization method during cryptanalysis of RSA-algorithm by hardware and software methods. – the Manuscript.

Dissertation for obtaining the scientific degree of the candidate of technical sciences (doctor of philosophy) in specialty 01.05.02 "Mathematical modeling and computational methods". – Institute of Modeling Problems in Energy named by G. E. Puhov NAS of Ukraine, Kyiv, 2017.

Annotation content.

Modern society is increasingly becoming information-driven. The task of ensuring the protection of information during its processing in information, telecommunication and information and telecommunication systems is currently one of the highest priority in many countries of the world, including Ukraine.

A prerequisite for the processing of restricted information in national information and telecommunication systems is the use of technical and / or cryptographic protection of information that has been put into service. The decision on admission is taken on the basis of the results of the thematic studies, one of the elements of which is to evaluate the cryptographic stability of the cryptographic algorithms used in the research objects.

Asymmetric cryptographic algorithms are widely used today in cryptographic protection systems, among which the RSA algorithm is particularly popular. The results of research on the methods of its cryptanalysis are presented in the works of the authors Song Y. Yan, Kocker PC, Shamir A., D. Genkin, B. Weger D., Sounak G. and Goutam P., Brown D., Avdoshin SM, Gorbenko I.D. and many others. The analysis of publications shows that known examples of RSA algorithm compromising are related to certain implementations, and in general, are not more effective than the factorization problem. Therefore, when investigating the stability of the RSA encryption system, much attention is being paid to solving the problem of factorization of its cryptmodule, which uses the best factorization algorithms and uses the latest technical achievements.

Among many factorization methods, the quadratic sieve method (QS) is ranked second in the list of fastest algorithms, second only to the numerical field sieve method, and

for the numbers up to 110 decimal places is still the best. Relative simplicity of the algorithm contributed to the emergence of many of its modifications. It is noted that the main problem is the complexity of the search for B-smooth numbers. The number of B-smooth is relatively large at $X = X_0 = \left[\sqrt{N} + 1\right]$ and decreases rapidly with deviations of X from X_0. Therefore, the best modification of QS is the multiple polynomial quadratic sieve (MPQS) method in which the B-smooth searches among the remaining $y_{a,b}(X) = (aX + b)^2 - N$, where a, b are specially selected integers . In comparison with the QS method for a> 1, for the polynomials $y_{a,b}(X)$, the number of possible values of the X samples at the same radius of sieve decreases a times. At the same time, the use of polynomials of the form $y_k(X) = X^2 - kN$ was not investigated, for which for the majority of values k the number of test X in the interval of sifting remains the same as for the QS method.

Therefore, it can be expected that when using polynomials $y_k(X)$ and the same sieve interval at different k, the algorithm's time of searching for a sufficient number of B-smooths will decrease, and the conditions under which factorization of numbers up to 2^{1024} will be possible, which is important in solving the tasks of the cryptanalysis of the RSA algorithm, will be revealed.

At the time of factorization a significant influence on the development of technical means. However, methods QS and MPQS require a significant amount of memory for its implementation, which makes it impossible to use modern hardware (including graphics cards). In connection with this, there is a contradiction between the technical capabilities of modern hardware and the existing methods of algorithmic implementation of such factorization methods.

Therefore, it is necessary to adapt the factorization method to the capabilities of hardware implementations in terms of data processing methods with restrictions on the amount of available memory and the use of standard data types. In this connection, the task of improving existing or developing new modern computing factorization methods based on QS and MPQS methods, which can be used in modern hardware software complexes, is to reduce computational complexity in comparison with existing QS and MPQS methods in solving tasks of RSA algorithm cryptanalysis.

In this connection, the task of improving existing or developing new modern computing factorization methods based on QS and MPQS methods, which can be used in modern hardware software complexes, is to reduce computational complexity in comparison with existing QS and MPQS methods in solving tasks of the cryptanalysis of the RSA algorithm, which has an important scientific and practical orientation in improving the existing and creating promising hardware and software tools for cryptanalysis of asymmetric crypto algebras chalk RSA.

Based on the research carried out, the following scientific results were obtained:

1. For the first time, the method of a plural quadratic k-sieve (MQkS), in which the new polynomial yk (X) = X2-kN (k is a natural number) is used, which, for most of its variants, provides the search for B-smooth among all test values in a single sieve intervals, in which unlike QS and MPQS methods:

- The common factor base and the current base are used (for each polynomial).
- The size of the sowing interval is adapted to the use of a new polynomial.
- The preliminary sifting of the tests X is carried out.
- With sifting of the tests X, the search for the divisors remains yk (X), whose degree of power can exceed one, is carried out for a limited number of primes from the current factor base, due to which account may be taken of restrictions on the amount of memory and available standard types of hardware data .

It is established that there is a range of values of parameters for a certain set of numbers of the order of 10^m, where m=20÷32, the value of the coefficient C < 1 is obtained in the estimation of the complexity of the MQkS method of the form $O\left(exp\left(C\sqrt{ln\,N\,lnln\,N}\right)\right)$ obtained. In the known estimates of the computational complexity of methods of QS and MPQS methods, the coefficient C ≥ 1. For the analyzed set of numbers N of the order of 10^m, where m=9÷32, it is also established that, in comparison with the QS method, the number of test X on the basis of which B-smooth is sought, in 6 or more times exceeds their number for a similar number of tests in the QS method and reduces the time for the search for B-smooth.

2. The method of diagonalization of the matrix "on the go" is proposed and the method of determining a sufficient number of B-smooth numbers, which in some cases can provide a decomposition of the cryptmodule N into factors before the B-smooth remains in a quantity larger than the size of the factor base (regardless of the magnitude numbers N). This result can be used for MQkS, QS, and MPQS methods.

3. By establishing the existence of the remaining $y_k(X)$ such that $y_k(X) = y1(X) \cdot y2(X)$, where $y1(X)$ is the product of prime numbers from the quotient base, and $y2(X)$ is a square an integer called conditionally B-smooth, modification of the methods MQkS, QS, and MPQS is proposed - conditionally B-smooth. It has been shown that there are cases where, based on the application of the proposed modification, it is possible to reduce the time for obtaining a sufficient number of B-smooth, although the method for detecting the conditionally B-smooth is very costly in the computational plan and further research is needed on how to obtain it.

4. The methods of implementing the MQkS method on hardware-software tools that include clusters and image processors, which take into account the limitations on standard data types and the amount of available memory, and the execution of arithmetic operations with multi-digit numbers, are replaced by operations with numbers such as long (or long long) and double.

The scientific value of the main provisions of the dissertation research is to develop a method of a plural quadratic k-sieve in which the reduction of the size of the factor base and the sowing interval is ensured. Organization of work with large numbers on the basis of their presentation by expansion coefficients on some basis and the developed method of working with the matrix allowed to implement the program for distributed distributed computing system with video cards, where there is a limit on the amount of memory and standard data types.

The practical value of the work consists in the fact that the developed computational method MQkS and its modifications allow to increase the speed of hardware and software used in case studies of asymmetric cryptographic algorithms, by using the polynomial $X^2 - kN$, and methods of diagonalization of the matrix on the run, or determination of sufficient

the number of B-smooth as well as the limits of hardware-software due to the choice of values of the parameters of the method MQkS, and methods of representation of large numbers through the stand art types int64.

Keywords. integer factorization, quadratic sieve method, multiple sieve, multi-digit numbers, MQkS, RSA.

Список публікацій здобувача

1. Факторизация числа N= pq при простых p и q методом дискретного логарифмирования. / ВН Мисько, СД Винничук, АВ Жилин. // Электронное моделирование . – 2013. – № 5 (35). – С. 3-10.
2. Ускорение метода Ферма методом прореживания с использованием нескольких баз. /В.М. Місько. // Журнал «Безпека інформації». Ukrainian Scientific Journal of Information Security. – 2015. – № 1 (21). – С. 64-68. DOI: 10.18372/2225-5036.21.8310
3. Удосконалення методу квадратичного решета на основі використання розширеної факторної бази та формування достатньої кількості B - гладких чисел. / В.М. Місько, С.Д. Винничук. // Збірник "Information technology and security". – 2017. – том. 5. выпуск. 2 (9). – С. 67-75.
4. Прискорення методу квадратичного решета на основі використання додаткового пошуку B-гладких чисел. / Місько В.М. // Моделювання та інформаційні технології. Збірник наукових праць. – 2017. – №78. – С. 51-57.
5. Acceleration analysis of the quadratic sieve method based on the online matrix solving. / V. Misko, S. Vynnychuk. // Eastern-European journal of Enterprise technologies. – 2018. – №10 (2). – С. 33-38. DOI: 10.15587/1729-4061.2018.127596
6. “Прискорення методу квадратичного решета на основі використання умовно B-гладких чисел” / Місько В.М. // Міжнародний науково–технічний журнал. Системні дослідження та інформаційні технології. – 2018. – №1. – С. 99-106. DOI: 10.20535/SRIT.2308-8893.2018.1.08

7. “Метод множинного квадратичного k-решета цілочисельної факторизації.” / В.М. Місько, С.Д. Винничук. // Електронне моделювання. – 2018. – №5 (40) С. 3-26. DOI: https://doi.org/10.15407/emodel.40.05.003
8. Просіювання пробних значень в методі множинного квадратичного k-решета на основі сигнальних остач. / С.Д. Винничук, В.М. Місько. // Безпека інформації. – 2019. – №1 (25). – С. 45-52. DOI: 10.18372/2225-5036.25.13446
9. “Метод множинного квадратичного k-решета з використанням сигнальних остач при просіюванні пробних значень.” / В.М. Місько, С.Д. Винничук. // Електронне моделювання. – 2019. – №2 (41) С. 3-22. DOI: https://doi.org/10.15407/emodel.41.02.003
10. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания.” / В.М. Мисько. // Матеріали XXXIV Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 13-14 січня 2015 року. – тези доп. – м. Київ. – С.7.
11. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания с использованием нескольких баз. ” / С.Д. Винничук, В.М. Місько. // Матеріали XXXVI Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 12-13 січня 2016 року. – тези доп. – м. Київ. – С.22.
12. “Прискорення методу квадратичного решета на основі використання розширеної факторної бази та формування достатньої кількості B-гладких чисел” / В.М. Місько. // III Міжнародна науково-практична конференція "Інформаційна безпека та комп’ютерні технології" 19-20 квітня 2018 року.: тези доп. – м. Кропивницький., – С. 106-108.
13. “Прискорення методу квадратичного решета на основі пошуку додаткових B-гладких чисел.” / В.М. Місько // Матеріали VI заочної наукової конференції «Наукові підсумки 2017 р», 13.11.2017. Науковий журнал «ScienceRise». – 2017. – №12(41). м. Харків. – С. 67-71. DOI: 10.15587/2313-8416.2017.118298
14. «Прискорення методу квадратичного решета на основі рішення матриці на ходу.» / В.М. Місько // Програма I Міжнародної науково-практична конференції

“Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS). 05-06 квітня 2018 р,: тези доп. – К., – С. 272-274.

15. «Прискорення методу квадратичного решета на основі рішення матриці на ходу». / В.М. Місько // Щорічна науково-технічна конференція молодих вчених та спеціалістів ІПМЕ ім. Г.Є. Пухова НАН України. 16 травня 2018 року, м. Київ. ,: тези доп. – К., – С. 29-30.
16. Множинне Квадратичне К-решето (MQKS). / С.Д. Винничук. В.М. Місько // Матеріали VI міжнародної наукової конференції «Моделювання-2018». 12-14 вересня 2018 р.: тези доп. – К., – С. 207-210 .
17. Множинне квадратичне k-решето факторизації чисел. / С.Д. Винничук В.М. Місько // Матеріали науково-практичної конференції «Сучасні інформаційні технології та кібербезпека», 15-16 листопада 2018р.: тези доп. – К., – С. 194-197.
18. Метод множинного квадратичного k-решета з використанням сигнальних остач при просіювання пробних значень. / В.М. Місько, С.Д. Винничук // XII Міжнародна науково-практична конференція «Комп’ютерні системи та мережеві технології» 28 –30 березня 2019 року.: тези доп. – К., – С. 27-28.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ІзОД – інформація з обмеженим доступом

КЗІ – криптографічний захист інформації

RSA – алгоритм Rivest, Shamir и Adleman

ІТС – інформаційні телекомунікаційні системи

АКА – асиметричний криптографічний алгоритм

QS – quadratic sieve

MPQS – multiple polynomial quadratic sieve

GPGPU – general-purpose computing for graphics processing units

MQkS – multiple quadratic k-sieve

КСЗІ – комплексна система захисту інформації

ЗФБ – загальна факторна база

ПФБ – поточна факторна база

SMP – symmetric multiprocessing

MPP – massive parallel processing

PVP – Parallel Vector Process

SISD – single instruction stream / single data stream

MISD – multiple instruction stream / single data stream

SIMD – single instruction stream / multiple data stream

MIMD – multiple instruction stream / multiple data stream

ВСТУП

Актуальність теми. Сучасне суспільство все більше стає інформаційно-обумовленим, успіх будь-якій сфері діяльності приходить до того, хто володіє певними відомостями посилює успіх також відсутність її у конкурентів. І чим сильніше проявляється зазначений ефект, тим більше потенційні збитки від зловживань в інформаційній сфері, і тим більше потреба в захисті інформації.

Завдання забезпечення захисту інформації при її обробці в інформаційних, телекомунікаційних та інформаційно-телекомунікаційних системах на даний час є одним з найбільш пріоритетних в багатьох країнах світу і, в тому числі, в Україні.

В Україні питання захисту інформації регулюються Законами України [19], [20] та іншими нормативними актами, згідно яких інформація поділяється на відкриту та з обмеженим доступом (ІзОД).

Обов'язковою умовою обробки інформації з обмеженим доступом в національних інформаційно-телекомунікаційних системах є застосування засобів технічного та/або криптографічного захисту інформації (КЗІ) [21], які допущено до експлуатації. Рішення про допуск приймається за результатами тематичних досліджень, одним з елементів яких є оцінка стійкості криптографічних алгоритмів [3], що використовуються в об’єктах досліджень.

Серед методів криптографічного захисту виділяють асиметричний криптографічний алгоритм (АКА) RSA, який де-факто вважається стандартом для багатьох криптографічних сервісів і додатків [22]-[25].

Не зважаючи на появу у останній час нових, більш досконалих АКА, RSA алгоритм в даний час залишається достатньо широко розповсюдженим і використовується в різних засобах і технологіях криптографічного захисту інформації сучасних інформаційно-телекомунікаційних системах (ІТС) [26].

В Україні в національних ІТС до недавнього часу для забезпечення задач КЗІ для вирішення завдань безпечного функціонування державних інформаційних ресурсів з 2015 року, згідно [27], дозволено використання RSA для забезпечення банківської таємниці та забезпечення захищеного електронного документообігу. Система для електронного банкінгу «iBank 2 UA», в якій механізм обміну ключами

заснований на АКА RSA, впроваджена в більш ніж в 30 банках України [28]. Центри сертифікації ключів багатьох державних і комерційних установ побудовано з використанням алгоритму RSA [29]. Крім того, на даний час вже існують сертифіковані засоби, побудовані з використанням RSA, які допущено для захисту інформації що становить державну таємницю (апаратно-програмні засоби SECURE TOKEN–337, CRYPTOCARD–337, Гепард 2.0, Гряда-61) [30].

Згідно Указу Президента [31] для КЗІ використовуються криптосистеми і засоби криптографічного захисту, які допущені до експлуатації. Допуск до експлуатації – це комплекс організаційно-технічних заходів щодо проведення тематичних досліджень засобів КЗІ, криптографічних алгоритмів, що використовуються в таких засобах, невід'ємною складовою яких є криптографічні дослідження, які проводяться відповідними підрозділами Державної служби спеціального зв'язку та захисту інформації України [32], або, за їх дорученням, відповідними експертними організаціями – ліцензіатами у сфері КЗІ [33]. Обов'язковим етапом проведення криптографічних досліджень засобів та комплексів КЗІ є оцінка стійкості криптографічних алгоритмів та протоколів, що використовуються. Оцінка криптостійкості ґрунтується на алгоритмічній складності проведення криптоаналізу таких алгоритмів і повинна враховувати сучасні обчислювальні методи, які можуть бути використані для оцінки криптографічних якостей алгоритму.

Результати досліджень щодо методів криптоаналізу алгоритму RSA представлені в роботах авторів Song Y. Yan, Kocker P.C., Koblitz N., Kraft J., Авдошин С.М., Горбенко І.Д. та багатьох інших [34-43]. Аналіз публікацій показує, що відомі приклади компрометації RSA алгоритму пов'язані з певними його реалізаціями, а в загальному випадку не ефективніші за задачу факторизації. Тому при дослідженні стійкості системи шифрування RSA значна увага приділяється вирішенню задачі факторизації її критптомодуля, для чого застосовуються кращі серед алгоритмів факторизації та використовуються останні технічні досягнення.

Однак аналіз відкритих публікацій [44 - 46] показує, що більшість відомих прикладів компрометації RSA алгоритму відносяться до певних їх практичних реалізацій, проте в загальному випадку не є ефективнішими задач криптоаналізу, заснованих на математичних проблемах побудови RSA алгоритму, пов'язаних з виконанням факторизації багаторозрядних чисел, що становлять криптомодуль RSA алгоритму.

При оцінці криптостійкості АКА RSA серед інших методів криптографічного аналізу перевіряється можливість розкладання на множники (факторизації) RSA криптомодуля. Для цього використовуються відомі та найбільш ефективні методи факторизації [22],[47]-[50].

У випадку, коли RSA криптомодуль не вдається розкласти на множники (на два багаторозрядних простих числа), даний етап перевірки криптостійкості вважається успішно пройденим.

Проблема полягає в тому що немає актуальної інформації про останні досягнення у методах фаткоризації криптомодуля RSA алгоритму, якою може володіти зловмисник. Остання обставина визначає необхідність застосування останніх досягнень стосовно методів факторизації та сучасних апаратних засобів.

Серед багатьох методів факторизації метод квадратичного решета (QS – quadratic sieve) займає друге місце у списку найшвидших алгоритмів, поступаючись тільки методу решета числового поля, а для чисел розміром до 110 десяткових знаків і досі є найкращим. Відносна простота алгоритму сприяла виникненню багатьох його модифікацій. При цьому відмічається що основна проблема це - складність пошуку B-гладких чисел. Число B-гладких є відносно більшим при $X = X_0 = \left[\sqrt{N} + 1\right]$ та швидко зменшується при відхиленнях X від X_0. Тому кращою модифікацією QS вважається метод множинного квадратичного решета (MPQS – multiple polynomial quadratic sieve) в якому B-гладкі шукають серед остач $y_{a,b}(X) = (aX + b)^2 - N$, де a, b – спеціально підібрані цілі числа. В порівнянні з методом QS при $a > 1$ для поліномів $y_{a,b}(X)$ в a раз зменшується кількість можливих значень пробних X при тому ж радіусі просіювання. В той же час не досліджувалося використання поліномів виду $y_k(X) =$

$X^2 - kN$, для яких при більшості значень k кількість пробних X в інтервалі просіювання залишається такою ж, як і для методу QS.

Тому можна очікувати, що при використанні поліномів $y_k(X)$ та однакового інтервалу просіювання при різних k зменшиться час роботи алгоритму з пошуку достатньої кількості B-гладких та будуть виявлені умови, при яких можливою стане факторизація чисел порядку до 2^{1024}, що важливо при вирішенні завдань криптоаналізу RSA алгоритму.

На час факторизації суттєво впливає розвиток технічних засобів. Проте методи QS та MPQS вимагають значного обсягу пам'яті для своєї реалізації що унеможливлює використання сучасних апаратних засобів (включаючи графічні карти). В зв'язку із чим виникає протиріччя між технічними можливостями сучасних апаратних засобів та існуючими способами алгоритмічної реалізації таких методів факторизації.

Тому необхідною є адаптація методу факторизації до можливостей апаратних реалізацій стосовно способів обробки даних при обмеженнях на обсяг доступної пам'яті та використання стандартних типів даних. У зв'язку з цим актуальною є задача удосконалення існуючих або розробка нових сучасних обчислювальних методів факторизації на основі методів QS та MPQS, які можна використовувати в сучасних апаратно програмних комплексах, що забезпечить зниження обчислювальної складності в порівнянні з уже існуючими методами QS та MPQS при вирішенні завдань криптоаналізу RSA алгоритму.

Таким чином, в дисертації вирішується актуальна **наукова задача** – удосконалення існуючих або розробка нових сучасних обчислювальних методів факторизації на основі методів QS та MPQS, які можна використовувати в сучасних апаратно програмних комплексах, що забезпечить зниження обчислювальної складності в порівнянні з уже існуючими методами QS та MPQS при вирішенні завдань криптоаналізу RSA алгоритму, що має важливу наукову та практичну спрямованість при удосконаленні існуючих і створенні перспективних апаратно-програмних засобів криптоаналізу АКА RSA.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційні дослідження проводились в рамках НДР «Розвиток методів зниження енергоспоживання обчислювальних систем за рахунок оптимізації обробки масивів даних (шифр – ФРІСК)», (д/р № 0114U000879), НДР «Дослідження та розробка методів оцінювання захищеності інформації в розподілених високопродуктивних інформаційних системах при вирішенні задач енергетики (Шифр МОД-Д)», (д/р 0114U002361), що виконувалась в Інституті проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України.

Основні наукові результати отримали **експериментальне підтвердження і практичне впровадження** в наступних організаціях*:*

1. Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, НДР «ФРІСК», (д/р 0114U000879);
2. Інститут проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, НДР «МОД-Д», (д/р 0114U002361);

Об'єкт дослідження – є процес факторизації багаторозрядних чисел при проведенні криптоаналізу RSA-алгоритму;

Предмет дослідження – обчислювальні методи факторизації багаторозрядних чисел, засновані на алгоритмі квадратичного решета, що дозволяють зменшити обчислювальну складність операцій з великими числами, включаючи їх адаптацію до можливостей апаратних реалізацій при використанні в апаратно-програмних комплексах.

Мета і завдання дослідження.

Метою роботи є зменшення обчислювальної складності методів факторизації багаторозрядних чисел заснованих на ідеях методу квадратичного решета, що надасть можливість підвищити швидкодію апаратно-програмних засобів при вирішенні завдань криптоаналізу АКА RSA.

Для досягнення поставленої наукової мети в дисертаційній роботі вирішуються такі *задачі дослідження*:

1. Провести аналіз математичних методів і підходів до оцінки захищеності АКА RSA при проведенні тематичних досліджень. Визначити роль і місце алгоритму

факторизації квадратичного решета при оцінці криптостійкості RSA криптоалгоритму.

2. Розробити спосіб максимального використання близьких до $X_0 = \left[\sqrt{N} + 1\right]$ значень для отримання необхідної кількості B-гладких чисел.

3. Здійснити аналіз та оцінити можливість ефективного аналізу множини B-гладких чисел для прискорення отримання нульового вектору при рішенні системи лінійних алгебраїчних рівнянь (СЛАР), без збільшення необхідного об'єму пам'яті.

4. Розробити рекомендації стосовно можливості використання запропонованих методів та алгоритмів для апаратних та апаратно-програмних засобів при оцінці криптографічної стійкості RSA-алгоритму.

Методи дослідження. Для вирішення наукової задачі використані методи теорії чисел, теорії оптимізації – для аналізу методів факторизації чисел та їх модифікацій, теорії складності обчислень – для дослідження степені прискорення запропонованих модифікацій, чисельні методи, теорії алгоритмів та комп'ютерного моделювання – перевірка адекватності запропонованого чисельного методу.

До реалізації чисельних методів застосовується системний підхід, а саме блочно-ієрархічний та об'єктно-орієнтований підходи. На етапі дослідження запропонованих та реалізованих чисельних методів використовуються чисельний експеримент та методи його обробки.

Наукова новизна отриманих результатів визначається наступними положеннями:

1. Вперше розроблено метод множинного квадратичного *k*-решета (MQkS – multiple quadratic k-sieve), в якому застосовується новий поліном $y_k(X) = X^2 - kN$ (k - натуральне число), який при більшості своїх варіантів забезпечує пошук B-гладких серед всіх пробних значень в єдиному інтервалі просіювання, в якому на відміну від методів QS та MPQS:

   – Використовуються загальна факторна база та поточна база (для кожного з поліномів).

– Розмір інтервалу просіювання адаптовано під використання нового поліному.
– Виконується попереднє просіювання пробних *X*.
– При просіюванні пробних *X,* пошук дільників остач $y_k(X)$, показник степеня яких може перевищувати одиницю, здійснюється для обмеженої кількості простих чисел з поточної факторної бази, за рахунок чого можливе врахування обмежень на обсяг пам'яті та доступні стандартні типи даних апаратних засобів.

Встановлено, що існує діапазон значень параметрів для визначеної множини чисел порядку 10^m, де m=20÷32, отримано значення коефіцієнту $C < 1$ в оцінці складності методу MQkS виду $O\left(exp\left(C\sqrt{ln\,N\,lnln\,N}\right)\right)$. У відомих оцінках обчислювальної складності методів методів QS та MPQS коефіцієнт $C \geq 1$. Для аналізованої множини чисел N порядку 10^m, де m=9÷32 встановлено також, що в порівнянні з методом QS кількість пробних *X*, на основі яких шукають B-гладкі, в 6 та більше разів перевищує їх кількість для аналогічного числа пробних в методі QS та зменшується час пошуку B-гладких.

2. Запропоновано метод діагоналізації матриці «на ходу» та метод визначення достатньої кількості B - гладких чисел, що в окремих випадках може забезпечити розкладання криптомодуля *N* на множники раніше ніж будуть знайдені B-гладкі остачі у кількості більших ніж розмір факторної бази (незалежно від величини числа *N*). Даний результат може бути використаний для методів MQkS, QS, та MPQS.

3. За рахунок встановлення існування серед остач $y_k(X)$ таких, що $y_k(X) = y1(X) \cdot y2(X)$, де $y1(X)$ є добутком простих чисел з факторної бази, а $y2(X)$ – квадратом цілого числа, які названо умовно B-гладкими, запропоновано модифікацію методів MQkS, QS, та MPQS – умовно B-гладкі. Показано, що існують випадки, коли на основі застосування запропонованої модифікації можливе скорочення часу отримання достатньої кількості B-гладких, хоча спосіб виявлення умовно B-гладких є дуже

затратним в обчислювальному плані та необхідні подальші дослідження стосовно способів їх отримання.

4. Запропоновано способи реалізації методу MQkS на апаратно-програмних засобах, які включають кластери та графічні процесори, в якому враховуються обмеження на стандартні типи даних та обсяг доступної пам’яті, а виконання арифметичних операцій з багаторозрядними числами, замінюються операціями з числами типу long (чи long long) та double.

Практичне значення отриманих результатів полягає у тому що розроблений обчислювальний метод MQkS та його модифікації дозволяють підвищувати швидкодію апаратно-програмних засобів, що використовуються при проведенні тематичних досліджень АКА, за рахунок використання поліномів X2 – kN, та методів діагоналізації матриці на ходу, або визначення достатньої кількості B-гладких а також врахування обмежень апаратно-програмних засобів за рахунок вибору значень параметрів методу MQkS.

Зокрема, результати роботи дозволяють:

– додати ще один етап криптоаналізу АКА RSA апаратно-програмними засобами, як наслідок, збільшити ефективність криптоаналізу комерційних та державних експертиз у сфері КЗІ нових криптоалгортимів;

– проектувати більш ефективні, з точки зору швидкодії, апаратно-програмні засоби проведення криптоаналізу АКА та, як наслідок, зменшити строки виконання державних експертиз у сфері КЗІ нових криптоалгортимів;

– здійснювати оцінку криптостійкості АКА RSA з використанням апаратно-програмної архітектури паралельних обчислень за допомогою технології GPGPU (General-purpose Computing for Graphics Processing Units).

Отримані результати складають теоретичну, методологічну та технічну основу удосконалювання існуючих і створення нових ефективних обчислювальних методів криптоаналізу АКА.

Результати роботи реалізовані в Інституті проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України, а також використані в навчальному процесі ІСЗЗІ КПІ ім. Ігоря Сикорського (в курсі лекцій та практичних занять з навчальної дисциплін

«Теоретична криптологія», «Математичні методи побудови та аналізу асиметричних криптосистем»).

Особистий внесок здобувача. Всі результати дисертаційної роботи, що винесені на захист, отримані автором самостійно та опубліковані в [1-18]. Тема, мета і завдання дослідження сформульовані дисертантом спільно з науковим керівником. Особисто дисертантом визначено актуальність роботи, проведений огляд і аналіз літератури, встановлений необхідний обсяг досліджень. Дві статті опубліковано без співавторів. У роботах, які опубліковано у співавторстві, особисто дисертантові належать наступні результати: [1] – розроблений програмний додаток, проведено розрахунки та здійснено їх аналіз; [2, 10, 11] – алгоритм застосування декількох баз, експериментальне порівняння декількох баз з однією; [3, 12] – обґрунтовано метод достатньої кількості B-гладких, проведено аналіз результатів тестування; [4, 6, 13] – алгоритм умовно B-гладких, експериментальне порівняння зі стандартним методом QS; [5, 14, 15] – алгоритм рішення матриці на ходу, експериментальне порівняння зі стандартним методом QS, та вирішені ряд задач з оцінки складності алгоритму; [7, 16] – розроблено алгоритм методу MQkS, проведено аналіз оцінки впливу розміру факторної бази та інтервалу просіювання на час факторизації; [8, 9, 17, 18] – обґрунтовано метод використання сигнальних остач, та розроблена методика його застосування;

Апробація результатів дисертації. Основні ідеї та конкретні наукові результати досліджень доповідались й обговорювались на:

1. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания.” / В.М. Мисько. // Матеріали XXXIV Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 13-14 січня 2015 року. – тези доп. – м. Київ. – С.7.
2. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания с использованием нескольких баз. ” / С.Д. Винничук, В.М. Міська. // Матеріали XXXVI Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 12-13 січня 2016 року. – тези доп. – м. Київ. – С.22.

3. “Прискорення методу квадратичного решета на основі використання розширеної факторної бази та формування достатньої кількості B-гладких чисел” / В.М. Міський. //III Міжнародна науково-практична конференція "Інформаційна безпека та комп’ютерні технології" 19-20 квітня 2018 року.: тези доп. – м. Кропивницький., – С. 106-108.
4. “Прискорення методу квадратичного решета на основі пошуку додаткових B-гладких чисел.” / В.М. Міський // Матеріали VI заочної наукової конференції «Наукові підсумки 2017 р», 13.11.2017. Науковий журнал «ScienceRise». – 2017. – №12(41). м. Харків. – С. 67-71. DOI: 10.15587/2313-8416.2017.118298
5. «Прискорення методу квадратичного решета на основі рішення матриці на ходу.» / В.М. Міський // Програма I Міжнародної науково-практична конференції “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS). 05-06 квітня 2018 р,: тези доп. – К., – С. 272-274.
6. «Прискорення методу квадратичного решета на основі рішення матриці на ходу». / В.М. Міський // Щорічна науково-технічна конференція молодих вчених та спеціалістів ІПМЕ ім. Г.Є. Пухова НАН України. 16 травня 2018 року, м. Київ. ,: тези доп. – К., – С. 29-30.
7. Множинне Квадратичне К-решето (MQKS). / С.Д. Винничук. В.М. Міський // Матеріали VI міжнародної наукової конференції «Моделювання-2018». 12-14 вересня 2018 р.: тези доп. – К., – С. 207-210.
8. Множинне квадратичне k-решето факторизації чисел. / С.Д. Винничук В.М. Міський // Матеріали науково-практичної конференції «Сучасні інформаційні технології та кібербезпека», 15-16 листопада 2018р.: тези доп. – К., – С. 194-197.
9. Метод множинного квадратичного k-решета з використанням сигнальних остач при просіювання пробних значень. / В.М. Міський, С.Д. Винничук // XII Міжнародна науково-практична конференція «Комп’ютерні системи та мережеві технології» 28 –30 березня 2019 року.: тези доп. – К., – С. 27-28.

Структура та обсяг дисертації. Структура та обсяг дисертації. Робота складається з анотації, вступу, чотирьох розділів, висновків, двох додатків та списку використаних джерел зі 147 найменувань. Загальний обсяг дисертації складає 245

сторінки. Основний зміст роботи викладено на 165 сторінках. Дисертація містить 12 рисунків та 38 таблиць.

РОЗДІЛ 1. ЗАСТОСУВАННЯ МЕТОДІВ ФАКТОРИЗАЦІЇ В ЧИСЕЛЬНИХ МЕТОДАХ ПРИ ПРОВЕДЕННІ КРИПТОАНАЛІЗУ МОДУЛІВ RSA АЛГОРИТМУ.

У вступі до роботи були відзначені актуальність, предмет і цілі дисертаційного дослідження, сформульовано його загальна і часткові завдання дослідження. Однак весь матеріал викладався на рівні змістовного опису. Тому основним змістом першого розділу роботи є аналіз сучасного стану предмету досліджень, на основі чого сформульовані загальна і часткові наукові завдання, що вирішуються в дисертації.

1.1. Аналіз сучасних обчислювальних задач інформаційної безпеки.

Згідно з доктриною інформаційної безпеки України [51], інформаційна безпека є невід'ємною частиною національної безпеки держави. Інформатизація українського суспільства, що йде все більшими темпами [52], з кожним роком збільшує значення інформаційної безпеки в питанні забезпечення загальної безпеки держави.

Законом України [21] визначено поняття захисту інформації в системі - діяльність, спрямована на запобігання несанкціонованим діям щодо інформації в системі. До основних видів захисту інформації відносять: технічний та криптографічний захист інформації.

Інформаційна безпека є одним з найважливіших аспектів інтегральної безпеки, на якому б рівні не розглядалася остання - національному, галузевому, корпоративному або персональному.

При аналізі задач інформаційної безпеки необхідно враховувати її специфіку, яка полягає в тому що інформаційна безпека це невід’ємна частина інформаційно-телекомунікаційних технологій, розвиток якої характеризується недосяжно високими темпами зросту.

В [20], [21] визначено, що захист інформації забезпечується застосуванням комплексних систем захисту інформації (КСЗІ) із застосуванням засобів технічної та/або криптографічного захисту. У цьому ж документі надано поняття

криптографічного захисту інформації (КЗІ) – як виду захисту інформації, що реалізується перетворенням інформації з використанням спеціальних (ключових) даних з метою приховування/відновлення змісту інформації, підтвердження її справжності, цілісності, авторства тощо. Перетворення інформації (криптографічні перетворення), реалізуються на основі криптографічних алгоритмів, що забезпечують відповідні функції щодо зашифровування, розшифровування, цифрового підпису, або інших перетворень ІзОД.

Згідно [31] криптографічний захист інформації здійснюється за допомогою криптосистем і засобів криптографічного захисту, які допущені до експлуатації. Законом України «Про Державну службу спеціального зв'язку та захисту інформації України» [53] визначено, що допуском до експлуатації є комплекс організаційно-технічних заходів щодо проведення тематичних досліджень засобів криптографічного захисту інформації, криптографічних алгоритмів, засобів, систем і комплексів спеціального зв'язку та державної експертизи їх результатів з метою встановлення можливості їх використання за призначенням. Державна експертиза включає перевірку відповідності об'єктів експертизи вимогам нормативних документів, оцінку рівня захисту інформації об'єктами експертизи або науково-технічного рівня об'єктів експертизи.

З метою визначення рівня захищеності від несанкціонованого доступу до інформації з обмеженим доступом проводяться криптографічні дослідження криптосистем і засобів криптографічного захисту [31].

Тематичні дослідження криптографічних систем - дослідження щодо встановлення відповідності засобів криптографічного захисту інформації, криптосистем, криптографічних алгоритмів, засобів, систем і комплексів спеціального зв’язку вимогам тактико-технічних завдань на їх створення, нормативних документів, включаючи оцінку криптографічної стійкості, під якою розуміють його здатність протистояти спробам проведення успішного криптоаналізу по розкриттю або підборі ключових даних. [31].

Криптостійкість визначається шляхом:

- розрахунку стійкості криптографічних алгоритмів при тотальному переборі ключів і при відомих окремих елементах ключових даних;
- розрахунку стійкості криптографічного алгоритму при повторному використанні ключових даних;
- дослідження можливості отримання відкритої інформації на основі шифрованої без застосування інформації про криптографічний алгоритм (безключове читання);
- дослідження можливості застосування відомих методів криптографічного аналізу для зниження криптографічних якостей алгоритму.

Для підтримки актуальності тематичних досліджень повинно враховуватись зростання обчислювальних потужностей та розвиток алгоритмів. Проблема полягає в тому що немає актуальної інформації про останні досягнення у методах фаткоризації криптомодуля RSA алгоритму, якою може володіти зловмисник. Остання обставина визначає необхідність застосування останніх досягнень стосовно методів факторизації та сучасних апаратних засобів.

Загальний висновок про стійкість криптографічного алгоритму ґрунтується на алгоритмічній складності проведення його криптографічного аналізу і має враховувати сучасні обчислювальні методи, які можуть бути використані для його дешифрування.

1.2. Алгоритм RSA та його криптоаналіз.

Алгоритм RSA, названий так з першою буквою сім'ї своїх розробників, був розроблений в 1978 році, який де-факто вважається стандартом для багатьох криптографічних сервісів і додатків [4-9].

У алгоритмі RSA з початку відбувається вибір відкритого ключа, який представляє собою пару цілих чисел (e, N), де число $N = p * q$ називається *модулем RSA* і є твором двох великих простих чисел p та q, які зберігаються в таємниці. Також обчислюється таємний ключ $d \equiv e^{-1} mod\, \varphi(n)$ ($\varphi(n)$ - функція Ейлера). Далі повідомлення m, яке передається по відкритому каналу разом з відкритим ключем, шифрується як $c \equiv m^e mod\, N$ і потім, згідно з теоремою Ейлера, розшифровується за

допомогою таємного ключа: $c^d \equiv m \, mod \, N$. Якщо зловмиснику стане відома факторизація N, то він легко зможе обчислити таємний ключ d та розшифрувати повідомлення [54-56].

Процедури генерації ключів, шифрування та дешифрування для цього алгоритму представлені на рис. 1.1.

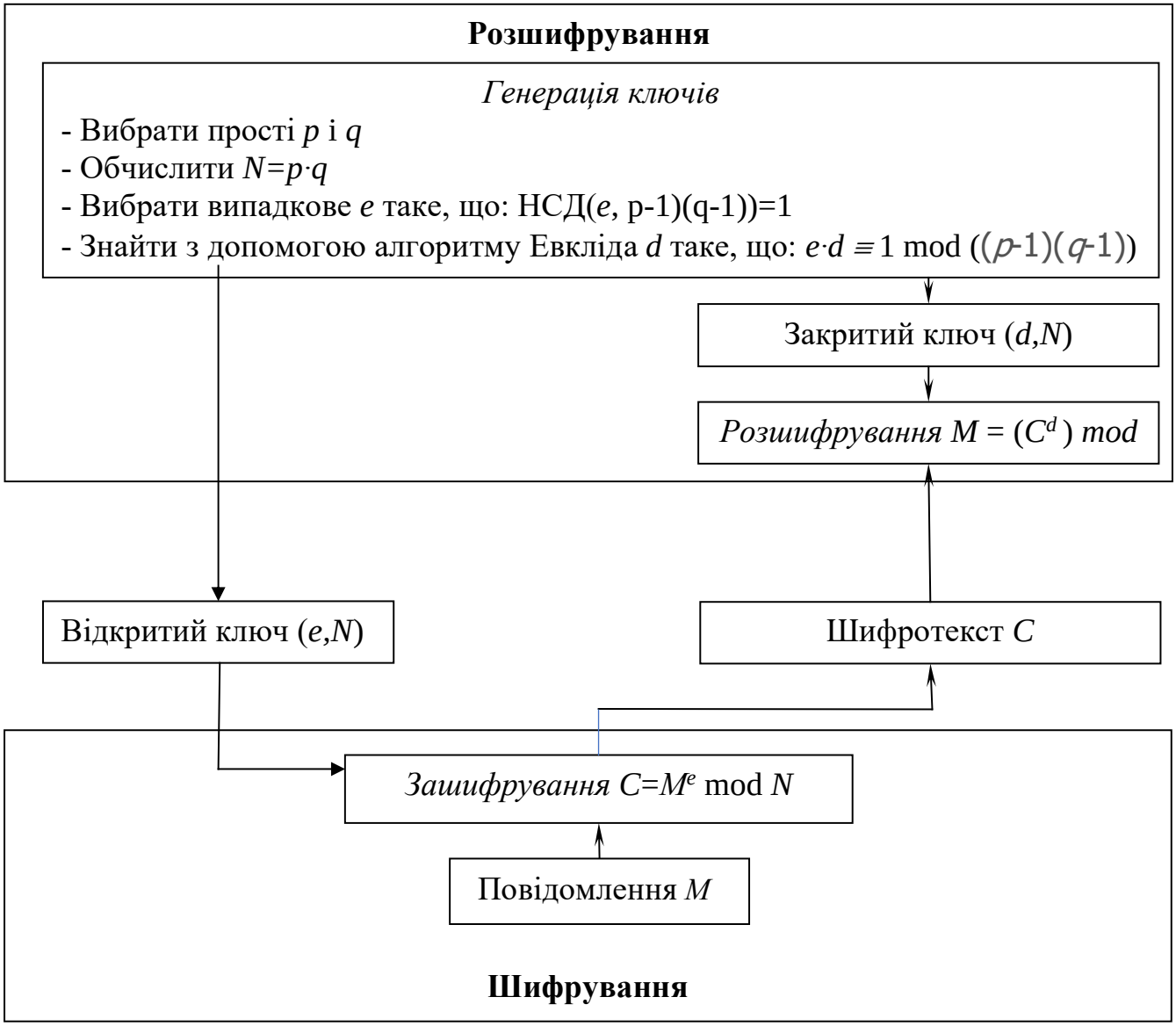


Рисунок 1.1 – Схема шифрування алгоритму RSA

Відмінності використання алгоритму RSA при створенні цифрового підпису полягають в наступному: для підпису застосовується закритий ключ, а відкритий – для його перевірки [57].

Очевидно, що основне завдання криптоаналітика при зломі цього шифру - дізнатися закритий ключ d. Для цього він повинен виконати ті ж дії, що і одержувач

при генерації ключа - вирішити в цілих числах рівняння $ed + y(p-1)(q-1) = 1$ щодо d і y. Тому безпека RSA алгоритму згідно [22], заснована на складності розкладання на множники (факторизації) великих чисел.

На даний час багатьма авторами проведені дослідження щодо можливих методів криптографічного аналізу RSA-алгоритму. Серед них, наприклад, роботи [37, 42 – 44, 47, 58-60] і багато інших. Ці питання постійно входять в число обговорюваних проблем на ряді авторитетних щорічних наукових конференцій, присвячених криптографії, наприклад, EUROCRYPT, ASIACRYPT і CRYPTO. У відповідності з перерахованими роботами на рис. 1.2 наведені деякі з найбільш відомих методів криптоаналізу стосовно RSA алгоритму.

Також широкого розповсюдження отримали так звані «не прямі» методи які, як правило, експлуатують неправомірне використання системи, неправильний вибір приватної експоненти d, або публічної експоненти e, послідовності у шифрованому тексті, що засновані на побічних каналах отримання інформації про криптосистеми (на основі енергоспоживання, «збою» у їх роботі та ін) [42], [61-70].

Однак Daniel Brown, Divesh Aggarwal і Ueli Maurer в роботах [44 - 46] показали, що більшість відомих прикладів компрометації RSA алгоритму відносяться до певних їх практичних реалізацій, проте в загальному випадку не є ефективнішими задач криптоаналізу, заснованих на математичних проблемах побудови RSA алгоритму, пов'язаних з виконанням факторизації багаторозрядних чисел, що становлять криптомодуль RSA алгоритму.

Слід зазначити, що на даний час так і не знайдено ефективного рішення обчислювальної задачі факторизації. Питання існування алгоритму факторизації з поліноміальною складністю на класичному комп'ютері є однією з відкритих проблем. Однак, найшвидший алгоритм факторизації довільних натуральних чисел, відомий на сьогоднішній день, має субекспоненціальноу оцінку часу. Це значить, що теоретична оцінка кроків до завершення його роботи значно більше ніж значення поліному $\ln n$. Функція складності такого алгоритму зображується як [71]:

$$L_n[\alpha, c] = \exp\left[(c + o(1))(\ln N)^{\alpha}(\ln \ln N)^{\alpha-1}\right] O\left(exp\left(C\sqrt{ln\, N\, lnln\, N}\right)\right) \quad (1.1)$$

Завдяки цьому прийнятний рівень безпеки у протоколі шифрування RSA досягається при використанні ключа (e, N) відносно не великого розміру, що дуже важливо для практичного використання. Прогрес в області комп'ютерних технологій та алгоритмів факторизації збільшує нижню границю розміру ключа, зокрема розмір модуля RSA шифрування за яким вважалося би на даний момент безпечним.

Згідно звіту про останній досягнутий рекорд факторизації [72] який був встановлений 12 грудня 2009, за допомогою алгоритму решета числового поля на прості множники було розкладено 768-бітне 232-значне число RSA-768. Загальний час, який був витрачений на виконання цієї роботи, складає два з половиною роки. При цьому розподілені обчислення на сотнях комп'ютерів вимагали більш 10^{20} операцій, що еквівалентно 2000 рокам обчислень на комп'ютері класу 2.2GHz AMD Opteron. Тим не менш автори дослідження справедливо оцінюють зусилля які були затрачені на факторизацію 768-бітного модуля як достатньо малі, щоб рекомендувати не використовувати у подальшому модулі такого розміру навіть для короткочасного захисту даних. Крім того, висувається припущення [73, 74], що факторизація 1024-бітного RSA модуля хоч і буде приблизно в тисячу більш складнішою задачею, але її рішення в рамках схожого проекту може отримане на протязі наступного десятиліття.

Слід зазначити що реалізація обчислювальної задачі факторизації повинна враховувати сучасні фактори, які дозволяють досягти значного приросту обчислювальної потужності доступних технічних засобів. Це дозволить зберегти адекватність процесу криптоаналізу. Що приводить до необхідності своєчасного перегляду вимог до апаратних реалізацій, в залежності від темпів росту основних показників обчислювальної техніки та появі принципово нових технічних рішень. Такі механізми повинні забезпечити своєчасність дослідження впливу нових апаратних (апаратно-програмних) засобів та чисельних методів на ефективність криптоаналізу.

1.3. Використання обчислювальних методів факторизації при криптоаналізі RSA-алгоритму.

Застосування нових або вдосконалених обчислювальних методів факторизації багаторозрядних чисел, що є одним з пріоритетних напрямків при вирішенні завдань криптоаналізу RSA алгоритму.

Серед всього розмаїття методів факторизації для виконання задач криптоаналізу виділяють чотири основних методи:

- *Метод Ферма* [75-77]. Метод застосовується для виявлення ключів, у яких множники p та q дуже близькі один до одного.
- *P-метод Полларда* [78]. Метод використовують до застосування більш сильних алгоритмів факторизації для того, щоб відокремити невеликі прості дільники ключа, який досліджується.
- *Метод Решешета числового поля* [79-81]. Метод є найбільш ефективним алгоритмом факторизації чисел довжиною понад 110 десяткових знаків.
- *Метод Квадратичного решета QS* [82-85] вважається другим за швидкістю після загального методу решета числового поля, і досі є найшвидшим для цілих чисел до 110 десяткових знаків і влаштований значно простіше, ніж загальний метод решета числового поля.

Питанням вдосконалення (зменшення обчислювальної складності) методів факторизації багаторозрядних чисел приділялася значна увага. Це класичні роботи Дж. Полларда (Pollard J.M.) [78], Х. Ленстри (Lenstra H.W.) [86], [87], К. Померанца (Pomerance C.) [88], Р. Леман (R. Lehman) [89] і багатьох інших вчених, загальний огляд методів факторизації виконали Василенко О.Н. [47] і Ш.Т. Ішмухаметов [90, 91]. Питанням проведення комп'ютерного обчислювального експерименту з факторизації багаторозрядних чисел, в тому числі і криптомодуля RSA алгоритму, присвячені, наприклад, роботи Я.І. Кінаха [92], С. Маітра (Maitra S.) [93], Kanabar K.A. [94] та інші [95-97].

Аналіз даних робіт показує, що серед методів факторизації багаторозрядних чисел на сьогоднішній день асимптотично найбільш швидкими є методи

квадратичного решета (QS – Quadratic Sieve) і решета числового поля (GNFS – General Number Field Sieve).

Квадратичне решето займає другий рядок у списку найбільш швидких алгоритмів факторизації, поступаючись тільки методом решета числового поля.

Відносна простота алгоритму сприяла виникненню багатьох його модифікацій при достатньо легкому його розпаралелюванню.

1.3.1. Метод Квадратичного решета.

У двадцятих роках XX століття Морис Крайчик (Maurice Kraitchik), узагальнюючи ідею Ферма, запропонував замість пар чисел, які задовільняють рівнянню

$$A^2 - B^2 = n \tag{1.2},$$

шукати пари чисел, які задовольняють рівнянню

$$A^2 \equiv B^2 \ (mod\ n) \tag{1.3}.$$

Крайчик помітив, що якщо розглянути поліном $q(x) = (m + x)^2 - n$; x = 1,2,3 ... то багато з чисел $q(x)$ розкладаються у твір невеликих простих чисел (тобто є гладкими).

Оберемо множину невеликих простих чисел та назвемо її факторною базою (**ФБ**) - B. Буд'яке ціле число, яке можна розкласти у твір множників з факторної бази, назвемо B-гладким відносно цієї факторної бази. Коли факторна база обрана, буд'яке B-гладке число можна представити вектором показників степенів $v = (r_1, r_2, \dots, r_k), k -$ розмір факторної бази.

Далі Крайчик помітив, що можна помножити деякі гладкі числа так, щоб отримати повні квадрати.

При множенні чисел, їх вектора паказників степенів додаються. Щоб твір B-гладких був повним квадратом, необхідно підібрати таку комбінацію множників, щоб сума векторів показників мала тільки парні координати.

Замість цілочисельних координат векторів степенів та іх сум достатньо розглядати їх залишки по модулю 2, тобто виконувати всі обрахунки у полі $F_2 = \{0,1\}$, тоді добуток елементів M є повним квадратом тоді і тільки тоді коли вектор

суми по модулю 2 всіх векторів – показників є нульовим вектором. Множина всіляких векторів розмірності k над полем $F_2 = \{0,1\}$ утворює лінійний простір L_k розмірності k, тому будякий простір векторів, яке містить більше ніж k елементів, є лінійно залежним, та існує нетривіальна лінійна комбінація таких векторів, яка дорівнює нульовому вектору. Коєфіцієнти цієї лінійної комбінації можна знайти, вирішуючи систему лінійних рівнянь, яка була складена з коефіцієнтів степенів взятих по модулю 2.

Оскільки коефіцієнти цієї лінійної комбінації дорівнюють або 0, або 1, то вона являє собою просто суму деякої підмножини векторів. Звідси виходить той факт, що для знаходження нетривіальної підмножини B-гладких чисел, добуток яких є повним квадратом, достатньо мати k+1 B-гладке число, де k – розмір факторної бази.

B-гладкі грають важливу в оцінці часу роботи алгоритмів факторизації. Існують оцінки розподілення гладких чисел [98-100] за допомогою функції Дікмана-Де Брюїна $\rho(u)$, яка має наступний графік:

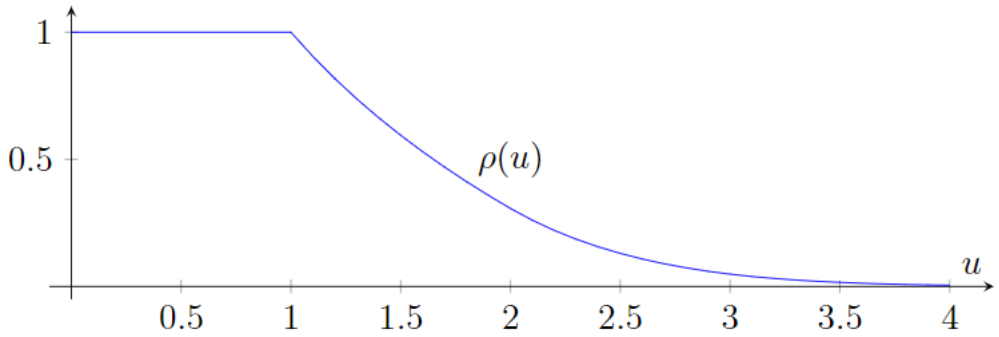


Рисунок 1.2 – Графік функції Дікмана-Де Брюїна.

З визначення цієї функції, ймовірність обрати $x^{1/u}$- гладке число серед рівномірно розподілених чисел, менших за x, дорівнює $\rho(u)$. Таким чином, так як, наприклад $\rho(2) = 0{,}3$ можна припустити що серед деякого випадкового набору цілих чисел приблизно 30% будуть володіти властивістю, що максимальний простий дільник не буде більший за кореня цього числа. Це дає деяке достатньо надійне представлення про ступінь поширеності гладких чисел.

У 1981 році Д. Діксон [101] запропонував метод фаткоризації, який використовує ідеї Крайчика. Однак, хоча цей метод забезпечував значну перевагу по

відношенню до існуючих методів, він працював повільно, бо пробне ділення на елементи факторної бази займало занадто багато часу.

У 1982 році Карл Померанс запропонував [82, 88, 102] нові ідею пошуку B-гладких чисел, використовуючи метод, який схожий на решето Ератосвена [103]. Ідея Померанса полягала у тому, що якщо для простого числа $p \in B$ знайдено аргумент x такий, що $q(x) \equiv 0(mod\ p)$, то на p ділитимуться усі елементи $q(x)$, де y відрізняється від x на аргумент кратний p, тобто $y = x + k \cdot p, k \in Z$. Тому, якщо для даного p знайдено корінь x рівняння $q(x) \equiv 0(mod\ p)$, то для всіх $y, y \equiv x\ (mod\ p)$ буде також виконана умова $q(y) \equiv 0(mod\ p)$.

Ряд вдосконалень цього методу був запропонований згодом у роботах [104-111]. Розглянемо кожен етам алгоритму окремо.

1.3.1.1. Ініціалізація.

Оберемо границі P та A порядку величини $e^{\sqrt{\log n \log\log n}}$, число A повинно бути більше за P, але не перевищувати відносно малої його степені, наприклад $P < A < P^2$. Порядок зростання функції $e^{\sqrt{\log n \log\log n}}$ (вона позначається як $L(n)$) знаходиться у проміжку між многочленами від $\log n$ та від n. Якщо $n \approx 10^6$, то $L(n) \approx 400$.

1.3.1.2. Побудова факторної бази.

Складаємо множину з усіх простих чисел, менших за верхню границю P. Для кожного простого числа $p \leq P$ перевіряємо умову $\frac{n}{p} = 1$ (символ Лежандра, його рівність одиниці означає, що n квадратичний залишок за модулем p, тобто n не ділиться на p та існує таке ціле x, що $x^2 \equiv n(mod\ p)$) та при його не виконанні видаляємо p з факторної бази.

1.3.1.3. Просіювання.

На практиці зазвичай використовується процес який називається просіювання. Якщо $Q(x)$ - многочлен $Q(x) = x^2 - n$ у нас є

$$Q(x) = x^2 - n \tag{1.4}$$
$$Q(x + kp) = (x + kp)^2 - n$$
$$Q(x + kp) = x^2 + 2xkp + (kp)^2 - n$$
$$Q(x + kp) = Q(x) + 2xkp + (kp)^2 \equiv Q(x)\ (mod\ p)$$

Таким чином, рішення $Q(x) \equiv 0\ (mod\ p)$ для x породжує цілу послідовність чисел y, для якої $y = Q(x)$, всі з яких діляться на p.

Тоді задача зводиться до знаходження квадратичного залишку по модулю простого числа, для якого існують ефективні алгоритми, такі як алгоритм Шенкса-Тонеллі.

Сенс процедури просіювання полягає в економії великої кількості операцій ділення великих чисел. Тобто замість того, щоб одразу для кожного $x \in [-M; M]$ намагатися розкласти $Q(x)$ за факторною базою, ми попередньо за допомогою простих операцій додавання та віднімання суттєво скорочуємо множину тих x, для яких ми далі будемо факторизувати числа $Q(x)$ пробним діленням. Ця економія дає на практиці дає дуже суттєвий ефект, із-за чого метод квадратичного решета перевершив усі попередні алгоритми факторизації.

Значення $x_i \in \mathrm{Z}$ для яких $Q(x_i)$ є гладкими, визначаються наступним чином.

Для кожного простого числа p з факторної бази знаходимо рішення $r_1^{(p)}$ та $r_2^{(p)}$ рівняння $Q(x) \equiv 0(mod\ p)$ (наприклад алгоритмом Шенкса-Тонеллі).

Наступним кроком ми змінюємо $x \in Z$ в достатньо великому проміжку $[-M; M]$, $M \in N$, заводимо масив $A[]$, який пронумерований цими значеннями x, та y елемент масиву з номером x розміщуємо достатньо грубо обчислені значення $\log |Q(x)|$.

Далі для кожного простого p з факторної бази S ми виконуємо процедуру просіювання: ми обчислюємо достатньо грубо значення $\log p$, та віднімаємо це значення від кожного елементу масиву, для якого $Q(x) = 0\ mod\ p$ ... тобто $A[kp + r_1^{(p)}]$ і $A[kp + r_2^{(p)}]$.

Сенс такого вирахування полягає в тому, що для елементів x в цих прогресіях значення $Q(x)$ буде ділитися на p, але ділення $Q(x)$ на p ми поки що замінюємо

відніманням $\log|Q(x)| - \log p$. Після закінчення процедури просіювання у елементі масиву з номером x буде зберігатися значення

$$\log|Q(x)| - \sum_{p\in S, p|Q(x)} \log p\,. \qquad (1.5)$$

Насправді необхідно проводити просіювання також і по степеням простих чисел p^l для декількох невеликих l. Тобто треба знаходити рішення $r_1^{(p,l)}$, $r_2^{(p,l)}$ рівняння $Q(x) \equiv 0\ (mod\ p^l)$, та потім у ході просіювання з елементів масиву з номерами $x \equiv r_1^{(p,l)}(mod\ p^l)$ та $x \equiv r_2^{(p,l)}(mod\ p^l)$ проводити віднімання $\log p$. Тоді після просіювання в елементі масиву з номером x буде знаходитись значення

$$\log|Q(x)| - \sum_{p\in S, p^l|Q(x)} l \cdot \log p\,. \qquad (1.6)$$

Після завершення просіювання ми йдемо по масиву та обираємо ті номера x, для яких значення елементів масиву маленькі за абсолютною величиною. Для цих номерів x значення $Q(x)$ скоріш за все розкладеться у обраній факторній базі.

Інформація про те, які саме прості ділять $y(x)$, була втрачена, але $y(x)$ має лише малі дільники, і є багато хороших алгоритмів для факторизації числа, які, як відомо, мають лише малі дільники, такі як «trial division by small primes», SQUFOF, Pollard rho, та ECM, які зазвичай використовуються в певній комбінації.

Факторизуємо число $Q(x)$ пробними діленнями та залишаємо ті x, для яких $A_i = Q(x_i)$ повністю розкладеться за нашою факторною базою.

Для багатьох значень $y(x)$ просіювання працює, але процес факторизації не є цілком надійним, часто процес має похибку приблизно для 5% вхідних даних, що вимагає невеликої кількості додаткового просіювання.

Зауваження. Оскільки кожне друге число ціле число ділиться на 2, кожне третє на 3 і т.д., то можна не проводити просіювання за степенями маленьких простих, тобто наприклад, за $p^k \leq 100$. Замість цього, після закінчення процедури просіювання окрім маленьких елементів масиву з результатами треба брати елементи які не є більшими за абсолютною величиною деякої невеликої границі, яка враховує значення $\log 2\,, \log 3\,, \log 5$, і т.д. які не відняли з $\log|Q(x)|$.

Етап просіювання зводиться до наступних кроків:

1. Якщо p – непарне, та n – квадратичний лишок по модулю p, вирішуємо рівняння $a^2 \equiv n(mod\ p^\beta)$ для $\beta = 1,2 \ldots$. Беремо β у порядку зростання, поки не з'ясується, що рівняння не має рішень а, які можна порівняти по модулю p^β з якимось із чисел в області $\sqrt{n} + 1 \le t \le \sqrt{n} + A$. Нехай β – найбільше з таких чисел, а a_1 та a – два рішення $a^2 \equiv n(mod\ p^\beta)$ та $a_1 \equiv -a_2(mod\ p^\beta)$ (не потрібно щоб a_1 та a_2 належали відрізку $([\sqrt{n}] + 1, [\sqrt{n}] + A)$)

2. При тому ж p розглядаємо значення $b = a^2 - n$, які були отримані у пункті 2. У стовпчику для p ставиться 1 навпроти усіх $a^2 - n$, для яких a відрізняється від a_1 на деяке кратне p, після чого замінюється 1 на 2 для всіх $a^2 - n$, для яких a відрізняється від a_1 на деяке кратне p^2, і т.д. до p^β. Далі проводяться всі ж ті самі операції, тільки для a_2 замість a_1. Найбільшим числом, яке може з'явитися у цій таблиці буде β.

3. Коли додається чергова одиниця до числа у стовбці у пункті 5, відповідне число $a^2 - n$ ділиться на p та отриманий результат зберігається.

4. У стовпчику під $p = 2$ при $n \neq 1\ (mod\ 8)$, ставимо 1 навпроти $a^2 - n$, з непарним a, та відповідне $a^2 - n$ ділиться на 2. При $n \equiv 1\ (mod\ 8)$ вирішується рівняння $a^2 = n\ (mod\ 2^\beta)$ та рішення продовжується.

5. Коли всі дії, які описані вище будуть проведені для всіх простих чисел менших за P, слід відкинути усі числа $a^2 - n$ які не дорівнюють одиниці. Після чого отримаємо таблицю у якій у a_i стовпці зберігаються усі такі значення a з інтервалу $[\sqrt{n}] + 1, [\sqrt{n}] + A$, що $a^2 - n$ є B-гладке число, а решта стовпчиків будуть відповідати тим значенням $p \le P$, для яких n – квадратичний лишок.

Таблиця 1.1

Пробні значення етапу просіювання.

a	$b = a^2 - n$	p_1	p_2	…	t_P
a_1	$(a_1)^2 - n$	t_{11}	t_{12}	…	t_{1P}
a_2	$(a_2)^2 - n$	t_{21}	t_{22}	…	t_{2P}
…	…	…	…	…	…
a_A	$(a_A)^2 - n$	t_{A1}	t_{A2}	…	t_{AP}

1.3.1.4. Рішення систем лінійних рівнянь.

За результатами попереднього кроку була сформована невизначена система лінійних рівнянь з нульовою правою чистиною та коефіцієнтами з поля $F_2 = \{1, 2\}$, число рівнянь m якої більш ніж кількість невідомих k.

Матриця отримана з рядків, кожен з яких є вектором розкладання кожного гладкого числа за факторною базою B (B-гладкого) по модулю 2. Система має як мінімум m-k нетривіальними векторами рішення. Особливістю системи є її сильна розрідженість, особливо в правій її частині, яка відповідає степеням старших простих чисел з факторної бази. Така система може бути вирішена з використанням звичайного метода Гауса. Складність полягає в тому, що розмірність системи дуже велика, і може досягати $10^6 \times 10^6$ та більше. Тому для рішення відповідної системи ЛР використовується спеціальні методи рішення розріджених систем ЛР, які будуть розглянуті у наступному параграфі.

1.3.1.5. Алгоритму квадратичного решета.

1. Визначимо розмір факторної бази та інтервалу просіювання.
2. Для $a = \lceil\sqrt{n}\rceil + 1, \lceil\sqrt{n}\rceil + 2, \dots, \lceil\sqrt{n}\rceil + A$, виписуємо по порядку цілі числа $b = a^2 - n$.
3. Формування факторної бази FB.
4. Просіювання варіантів $Q(x)$ через факторну базу, та формування списку B-гладких чисел.
5. Рішення СЛАУ для визначення набору векторів, що утворюють нульовий вектор.
6. У результаті отримаємо два числа, які є квадратами за модулем n, $a^2 = b^2 \ mod \ n$.
7. Тепер ми отримали бажане рівняння $(a + b)(a - b) \equiv 0(mod\ n)$. Обчислюємо НСД($n$, a+b) або НСД(n, a-b). Якщо отримане значення дорівнює n або 1 – ми отримали хибне рішення, треба продовжити пошук нових B-гладких чисел для формування нового нульового вектору. В усіх інших випадках ми отримали один з дільників n.

Існують підходи в яких для збільшення ефективності просіювання використовуються попередні оцінки тому потребує дослідження ефективності такого підходу.

1.3.1.6. Приклад.

Цей приклад описує роботу стандартного алгоритму квадратичного решета без застосування прискорення за логарифмами.

Нехай *N*=15347 число яке ми намагаємося факторизувати. Тоді $\sqrt{N} = 124$.

Сформуємо многочлен $y(x) - y(x) = (x + 124)^2 - 15347$;

Оберемо границі *P*=4 та *A*=100.

Наша факторна база складається з чотирьох простих чисел {2,17,23,29}, для яких 15347 – квадратичний залишок. Ці прості числа є основою для просіювання.

Тепер почнемо процес просіювання V_x для $Y(X) = (X + \lceil\sqrt{N}\rceil)^2 - N = (X + 124)^2 - 15347$ та кожного простого числа з факторної бази та значень X від 0 до 100:

$V = [Y(0)\ Y(1)Y(2)Y(3)Y(4)Y(5)\ \ldots\ Y(99)] = [29\ 278\ 529\ 782\ 1037\ 1294\ \ldots 34382]$

Наступний крок саме просіювання. Для кожного p з факторної бази, яка була сформована {2,17,23,29} вирішується рівняння

$$Y(X) = (X + \lceil\sqrt{N}\rceil)^2 - N \equiv 0\ (mod\ p)$$

для знаходження значень $Y(X)$ з масиву V які діляться націло на p.

Для p=2 вирішується рівняння $(X + 124)^2 - 15347 \equiv 0\ (mod\ 2)$, рішенням якого є $X \equiv \sqrt{15347} - 124 \equiv 1\ (mod\ 2)$.

Отже починаючи з значення *X*=1 та збільшуючи це значення на 2, кожен варіант $Y(X)$ буде ділитися на 2. Таке ділення приводить масив *V* до наступного вигляду:

$$V = [29\ 139\ 529\ 391\ 1037\ 647\ \ldots 17191]$$

Аналогічно для наступних простих чисел з факторної бази {17,23,29} вирішується рівняння $X \equiv \sqrt{15347} - 124\ (mod\ p)$. Зауважимо, що для простих p>2, існує два модульних рівняння, адже існують два квадратичних залишку для цих чисел

$$X \equiv \sqrt{15347} - 124 \equiv 8 - 124 \equiv 3\ (mod\ 17)$$
$$\equiv 9 - 124 \equiv 3\ (mod\ 17)$$
$$X \equiv \sqrt{15347} - 124 \equiv 11 - 124 \equiv 2(mod\ 23)$$
$$\equiv 12 - 124 \equiv 3\ (mod\ 23)$$
$$X \equiv \sqrt{15347} - 124 \equiv 8 - 124 \equiv 0(mod\ 23)$$
$$\equiv 21 - 124 \equiv 13\ (mod\ 23)$$

Кожне рівняння $X \equiv a\ (mod\ p)$ призводить до знаходження V_x яку ділиться на p для $x = a$ та кожне +p наступне. Розділимо *V* на *p* для наступних значень *x*: *a*, *a*+*p*, *a*+2*p*, *a*+3*p*; Таким чином знайдено гладкі числа для простих чисел з факторної бази першої степені.

$$V = [1\ \ 139\ \ 23\ \ 1\ \ 61\ \ 647\ \ \ldots\ 17191]$$

Кожне значення масиву *V* яке дорівнює одиниці є гладким числом. Оскільки V_0, V_3 та V_{71}дорівнюють одиниці, які відповідають наступним значенням *X*:

Таблиця 1.2

Пробні значення.

X+124	*Y*	множники
124	29	$2^0 17^0 23^0 29^1$
127	782	$2^1 17^1 23^1 29^0$
195	22678	$2^1 17^1 23^1 29^1$

Сформуємо матрицю А із значень степенів множників *Y*. Перший стовпчик визначає знаки відповідних значень *Y*(*x*).

$$S\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \equiv [0\ 0\ 0\ 0\ 0\ ]\ (mod\ 2)$$

Сума цих трьох рядків утворюють нульовий вектор за модулем 2. Таким чином добуток цих трьох варіантів *Y*(*x*) утворює квадрат за модулем *n*.

$$29 \cdot 782 \cdot 22678 = 22678^2$$

та

$$124^2 \cdot 127^2 \cdot 195^2 = 3070860^2$$

Таким чином отримано порівняння:

$$22678^2 \equiv 3070860^2 \ (mod\ 15347)$$

Це дає не тривіальний дільник GCD(3070860 - 22678, 15347) = 103, відповідно другий дільник дорівнює 149.

1.3.1.7. Оцінка складності.

Якщо кількість простих чисел у факторній базі (розмір факторної бази) дуже малий, то розмір вектора степенів буде малим, це значно зменшує кількість операцій. Проблема в тому, щоб знайти такі B-гладкі числа, які б входили в цю факторну базу. Чим менше факторна база, тим суттєво меншою є кількість B-гладких чисел, тобто необхідно значно збільшувати інтервал просіювання. Якщо створити велику за розміром факторну базу, то перед нами б постала проблема вирішення системи лінійних алгебраїчних рівнянь (СЛАУ) спеціального виду з матрицею великої розмірності, що потребує великої кількості пам'яті та ресурсів.

Оптимальне значення розміру факторної бази пропонується в роботі [112], яке обчислюється за формулою:

$$A = L^a = \ (e^{\sqrt{\ln(n)\ln(\ln(n))}})^{\sqrt{2}/4} = L(n)^{\sqrt{2}/4} = L^{\sqrt{2}/4}. \qquad (1.7)$$

Ця формула не дає остаточної відповіді. Для кожного випадку найкращий розмір факторної бази є індивідуальним і може відрізнятися від значення отриманого за формулою.

Наприклад, коли факторизували RSA-129 в 1994 році, була задіяна мережа 1600 комп'ютерів, що пропрацювала 220 днів, була сформована матриця лінійних рівнянь з 524338 невідомими, яка вирішувалася на суперкомп'ютері протягом двох днів [113]

Цей факт був сприйнятий як великий сюрприз, адже вважалося, що число RSA-129 дуже важко факторизувати.

Інтервал просіювання повинен бути таким щоб кількість B-гладких була більше, за кількість елементів у кожному векторі. Але цієї умови не достатньо. Ми можемо скласти матрицю де кількість векторів більше за кількість елементів у кожному векторі, та отримати хибне рішення. В такому випадку нам знадобиться

розширити інтервал просіювання, для отримання додаткових векторів. Для загального випадку (згідно з [112]), отримати розмір інтервалу просіювання можна за формулою:

$$M_{max} = L^b = (e^{\sqrt{\ln(n)\ln(\ln(n))}})^{3\sqrt{2}/4} = L(n)^{3\sqrt{2}/4} = L^{3\sqrt{2}/4}. \quad (1.8)$$

При обмеженнях на розмір факторної бази можливі випадки, коли неможливо отримати достатню кількість B-гладких чисел. Тоді збільшують границю гладкості та число елементів факторної бази і пошук B-гладких повторюється.

Згідно з [125] час просіювання приблизно втричі більше за час вирішення матриці. Маючи евристичну оцінку розміру факторної бази та інтервалу просіювання маємо асимптотичний час роботи алгоритму QS:

$$T_{QS}(n) = L_n\left[\frac{1}{2}, c\right] = e^{(c+o(1))\sqrt{\ln n \ln\ln n}},\ c + o(1) > 1 \quad (1.9)$$

Метод використовує дуже великий розмір пам'яті (також порядку $O\left(e^{C\sqrt{\ln(n)\ln(\ln(n))}}\right)$).

1.3.2. Модифікації метода квадратичного решета.

Як вже було описано раніше вибір розміру факторної бази та інтервалу просіювання є визначною для алгоритму QS. Це місце де метод стає евристичним.

Метод проріджения пробних значень *x*, який був застосований для методу Ферма[8, 10] не є ефективним для методу квадратичного решета, адже серед багатьох значень пробних x які були відсіяні також були відсіяні і частина B-гладких.

Розглянемо на прикладі, оберемо *n*=2041. Найближчим до n числом, яке є повним квадратом є *m*=45, m^2=2025, розглянемо послідовність пар чисел $\{x, y(x)\}$, де $y(x) = (m + x)^2 - n$, яка зображена у таблиці 1.3.

Таблиця 1.3

Пробні значення.

x	-2	-1	0	1	2	3	4	5	6
y	-192	-105	-16	75	168	263	360	459	560

Розглянемо B=16 як базу для просіювання. Залишки повних квадратів для бази 16 приведені в таблиці 1.4.

Таблиця 1.4.

Значення вектора степенів.

Показники квадратів для модуля 16			
0	1	4	9

Серед отриманих значень пробних *y*, значення $y = \{-192, -16{,}75\}$ є B-гладкими та утворюють нульовий вектор. Але із застосуванням просіювання за модулем 16, отримаємо результати зображені в таблиці 1.5.

Таблиця 1.5.

Пробні значення.

x	y	Показники степенів елементів факторної бази				mod 16
		-1	2	3	5	
-2	-192	-	6	1	0	0
0	-16	-	4	0	0	0
1	75	-	0	1	2	11

Значення *y*=75 має остачу 11 за модулем 16. Тобто не є квадратичним залишком. Отже ми відсіємо значення *y*=75, і втратимо варіант рішення.

Втрата B-гладких для методу квадратичного решета при оптимально обраних розмірах факторної бази та інтервалу просіювання призводить до необхідності розширення інтервалу просіювання.

При аналізі оцінки обчислювальної складності методу QS згідно з [114] використовується середнє значення пробних *x* з інтервалу просіювання для отримання одного B-гладкого числа. Проте, як було описано вище B-гладкі числа розміщуються на інтервалі просіювання в середньому згідно деякого закону, загальною характеристикою якого є те, що зі збільшенням модуля *x* росте кількість *x*, які слід просіяти.

Тому відсіювання B-гладких, при малих значеннях пробних *x* призводить до значного розширення інтервалу просіювання, що в свою чергу призводить до збільшення часу факторизації.

Основною проблемою для методу квадратичного решета - є пошук достатньої кількості B-гладких чисел. Тому подальші дослідження модифікацій методу

квадратичного решета направлені на одночасне збільшення кількості B-гладких при сталому значенні розміру факторної бази та зменшенню часу обчислень.

Ряд удосконалень цього методу був запропонований у роботах [82, 104 - 114]. Їх можна умовно поділити на два напрямки: 1. Прискорення процесу просіювання (збільшення числа B-гладких, при сталому розмірі факторної бази), 2. прискорення вирішення матриці.

1.3.2.1. Прискорення процесу просіювання.

Беручи до уваги той факт що один поліном як правило дає недостатню кількість B-гладких чисел, тому одним з методів прискорення процесу просіювання є застосування множини поліномів.

Поліноми які будуть використовуватися повинні бути спеціальної форми:

$$y(x) = (Ax + B)^2 - n \quad A, B \in Z \tag{1.10}$$

де A, B – спеціально підібрані цілі числа, x вибирається з інтервалу просіювання [$-L$, L].

Такий підхід (називають MPQS методом Множинного Поліноміального Квадратичного Решета) [93] ідеально підходить для разпаралелювання, бо кожен процесор може працювати з конкретним поліномом. І йому нема потреби зв'язуватися з центральним процесором до кінця процесу просіювання.

Питання полягає в тому, наскільки великим потрібно брати L довжину інтервалу просіювання для кожного полінома. При великому значенні L на кожному поліномі буде просіватися багато чисел великого розміру, серед яких гладких чисел буде порівняно мало що небажано. Якщо L взяти малим, то кількість варіантів (x, y) для просіювання також будуть малим, в такому випадку потрібно генерувати велику кількість поліномів, підбираючи коефіцієнти a, b. Але для кожного нового полінома потрібно заново формувати факторну базу, тобто заново виконувати етап ініціалізації, що також є витратною процедурою. Таким чином якщо L занадто мало, то велика частина часу буде витрачена не на просіювання, а на ініціалізацію поліномів. Також при збільшенні X на одиницю значення Y у A^2 разів, що значно пришвидшує віддалення від початкового значення X_0 у порівнянні з стандартним

методом квадратичного решета. У зв'язку з чим різко зростає складність пошуку B-гладких чисел.

Померанс [114, 106] вказує, що трудомісткість від заміни многочленів складає 25-30% від усього алгоритму. При зміні многочлена необхідно виконати процес ініціалізації, який включає в себе формування нової факторної бази.

Коен [124] також не рекомендує занадто часто міняти многочлени.

Тому наступні дослідження направлені на створення алгоритму який би дозволяв змінювати поліноми за меншу кількість часу, та більш ефективніше використовував більш щільне знаходження B-гладких чисел біля $\sqrt{N}$.

Одне з важливих покращень у методі квадратичного решета є ідея варіація великого множника - Large Prime Variations (LVP). Ця версія алгоритму була представлена Ленстрой Манассе (Lenstra, Manasse) та декількома іншими вченими в 1993-1994 роках та використовувалася для факторизації числа RSA-129 [72, 112, 115-117].

Ідея полягає в наступному. У процесі просіювання накопичується багато значень полінома q(x), $q(x), x \in [-L; L]$, які представлені у вигляді $q(x)P_x \cdot C_x$ де множник C_x є гладким числом а P_x є дільником який виходить за границю гладкості B але менший за B^2 такий, що P_x не містить дільників з факторної бази. Множник P_x є простим числом, інакше можливо було б розкласти у добуток двох множників менших за B. Такі значення називають напівгладкими. При появі напівгладких чисел з однаковими P_x, їх можна використовувати разом для знаходження нульового вектору.

Відмітимо що додатковий час яке необхідне для сортування напівгладких чисел є мізерно малим на тлі повного часу просіювання, а виграш є значним, оскільки не змінюючи розміру факторної бази ми можемо отримувати більше число гладких чисел. Таким чином, використовуючи варіацію великих множників, можна щільність B-гладких чисел на інтервалі просіювання що дозволяє зменшити кордон гладкості B для факторної бази і зменшити всі основні витрати з пошуку рішення.

Подальшим розвитком ідеї LPV є використання напівгладких з двома великими множниками замість одного. Хоча така реалізація вимагає набагато більше додаткової роботи, але забезпечує більше можливостей для вибору напівгладких чисел.

Подальший розвиток ідеї великих множників до трьох або більше множників викликає додаткові складності і вважається неефективним.

Окрім перевірку на те що P_x є простим інших додаткових перевірок в літературі не було виявлено. Тому подальші дослідження можуть бути проведені у напрямку додаткового аналізу P_x.

1.3.2.2. Прискорення вирішення матриці.

Відмічається, що крок розв'язання матриці не може реалізована у паралельному режимі, тому був прийняті кроки для його прискорення [84].

У [118] описано, що кількість одиниць у матриці степенів значно менша за кількість нулів. Для великих чисел розміром 10^{100} та більше відношення кількості нулів до кількості одиниць тільки зростає. Більша частина пам'яті виділена для зберігання матриці використовується для зберігання нулів. Тому замість зберігання двомірної матриці запропоновано зберігати тільки позиції одиниць.

У всіх наведених публікаціях вважалося, що етап вирішення матриці потребує обов'язкового знаходження кількості B-гладких чисел, не меншої за $L^a + 2$.

Після проходження етапу просіювання було отримано набір B-гладких чисел. Тепер необхідно знайти підмножину B-гладких чисел таку щоб добуток її членів був повним квадратом. Щоб зробити це алгоритм QS трансформує проблему множення в проблему додавання по модулю два. Спочатку знаходиться вектори степенів для кожного B-гладкого числа. Після чого необхідно знайти такі вектори, які при сумі по модулю два створювали нульовий вектор. Це класична задача пошуку нульового вектору для матриці, в якій вектори степенів B-гладких чисел виступають рядками чи стовпчиками (в залежності від алгоритму).

Тому буде сформована матриця розміром $|F| \times |R|$, де $|F|$ розмір факторної бази та $|R|$ кількість знайдених B-гладких. Серед багатьох алгоритмів для

знаходження нульового вектора виділяють, для матриць малого розміру, метод Гауса. Для матриць розміром більше ніж 100000×100000 необхідно використовувати більш ефективний алгоритм, такий як метод Ланцоша. Цей алгоритм був розроблений спеціально для пошуку нульового вектору у великих матрицях. Для цього алгоритму є спеціальна модифікація по роботі з матрицею за модулем 2.

Проблема у блочному методі Ланцоша полягає в тому, що він здатний вирішувати лише симетричні матриці.

Як вже говорилося розмір факторної бази L^a – це один з ключових параметрів, що визначають ефективність алгоритму просіювання та розмір матриці. Пошук нульового вектора у матриці починається коли знайдено B-гладких чисел у кількості принаймні $L^a + 2$.

Проте існують випадки, коли вирішення задачі факторизації можливе при значно меншому числі B-гладких чисел. Умова накопичення B-гладких у кількості $L^a + 2$ поставлена для гарантованого отримання нульового вектору. Але нульовий вектор можливо отримати маючи меншу кількість B-гладких. Зменшення значення $L^a + 2$ призведе до зменшення інтервалу просіювання. Подальші дослідження будуть направлені на аналіз можливості отримання нульового вектора при наявності B-гладких у кількості менших за $L^a + 2$.

Слід зауважити, що метод Квадратичного решета застосовується не у чистому вигляді, а з рядом модифікацій, що дозволяють у деяких випадках факторизувати число розміром 1024 біт швидше, ніж метод Решета числового поля.

Сучасні реалізації метода Квадратичного решета передбачають виконання арифметичних операцій з багаторозрядними числами. Цей аспект вимагає застосування спеціальних типів даних з підтримкою багаторозрядних чисел, що в свою чергу ускладнює використання сучасних засобів високопродуктивних розподілених систем обчислень, наприклад, графічних карт, де суттєвим є обмеження на типи даних [106], [111].

1.4. Аспекти реалізації методів факторизації.

Одним з основних способів підвищення швидкості обчислень є використання гетерогенної архітектури паралельних обчислень, при цьому існують різні способи і засоби їх організації [119].

1.4.1. Вимоги до платформи.

Ідеальна платформа має ряд критеріїв, які необхідно виконати. Ці критерії ідентифіковані при аналізі структури алгоритму, що реалізується. Для алгоритму квадратичного решета, вони наступні:

Можливість обробки великих цілих чисел: Числа, що розглядаються, перевищують розмір 64-розрядного цілого числа, що є межею більшості платформ. Тому мову програмування має мати відповідний спосіб представити ці цифри, а апаратні засоби - швидкий спосіб роботи з ними.

Різноманітна швидка пам'ять: обидва етапи просіювання та лінійної алгебри метода квадратичного решета залежать значною мірою від доступної пам'яті, але у різному сенсі.

Лінійна алгебра вимагає зберігання великих матриць. Чим більше число факторизується тим більшого розміру стає матриця, тим більша кількість пам'яті необхідно. Наприклад при факторизації числа RSA-129 методом квадратичного решета розмір матриці становив 524338x524338.

Процес просіювання вимагає швидких операцій запису та зчитування у пам'ять. Чим ближче пам'ять до блоку обробки математики, чим менше тактів процесора необхідно для доступу до комірок пам'яті, тим нижче затримка, і тим швидше виконуються операції зчитування та запису у пам'ять. Якщо ієрархія пам'яті із застосуванням кешу даних на чіпі та механізмом прогнозування, це було б ідеально, оскільки це може допомогти приховати затримку основної пам'яті. Хоча такий механізм не підходить для запису даних на пам’ять яка розташована на мікросхемі, це може бути корисним для отримання даних, наприклад, від факторної бази, тому що цілком імовірно, що розмір факторної бази не дозволить повне зберігання факторної бази у швидкій пам’яті що розташована на чіпі. Механізми прогнозування

спроможні сканувати скомпільований код, який чекає на виконання та попередньо завантажувати сторінки інструкцій та даних в кеш на основі можливих шаблонів виконання.

Швидкі модулі обробки математики: Цілком очевидно, що зменшення вузького місця на операціях додавання, віднімання, ділення, множення зменшить час, витрачений на арифметичні обчислення, і тим самим дозволяє програмі працювати швидше.

Паралелізація: вже визначено, що багато областей алгоритму квадратичного решета з легкістю реалізовані у паралельному режимі. Якщо вони доступні і правильно використовуються, кожен блок паралельної обробки має свою власну кеш-пам'ять, що приховує затримку при звертанні до пам'яті, а також збільшення числа операцій в секунду, і тому в обох випадках прискорити обчислення.

Вартість і доступність: повинно бути можливо придбати апаратний пристрій і реалізувати програму за розумну ціну.

Для реалізації факторизації великих чисел використовувалися різні комп’ютерні системи. Зробимо невеликий огляд найбільш успішних реалізацій.

1.4.2. CPU і розподілені обчислення.

З усіх можливих платформ для реалізації, CPU може розглядатися як найчастіше використаний метод для комплексного обчислення. Щоб бути більш конкретним, CPU розглядається як основний процесор до комп'ютерної системи. Коли ми порівнюємо CPU з переліком вище, майже всі критерії виконуються, за винятком:

Перше - швидкість обробки математичних операцій. CPU є процесором загального призначення, і тому він розроблений з метою, щоб підтримувати якомога більшу функціональність, яку потребують сучасні системи, а не чисто математичні обчислення.

Друге - це паралелізація. Хоча це правда, що нинішні покоління CPU мають кілька ядер, розробники CPU на сьогоднішній день зіткнулися з такими

проблемами як охолодження системи та іншими. Тому ріст ядер для CPU значно зменшився в останні роки.

Наступний логічний крок (коли обчислення вичерпують ресурс одного комп’ютера) - це об'єднати багато з систем. Це зазвичай досягається за допомогою мережі і називається розподіленим обчисленням, оскільки обчислення розподіляється між кожним комп'ютером в мережі, яка, в свою чергу, може бути розподілена на великій відстані. Найдавніший приклад цього з стосується факторизації 116-значного числа (Lenstra і Manasse, 1990). У цьому випадку етап просіювання програми MPQS був розподілений між добровольцями, які потім використовували свої комп'ютери щоб генерувати відносини та відправляли їх електронною поштою на центральний вузол, який потім обчислював результати. Хоча безпосереднього зв'язку між комп'ютерними системами не було, передавання даних за допомогою електронної пошти або навіть звичайної пошти все-таки являє собою мережу окремих вузлів які працюють на спільну мету.

Перевагою цієї особливої техніки є те, що багато вузлів дешево доступні. Недавніми прикладами є проекти BOINC та PrimeGrid.

Однак існує один недолік, який полягає у затримці на передачу інформації між вузлами. На відміну від етапу просіювання, лінійна алгебра QS має високий рівень співвідношення зв'язку/обчислення (Montgomery, 2000) і тому вимагає постійного обміну даними для роботи над набором даних. Накладні витрати які включають розбиття цього набору даних, передача даних між вузлами через Інтернет і тоді поєднання отриманих результатів, що вважаться занадто дорогим для цих задач.

Існує також локальної мережі комп’ютерів ріст продуктивності таких проектів сповільнився великою вартістю та складнощами в реалізації.

1.4.3. Графічні процесори GPGPU.

Всім суперкомп'ютерам описаним вище притаманні суттєві недоліки, головний з яких це вартість, яка робить їх недоступними для регіональних університетів, дослідних центрів, лабораторій і тим більше рядових дослідників.

До числа перспективних і інтенсивно розвиваються, у цій області в даний час відноситься паралельне програмування на графічних процесорах GPGPU (General-purpose graphics processing units) [120, 121]. Графічний процесор, на відміну від центрального, володіє меншим набором виконуваних команд (RISC), але більшою продуктивністю.

Головна особливість і перевага GPU перед CPU це висока ступінь паралелізму обчислень - сучасні GPU містять від декількох десятків, до декількох сотень скалярних процесорів, які працюють в паралельному режимі. Також GPU має свою внутрішню пам'ять, яка володіє більш високою швидкодією в порівнянні з RAM.

В останні кілька років графічний процесор стали активно застосовувати не тільки для вирішення спеціалізованих завдань (обробка графічної інформації), але і для обчислень загального призначення [122].

Технологія GPGPU дозволяє на одному обчислювачі досягати досить високого рівня паралелізму, при цьому не буде значних тимчасових витрат на передачу даних між обчислювальними вузлами і синхронізацію результатів обчислень.

Концепцію GPGPU, підтримали основні виробники графічних прискорювачів, зокрема компанія NVIDIA, яка розробила технологію CUDA (Compute Unified Device Architecture) і відповідну специфікацію [22], що полегшує процес створення алгоритмів для GPU.

Відеоадаптери компанії nVidia з архітектурою CUDA займають більшу частину ринку графічних рішень як в Україні, так і за кордоном [123, 124]. Відносна низька вартість і низьке енергоспоживання в поєднанні з високою питомою продуктивністю графічних процесорів дозволяють говорити про можливості масових розподілених паралельних обчислень, які доступні більш широкому колу потенційних порушників, ніж спеціалізовані мережеві багатопроцесорні системи.

Протягом останніх декількох років графічний процесор перетворився на потужні паралельні обчислювальні пристрої, що мають високу вартість. Хоча вона спочатку була розроблена тільки для прискорення графічних додатків, графічні процесори досліджувалися як ефективний апаратний прискорювач для неграфічних

загальних обчислень. Таким чином, відтепер графічний процесор оброблявся з GPGPU (Обчислювальна програма для графічного процесора)

Додавання обчислювальних модулів на базі GPU (graphics processing unit) дозволяє істотно збільшити загальну продуктивність суперкомп'ютерів з гетерогенної архітектурою. модернізація, проведена в 2015 році, обчислювального комплексу «СКИТ» (Інститут кібернетики імені В.М. Глушкова НАН України) дозволила збільшити його продуктивність з 0.28 до 18 тера-флопс [125]. Це стало можливо завдяки використанню, так званої, гібридної конфігурації «CPU + GPU», яка об'єднує 38 графічних процесорів nVidia Tesla M2075 [126].

Використовуючи графічні процесори і алгоритми розпаралелювання, можна підвищувати продуктивність не тільки суперкомп'ютерів. Одними з перших підтримку GPGPU додали в свої продукти розробники різних програм злому і систем «брутфорса» паролів. Наприклад, швидкість підбору MD5 - паролів на nVidia GTX580 становить до 15800 мільйонів комбінацій в секунду, що дозволяє знайти середній за складністю пароль довжиною вісім символів всього за 9 хвилин [127].

Таким чином для реалізації факторизації чисел методом MQkS на розподілених обчислювальних системах було обрано обчислювальну платформу CUDA на графічних процесорах загального призначення GPGPU.

Переваги графічних процесорів обертаються основною проблемою при складанні алгоритмів, так як для того, щоб досягти високої ефективності при таких масивно-паралельних обчисленнях, потрібно враховувати безліч чинників: архітектурні особливості GPU, швидкодія і порядок доступу до пам'яті, механізми синхронізації між обчислювальними потоками. Важливу роль відіграє також пристосованість самого алгоритму до виконання в паралельному режимі.

Можна виділити такі основні ключові моменти при реалізації методів перевірки криптостійкості RSA з використанням чисел розміром 10^{10} на GPU:

- швидкість доступу до пам'яті;
- паралельна реалізація алгоритмів простих арифметичних операцій;
- синхронізація між блоками паралельних ниток.
- відсутність бібліотеки великих чисел з підтримкою розпаралелювання.

Тому реалізація алгоритму MQkS повинна застосовувати переваги GPU такі як легкість створення потоків, та враховувати недоліки – нестачу пам’яті та наявність різних видів цієї пам’яті. Що, потребує реалізації спеціального представлення великих чисел та роботи з ними.

Постановка завдання. Популярність і інтенсивне розвиток технології CUDA, а також відсутність готових рішень для застосування в криптографічних цілях привели до ідеї створення бібліотеки длинної модульної цілісної та поліноміальної арифметики, яка використовує ГПУ як основний вичислительного ядра. Головна особливість і перевага GPU перед CPU це висока ступінь паралелізму розрахунків - сучасні GPU містять від декількох десятків, до декількох сотень скалярних процесорів, які працюють у паралельній режимі. Також GPU має свою внутрішню пам'ять, яка володіє більш високим швидкодією порівняно з ОЗУ. Переваги графічних процесорів обертаються основною проблемою при складанні алгоритмів, так як для досягнення високої ефективності при таких масивно-паралельних розрахунках слід враховувати безліч факторів: архітектурні особливості GPU, швидкодія і порядок доступу до пам'яті, механізми синхронізації між розрахунковими потоками Важливу роль грає також пристосування самого алгоритму до виконання в паралельній режимі.

1.5. Висновки до розділу 1.

Використання нових (модифікованих) обчислювальних методів із застосуванням нової архітектури високопродуктивних обчислювальних пристроїв при оцінки криптостійкості RSA криптомодулів завжди є актуальною задачею.

Відомі приклади компрометації RSA алгоритму працюють тільки для конкретних практичних реалізацій та, як правило, у загальному випадку не є ефективніше задачі факторизації.

Серед багатьох методів факторизації метод квадратичного решета (QS – quadratic sieve) займає друге місце у списку найшвидших алгоритмів, поступаючись тільки методу решета числового поля, а для чисел розміром до 110 десяткових знаків і досі є найкращим.

Відносна простота алгоритму сприяла виникненню багатьох його модифікацій.

Удосконалення методу QS можна умовно поділити на два напрямки:

1. Прискорення процесу просіювання (збільшення числа B-гладких, при сталому розмірі факторної бази)

2. прискорення вирішення матриці.

3. Адаптація алгоритму до обраної апаратної реалізації.

При цьому відмічається що основна проблема це - складність пошуку B-гладких чисел. Число B-гладких є відносно більшим при $X = X_0 = \left[\sqrt{N} + 1\right]$ та швидко зменшується при відхиленнях X від X_0. Тому кращою модифікацією QS вважається метод множинного квадратичного решета (MPQS – multiple polynomial quadratic sieve) в якому B-гладкі шукають серед остач $y_{a,b}(X) = (aX + b)^2 - N$, де a, b – спеціально підібрані цілі числа. В порівнянні з методом QS при $a > 1$ для поліномів $y_{a,b}(X)$ в a раз зменшується кількість можливих значень пробних X при тому ж радіусі просіювання. В той же час не досліджувалося використання поліномів виду $y_k(X) = X^2 - kN$, для яких при більшості значень k кількість пробних X в інтервалі просіювання залишається такою ж, як і для методу QS.

Тому наступні дослідження направлені на створення алгоритму який би дозволяв змінювати поліноми за меншу кількість часу, та більш ефективніше використовував більш щільне знаходження B-гладких чисел біля $\sqrt{N}$. Що створить умови при яких можливою стане факторизація чисел порядку до 2^{1024}, що важливо при вирішенні завдань криптоаналізу RSA алгоритму.

Одне з важливих покращень у методі квадратичного решета є ідея варіація великого множника - Large Prime Variations (LVP).

Додатковий аналіз B-гладких чисел згадується в літературі як LVP стартегія [84, 125]. Пропонується запам’ятовувати B-гладкі з неодиничним простим залишком. При знаходженніі B-гладких з однаковими залишками використовувати їх разом.

Подальшим розвитком ідеї LPV є використання напівгладких з двома великими множниками замість одного. Хоча така реалізація вимагає набагато більше

додаткової роботи, але забезпечує більше можливостей для вибору напівгладких чисел.

Популярність і інтенсивне розвиток технології CUDA, повна апаратна підтримка цілочисельних та побітових операцій, висока ступінь паралелізму розрахунків, наявність своєї внутрішньої пам'яті, яка володіє більш високою швидкодією у порівнянні з ОЗУ – причини застосування архітектури CUDA.

Переваги графічних процесорів обертаються основною проблемою при складанні алгоритмів, так як для досягнення високої ефективності при таких масивно-паралельних розрахунках слід враховувати безліч факторів: архітектурні особливості GPU, швидкодія і порядок доступу до пам'яті, механізми синхронізації між розрахунковими потоками Важливу роль грає також пристосування самого алгоритму до виконання в паралельній режимі, та в умовах обмеженого об’єму пам’яті.

РОЗДІЛ 2. МЕТОД МНОЖИННОГО КВАДРАТИЧНОГО k-РЕШЕТА ФАКТОРИЗАЦІЇ ЦІЛИХ ЧИСЕЛ.

Серед багатьох методів факторизації метод квадратичного решета (QS) займає друге місце у списку найшвидших алгоритмів, поступаючись тільки методу решета числового поля, а для чисел розміром до 110 десяткових знаків і досі є найкращим.

Основна ідея методу QS полягає у пошуку *B*-гладких чисел, тобто таких, що різниця

$$y = x^2 - N \tag{2.1}$$

розкладається у добуток простих чисел - елементів факторної бази.

Основною проблемою для методу квадратичного решета - є пошук достатньої кількості B-гладких чисел.

В науковій літературі описується два підходи до визначення елементів факторної бази. Згідно [103] для цього визначається границя гладкості B – число, що обмежує зверху величину простих чисел – елементів бази. Самі ж елементи бази визначаються на основі обчислення значення символу Лежандра.

Проте при обмеженнях на границю гладкості можливі випадки, коли неможливо отримати достатню кількість B-гладких чисел. Тоді збільшують границю гладкості та число елементів факторної бази і пошук B-гладких повторюється.

В роботі [125] пропонується не обмежуватися границею гладкості, а визначати кількість елементів факторної бази. Згідно [125] оптимальна кількість елементів факторної бази L^a визначається за формулою

$$L^a = (e^{\sqrt{\ln N \ln\ln N}})^{\sqrt{2}/4} \tag{2.2}.$$

Для факторизації *N* потрібно знайти L^a + 2 *B*-гладких чисел, що призводить до необхідності визначати остачі в (1) та перевіряти можливість розкладання на прості множники елементів факторної бази для великої кількості значень *x*. Кількість пробних *x* рекомендується [125] вибирати з інтервалу просіювання [$-L^b$, L^b], де радіус просіювання L^b визначається за формулою

$$L^b = (e^{\sqrt{\ln N \ln\ln N}})^{3\sqrt{2}/4} \tag{2.3}.$$

Факторизація чисел порядку 10^{129} та більших потребує значних обчислювальних ресурсів (для числа N відомого як RSA-129 була задіяна мережа 1600 комп'ютерів, що пропрацювала 220 днів, була сформована матриця лінійних рівнянь з 524338 невідомими, яка вирішувалася на суперкомп'ютері протягом двох днів [113]). Тому розробка способів зниження обчислювальної складності алгоритму методу QS є актуальними.

Основні ідеї підходів, що можуть забезпечувати зниження обчислювальної складності алгоритму методу QS, пов'язані зі зменшенням області просіювання та розміру факторної бази. В літературних джерелах (див., наприклад, [114, 128, 129]) відмічається, що спроба зменшення числа елементів факторної бази призводить до збільшення інтервалу просіювання та може призводити до зростання обчислювальної складності. При її ж збільшенні необхідно отримувати більшу кількість B-гладких та вирішувати систему рівнянь більш високого порядку, що також може призводити до зростання обчислювальної складності.

Для алгоритму квадратичного решета було запропоновано ряд варіантів модифікацій, які пов'язані з прискоренням процесу просіювання та вирішенням матриці.

Серед модифікацій алгоритму QS окремо слід виділити ідею П. Мантгомері [109, 130] методу множинного поліноміального квадратичного решета (MPQS). В запропонованому методі крім полінома (1) розглядаються ряд інших поліномів виду

$$z_{a,b}(x) = (ax + b)^2 - N = a^2x^2 + 2abx + b^2 - N, \qquad (2.4)$$

де a, b – спеціально підібрані цілі числа, x вибирається з інтервалу просіювання $[-L, L]$, а для кожного з поліномів $z_{a,b}(x)$ формується своя факторна база. Існує зв'язок між величиною інтервалу просіювання, що визначається числом L, та кількістю поліномів $z_{a,b}(x)$. При малих L необхідно генерувати багато поліномів $z_{a,b}(x)$, для яких слід визначати параметри a, b та формувати факторну базу, що є затратною процедурою. При великих L число поліномів буде меншим, але при просіюванні необхідно обробляти досить великі значення залишків $q(x) = z_{a,b}(x)/a = ax^2 + 2bx + c$, де $c = (b^2-N) / a$.

При аналізі оцінки обчислювальної складності методу квадратичного решета

згідно [129] використовується середнє значення пробних x з інтервалу просіювання для отримання одного B-гладкого числа. Проте, як показують чисельні експерименти, B-гладкі числа розміщуються на інтервалі просіювання в середньому згідно деякого закону, який потребує додаткового дослідження.

2.1. Розміщення B-гладких в діапазоні інтервалу просіювання.

Інтуїтивно зрозуміло, що чим меншою буде остача в $y = x^2 - N$, $z_{a,b}(x) = (ax + b)^2 - N = a^2x^2 + 2abx + b^2 - N$, чи $y = x^2 - kN$ тим більшою повинна бути ймовірність розкладання її на прості множники – елементи факторної бази. Тому перше завдання, що вирішувалося при розробці методу MQkS, полягало в отриманні оцінок стосовно розміщення.

B-гладких в діапазоні інтервалу просіювання. При проведенні чисельних експериментів вияснилося, що в кожному конкретному випадку чисел N при однакових значеннях кількості елементів факторної бази та розміру області просіювання різною була кількість B-гладких чисел і розподіл відповідних їм x в області просіювання. Тому для отримання оцінок, які можна було б використати в подальшому, було прийнято наступні правила проведення чисельних експериментів.

П1. Множина чисел N, що використовуються при отриманні середніх оцінок розміщення B-гладких в діапазоні інтервалу просіювання, формується як добуток двох різних простих чисел p і q ($p < q$), де p і q вибиралися з множини простих, з порядковими номерами, що слідують підряд.

П2. Для кожного N зі сформованої їх множини встановлюються єдині значення розміру факторної бази L^a і для кожного N визначається радіус просіювання L^b за формулою (2.3).

П3. Для оцінки розміщення B-гладких в діапазоні інтервалу просіювання використовуються дані тільки для тих N, для яких отримано L^a B-гладких остач (2.2).

Перша серія чисельних експериментів була проведена для 14 множин чисел $N=pq$, де кожна з множин містила 500000 варіантів. В кожній з множин m множники p - це прості числа з порядковими номерами в діапазоні від 1001 до 2000, де номер 1

– це число 2, 2 – число 3, 3 – число 5 і.т.д. Для m – ї множини прості q вибиралися як порядкові номери простих чисел в діапазоні їх номерів від 2001+500m до 2500+500m. Для кожної з таких множин розглядалися варіанти числа fb елементів факторної бази, що дорівнювало 8, 16, 24, 32, 48 та 64, де 16 відповідає L^a, що визначалася за формулою (2) для середнього значення N серед аналізованих їх множин, 8 - $0.5L^a$, 16 - L^a, 24 - $1.5L^a$, 32 - $2L^a$, 48 - $3L^a$ та 64 - $4L^a$.

Для визначення середнього значення кількості x, необхідних для отримання кожного j-го В-гладкого числа (k=1÷fb) розраховувалося сумарна величина $sum(j)$ значень x для всіх sc значень N, що відповідають правилу П3, після чого отримане сумарне значення ділилося на sc. Тобто середнє значення $xx(j)$ розраховувалося за формулою $xx(j) = sum(j)/sc$.

З метою оцінки характеристики залежності, що відповідає варіанту множини та розміру факторної бази, формувалися перші ($dx(j)=xx(j+1)-xx(j)$), другі ($d2x(j)=dx(j+1)-dx(j)$) та треті ($d3x(j)=d2x(j+1)-d2x(j)$) різниці середніх значень. Характерні фактичні дані для m рівних 1, 7 та 14 при fb = 8 наведені в табл. 2.1 з використанням чотирьох десяткових цифр. Аналогічні дані для m = 1 і fb = 16, 32 та 64 для перших 16 В-гладких приведено в табл. 2.2.

Таблиця 2.1 **Дані про розподіл середніх значень В-гладких чисел в діапазоні інтервалу просіювання для m=1, 7, 14 для fb = 8.**

m	m=1				m=7				m=14			
	xx	dx	$d2x$	$d3x$	xx	dx	$d2x$	$d3x$	xx	dx	$d2x$	$d3x$
j=0	0	21.35	39.01	14.87	0	26.69	50.22	19.83	0	30.53	58.71	24.52
j=1	21.35	60.36	53.88	10.53	26.69	76.91	70.05	16.71	30.53	89.24	83.24	20.89
j=2	81.71	114.2	64.42	9.581	103.6	147.0	86.75	17.41	119.8	172.5	104.1	20.26
j=3	196.0	178.7	73.99	13.70	250.5	233.7	104.2	15.69	292.3	276.6	124.4	22.45
j=4	374.6	252.6	87.69	8.859	484.3	337.9	119.8	14.88	568.9	401.0	146.8	12.01
j=5	627.2	340.3	96.55	8.321	822.1	457.7	134.7	10.68	969.9	547.8	158.9	14.60
j=6	967.6	436.9	104.9		1280	592.4	145.4		1517	706.7	173.5	
j=7	1404	541.8			1872	737.7			2224	880.2		
j=8	1946				2610				3104			
sc	295368 (59.07%)				240086 (48.02%)				208923 (41.78%)			

Таблиця 2.2 **Дані про розподіл середніх значень B-гладких чисел в діапазоні інтервалу просіювання при m= 1 та fb = 16, 32, 64 для перших 16 B-гладких.**

fb	*fb* =16				*fb* =32				*fb* =64			
	xx	*dx*	*d*2*x*	*d*3*x*	*xx*	*dx*	*d*2*x*	*d*3*x*	*xx*	*dx*	*d*2*x*	*d*3*x*
j=0	0	6.463	5.315	0.541	0	2.848	1.079	-0.100	0	1.897	0.382	-0.078
j=1	6.463	11.78	5.856	-0.135	2.848	3.927	0.980	-0.122	1.897	2.279	0.304	0.039
j=2	18.24	17.63	5.721	-0.164	6.775	4.907	0.858	-0.122	4.177	2.583	0.342	-0.098
j=3	35.87	23.36	5.558	-0.262	11.68	5.765	0.736	-0.084	6.759	2.925	0.244	-0.005
j=4	59.23	28.91	5.296	-0.070	17.45	6.501	0.652	-0.037	9.684	3.169	0.240	-0.032
j=5	88.14	34.21	5.226	-0.078	23.95	7.153	0.615	-0.014	12.85	3.409	0.207	-0.009
j=6	122.4	39.43	5.148	0.107	31.10	7.768	0.601	-0.019	16.26	3.616	0.199	-0.002
j=7	161.8	44.58	5.255	-0.265	38.87	8.369	0.582	-0.004	19.88	3.815	0.197	-0.045
j=8	206.4	49.84	4.989	0.134	47.24	8.950	0.578	-0.048	23.69	4.012	0.150	0.003
j=9	256.2	54.83	5.123	0.280	56.19	9.529	0.530	0.032	27.71	4.161	0.153	-0.009
j=10	311.0	59.95	5.403	0.057	65.72	10.06	0.561	-0.064	31.87	4.315	0.144	-0.001
j=11	371.0	65.35	5.461	-0.213	75.78	10.62	0.497	-0.003	36.18	4.459	0.143	-0.023
j=12	436.3	70.81	5.247	0.638	86.40	11.12	0.494	0.025	40.64	4.602	0.120	0.026
j=13	507.1	76.06	5.886	-0.513	97.51	11.61	0.520	-0.042	45.24	4.721	0.146	-0.048
j=14	583.2	81.95	5.372		109.1	12.13	0.478	0.002	49.96	4.867	0.097	0.028
j=15	665.2	87.323			121. 3	12.61	0.480	-0.013	54.83	4.964	0.125	-0.016
j=16	752.3				133.9	13.09	0.467	0.030	59.80	5.090	0.109	-0.014
sc	495238 (99.0476%)				499966 (99.9932%)				500000 (100%)			

Таблиці 2.1 і 2.2 ілюструють ряд висновків, отриманих за результатами чисельних експериментів.

B1. В усіх випадках даних про числа *N* та кількість елементів факторної бази має місце зростання змінних *xx* та *dx*.

B2. Для числа елементів факторної бази $fb = L^a$ функція *xx*(*j*) практично є параболою, оскільки треті різниці *d*3*x* постійно змінюють свій знак.

B3. Для числа елементів факторної бази $fb < L^a$ функція *xx*(*j*) росте швидше ніж парабола, оскільки треті різниці *d*3*x* додатні, їх значення спочатку спадають, а потім стабілізуються на додатному значенні.

B4. Для числа елементів факторної бази $fb > L^a$ функція *xx*(*j*) росте як парабола, проте додатні значення других різниць *d*2*x* спочатку спадають, а потім стабілізуються на додатному значенні.

B5. З ростом N зростають середні значення *xx*(*j*) для всіх *j*.

В6. Зі збільшенням числа елементів факторної бази зменшуються середні значення $xx(j)$ для всіх j.

В7. Для поліноміальної функції $xx(j)$ значення коефіцієнта при j^1 не перевищує значення коефіцієнта при j^2.

На основі отриманої інформації та наведених висновків можна припустити, що при пошуку B-гладких на основі використання багатьох поліномів з малою областю просіювання є ймовірність отримати модифікацію алгоритму методу QS (MPQS), обчислювальна складність якого буде нижчою ніж QS. Цього можна очікувати, оскільки B-гладких чисел більше при малих x з інтервалу просіювання. Використання такої обставини при вирішенні задачі факторизації в літературних джерелах не виявлено.

Метод MPQS враховує проблему зменшення кількості B-гладких при великих значеннях чисел з інтервалу просіювання, та використовує декілька поліномів.

Існує зв'язок між величиною інтервалу просіювання, що визначається числом L, та кількістю поліномів $z_{a,b}(x)$. При малих L необхідно генерувати багато поліномів $z_{a,b}(x)$, для яких слід визначати параметри a, b та формувати факторну базу, що є затратною процедурою. При великих L число поліномів буде меншим, але при просіюванні необхідно обробляти досить великі значення залишків $q(x) = z_{a,b}(x)/a = ax^2 + 2bx + c$, де $c = (b^2\text{-}N) / a$.

Трудомісткість від заміни многочленів складає 25-30% від усього алгоритму [114, 106]. При зміні многочлена необхідно виконати процес ініціалізації.

Метод ідеально підходить для разпаралелювання, бо кожен процесор може працювати з конкретним поліномом. І йому нема потреби зв’язуватися з центральним процесором до кінця процесу просіювання.

2.2. Середнє значення числа елементів факторної бази при фіксованій границі гладкості для поліномів.

Принциповий момент для методу QS і MPQS – це використання поліномів (2.1) та (2.4), у яких входить значення (- N). В той же час при пошуку пари чисел A і B для досягнення рівності

$$A^2 = B^2 (\text{mod } N), \quad (2.5)$$

що відображає ідею М. Крайчека, можна використати множину поліномів

$$y = x^2 - kN, \quad (2.6)$$

де k - натуральне число, а побудова нових поліномів є дуже простою. Покажемо це на прикладі.

Нехай N = 86327. Значення L^a = 6 (згідно (2.1)). Приймемо радіус просіювання $L^b = 2L^a = 12$ та $k = 1 \div 2L^a$. В якості елементів факторної бази вибираємо перших L^a найменших простих чисел. Перебираючи всі варіанти значень k та x отримаємо B-гладкі числа, представлені в табл. 2.3.

Таблиця 2.3

Дані про B-гладкі для N=86327.

k	x_0	x	x_0+x	B-гладке	Показники степенів елементів факторної бази						
					-1	2	3	5	7	11	13
1	294	B-гладкі відсутні									
2	416	-1	415	-429	1	0	1	0	0	1	1
		12	428	10530	0	1	4	1	0	0	1
3	509	0	509	100	2	0	2	0	0	0	0
4	588	B-гладкі відсутні									
5	657	0	657	14	0	1	0	0	1	0	0
		2	659	2646	0	1	3	0	2	0	0
		7	664	9261	0	0	3	0	3	0	0
6	720	-1	719	-1001	1	0	0	0	1	1	1
7	778	-1	777	-560	1	4	0	1	1	0	0
		5	783	8800	0	5	0	2	0	1	0
		-1	767	-16000	1	7	0	3	0	0	0
8	832	-1	831	-55	1	0	0	1	0	1	0
9	882	B-гладкі відсутні									
10	930	B-гладкі відсутні									
11	975	B-гладкі відсутні									
12	1018	-11	1007	-21875	1	0	0	5	1	0	0

В табл. 2.3 для значень k, що містять дільники, які є квадратами простого числа t, значення $x_0 + x$ не діляться на t, де

$$x_0 = \left[\sqrt{kN}\right] + 1. \quad (2.7)$$

Аналіз даних табл.2.3 показує, що рівність (2.5) можна отримати на основі добутку B-гладких, що відповідають x, які дорівнюють 777 і 1007. Рішення отримаємо

при x=509. В обох випадках це дозволяє знайти множники N числа 173 та 499. Але при отриманні рівності (2.5) на основі добутку значень x, рівних 657, 659 та 664, корінь буде хибним.

Наведений приклад демонструє можливість отримання кореня при використанні співвідношень (2.6). В результаті було отримано 12 B-гладких чисел при кількості просіяних x рівній 25*12 = 300. У випадку ж базового методу квадратичного решета факторна база з шести елементів міститиме прості числа 2, 17, 23, 29, 53 і 61, а для просіяних x від 30 до 558 (всього 529 пробних x) буде отримано всього 5 B-гладких замість необхідних 8.

Можна очікувати, що метод, заснований на ідеї використання рівності (6), зможе забезпечити зменшення обчислювальної складності процесу просіювання та зменшення загального часу вирішення задачі факторизації. Метод факторизації, в якому пропонується використати поліноми (2.6), будемо називати в подальшому множинним квадратичним k-решетом (Multiple Quadratic k-Sieve) (MQkS).

Однією з основних проблем, пов'язаних з використанням різних функцій для формування множини B-гладких чисел полягає у тому, що для кожної з них необхідно формувати свою факторну базу. В методі MQkS пропонується використовувати загальну факторну базу (ЗФБ) для всіх значень k, що містить прості числа до деякої границі гладкості B.

Згідно результатів чисельних експериментів та висновку В6 при одній і тій же області просіювання число отримуваних B-гладких чисел залежить від кількості елементів факторної бази. Тому важливою є оцінка середнього числа елементів факторної бази для різних N та множини k при фіксованій границі гладкості. Для отримання такої інформації проводилася друга серія чисельних експериментів. Аналогічно як і для першої серії формувалися множини чисел N. Було сформовано 28 множин, кожна з яких містила 100000 варіантів. Для множин m = 1÷7 множники p - це прості числа з порядковими номерами в діапазоні від 501 до 600, а для m – ї множини прості q вибиралися як порядкові номери простих чисел в діапазоні їх номерів від 1501+1000m до 2500+1000m. Для множин m = 8÷14 множники p - це прості

числа з порядковими номерами в діапазоні від 1001 до 1100, а для m – ї множини прості q вибиралися як порядкові номери простих чисел в діапазоні їх номерів від 1501+1000*(m-7) до 2500+1000*(m-7). Для множин m = 15÷21 множники p - це прості числа з порядковими номерами в діапазоні від 1501 до 1600, а для m – ї множини прості q - це порядкові номери простих чисел в діапазоні їх номерів від 1501+1000*(m-14) до 2500+1000*(m-14). Для множин m = 22÷28 множники p - це прості числа з порядковими номерами в діапазоні від 2001 до 2100, а для m – ї множини прості q - це порядкові номери простих чисел в діапазоні їх номерів від 1501+1000*(m-21) до 2500+1000*(m-21).

Для кожної з таких множин розглядалися варіанти границі гладкості, яка визначалася порядковим номером простого числа, який рівний L^a та $2L^a$. При цьому K - максимальне значення для k у співвідношеннях (2.6) вибиралися рівними $2L^a$, $4L^a$ та $(L^a)^2$, але не розглядалися k , що ділиться націло на квадрат простого числа, яке не перевищує $\sqrt{B}$. Для множин m = 22÷28 додатково розглядалися варіант границі гладкості $B = 4L^a$ Загальне число варіантів розрахунків склало 28 * 3 * 2 + 7*3 = 189.

За результатами розрахунків встановлено, що середнє число елементів факторної бази для всіх варіантів розрахунків перевищує половину кількості простих чисел з ЗФБ, тобто не більших за границю гладкості B. При цьому найменше значення становило 50.1572% (при m = 2, $B = L^a$, $K = 2L^a$), а найбільше - 54.3208% (при m = 5, $B = L^a$, $K = (L^a)^2$).

На основі отриманих даних можна попередньо оцінити затрати на формування множини B-гладких чисел.

Нехай границя гладкості B визначена з умови, що число простих, які не перевищують B, дорівнює $2L^a$. Радіус інтервалу просіювання становить $2L^a$, а значення $k = 1 \div 2L^a + 2$. Визначимо ймовірне середнє число B-гладких чисел, яке можна при цьому отримати.

Оскільки середнє число елементів факторної бази перевищує половину кількості простих чисел не більших за границю гладкості B, то можна припустити, що для кожного з k розмір факторної бази $fb \geq L^a$. Але тоді B-гладкі в діапазоні інтервалу

просіювання розміщуються за квадратичним законом, де нульовому B-гладкому відповідає 0, а B-гладкому з номером L^a деяке значення з інтервалу просіювання. Інтервал просіювання містить $2L^b+1$ пробне значення x, де $L^b = (L^a)^3$. Тому в гіршому випадку порядковий номер x з інтервалу просіювання може дорівнювати $2(L^a)^3 + 1$. Виходячи з квадратичного закону розміщення B-гладких, де для полінома другої степені $xx(j)$ значення коефіцієнта при j^1 не перевищує значення коефіцієнта при j^2, обчислимо необхідну кількість пробних x, щоб знайти перше B-гладке.

Нехай коефіцієнти при j^1 та j^2 для полінома співпадають і дорівнюють c. Тоді $xx(j) = c\,j^2 + c\,j$. Коефіцієнт c визначимо з умови, що при $j = L^a$

$$xx(j) = 2(L^a)^3 + 1. \tag{2.8}$$

З рівності (8) тоді отримаємо, що

$$c = \frac{2(L^a)^3+1}{(L^a)^2+L^a} < 2L^a. \tag{2.9}$$

Згідно (2.9) для знаходження першого B-гладкого необхідно не більше ніж $2L^a(1+1) = 4L^a$ пробних x, що відповідає радіусу просіювання $2L^a$. Оскільки така умова має місце для всіх значень k, то при $2L^a + 2$ значеннях k отримаємо не менше ніж $2L^a + 2$ B-гладких чисел, що достатньо для формування матриці та вирішення задачі факторизації.

Наведені оцінки показують, що при такому підході число пробних x, необхідних для факторизації, оцінюється величиною $4(L^a)^2$, що суттєво менше за $2(L^a)^3$.

Проте в такому алгоритмі факторизації наявні ряд додаткових операцій, а саме:

- $2L^a + 2$ обчислення кореня з великого числа;
- $2L^a + 2$ формування факторної бази і для кожного з елементів p факторної бази пошук числа $0 \le t < p$, для яких

$$((x_0 + t)^2 - kN)\ mod\ p^i = 0, \tag{2.10}$$

де $i \ge 1$, а $p^i < B$.

Додаткове збільшення обчислювальної складності отримуємо і при вирішенні матриці, розмір якої збільшується вдвічі та визначатиме асимптотичну оцінку

обчислювальної складності того ж порядку, що і для алгоритму методу QS. Тому необхідно оцінити обчислювальну складність методу MQkS для випадків, коли границя гладкості B менша за L^a.

2.3. Використанням сигнальних остач при просіюванні пробних значень.

У попередньому розділі описано процес просіювання. Який полягає у суттєвому скороченні множини пробних x, за допомогою простих операцій додавання та віднімання. Сенс процедури просіювання полягає в економії великої кількості операцій ділення великих чисел. Ця економія дає на практиці дуже суттєвий ефект, із-за чого метод квадратичного решета перевершив усі попередні алгоритми факторизації. Тому імплементація просіювання для методу MQkS має ключове значення.

Для просіювання пробних значень X пропонується скористатися їх відсіюванням на основі порівняння сигнальних остач $y^*(X)$ з остачами $y_k(X) = X^2 - kN$ ($k \geq 1$), де сигнальні остачі – це добуток перших степенів елементів факторної бази, що є множниками $y_k(X)$.

2.3.1. Оцінка кількості B-гладких, що втрачаються при попередньому просіюванні.

Нехай ЗФБ є множиною всіх найменших простих чисел починаючи з 2, що містить fa елементів, $fa = (L^a)^{pla}$, де $L^a = exp\left(\frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)$– розмір ФБ, рекомендований в [112], pla – параметр, значення якого знаходиться в межах від 0.5 до 1.5. Оскільки для остач $y_k(X)$ при різних k множини можливих дільників є різними, то в методі MQkS для кожного k формується поточна факторна база (ПФБ), число елементів у якій позначено через pfa.

Визначимо розмір радіусу просіювання $fb = (L^a)^{plb}$, де plb – параметр, значення якого знаходиться в межах від 0.5 до 4, $fb = L^b$ при $plb = 3$, а L^b є рекомендованою в [125] величиною радіусу просіювання.

Вважатимемо, що при пошуку B-гладких остач $y_k(X)$ пробне X не буде виключатися з розгляду (не буде відсіюватися), якщо виконана умова

$$log(y_k^*(X)) \geq h \cdot log(y_k(X)), \quad (2.11)$$

де параметр h – це деяке число $h \in [0, 1]$.

Множину пробних X для яких виконана умова (2.11) позначимо через MV(h). Тоді констатація факту того, що пробне X відсіюється, позначається як $X \notin$ MV(h), а в інших випадках $X \in$ MV(h).

Слід відмітити, що при h = 1 до множини B-гладких ввійдуть остачі тільки $y_k(X)$, для яких виконується умова

$$y_k(X) = y_k^*(X) \quad (2.12)$$

тобто $y_k(X)$ є добутком простих чисел з поточної ФБ, показник степеня яких дорівнює 1.

Оцінимо степінь впливу параметру h на час знаходження достатньої кількості B-гладких при фіксованих значеннях параметрів *pla* та *plb*.

Значення параметру *plb* вибиралося на основі чисельних експериментів з числами N порядку 10^m, де m = 20 ÷ 32, із умови, що час знаходження достатньої кількості B-гладких при h = 0 та *pla* = 0.8 ÷ 1 був близький до мінімального. Отримане значення *plb* дорівнює 1.4.

Числа N формувалися як добуток двох простих, кожне з яких задавалося як сума опорного (як правило є складеним числом) та одного з 5 значень приростів до нього. Значення опорних та прирости до них наведені в табл.1. Результати чисельних експериментів з визначення fa + 3 B-гладких для кожного з 25 варіантів чисел N порядку 10^m (m = 20 ÷ 32) та визначення серед них кількості тих, для яких має місце нерівність

$$log(y_k^*(X)) < h \cdot log(y_k(X)) \quad (2.13)$$

при значеннях h = 0.1·j (j=0÷10), наведені в табл. 2.4.

Таблиця 2.4 **Дані про прості числа, що використовувалися в чисельних експериментах, представлені значеннями опорних та приростами до них**

m	Опорні значення та прирости до них											
	Опорне 1	Прирости					Опорне 2	Прирости				
		1	2	3	4	5		1	2	3	4	5
20	6 471 594 853	16	36	54	96	120	15 452 141 593	58	76	78	106	118
21	27 749 160 899	32	42	54	92	140	36 037 125 721	42	88	100	112	120
22	87 614 993 781	8	82	86	110	116	114 135 715 457	2	12	14	36	54
23	274 858 909 339	34	54	94	114	130	363 823 025 568	1	25	29	43	79
24	651 133 587 339	4	14	58	98	104	1 535 783 162 540	53	63	83	149	219
25	2 095 629 580 239	142	160	184	248	292	4 771 835 678 545	28	46	48	108	118
26	6 787 809 030 482	39	41	77	111	195	14 732 294 257 385	24	78	122	158	194
27	22 126 102 809 573	8	16	34	58	76	45 195 487 366 502	131	167	195	215	225
28	72 582 191 787 419	14	32	68	108	222	137 774 841 923 874	89	107	119	133	137
29	239 604 396 594 260	47	81	209	293	357	417 354 612 108 130	21	39	67	123	133
30	687 691 741 189 047	100	104	176	184	244	1 454 139 900 343 951	16	22	130	162	190
31	2660737903883030	101	153	219	243	329	3758355900220833	20	28	38	70	80
32	8950030870996722	5	31	119	149	167	11173145818307493	4	20	26	86	104

В табл. 2.4. окремо виділені ті з B-гладких, що отримані при $X = X_0 = \left\lfloor \sqrt{N} + 1 \right\rfloor$ та $X = X_0 - 1 = \left\lfloor \sqrt{N} \right\rfloor$. Інформацію про їх кількість представлено в колонці, позначеною D0. Значення пробних *X*, які дорівнюють X_0 та $X_0 - 1$, не відсіюються незалежно при довільних значеннях параметра *h*. За такої умови виявилося, що при *h* < 0.3 відсутні B-гладкі, для яких виконана умова (2.13) і в табл. 2.5 не виділяються колонки, що відповідають значенням *h* = 0, *h* = 0.1, *h* = 0.2 та *h* = 0.3. Тобто жодне з B-гладких не втрачено при виконанні умови (2.13) для *h* < 0.3. Дані про число втрачених B-гладких за рахунок відсіювання пробних *X* при *h* = 0.1·*j* для *j* = 4÷10 наведені в колонках, позначених D(*j*) для відповідних значень *j*. В колонці, позначеній як DSum, наведено сумарну кількість B-гладких. Значення відношень D(*j*) / DSum представлено в табл. 2.6.

Таблиця 2.5 **Дані про кількість втрачених B-гладких за умови виконання обмеження (4) для $h = 0.1 \cdot j$ ($j = 4 \div 10$)**

m	D(4)	D(5)	D(6)	D(7)	D(8)	D(9)	D(10)	D0	DSum
20	9	43	180	495	1224	1966	2728	72	2800
21	2	19	135	490	1366	2274	3184	66	3250
22	6	22	118	488	1472	2553	3654	71	3725
23	2	18	110	506	1617	2904	4209	66	4275
24	0	17	116	479	1725	3346	4809	91	4900
25	0	13	106	552	1817	3685	5506	94	5600
26	0	12	89	482	1962	4140	6314	86	6400
27	0	11	86	542	2197	4822	7209	91	7300
28	0	9	82	465	2204	5388	8176	99	8275
29	1	9	73	491	2451	6094	9306	94	9400
30	0	7	82	548	2549	6777	10551	99	10650
31	0	3	70	480	2801	7660	11942	108	12050
32	0	3	52	491	2903	8565	13498	102	13600

Таблиця 2.6 **Значення відношень d(j) = D(j) / DSum**

m	d(4)	d(5)	d(6)	d(7)	d(8)	d(9)	d(10)
20	0,003214	0,015357	0,064286	0,176786	0,437143	0,702143	0,974286
21	0,000615	0,005846	0,041538	0,150769	0,420308	0,699692	0,979692
22	0,001611	0,005906	0,031678	0,131007	0,395168	0,685369	0,98094
23	0,000468	0,004211	0,025731	0,118363	0,378246	0,679298	0,984561
24	0	0,003469	0,023673	0,097755	0,352041	0,682857	0,981429
25	0	0,002321	0,018929	0,098571	0,324464	0,658036	0,983214
26	0	0,001875	0,013906	0,075313	0,306563	0,646875	0,986563
27	0	0,001507	0,011781	0,074247	0,300959	0,660548	0,987534
28	0	0,001088	0,009909	0,056193	0,266344	0,651118	0,988036
29	0,000106	0,000957	0,007766	0,052234	0,260745	0,648298	0,99
30	0	0,000657	0,0077	0,051455	0,239343	0,636338	0,990704
31	0	0,000249	0,005809	0,039834	0,232448	0,635685	0,991037
32	0	0,000221	0,003824	0,036103	0,213456	0,629779	0,9925

Для оцінки можливості використанні обмежень (2.11) при пошуку B-гладких в чисельних експериментах визначалося кількість $XX(j)$ пробних X, для яких мають місце обмеження

$$0.1 \cdot j \cdot log(y_k(X)) \le log(y_k^*(X)) \le 0.1 \cdot (j+1) \cdot log(y_k(X)) \; (j = 0 \div 9)$$

При отриманні таких даних для кожного з пробних X обчислювалися $y_k(X)$ та $y_k^*(X)$, їх логарифми та відношення $log(y_k^*(X)) \,/\, log(y_k(X))$. Сумарне фактичне

число *XSum* пробних обчислювалося для 25 варіантів чисел N 10^m (m = 20 ÷ 32). В табл. 2.7 наведено значення відношень $xd(j) = XX(j) / XSum$ при j = 0÷9.

Таблиця 2.7 **Значення відношень $xd(j) = XX(j) / XSum$ для j = 0÷9 при m = 20 ÷ 32.**

m	Значення j									
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	0.4654	0.2250	0.1832	0.1414	0.0786	0.0532	0.0289	0.0168	0.0114	0.0029
21	0.4563	0.2222	0.1763	0.1366	0.0729	0.0465	0.0311	0.0200	0.0114	0.0040
22	0.4282	0.2097	0.1710	0.1251	0.0585	0.0365	0.0228	0.0140	0.0102	0.0038
23	0.4311	0.2028	0.1635	0.0931	0.0536	0.0335	0.0208	0.0126	0.0075	0.0023
24	0.4071	0.1973	0.1578	0.0888	0.0514	0.0322	0.0198	0.0143	0.0096	0.0027
25	0.4259	0.1850	0.1450	0.1050	0.0539	0.0325	0.0196	0.0120	0.0071	0.0018
26	0.3973	0.1920	0.1591	0.1148	0.0498	0.0300	0.0208	0.0131	0.0061	0.0016
27	0.3819	0.1790	0.1458	0.1052	0.0456	0.0312	0.0211	0.0122	0.0071	0.0032
28	0.3674	0.1679	0.1344	0.0721	0.0358	0.0218	0.0135	0.0094	0.0057	0.0017
29	0.3679	0.1694	0.1338	0.0730	0.0364	0.0233	0.0135	0.0077	0.0039	0.0010
30	0.3561	0.1593	0.1286	0.0729	0.0355	0.0207	0.0126	0.0066	0.0039	0.0018
31	0.3915	0.1546	0.1191	0.0767	0.0324	0.0204	0.0124	0.0079	0.0049	0.0021
32	0.3543	0.1617	0.1093	0.0752	0.0387	0.0223	0.0140	0.0063	0.0030	0.0012

На основі отриманих даних табл. 2.5 – 2.7 можна зробити ряд висновків.

1. З ростом значення параметру h у співвідношенні (2.11) збільшується кількість відсіяних X та суттєво зменшується кількість тих з пробних X, серед яких слід шукати B-гладкі остачі $y_k(X)$. Обчислювальні затрати на попереднє просіювання та відсіювання значної частини пробних X є меншими ніж на перевірку чи буде $y_k(X)$ B-гладким для всіх цих пробних X. Тому можна сподіватися, що при використанні процедури попереднього просіювання зменшиться час пошуку достатньої кількості B-гладких.
2. Загальний час розрахунку достатньої кількості B-гладких для чисел N порядку 10^m, де m = 20 ÷ 32, повинен зменшуватися при значеннях $h < 0.3$ в умові (2.13), оскільки жодне з B-гладких не втрачається. Але і при

подальшому рості h серед відсіяних X можуть бути ті, для яких остача $y_k(X)$ буде B-гладким числом. Тоді для отримання достатньої кількості B-гладких необхідно буде розширювати область просіювання, що може призводити та росту обчислювальної складності та часу розрахунку. Тому доцільно отримати оцінки такого часу при різних h.

3. З даних табл. 2 випливає також, що зростає відносна кількість B-гладких, для яких має місце рівність (2.12) та для переважної більшості B-гладких виконується умова (2.11) при $h \geq 0.7$. Крім того, із аналізу даних про показники степенів множників можна зробити припущення, що більші від одиниці показники степеня простих множників p для остач $y_k(X)$ характерні для відносно малих значень простих p та рідко зустрічаються при більших p. Тому можна припустити, що при пошуку B-гладких доцільно обмежувати величину дільників p в остачах $y_k(X)$, для яких показник степеня може перевищувати одиницю. Це, безперечно, призведе до розширення області просіювання, але простішою буде процедура просіювання. Тому доцільно отримати оцінки часу отримання достатньої кількості B-гладких в залежності від ступеня такого обмеження.

2.3.2. Вплив коефіцієнта h на час пошуку достатньої кількості B-гладких

Для отримання оцінок впливу обмежень (2.11) при різних значеннях параметру h на час знаходження необхідної кількості B-гладких проводилися чисельні експерименти для множини чисел, представлених в табл. 2.4, що відповідають $m = 20 \div 32$. Дані про час розрахунків наведено в табл. 2.7 для $h = 0.1 \cdot j$ при $j = 0 \div 9$, де виділено мінімальний час.

Таблиця 2.7 **Час розрахунку достатнього числа B-гладких при різних значеннях параметра h для $m = 20 \div 32$.**

m	Значення параметра h при фіксованих параметрах pla=1.0 та plb=1.4									
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	2.219	2.047	1.860	1.531	1.328	1.203	1.140	*1.109*	1.125	1.203
21	3.422	3.125	2.812	2.453	2.017	1.828	1.860	*1.704*	1.710	1.922
22	4.953	4.516	4.142	3.394	3.031	2.844	2.593	2.640	*2.593*	2.687
23	7.702	6.945	6.156	5.209	4.609	4.194	3.860	3.844	*3.828*	3.993
24	11.171	10.265	8.906	7.437	6.534	6.281	5.750	5.609	*5.546*	5.812
25	15.812	14.594	12.672	10.516	9.281	8.531	8.030	8.000	*7.931*	8.375
26	23.046	21.476	18.453	15.567	13.523	12.281	12.542	11.835	*11.765*	11.921
27	33.843	31.280	27.046	22.514	19.906	18.203	17.797	*17.218*	*17.218*	17.547
28	49.436	45.437	39.445	32.999	29.077	26.999	25.906	25.603	*25.421*	26.240
29	71.698	65.341	56.342	47.255	41.702	38.608	37.468	36.999	*36.905*	60.826
30	103.58	97.341	80.981	68.310	60.404	56.279	54.701	53.920	*53.858*	68.357
31	145.03	130.92	113.37	96.122	85.895	80.310	77.919	77.059	*77.039*	92.513
32	210.07	188.74	160.91	136.73	122.12	115.30	112.00	110.90	*110.86*	129.67

Згідно даних табл. 2.7 кращий час отримано при для $h = 0.7$, та для $h = 0.8$. Тоді час розрахунку виявився майже вдвічі меншим, за відповідний час розрахунку для випадку $h = 0$, коли B-гладкі шукали серед остач $y_k(X)$ для всіх пробних X. Тому надалі в чисельних експериментах буде використовуватися параметр $h = 0.7$.

2.3.3. Характер розподілу більших за одиницю показників степеня множників B-гладких.

Вище відмічалося, що

- з ростом N зростає відносна кількість B-гладких, у яких всі показники степеня множників $y_k(X)$ дорівнюють одиниці,
- більші від одиниці показники степеня простих множників p для остач$y_k(X)$ характерні для відносно малих значень простих p та рідко зустрічаються при більших p,
- при перевірках подільності $y_k(X)$ лише на першу степінь простого для всіх його множників зменшиться кількість операцій з визначення B-гладких, що може призвести до зменшення часу визначення достатнього числа B-гладких навіть за умови використання додаткових k у співвідношеннях (2.6).

Для перевірки такого припущення проводилися чисельні експерименти, в яких:

1) допустимими B-гладкими вважалися ті, для яких показники степені їх дільників p могли бути більшими за одиницю при виконанні умови

$$f_p \le ff = (L^a)^{kff}, \qquad (2.14)$$

де f_p – порядковий номер простого p у списку простих чисел, kff – дійсне число, значення якого змінюються в діапазоні від 0 до 1, при значення kff = 1 відсутні обмеження для f_p, а при kff =0 до B-гладких будуть віднесені тільки ті з остач $y_k(X)$, для яких має місце рівність (2.12);

2) серед множини B-гладких, що визначалися за умови відсутності обмежень для f_p, визначалося число B-гладких, для яких виконувалися умови (2.14) при $kff = j/10$ ($j = 1 \div 10$).

Проведені чисельні експерименти підтвердили таке припущення. Їх результати, стосовно часу отримання достатньої кількості B-гладких для 25 варіантів чисел N порядку 10^m при m = 20÷32, представлені в табл.2.8, де виділені отримані найменші значення часу.

Таблиця 2.8 **Час розрахунку достатнього числа B-гладких при різних значеннях параметра *kff* для *m* = 20 ÷ 32.**

m	Параметри *kff* (*pla*=1.0, *plb*=1.4, *h* =0.7)									
	kff = 0.1	*kff* = 0.2	*kff* = 0.3	*kff* = 0.4	*kff* = 0.5	*kff* = 0.6	*kff* = 0.7	*kff* = 0.8	*kff* = 0.9	*kff* = 1.0
20	1.794	1.531	1.360	1.156	1.079	0.969	*0.953*	0.969	1.031	1.125
21	2.688	2.244	1,984	1.719	1.578	1.468	*1.422*	1.468	1.562	1.718
22	3.593	3.000	2.703	2.344	2.125	2.016	*2.000*	2.063	2.250	2.500
23	5.453	4.578	3.859	3.405	3.171	3.015	*3.000*	3.125	3.370	3.813
24	7.641	6.469	5.414	4.844	4.469	*4.328*	4.500	4.515	4.906	5.609
25	10.370	9.093	7.703	6.687	7.814	*5.969*	6.156	6.344	6.906	7.922
26	14.890	13.053	11.140	9.422	8.812	*8.640*	8.702	9.140	10.047	11.546
27	21.968	18.702	15.656	13.421	12.687	*12.614*	12.678	13.250	14.765	17.093
28	30.499	26.203	21.277	19.389	*18.490*	17.921	18.281	19.392	21.859	25.265
29	42.871	36.537	30.233	27.233	*25.514*	25.171	25.843	27.500	31.435	36.514
30	58.936	50.655	42.608	38.150	*36.389*	35.811	36.483	39.561	44.722	53.022
31	81.075	69.575	58.534	51.530	*50.077*	50.092	51.940	55.654	63.353	76.013
32	111.871	90.857	80.607	75.091	*71.731*	71.937	73.664	80.340	97.621	110.512

Дані чисельних експериментів підтвердили, що з ростом величини простих чисел p зменшується відносне число B-гладких, для яких показник степеня множників p B-гладкого перевищують одиницю. Тобто росте відносна величина B-гладких, для яких виконується умова (2.14). Тому при фіксованому деякому значенні *kff* < 1 при виконанні (2.14) отримуватимемо меншу кількість B-гладких ніж при *kff* = 1. Проте їх отримання потребуватиме меншого числа операцій, оскільки для всіх множників p остач $y_k(X)$, порядкові номери f_p яких більші за *ff*, не допускається значення показника степеня вище 1, а те, що від є дільником $y_k(X)$, визначається на основі попереднього просіювання.

Згідно даних, наведених в табл.2.8, найкращий час розрахунку отримано:

- для m = 20 ÷ 23 при *kff* = 0.7;

- для m = 24 ÷ 26 при *kff* = 0.6.

- для m = 27 ÷ 32 при *kff* = 0.5.

Тобто з ростом N доцільно зменшувати значення параметра *kff*.

Із даних табл.2.8. слідує також, що з ростом N мінімальне значення часу розрахунку зміщується в сторону менших значень *kff*.

Далі представлено узагальнений опис алгоритму MQkS, вибір параметрів для нього, а також результати порівняльних чисельних експериментів з методом QS.

2.4. Алгоритм методу множинного квадратичного k-решета.

В рамках загального опису алгоритму будуть використані такі параметри, як границя гладкості B та розмір радіусу просіювання L, значення яких слід буде вибрати. В описаному нижче алгоритмі передбачено визначати необхідну кількість B-гладких чисел, і тільки після цього виконувати їх обробку. Тобто кроки алгоритму не передбачають діагоналізацію матриці "на ходу". Додатково враховується спосіб попереднього просіювання на основі сигнальних остач.

При його використанні отримано зниження часу знаходження достатньої кількості B-гладких для тієї ж множини значень N для m = 20 ÷ 30 в 14-22 рази для значень параметрів *plb* =1.4, *h* = 0.7, *kff* = 1.0 та значень *pla* = 1.0, *pla* = 0.95 і *pla* = 0.9. При порівнянні ж даних табл. 5 роботи [7] та представлених тут даних в табл. 7 роботи [9] час розрахунку знизився в 16-30 разів, де більші зниження відповідають більшим значенням N. Це підтверджує високу ефективність процедури попереднього просіювання.

Загальний алгоритм **A** методу MQkS.

1. Для заданого N задати значення параметрів *pla*, *plb*, *h* та *kff*. За їх значеннями та числом N визначити: кількість елементів *fa* загальної факторної бази за формулою $fa = exp\left(pla \cdot \frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)$; базове значення розміру радіусу просіювання *fb* за формулою $fb = exp\left(plb \cdot \frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)$; границю гладкості B згідно зі значенням *fa*; граничне значення *ff* порядкового

номера простого числа, що визначає множину можливих простих дільників p в остачах $y_k(X)$, показники степенів яких можуть перевищувати одиницю.

Обчислити значення логарифмів для всіх елементів ЗФБ. Дані запам'ятати в масиві *p_log*[*fa*].

Для всіх простих чисел, порядкові номери яких не перевищують *ff*, визначити їх степені, що не перевищують деякого вибраного значення, наприклад, границю гладкості B чи обмеження на тип даних. Дані запам'ятати їх в масиві *mpb*[*ff*] (наприклад, при B > 256 і B < 300 для числа 2, порядковий номер якого в списку простих дорівнює одиниці, *mpb*[1] = 256, для числа 3 *mpb*[2] = 243 і т.д.).

Лічильнику *k* присвоїти значення нуль.

2. $k = k + 1$.
3. У випадках, коли *k* ділиться на квадрат двох чи більше різних простих чисел, перейти до кроку 2.
4. Для числа *kN* виконати:

   4.1. Сформувати множину елементів поточної факторної бази, до якої будуть віднесені такі прості *p*, що є елементами ЗФБ:

   - прості *p*, для яких символ Лежандра $\left(\frac{kN}{p}\right) = 1$;
   - прості *p*, що є дільниками *k*, якщо $k(\mathrm{mod}p^2) > 0$.

   У випадках, коли для деякого одного простого *p*, $k(\mathrm{mod}p^2) = 0$, виключити таке p з множини елементів поточної ФБ незалежно від значення символу Лежандра $\left(\frac{kN}{p}\right)$.

   4.2. Визначити число *pfa* елементів поточної ФБ. Якщо виявиться, що $(2 \cdot pfa/fa)^5 < 0.75$, перейти до кроку 2. Якщо $(2 \cdot pfa/fa)^5 \geq 0.75$ обчислити значення поточного (залежного від *k*) радіусу просіювання $pfb = fb \cdot (2 \cdot pfa/fa)^5$ та перейти до кроку 4.4.

   4.4. Визначити $x_0 = \left[\sqrt{kN}\right] + 1$ та присвоїти значення: $xp = x_0$, $xm = x_0\text{-}1$, $yp = xp^2 - kN$ і $ym = kN - xm^2$. Визначити коефіцієнти розкладання *xp*, *xm*, *yp* і

ym за основою 1000. Результати розкладання записати в масиви, наприклад, *xp*0[], *xm*0[], *yp*0[] і *ym*0[]. Визначити перші 5 коефіцієнтів розкладання чисел *xp, xm, yp* і *ym* за основами степенів простих чисел, записаних в масиві *mpb*[*ff*] (кількість коефіцієнтів може розглядатися як параметр, а їх число 5 вибрано на основі чисельних експериментів). Результати розкладання записати в масиви, наприклад, *xpp*[], *xmp*[], *ypp*[] і *ymp*[].

4.5. Обчислити значення логарифму *lxp* = log(*xp*).

4.6. Для всіх простих з поточної ФБ знайти корені рівняння

$$(y_k(X))(mod\ p) = 0 \qquad (2.15)$$

користуючись алгоритмом Шенкса. Значення меншого з коренів запам'ятати в масиві, наприклад *mx*1[f_p], де f_p – порядковий номер простого p в списку простих.

5. c=0. Пробні $X = x_0$ та $X = x_0 - 1$ вважати елементами множини MV(h). Перевірити чи будуть B-гладкими остачі $y_k(x_0)$ та $y_k(x_0 - 1)$.
6. Виділити дві підмножини з інтервалу просіювання: (x_0+c, x_0+c+z] та [x_0–c–z, x_0–c), де z = min(1000, (*pfb*–x_0–c)).
7. Для $t = 1 \div z$ присвоїти значення нуль елементам масивів $mzp[t]$ і $mzm[t]$ та знайти наближене значення логарифму $log(y_k(x_0 + c + t)) \approx lxp + log(2c + t) = mlp[t]$.
8. Для кожного з елементів p поточної факторної бази визначити ті зі значень пробних $X = x_0 + c + t$ з підмножини (x_0+c, x_0+c+z], для яких має місце рівність (2.15), і для них додати значення логарифму від p, значення якого записано в p_log[f_p], до елементу масиву *mzp*[t].
9. На основі порівняння значень $mzp[t]$ та $mlp[t]$ з урахуванням значення параметра h визначити пробні X, що належать множині MV(h).
10. Для $X \in$ MV(h) перевірити чи існують дільники p з показником степеня s вище одиниці для остачі$y_k(X)$ серед простих чисел з поточної факторної бази, для яких $f_p \le ff$. Якщо такі існують, то для кожного з таких p при $X =$

$x_0 + c + t$ до елемента масиву $mzp[t]$ додати значення $(s\text{-}1)\cdot \text{p\_log}[f_p]$. Якщо в результаті виявиться, що

$$|mzp[t] - mlp[t]| < 0.1, \tag{2.16}$$

то $y_k(X)$ буде B-гладкою остачею.

11. Для кожного з елементів p поточної факторної бази визначити ті зі значень пробних $X = x_0 - c - t$ з підмножини [x_0–c–z, x_0–c), для яких має місце рівність (2.15), і для них додати значення логарифму від p, значення якого записано в $p\_log[f_p]$, до елементу масиву $mzm[t]$.
12. На основі порівняння значень $mzm[t]$ та $mlm[t]$ з урахуванням значення параметра h визначити пробні X, що належать множині MV(h).
13. Для $X \in$ MV(h) і $X = x_0 - c - t$ перевірити чи існують дільники p з показником степеня s вище одиниці для остачі$y_k(X)$ серед простих чисел з поточної факторної бази, для яких $f_p \leq ff$. Якщо такі існують, то для кожного з таких p до елемента масиву $mzm[t]$ додати значення $(s\text{-}1)\cdot \text{p\_log}[f_p]$. Якщо в результаті виявиться, що

$$|mzm[t] - mlm[t]| < 0.1, \tag{2.17}$$

то $y_k(X)$ буде B-гладкою остачею.

14. Якщо знайдене число B-гладких перевищує fa, перейти до п. 16, а інакше до п. 15.
15. $c = c + z$. Якщо $c = pfb$, перейти до п. 2, а інакше до п. 6.
16. Діагоналізувати матрицю та знайти нульовий рядок. Якщо нульовому рядку відповідає тривіальний корінь рівняння $A^2(mod\ N) = B^2(mod\ N)$, де A – це добуток ряду пробних значень X , а B – добуток відповідних їм остач (2.6), замінити його іншим B-гладким, за наявності, а інакше перейти до кроку 2. Якщо ж отримано нетривіальний корінь, то вивести значення множників числа N і закінчити роботу алгоритму.

В описаному алгоритмі A1 передбачається, що вирішення матриці здійснюється тільки тоді, коли знайдено достатнє число B-гладких. Проте такий пошук можна здійснювати на основі діагоналізації матриці на ходу, що

запропоновано в роботі [5]. В останньому випадку діагоналізація матриці може здійснюватися на кожен раз після виконання кроку 13.

2.5. Реалізація алгоритму *A* методу MQkS.

На кроці 1 алгоритму ***A*** для визначення числа *La* елементів ЗФБ, границі гладкості *B* і радіусу просіювання *Lb* пропонується використовувати співвідношення:

$$L_a = (e^{\sqrt{\ln N \ln\ln N}})^{ka\sqrt{2}/4} = (L^a)^{ka}, \quad (2.18)$$

$$L_b = (e^{\sqrt{\ln N \ln\ln N}})^{kb\sqrt{2}/4} = (L^b)^{kb}, \quad (2.19)$$

де коефіцієнти *ka* і *kb* – параметри, що використовуються при проведенні чисельних експериментів, границя факторної бази *B* – це просте число, що в списку зростаючих значень простих відповідає номеру *La*.

Кроки 2 і 3 – це робота з лічильником значень *k*. Кроки 5.1 - 5.5, та 5.9 визначають правила роботи для лічильника значень з області просіювання.

Число *N* – це велике число, для якого перевищені обмеження для базових типів long та double. Тому для виконання арифметичних операцій слід або скористатися бібліотекою для роботи з великими числами, або представляти великі числа масивами коефіцієнтів їх розкладання за деякою основою та виконувати операції для масивів чисел.

При комп'ютерній реалізації алгоритму ***A*** використано другий підхід. При цьому для великих чисел *u* і *v* при їх розкладанні за основою *b* у вигляді

$$u = \sum_{i=0}^{m_u} u_i b^i, \quad v = \sum_{i=0}^{m_v} v_i b^i$$

у відповідних їм поданні масивах *U* і *V* чисел буде записано:

$U[0]=m_u+1$; $U[j]=u_{j-1}$ ($j=1 \div m_u+1$); $V[0]=m_v+1$; $V[j]=v_{j-1}$ ($j=1 \div m_v+1$).

Тоді при обчисленні суми чисел *u* і *v* додаються відповідні значення елементів масиву, а при перевищенні сумою значення *b*-1 від суми віднімається *b* і добавляється одиниця до наступного розряду. При $u > v$ віднімання виконується подібним чином, але при від'ємній величині різниці значень у відповідних клітинках масиву до

результату додається b, а від значення в наступній клітинці масиву U віднімається одиниця.

При множенні u і v отримуємо масив UV, довжиною на більше ніж $U[0] + V[0] + 1$, коефіцієнти якого визначаються за правилом:

$$u \cdot v = \sum_{i=0}^{m_u} u_i b^i \cdot \sum_{j=0}^{m_v} v_j b^j = \sum_{t=0}^{m_u+m_v} b^i \cdot \sum_{s=0}^{t} u_s v_{t-s} = \sum_{t=0}^{m_u+m_v} b^i \cdot z_t,$$

де при $z_0 \geq b$ приймаємо значення $UV[1]$ рівним остачі від ділення z_0 на b ($UV[1] = z_0 (\mathrm{mod} b)$), цілу частину від ділення z_0 на b додаємо до z_1: $z_1^* = z_1 + [z_0 / b]$, а при $i > 0$ $UV[i+1] = z_i^* (\mathrm{mod} b)$, де $z_i^* = z_i + [z_{i-1}^* / b]$.

При діленні u / v необхідно буде шукати цілі значення частки і остачі. Для їх обчислення визначається число a типу double $a = V[m_v+1] \cdot b + V[m_v] + 1$, на яке ділилася величина $U^*[j+2] \cdot b^2 + U^*[j+1] \cdot b + U[j]$ при $j = 0 \div m_u - m_v$, де $U^*[j+2]$ та $U^*[j+1]$ – це результат віднімання від u попередніх значень частки, помноженої на v.

Для обчислення квадратного кореня з N використовується відома ітераційна формула $x_{i+1} = \left(x_i + \frac{N}{x_i}\right) / 2$, яка представлялась у вигляді

$$x_{i+1} = x_i + \frac{y_i}{2x_i} = x_i + dx_i,$$

$$y_i = y_{i-1} - 2x_i dx_i - dx_i^2, \qquad (2.20)$$

де початкове значення y_0 обчислювалося з використанням функції sqrt().

При обчисленнях за формулою (2.20) при відніманні від N отриманої частки на кожній ітерації при діленні уточнювалося значення x_i до тих пір, поки $N - x_i^2$ не стане меншим за $2x_i$.

Для роботи з великими числами вибиралася деяка «зручна» основа числення. В якості неї використовувалося значення 1000 чи 1024. При цьому у масивах даних, що відповідають великому числу, в нульовому елементі масиву вказувалося значення найбільшого номера з ненульовим коефіцієнтом.

Для реалізації кроку 4.1 визначалися значення символів Лежандра $\left(\frac{N}{p}\right)$, де p – просте число із загальної факторної бази. Враховуючи те, що $\left(\frac{kN}{p}\right) = \left(\frac{N}{p}\right) \cdot \left(\frac{k}{p}\right)$ значення $\left(\frac{N}{p}\right)$ зберігалися в пам’яті на протязі всього часу розрахунків, а для кожного

k обчислювалися значення $\left(\frac{k}{p}\right)$, де k і p – не відносяться до великих чисел, що дозволяло просто обчислювати символи Лежандра $\left(\frac{kN}{p}\right)$.

Всі p, для яких $\left(\frac{kN}{p}\right) = 1$, – це елементи факторної бази, визначеної для простих p, значення яких не перевищує границю гладкості B. Їх число Lk може бути різним при різних k, але завжди менше за La. Відмічалися випадки, коли Lk може бути значно меншим за La. Наприклад, Lk <10 при La > 250. Тоді практично завжди не вдавалося отримати хоча б одного B-гладкого числа на всьому інтервалі просіювання. Тому для випадків таких k в алгоритмі обчислень пропонувалося не шукати B-гладкі, а збільшувати k на одиницю. Але у випадках, коли $2 \cdot Lk > La$ виявилося, що доцільно збільшувати інтервал просіювання. Тому в залежності від k інтервал просіювання встановлювався рівним $Lb \cdot (2 \cdot Lk / La)^5$. Показник степені 5 був підібраний за результатами чисельних експериментів.

На кроці 4.2 обчислено значення кореня з kN та введено змінні:

xp – значення пробного x, при якому $x^2 > kN$;

xm – значення пробного x, при якому $x^2 < kN$;

yp – значення $xp^2 - kN$;

ym – значення $kN - xm^2$ – додатне число при $x^2 < kN$.

Ці значення в подальшому використовуються часто. Тому на етапі підготовки до просіювання пробних x для фіксованого kN вони представлялися масивами розкладань за основою степенів простих чисел – елементів ЗФБ. Степені m простих p вибиралися з умов:

$$\begin{cases} p^m > 100, \\ p^m < 20000 \end{cases}.$$

При виконанні цих умов число елементів в масивах $mpxp[i][*]$, $mpxm[i][*]$, $mpyp[i][*]$, $mpym[i][*]$ (i – порядковий номер простого p в ЗФБ), що відповідають розкладанням xp, xm, yp та ym за основою p^m не перевищували 200 клітинок пам'яті для чисел N порядку 2^{1024}. Такі представлення xp, xm, yp та ym дозволяють в

подальшому відносно просто визначати дільники *yp* та *ym*, що є найбільш затратною за часом процедурою. Пошук дільників та виявлення *B*-гладких здійснюється так.

1. Для *yp* обчислити наближене значення $z_0 = \ln(yp)$ та присвоїти $z = z_0$.
2. В циклі по *i* для елементів для поточної факторної бази p_i з порядковим номером *j* в ЗФБ виконати операції:
   2.1. Обчислити $t = mpyp[j][1] + 2c \cdot mpxp[j][1] + c^2$.
   2.2. Перевірити, чи ділиться *t* на p_i без остачі. Якщо ні, то перейти до аналізу наступного простого *p* для поточної факторної бази. Якщо ж так, то перейти до кроку 2.3.
   2.3. Вияснити показник *m* степені *p* такий, що для остач від ділення *t* на p^m та *t* на p^{m+1} виконані умови: $t \,\%\, p^m = 0$, а $t \,\%\, p^{m+1} > 0$.
   2.4. Обчислити значення різниці $z_j = z - m \cdot \ln(p_i)$ та присвоїти $z = z_j$.
   2.5. Якщо множина елементів поточної факторної бази вичерпана, то перейти до п.3, а інакше перейти до аналізу наступного простого *p* для поточної факторної бази.
3. Якщо отримане значення *z* буде близьким до нуля, то отримано *B*-гладке число.

Описані алгоритми були реалізовані програмно на мові С для двох варіантів:

- для алгоритму базового методу квадратичного решета, коли для пошуку B-гладких використовується тільки співвідношення (1.4), число елементів факторної бази визначається згідно (2.1), а радіус інтервалу просіювання – згідно (2.2) (при пошуку B-гладких використано описаний вище алгоритм);
- для методу MQkS, коли для пошуку B-гладких використовується співвідношення (2.6), число елементів факторної бази визначається згідно (2.18), а радіус інтервалу просіювання – згідно (2.19).

Метод квадратичного решета працює в два етапи: на першому визначається множина B-гладких чисел, а на другому на їх основі визначаються нетривіальні множники *N*. Враховуючи те, що для різних алгоритмів різною буде множина B-гладких чисел, для можливості порівняння методів вирішувалася тільки задача

знаходження числа B-гладких, рівного L^a+3 при однакових значеннях *La*. Додатково проводилися розрахунки для методу MQkS при менших значеннях *La*.

2.6. Вплив розміру загальної факторної бази на час отримання достатньої кількості B-гладких.

Для методів QS і MPQS відомо, що при зниженні розміру ФБ збільшується розмір радіусу просіювання та може зростати час пошуку достатньої кількості B-гладких. Це характерно і для методу MQkS. Тому для методу MQkS важливо оцінити ступінь впливу розміру *fa* ЗФБ час пошуку достатньої кількості B-гладких. Для отримання таких оцінок проводилися чисельні експерименти, в яких постійними були значення параметрів *plb* = 1.4 та *h*=0.7. А для ряду фіксованих значень параметру *pla* визначалися значення параметру *kff* (*kff* = *j*/10, *j* > 3, але *kff* ≤ *pla*) при яких розрахунковий час *t* (сек) отримання достатньої кількості B-гладких для чисел *N* порядку 10^m при *m* = 20 ÷ 32 був найменшим. Дані обчислень представлені в табл. 2.9.

Таблиця 2.9

Значення найменшого часу розрахунку достатньої кількості B-гладких в залежності від *kff* для ряду значень параметра *pla*.

m	*pla* = 1.00		*pla* = 0.95		*pla* = 0.90		*pla* = 0.85		*pla* = 0.80	
	kff	*t* (c)	*kff*	*t* (c)	*kff*	*t* (c)	*kff*	*t* (c)	*kff*	*t* (c)
20	0.7	0.953	0.8	1.344	0.9	2.312	0.85	4.234	0.8	8.719
21	0.7	1.422	0.8	2.078	0.8	3.672	0.85	6.753	0.8	15.343
22	0.7	2.000	0.7	2.906	0.8	4.969	0.8	10.140	0.8	22.625
23	0.7	3.000	0.7	4.500	0.7	7.922	0.8	15.293	0.8	35.858
24	0.6	4.328	0.7	6.656	0.7	11.750	0.7	23.640	0.8	58.607
25	0.6	5.969	0.6	9.359	0.7	17.547	0.7	34.640	0.8	89.247
26	0.6	8.640	0.6	13.540	0.7	24.431	0.7	54.107	0.8	138.225
27	0.6	12.614	0.6	19.749	0.7	36.390	0.7	80.310	0.8	201.019
28	0.5	18.490	0.6	28.405	0.7	52.611	0.7	118.652	0.7	321.223

Продовж. табл. 2.9

29	0.5	25.514	0.6	41.171	0.6	82.833	0.7	170.076	0.7	455.547
30	0.5	36.389	0.6	58.623	0.6	112.50	0.6	268.829	0.7	702.967
31	0.5	50.077	0.6	83.538	0.6	162.32	0.6	362.158	0.7	1051.68
32	0.5	71.731	0.6	114.969	0.6	243.98	0.6	563.700	0.7	1538.51

Згідно отриманих даних, представлених в табл.2.9, при зменшенні значення параметра *pla* зростає час розрахунку для всіх аналізованих чисел N порядку 10^m при $m = 20 \div 32$. Має місце зростання часу при рості m, при чому темп зростання збільшується при зменшенні *pla*. Характерним також є зменшення значення параметра *kff* при збільшенні m. Тому для вибору розміру ЗФБ та параметру *kff* доцільно отримати оцінку обчислювальної складності алгоритму пропонованого методу MQkS з просіюванням на основі сигнальних остач.

2.7. Оцінка обчислювальної складності алгоритму формування достатньої кількості B-гладких з проріджуванням пробних X на основі сигнальних остач.

В описаному вище методі формування достатньої кількості B-гладких з проріджуванням пробних X на основі сигнальних остач використовуються описані вище параметри *pla*, *plb*, *h* та *kff*. На основі значення *pla* визначався розмір *fa* ЗФБ за формулою $fa = exp\left(pla \cdot \frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)$, а базове значення розміру радіусу просіювання *fb* за формулою $fb = exp\left(plb \cdot \frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)$. Такі варіанти визначення значень *fa* та *fb* корелюють з оцінкою обчислювальної складності виду $T(N) = O\left(exp(\,C\sqrt{ln\,N\,lnln\,N})\right)$ для методу QS та його модифікацій, включаючи і метод MQkS. Тому при оцінці обчислювальної складності можна скористатися наближеною формулою для часу розрахунку виду

$$T(m) \approx Ae^{C\sqrt{ln\,N\,lnln\,N}} = A \cdot \left(e^{\sqrt{ln\,N\,lnln\,N}}\right)^{\cdot C} = A \cdot (L^a)^{2\sqrt{2}\cdot C}, \qquad (2.21)$$

де A та C деякі постійні коефіцієнти. При логарифмуванні правої та лівої частини співвідношення (11) отримаємо

$$log_2(T(m)) \approx log_2(A) + C\sqrt{ln\,N\,lnln\,N} = log_2(A) + 2\sqrt{2\cdot}\,C \cdot log_2(L^a).$$

Якщо означити: $v_1 = log_2(A)$, $v_2 = 2\sqrt{2\cdot}\,C$, $a(m) = log_2(L^a)$, $b(m) = log_2(T(m))$, то для множини значень m можна записати рівняння:

$$v_1 + a(m) \cdot v_2 = b(m), \qquad (2.22)$$

корені якого v_1 та v_2 будуть визначатися з умови мінімального квадратичного відхилення.

При *plb* = 1.4 та *h*=0.7 значення коефіцієнта С (змінна v_2) визначалися на основі чисельних експериментів для таких варіантів значень параметрів *pla* та *kff* при зміні *m*:

В1). *kff* = 1, *kff* = 0.9, *kff* = 0.8, *kff* = 0.7, *kff* = 0.6, *kff* = 0.5 та *kff* = 0.4, де *pla* =1;

В2). *kff* = 1, *kff* = 0.9, *kff* = 0.8, *kff* = 0.7, *kff* = 0.6, *kff* = 0.5 та *kff* = 0.4, де *pla* =0.95;

В3). *kff* = 1, *kff* = 0.8, *kff* = 0.7, *kff* = 0.6, *kff* = 0.5 та *kff* = 0.4, де *pla* =0.9;

В4). *kff* = 1, *kff* = 0.8, *kff* = 0.7, *kff* = 0.6, *kff* = 0.5 та *kff* = 0.4, де *pla* =0.85;

В5). *kff* = 1, *kff* = 0.7, *kff* = 0.6, *kff* = 0.5 та *kff* = 0.4, де *pla* =0.8.

Отримані на основі чисельних експериментів дані про час розрахунку достатньої кількості В-гладких для варіанту В1) представлені в табл. 2.10.

Таблиця 2.10

Час розрахунку достатнього числа В-гладких при *pla*=1.0, *plb*=1.4, *h* =0.7, різних значеннях параметра *kff* та *m* = 20 ÷ 32.

m	*kff*						
	0.4	0.5	0.6	0.7	0.8	0.9	1.0
20	1.156	1.079	0.969	0.953	0.969	1.031	1.125
21	1.719	1.578	1.468	1.422	1.468	1.562	1.718
22	2.344	2.125	2.016	2.000	2.063	2.250	2.500
23	3.405	3.171	3.015	3.000	3.125	3.370	3.813
24	4.844	4.469	4.328	4.500	4.515	4.906	5.609
25	6.687	7.814	5.969	6.156	6.344	6.906	7.922
26	9.422	8.812	8.640	8.702	9.140	10.047	11.546

Продовж. табл. 2.10

27	13.421	12.687	12.614	12.678	13.250	14.765	17.093
28	19.389	18.490	17.921	18.281	19.392	21.859	25.265
29	27.233	25.514	25.171	25.843	27.500	31.435	36.514
30	38.150	36.389	35.811	36.483	39.561	44.722	53.022
31	51.530	50.077	50.092	51.940	55.654	63.353	76.013
32	75.091	71.731	71.937	73.664	80.340	97.621	110.512

При формуванні системи рівнянь (2.22) обчислювалися значення логарифму за основою 2 для значень рекомендованого в [112] розміру ФБ $L^a(m) = exp\left(\frac{\sqrt{2}}{4}\sqrt{ln\,N\,(m)\cdot lnln\,N\,(m)}\right)$, де N(m) – це числа порядку 10^m при m = 20 ÷ 32. Для значень часу, наведених в табл. 8 також обчислювався логарифм за основою 2 від отриманого часу розрахунку.

При вирішенні системи лінійних рівнянь з двома невідомими та більшою кількістю рівнянь виявилося, що при різній кількості рівнянь отримуються різні значення коефіцієнта С. Тому для даних кожного зі стовпчиків даних табл.8 формувалися ряд систем рівнянь, починаючи з деякого рядка, що відповідає значенню $m = m_0$ таблиці 2.10 до рядка, що відповідає m = 32. В кожному випадку m_0 та *pla* отримували різні значення коефіцієнта *C*. Серед них вибирали те, при якому різниці між даними табл. 2.10 та отриманими значеннями часу постійно змінювали свій знак. В результаті проведених порівнянь значень С були виділені його значення, при яких розрахунковий час для *m*=32 менший (С(-)) та більший (С(+)) за табличне значення. Для даних табл. 2.23 отримані значення С(-) та С(+) наведено в табл.2.11.

Таблиця 2.11 **Значення коефіцієнтів С(-) та С(+) при *pla* = 1.0.**

kff	0.4	0.5	0.6	0.7	0.8	0.9	1.0
С(-)	-	-	-	-	-	-	1.04598
С(+)	0.95765	0.97725	0.995949	0.990047	1.01645	1.07049	1.04163

Отримані на основі чисельних експериментів дані про час розрахунку достатньої кількості В-гладких для варіанту В1) представлені в табл. 2.12.

Таблиця 2.12

Час розрахунку достатнього числа В-гладких при *pla*=0.95, *plb*=1.4, *h* =0.7, різних значеннях параметра *kff* та *m* = 20 ÷ 32.

m	*kff*						
	0.4	0.5	0.6	0.7	0.8	0.9	0.95
20	1.781	1.610	1.406	1.375	1.344	1.407	1.484
21	2.641	2.484	2.156	2.188	2.078	2.187	2.297
22	3.656	3.219	2.984	2.906	2.937	3.172	3.315
23	5.516	4.925	4.563	4.500	4.551	4.920	5.234
24	7.905	7.078	6.656	6.656	6.734	7.281	7.765
25	10.891	10.109	9.359	9.577	9.812	10734	11.531
26	15.640	14.375	13.540	13.901	14.328	15.828	16.972
27	22.390	21.225	19.749	20.390	20.962	23.202	25.202
28	32.561	30.843	28.405	29.026	30.592	34.350	37.686
29	47.123	43.230	41.171	42.030	45.030	51.467	57.042
30	65.029	59.373	58.623	58.748	63.040	71.700	79.528
31	92.685	85.601	83.538	85.341	90.814	105.998	115.653
32	125.293	117.658	114.969	119.825	129.433	149.511	165.775

Значення C(-) та C(+), отримані на основі даних табл. 2.10, наведено в табл. 2.13.

Таблиця 2.13.

Значення коефіцієнтів C(-) та C(+) при *pla* = 0.95.

kff	0.4	0.5	0.6	0.7	0.8	0.9	0.95
C(-)	0.957962	0.960859	0.985946	1.00466	1.02279	1.04022	1.05352
C(+)	0.950964	0.958305	0.979411	1.00113	1.01919	1.03619	1.05073

Отримані на основі чисельних експериментів значення C(-) та C(+) для варіантів B3), B4) та B5) наведені в табл. 2.14.

Таблиця 2.14.

Значення коефіцієнтів C(-) та C(+) при *pla* = 0.9, *pla* = 0.85 та *pla* = 0.8.

pla	*kff*	0.4	0.5	0.6	0.7	0.8	0.85	0.9
0.9	C(-)	-	-	-	1.07631	1.06094		1.06763
	C(+)	1.05159	1.04601	1.05925	1.07416	1.06032		1.01483
0.85	C(-)	-	-	-	1.12124	-	1.13575	
	C(+)	1.01827	1.03077	1.09658	1.10882	1.12937	1.12172	
0.8	C(-)	1.03691	1.04564	-	1.13696	1.15366		
	C(+)	1.02762	1.04291	1.07915	1.12591	1.15128		

В табл. 2.11 та 2.14 наявні пропуски, оскільки при різних варіантах формування системи рівнянь (12) не було жодного, при якому прогнозований час розрахунку достатньої кількості B-гладких для чисел порядку 10^{32} перевищував би отримане значення часу. Це може означати, що значення коефіцієнта C, що використовується при оцінці обчислювальної складності є більшим за C(+).

На основі аналізу даних, наведених в табл. 2.11, 2.13 та 2.14, можна зробити висновок, що коефіцієнт C може приймати середнє значення між C(+) та C(-) та може бути меншим за одиницю, що підтверджено результатами розрахунків для відносно малих чисел. Узагальнити такий висновок на довільні числа N немає можливості, оскільки оцінки його значення отримані на основі чисельних експериментів над обмеженою множиною відносно малих чисел. Проте плавний характер зміни C(+) та C(-) при зміні *kff* та *pla* дозволяє вважати правдоподібною можливість отримання модифікованого методу квадратичного решета, для алгоритму якого коефіцієнт C виявиться меншим за одиницю.

Для аналізованих чисел N при *pla* = 0.94 при пошуку достатньої кількості B-гладких отримуємо C(+) = 0.989129, C(-) = 0.99512. При цьому розмір матриці визначається як $(L^a)^{0.94} = exp\left(0.94 \cdot \frac{\sqrt{2}}{4} \cdot \sqrt{ln\,N \cdot lnln\,N}\right)$. Обчислювальна складність

методу Гауса при вирішенні матриці буде величиною порядку $O\left(exp\left(3 \cdot 0.94 \cdot \frac{\sqrt{2}}{4} \cdot \sqrt{ln\,N \cdot lnln\,N}\right)\right) = O\left(exp\left(0.997021 \cdot \sqrt{ln\,N \cdot lnln\,N}\right)\right)$, тобто і для методу MQkS в цілому значення C виявляється меншим за одиницю. Хоча таку оцінку слід вважати евристичною.

2.8. Висновки до розділу 2.

Розроблено метод множинного квадратичного *k*- решета (MQkS), в якому для пошуку B-гладких остач використовуються остачі $y_k(X)=X^2-kN$, що при більшості значень *k* забезпечує пошук B-гладких серед всіх пробних $X = X_0 + x = \left[\sqrt{N} + 1\right] + x$ в єдиному інтервалі просіювання без обмежень на *x*, в якому, на відміну від методів QS та MPQS:

- використовується загальна факторна база (ЗФБ), утворена всіма найменшими простими числами починаючи з 2, кількість яких $fa = \left(exp\left(\frac{\sqrt{2}}{4}\sqrt{ln\,N \cdot lnln\,N}\right)\right)^{pla} = (L^a)^{pla}$, де, $pla \in [0.5, 1.5]$ – параметр, а при кожному зі значень *k* з елементів ЗФБ формується ПФБ;
- розмір радіусу просіювання $fb = \left(L^a\right)^{plb}$, де $plb \in [0.5, 4]$ – параметр;
- на етапі просіювання пробних *X* реалізується попереднє просіювання на основі використання сигнальних остач $y^*(X)$, що є добутками перших степенів дільників $y_k(X)$ з числа елементів ЗФБ, при якому до множини відсіяних *X* відносяться ті, для яких виконана умова $log(y_k^*(X)) < h \cdot log(y_k(X))$, де $h \in [0, 1]$ – параметр;
- при просіюванні пробних *X*, які не були відсіяні, пошук дільників остач $y_k(X)$, показник степеня яких може перевищувати одиницю, здійснюється для простих чисел з поточної факторної бази за умови, що для порядкового номера f_p простого p у списку простих чисел виконана умова $f_p \le ff = (L^a)^{kff}$, де $kff \in [0, 1]$ – параметр;
- максимальне значення дільника $y_k(X)$, що є степенем простого числа *p*, номер якого $f_p \le ff$, не перевищує значення B^λ, де λ – параметр, значення якого

вибирається на основі даних про обмеження на обсяг пам’яті та доступні стандартні типи даних апаратних засобів;

- при пошуку нульового рядка матриці, елементи якої дорівнюють одиниці для непарних показників степеня дільників B-гладких чисел при рівних нулю інших значеннях, за рахунок перенумерації стовпчиків замість двох матриць з числом стовпчиків, що дорівнює *fa*, використовується одна, за рахунок чого розмір необхідної пам’яті суперкомп’ютера можна скоротити вдвічі.

При значеннях параметрів *pla* 0.9÷0.94, *plb* =1.4, h=0.7, *kff* = 0.4÷0.6 та λ=1 для визначеної множини чисел порядку 10^m, де m=20÷32, отримано значення коефіцієнту $C < 1$ в оцінці складності методу MQkS виду $O\left(exp\left(C\sqrt{ln\,N\,lnln\,N}\right)\right)$. У відомих оцінках обчислювальної складності методів QS та MPQS коефіцієнт $C \geq 1$. Для аналізованої множини чисел N порядку 10^m, де m=9÷32 встановлено також, що в порівнянні з методом QS кількість пробних *X*, на основі яких шукають B-гладкі, в 6 та більше разів перевищує їх кількість для аналогічного числа пробних в методі MQkS та зменшується час пошуку B-гладких.

РОЗДІЛ 3. РІШЕННЯ СЛАУ ДЛЯ МАТРИЦІ КВАДРАТИЧНОГО РЕШЕТА.

3.1. Методи рішення систем лінійних рівнянь.

У результаті виконання кроку просіювання у методі Квадратичного решета було знайдено послідовність B-гладких чисел

$$a_j^2 \equiv \prod_{i=1}^{m} p_i^{b_{ij}} \ (mod\ M) \quad (1 \le j \le n).$$

Тут p_i прості числа (або -1) та b_{ij} степінь простих чисел, які у більшості випадках дорівнюють нулю. QS намагається знайти $S \subseteq \{1, 2, \dots, n\}$, такий щоб дві сторони рівняння а $\prod_{j \in S} a_j^2 \equiv \prod_{j \in S} \prod_{i=1}^{m} p_i^{b_{ij}} \ (mod\ M)$ є повними квадратами. Ліва сторона автоматично є квадратом, але права сторона буде квадратом, тоді коли всі степені простих чисел парні, тобто $\sum_{j \in S} b_{ij} \equiv 0 \ (mod\ 2)$ для $1 \le i \le m$. Це є еквівалентом для $Bx \equiv 0 \ (mod\ 2)$, де $B = \left(b_{ij}\right), x = (x_j)$, та де $x_j = 1$ якщо $j \in S$ та $x_j = 0$ якщо $j \notin S$.

Тобто матриця отримана шляхом з'єднання строчок, кожна з яких представляє собою вектор розкладання j-го гладкого числа по факторній базі по модулю 2. Система має як мінімум k-m нетривіальних рішень. Особливістю системи є її сильна розрідженість, особливо в правій її частині, яка відповідає степеням старших простих чисел з факторної бази.

Така система $Bx = 0$ з коефіцієнтами з поля $F_2 = \{0,1\}$ може бути вирішена за допомогою звичайного метода Гауса, що потребує mn бітів.

Адже метод потребує застосування додаткової одиничної матриці. Наприклад, якщо необхідно розв'язати матрицю 5×8, необхідно застосовувати одиничну матрицю 8×8.

$$\begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & \quad 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix} \end{array}$$

Коли матриця B розріджена, застосовують структурований метод Гауса [10], який заміняє матрицю B на щільну матрицю у якій втричі менша кількість рядків та стовпців.

Гігабайта пам'яті не вистачає для збереження матриці $10^5 \times 10^5$, в той час як необхідно розв'язувати матриці розміром $10^6 \times 10^6$ та більше.

Тому для рішення відповідної системи рівнянь використовуються спеціальні методи рішення розріджених систем ЛУ.

LaMacchia та Odlyzko [131] реалізували варіант методів Ланцоша та метода сполучених градієнтів. Ці методи ітераційно множать симетричну матрицю $n \times n$ на вектор. Вони використовують тільки декілька тимчасових векторів та початкову матрицю, таким чином полегшуючи проблему нестачі пам'яті. Ці методи були розроблені для застосування з реальними матрицями, але працюють з іншими полями доти поки не зустрінуть вектор ортогональний сам до себе. Для уникнення векторів ортогональних до себе, вони працюють з розширенням поля $F_2 = \{0,1\}$, аніж з $F_2 = \{0,1\}$ на пряму.

Wiedemann [132] запропонував інший ітераційний метод. Його алгоритм множить $n \times n$ матрицю B на вектор приблизно 2n разів, та формує невеликий многочлен B. Використовуючи цей мінімальний многочлен, можна знайти вектори в нульовому просторі B, якщо B є виродженою. Метод так само потребує місця тільки для матриці B, та для декількох додаткових векторів.

Пітер Монтгомері [133, 134] застосував метод Ланцоша для рішення СЛАУ у методі решета числового поля. Опис цього методу можна знайти у огляді [147].

Всі методи Ланцоша, сполучених градієнтів, та Wiedemann множать

симетричну матрицю $n \times n$ на $O(n)$ векторів. Та можуть бути застосовані тільки для симетричних матриць. Інших модифікацій які б стосувалися вирішення матриці не було виявлено.

Проаналізувавши наведені методи оптимізації було запропоновано ряд модифікацій які направлені на прискорення рішення СЛАУ для методу квадратичного решета та поліноміального квадратичного К-решета, які представлені у даному розділі.

Викладений нижче матеріал представлений в наступному порядку:

– метод рішення матриці «на ходу»;
– зменшення об'єму пам'яті при рішення СЛАУ.

3.2. Метод рішення матриці «на ходу».

У рамках цього дослідження будемо вважати вирішеною задачу пошуку розмірів інтервалу просіювання та факторної бази, де факторна база містить L^a елементів.

В пропонованому алгоритмі, що реалізує вирішення матриці на ходу, використовується додатковий вектор $Vs[L^a+1]$.

Пошук нульового вектора для матриці степенів представлені нижче наступними кроками:

1. При прояві нового B-гладкого числа, вектор степенів *Vnew*, який йому відповідає, заноситься у матрицю.

2. Обчислюється позиція *k0* першого не нульового значення вектора *Vnew* та позиція самого вектора у матриці – *kv*.

a. Якщо нульове значення вектора *Vnew* відсутнє, то нульовий вектор знайдено. Перехід до пункту 4.

3. Перевіряється елемент вектора *Vs* з номером, який дорівнює першому не нульовому значенню доданого вектора *Vnew – k0*.

a. Якщо елемент пустий, тоді заноситься в цей елемент позицію доданого вектора *Vnew – kv: Vs[k0] = kv*. Переходимо до пункту 1.

b. Якщо цей елемент не пустий, то за допомогою значення в цьому елементі

знаходиться рядок, який необхідно додати до вектора *Vnew*. Перехід до пункту 2.

4. Вияснюється, чи отримане значення кореня не дорівнює *N*. Якщо ні, то задача факторизації вирішена, а інакше здійснюється перехід до пункту 5.

5. Видаляється інформацію про *B*–гладке число з нульовими значеннями елементів перетвореної матриці. Здійснюється перехід до пункту 1.

3.2.1. Приклади застосування алгоритму вирішення матриці «на ходу».

Розглянемо на прикладі ефективність запропонованої модифікації.

Приклад 1. Оберемо *p=401* та *q=103*, ці прості числа утворюють число для факторизації *p*q=N=41303*. Обчислимо за формулою (2.1) розмір факторної бази *A=6*. За допомогою формули (2.2) отримаємо інтервал просіювання *M=203*. Початкове значення $X0 = xx = \sqrt{N} = 204$.

Після просіювання варіантів *y(x)* через факторну базу отримуємо *B*-гладкі числа. Ці числа зображені у табл. 3.1.

Таблиця 3.1.

Результати просіювання варіантів *y(x)*

Вектор степенів *B*-гладких чисел							*B*-гладкі
Знак числа	2	11	19	23	29	37	
1	1	1	0	0	2	0	-18502
1	**0**	**1**	**0**	**0**	**0**	**2**	**-15059**
1	**1**	**0**	**0**	**1**	**0**	**1**	**-1702**
0	1	0	2	0	0	0	722
0	0	0	0	2	1	0	15341
0	**1**	**1**	**0**	**1**	**0**	**1**	**18722**

Для базового алгоритму квадратичного решета цих чисел недостатньо для формування матриці та отримання рішення.

Зауважимо, що вектори, виділені жирним в табл. 2.1, створюють нульовий вектор.

Метод вирішення матриці на ходу зміг знайти нульовий вектор при поточній кількості *B*-гдадких чисел та факторизувати *N*.

Приклад 2. Нехай *p*=7624217, *q*= 98269, *N* = 749224180373, розмір факторної бази: $L^a = 29$, інтервал просіювання: $L^b = 24052$, початкове значення $X0 = xx = \sqrt{N}$

= 865578.

Елементи факторної бази: 2, 7, 11, 19, 23, 31, 37, 41, 43, 61, 67, 71, 101, 127, 131, 157, 163, 167, 173, 179, 181, 191, 193, 211, 223, 227, 229, 241, 263.

Базовому методу квадратичного решета знадобилося перевірити 5535 варіантів *x*, щоб отримати 30 *B*-гладких і факторизувати число.

Як видно з табл. 3.2, модифікований метод перевірив 194 варіанта *x*, отримавши всього 3 *B*-гладких.

Таблиця 3.2

Результати просіювання варіантів *x*

X	Y	Елементи факторної бази																													
865615	65147852	0	2	2	2	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
865533	-76806284	1	2	0	0	1	0	0	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	1
865481	-166819012	1	2	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1

Цього виявилося достатньо для отримання нульового вектора.

Модифікований алгоритм отримав рішення у *тридцять* раз швидше ніж базовий алгоритм.

3.2.2. Аналіз ефективності метода рішення матриці «на ходу».

У табл. 3.3 наведені приклади отриманих коефіцієнтів прискорення модифікованого алгоритму з приблизним кроком 20. Прискорення вираховується відношенням кількості просіяних *X* базовим методом до кількості просіяних *X* модифікованим методом.

Таблиця 3.3

Порівняльна характеристика модифікованого методу квадратичного решета зі базовим методом квадратичного решета.

P	Q	Кількість X просіяних модифікованим методом	Кількість X просіяних базовим методом	L^a	Кількість отриманих B-гладких модифікованим методом	Кількість отриманих B-гладких базовим методом	Прискорення раз.
152065567	21386557	7739	7739	54	59	59	1
24359	75511	62	655	18	7	19	10.5
170537	47087	37	769	20	4	21	20.8
44351	60343	33	1125	18	3	19	34.0

Продовж. табл. 3.3

111121	41381	20	1377	19	3	20	68.8
853529	333367	226	19025	27	3	22	84.1
51749	32609	3	305	18	2	19	101.6

Метод вирішення матриці на ходу в ряді випадків дозволяє сформувати нульовий вектор з уже отриманих векторів для B-гладких чисел навіть тоді, коли базовий метод не зміг отримати достатньої кількості B-гладких для формування матриці. Так для одного з чисельних експериментів було обрано 200 простих чисел на інтервалі від 23663 до 152065567 з плаваючим кроком та згенеровано десять тисяч N для факторизації. Базовий алгоритм не зміг факторизувати у 686 випадках з 10000. Модифікований алгоритм зменшив цю цифру до 503 випадків. Час факторизації всіх вдалих випадків теж зменшився з 3941 секунд до 3301 секунд, що складає 16 відсотків.

Факторизація базовим методом квадратичного решета одного числа розміром 10^{22} може займати до 2 хвилин, тому проведення досліджень для великої кількості чисел значно більшого розміру не проводилося, оскільки потребує великих обчислювальних ресурсів. Для аналізу ефективності були обрані контрольні діапазони для N, близьких до $log\ _{10}N^{12+(k+2)}$, де k=0,1,..,5. Для кожного k було сформовано 10000 варіантів N. p та q обирались за формулою $p = i * 10^{log_{10}N \mp j} + 200 * f$, де $i = 1,2, \dots, 9; j = 0, \dots, 5; f = 0,1, \dots, 200$. На основі отриманих результатів здійснювалася їх екстраполяція на числа порядку до 10^{130}.

Аналіз ефективності проводився за декількома ознаками:

1. Кількість просіяних X для базового та модифікованого методу.

Відносне зменшення загальної кількості просіяних X (в процентах), дано у табл. 3.4.

Таблиця 3.4

Процент прискорення модифікованого методу квадратичного решета відносно базового методу квадратичного решета, відповідно до розміру числа що факторизується

lgN	%
14	13.06974138
16	12.20758206
18	10.23803938
20	9.047552192
22	8.337249169

Оскільки процедура просіювання пробних значень x є найбільш затратною за часом, то оцінка числа просіяних x є важливою складовою, що характеризує ефективність алгоритму, і така оцінка є точною. При цьому неможливо врахувати час на повторні вирішення матриці за умови появи хибних рішень. Тому оцінка за кількістю просіяних X не є повною.

2. Загальний час виконання завдання факторизації. Вирахуваний процент зменшення загального часу виконання програми факторизації дано в табл. 3.5.

Таблиця 3.5

Процент прискорення модифікованого методу квадратичного решета відносно базового методу квадратичного решета, відповідно до розміру числа, що факторизується

lgN	%
14	15.42857143
16	12.47148289
18	11.89327278
20	10.65345846
22	8.647172602

Слід відмітити, що оцінка за часом розрахунку не є точною. Одне і те ж завдання може виконуватися на протязі різного (близького за значенням) часу, що пов’язано з роботою планувальника задач операційної системи. Тому наведені в таблиці 3.5 дані є наближеними, які отримані на основі одного розрахунку для кожного з методів. Оцінка похибки таких даних не проводилася, оскільки кожен з

розрахунків потребував значного часу, що робить неможливим проведення статистично значимого числа експериментів.

За отриманими результатами з табл. 3.4 та 3.5 було сформовано функції методом найменших квадратів,

$$T = \frac{194.39}{\lg N} - 0.49, \tag{3.1}$$

та

$$T = \frac{134.57}{\lg N} + 4.43,$$

відповідно.

На рис. 3.1 зображено графік цих функцій.

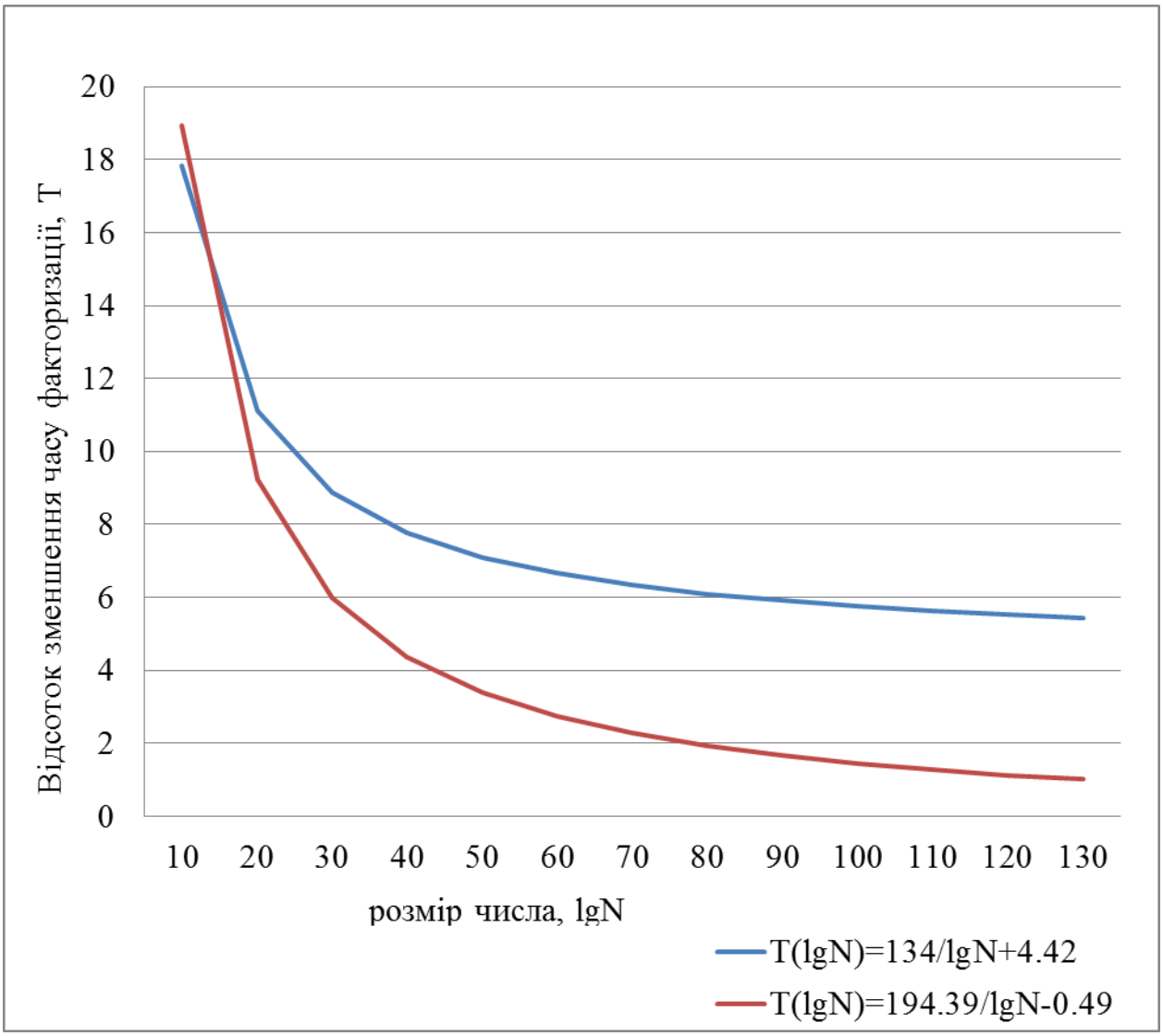


Рисунок 3.1 – Графік прискорення методу рішення матриці на ходу

Для проведення подальшої оцінки будемо використовувати формулу (3.1), оскільки порівняння за часом виконання хоч і має похибки, але враховує час на знаходження p та q. Тому за формулою (3.1) отримаємо, що для чисел розміром 10^{100} модифікований алгоритм квадратичного решета на основі вирішення матриці на ходу має прискорення приблизно 5.76 відсотків, та для чисел розміром 10^{130} – 5.45 відсотків.

3.2.3. Порівняльна оцінка складності методу квадратичного решета із застосуванням рішення СЛАУ «на ходу» з решетом числового поля.

Для порівняння відносної ефективності методів QS і GNFS приймемо, що

$$T^*_{QS}(N) = \exp((\ln N \ln\ln N)^{1/2}),$$

$$T^*_{GNFS}(N) = \exp\left((\ln N \ln\ln N)^{\frac{1}{2}}\right),$$

$$T^*_{QS}(N) = K * T^*_{GNFS}(N),$$

де коефіцієнт К визначимо з умови, що метод QS кращий за GNFS для чисел 10^{110}, але при $N > 10^{129}$ кращим буде GNFS, а при деякому $10^{110} < N < 10^{129}$ співпадають відношення

$$\frac{T^*_{QS}(N)}{T^*_{GNFS}(N)} = \frac{T_{qs}(N)}{T_{GNFS}(N)}.$$

Обчислимо значення $T^*_{QS}(N)$, $T^*_{GNFS}(N)$ та $T_K(N) = T^*_{GNFS}(N)/T^*_{QS}(N)$, значення якої дозволить охарактеризувати коефіцієнт K [18, 19].

Таблиця 3.6

Порівняльні дані для методів QS та GNFS

lg$N$	$T^*_{QS}(N)$	$T^*_{GNFS}(N)$	$T_K(N) = T^*_{GNFS}(N) / T^*_{QS}(N)$
110	1.8209e+016	3.4122e+016	0.53365
111	2.2252e+016	3.9907e+016	0.5576
112	2.7172e+016	4.6635e+016	0.58266
113	3.3154e+016	5.4452e+016	0.60887
114	4.0422e+016	6.3527e+016	0.63629
115	4.9245e+016	7.4056e+016	0.66498
116	5.995e+016	8.6261e+016	0.69498
117	7.2927e+016	1.004e+017	0.72637

Продовж. табл. 3.6

118	8.8648e+016	1.1677e+017	0.7592
119	1.0768e+017	1.357e+017	0.79355
120	1.3071e+017	1.5758e+017	0.82947
121	1.5854e+017	1.8285e+017	0.86706
122	1.9218e+017	2.1203e+017	0.90637
123	2.3278e+017	2.4568e+017	0.9475
124	2.8177e+017	2.8447e+017	0.99052
125	3.4085e+017	3.2915e+017	1.0355
126	4.1203e+017	3.8059e+017	1.0826
127	4.9776e+017	4.3977e+017	1.1319
128	6.0093e+017	5.0781e+017	1.1834
129	7.2501e+017	5.8598e+017	1.2373
130	8.7416e+017	6.7574e+017	1.2936

З табл. 3.6 слідує, що $T_K(N)$ росте при збільшенні числа десяткових розрядів N. Проте цікавим є той факт, що при збільшенні числа десяткових розрядів N на одиницю, $T_K(N)$ збільшується в 1.045 раз для lg$N$ =113 та досягає величини 1.0459 для lg$N$ = 160, при плавному монотонному рості. Тобто незалежно від граничного значення lg$N$, при якому методи QS та GNFS мають однакову обчислювальну складність, довільний варіант удосконаленого методу QS, для якого граничне значення lg$N$ збільшиться на одиницю, потребує, щоб його обчислювальна складність була знижена не менше ніж в 1.045 рази.

Для проведених розрахунків було обрано $o(1) = 1$. Можна, стверджувати з повною впевненістю, що для $o(1) \neq 1$ динаміка буде аналогічна тій, яка наведена вище.

3.3. Метод діагоналізації матриці «на ходу» зменшеного розміру.

3.3.1. Алгоритм діагоналізації матриці «на ходу» зменшеного розміру.

Проаналізувавши методи прискорення квадратичного решета які були розглянуті у, були виявлені модифікації які спрямовані на прискорення вирішення матриці.

А саме у [133] описано, що кількість нулів у матриці степенів значно більша за кількість одиниць. Для великих чисел розміром 10^{100} та більше відношення кількості нулів до кількості одиниць тільки зростає. Більша частина пам'яті виділена для зберігання матриці використовується для зберігання нулів. Тому замість зберігання двомірної матриці запропоновано зберігати тільки позиції одиниць.

Всі методи Ланцоша, сполучених градієнтів, та Wiedemann множать симетричну матрицю $n \times n$ на $O(n)$ векторів. Та можуть бути застосовані тільки для симетричних матриць.

Метод рішення матриць «на ходу» застосується для не симетричних матриць та базується на методі Гауса, який використовується для матриць малого розміру.

Для зменшення необхідної кількості пам'яті пропонується застосовувати метод рішення матриці «на ходу» зменшеного розміру, який потребує тільки збереження початкової матриці та одного тимчасового вектора, який може бути застосований для не симетричної матриці.

В рамках загального опису алгоритму будуть використані такі параметри:

A_list – список в якому міститься інформація про B-гладкі. Це двобічно зв'язаний список з елементами в яких міститься два масиви:

prime_num – номери простих які мають не нульовій степінь,

power – степінь простих чисел відповідно до масиву prime_num.

Елементи списку створюються динамічно під кожне B-гладке число.

A - матриця використовується для проведення операцій з пошуку нульового вектору. Строки у матриці виділяються динамічно під кожне отримане B-гладке. Розмір матриці становить $m \times j$ де m - розмір факторної бази, j – номер отриманого B-гладкого.

v_pos – масив розміром m, для запам'ятовування номерів стовпчиків.

Загальний алгоритм рішення матриці зменшеного розміру:

1. j=j+1;
2. Створення нового рядка у матриці A, та заповнення його інформацією про B-

гладке.

3. Створення нового елемента списку A_list та заповнення його інформацією про B-гладке.
4. Замінимо коефіцієнти матриці їх остатками за модулем 2.
5. Для отримання на головній діагоналі одиниць будемо виконувати еквівалентні перестановки:
    a. стовпчиків, з відповідним збереженням наших дій у векторі v_pos.
    b. після отримання одиниці у клітинці на головній діагоналі, проводимо еквівалентні перетворення зі строками (додавання провідного рядку до нижчих за модулем 2) для отримання нуля під одиницею у клітинці на головній діагоналі. Після отримання цього нуля, у клітинах строк до яких додавався провідний рядок ставимо одиницю. При пошуку нульового вектора не будемо зважати на одиниці під головною діагоналлю.
    c. Після будь-якого додавання проводиться перевірка на нульовий вектор. Якщо вектор нульовий переходимо до кроку 7. Якщо ні, продовжуємо перестановки.
6. Після отримання головної діагоналі. Для отримання нульового вектора, обирається рядок який не належить до головної діагоналі. Будь-який рядок можна перетворити на нульовий вектор.
7. Після отримання нульового вектору одиниці які залишилися під головною діагоналлю вкажуть на вектори, які додавалися до поточного вектора.
8. A_list надасть інформацію про початкові показники степенів простих множників B-гладких чисел які утворюють нульовий вектор.

Представлена блок-схема діагоналізації матриці зменшеного розміру.

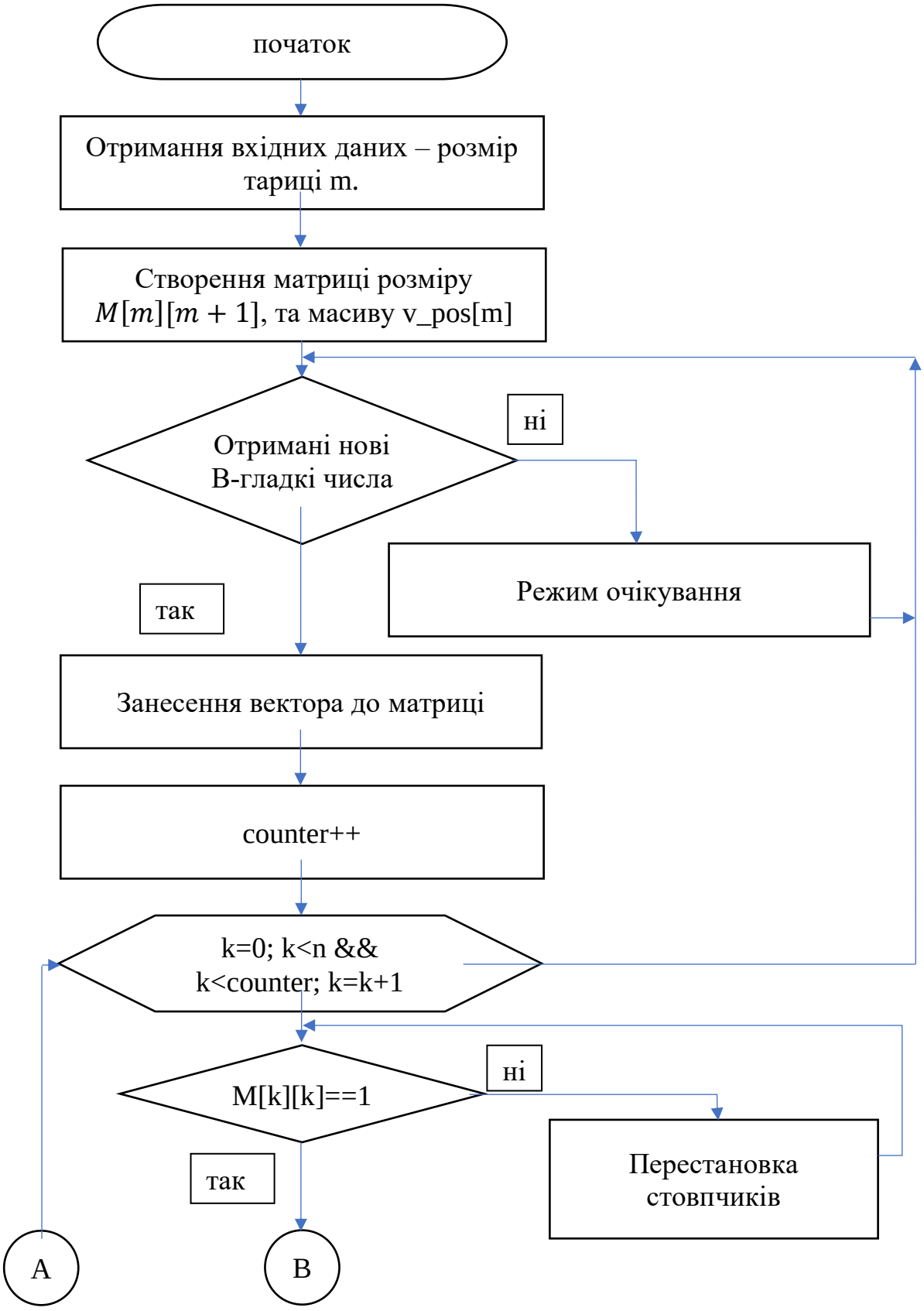


Рисунок 3.2 – Схема алгоритму діагоналізації матриці зменшеного розміру частина 1.

Рисунок 3.3 – Схема алгоритму діагоналізації матриці зменшеного розміру частина 2.

3.3.2. Приклад діагоналізації матриці зменшеного розміру.

Виберемо наступну матрицю як вихідну.

$$\begin{array}{c} 1\ \ 2\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Перший елемент в першому стовпчику нульовий, тому знайдемо перший не нульовий елемент в цьому рядку. Якщо такого нема – ми знайшли нульовий вектор. У нашому випадку не нульовий елемент знаходиться у другому стовпчику.

Зробимо заміну першого та другого стовпчика.

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Це перша знайдено одиниця у процесі діагоналізації матриці.

З метою приведення матриці до трикутного виду додамо перший рядок до сьомого та восьмого.

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Елементи матриці під одиницею у першому стовпчику дорівнюють нулю та можуть бути використані для збереження інформації про додавання строк.

Строки до яких додавався перший рядок помічаються одиницею.

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Другий елемент у другому стовпчику дорівнює одиниці

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Додамо другий рядок до всіх рядків які мають одиницю у другому стовпчику та знаходяться нижче чим другий рядок.

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Строки до яких додавався другий рядок помічаються одиницею.

$$\begin{array}{c} 2\ \ 1\ \ 3\ \ 4\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Третій елемент у третьому рядку не нульовий. Тому поміняємо місцями третій та четвертий стовпчик. Тепер третій елемент у третьому рядку дорівнює одиниці.

$$\begin{array}{ccccc} 2 & 1 & 4 & 3 & 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Додамо тертій рядок до всіх рядків які мають одиницю у третьому стовпчику та знаходяться нижче чим третій рядок.

$$\begin{array}{ccccc} 2 & 1 & 4 & 3 & 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Строки до яких додавався третій рядок помічаються одиницею.

$$\begin{array}{ccccc} 2 & 1 & 4 & 3 & 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Четвертий елемент у четвертому стовпчику дорівнює одиниці.

$$\begin{array}{c} \begin{matrix} 2 & 1 & 4 & 3 & 5 \end{matrix} \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Додамо четвертий рядок до всіх рядків які мають одиницю у четвертому стовпчику та знаходяться нижче чим четвертий рядок.

$$\begin{array}{c} \begin{matrix} 2 & 1 & 4 & 3 & 5 \end{matrix} \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Строки до яких додавався четвертий рядок помічаються одиницею.

$$\begin{array}{c} \begin{matrix} 2 & 1 & 4 & 3 & 5 \end{matrix} \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

П’ятий елемент у п’ятому стовпчику дорівнює одиниці.

$$\begin{array}{c} \begin{matrix} 2 & 1 & 4 & 3 & 5 \end{matrix} \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & \mathbf{1} \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Додамо п’ятий рядок до всіх рядків які мають одиницю у п’ятому стовпчику та знаходяться нижче чим п’ятий рядок.

$$\begin{array}{c} 2\ \ 1\ \ 4\ \ 3\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & \mathbf{1} \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Строки до яких додавався п’ятий рядок помічаються одиницею.

$$\begin{array}{c} 2\ \ 1\ \ 4\ \ 3\ \ 5 \\ \hline \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 1 \\ 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & 0 & \mathbf{1} \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Строки які знаходяться нижче головної діагоналі є нульовими векторами, а одиниці в цих векторах свідчать про додавання відповідного рідка. Переміщення стовпчиків збережені у масиві розміром n.

3.4. Використання розширеної факторної бази та формування достатньої кількості B - гладких чисел.

Етап просіювання для методу QS проводиться до формування множини B-гладких у кількості не менше ніж $L^a + 2$, де L^a - розмір факторної бази.

Розмір факторної бази та інтервал просіювання це евристичні значення, які обчислюються за формулами які описані у Першому розділі.

Алгоритм дуже чутливий до змінення розмірів факторної бази та інтервалу просіювання. При зменшенні розміру факторної бази для методу QS призводить до збільшення часу обчислення яке пов’язане з пошуком B-гладких чисел. Збільшення розміру факторної бази призводить до збільшення необхідної кількості B-гладких, але

при цьому суттєво збільшується час на рішення матриці (наступний етап алгоритму) також збільшується розмір необхідної пам’яті.

Ідея досліджень, результати яких подаються в даному розділі, полягає у використанні суттєво меншої кількості B-гладких чисел при збереженні розміру факторної бази, за рахунок чого можливе суттєве зниження часу факторизації тобто й зменшення інтервалу просіювання. Використання меншої кількості B-гладких виконується завдяки початковому розміру факторної бази $L_{\max} > L^a$ та визначенні достатнього такого розміру L^*, який може виявитися меншим за L^a.

3.4.1. Метод вибору достатньої кількості B – гладких чисел.

У рамках цього дослідження будемо вважати вирішеною задачу пошуку розмірів інтервалу просіювання та факторної бази, де факторна база містить $L_{\max}$ елементів.

В пропонованому алгоритмі **MLB**, що реалізує вибір достатньої кількості B-гладких чисел при розмірі факторної бази $L_{\max} > L^a$, використовуються два додаткові вектори $Ve[L_{\max}+1]$ та $Vf[L_{\max}+1]$, кожен елемент яких співвіднесений відповідному елементу факторної бази. В цих векторах нульовій клітинці відповідає знак B – гладкого числа, а довільній іншій клітинці – відповідний порядковий номер елемента факторної бази, де елементи факторної бази розміщені в порядку зростання їх значень.

Вектор $Ve[L_{\max}+1]$ – це інформація про показники степенів отриманого нового B – гладкого числа, де в нульовій клітинці значенню 1 відповідає від’ємне значення B – гладкого числа, а значенню 0 – додатне. В довільній іншій клітинці k вектора Ve вказано показник степеня елемента факторної бази за номером k, що є дільником B – гладкого числа, а інакше нуль.

В кожній клітинці k вектора $Vf[L_{\max}+1]$ вказано кількість B – гладких чисел, для яких порядковий номер s максимального за значенням елемента факторної бази, що є дільником B – гладкого числа з непарним показником степеня, не перевищує k ($s \le k$).

В пропонованому алгоритмі **MLB** буде використовуватися також вектор $VB[L_{\max}+2]$ – в кожній клітинці t якого міститься інформація про значення числа з інтервалу просіювання $(-L^b, L^b)$ на основі якого отримано B – гладке число з номером t, а також вектор $VM[L_{\max}+2]$. Клітинці t вектора VM відповідає B – гладке число з номером t, для якого задається порядковий номер s максимального за значенням елемента факторної бази, що є дільником цього B – гладкого числа з непарним показником степеня.

Визначення початкових значень елементів векторів Vf, VM та VB, їх зміни при отриманні нового B – гладкого числа та умови достатності кількості B – гладких чисел представлені нижче кроками алгоритму **MLB**:

1. Присвоїти довільному k-у елементу вектора Vf значення $k + 2$, довільному з елементів векторів Ve та VB значення нуль. Лічильнику nb B – гладких чисел присвоїти значення 0.
2. На етапі проріджування при отриманні B – гладкого числа B з номером nb, на основі числа x з інтервалу просіювання, сформувати вектор Ve показників степенів для дільників B та визначити найбільший порядковий номер s ненульового непарного елемента в ньому. Присвоїти $nb = nb + 1$; $VB[nb] = x$; $VM[nb] = s$.
3. Для всіх елементів вектора Vf, починаючи з номера s, зменшити їх значення на одиницю.
4. Кроки 2 та 3 продовжувати до тих пір, поки для одного з елементів вектора Vf, наприклад k, не буде виконана умова $Vf[k] = 0$, або для додатного B всі показники степенів у векторі Ve парні. Якщо для додатного B всі показники степенів у векторі Ve парні, перейти до кроку 5, а при $Vf[k] = 0$ –до кроку 6.
5. Отримуємо множники N згідно методу факторизації Ферма і завершити роботу алгоритму.
6. Прийняти $L^* = k$. Сформувати матрицю M, що відповідає $k + 2$ - м B – гладким числам, для кожного з яких з порядковим номером t $VM[t] \le k$. Для формування j – го рядка матриці M необхідно:
   a. знайти t_j, для якого $VM[t] \le k$;

b. знайти значення B – гладкого числа за формулою $B = x^2 - N$, де $x = VM[t]$;

c. сформувати вектор *Ve* показників степенів елементів факторної бази, дільників *B* та для непарних їх значень у відповідному стовпчику матриці M записати 1, а в інших випадках 0.

7. Опрацювати матрицю та виявнити чи отримане значення кореня не дорівнює N. Якщо ні, то задача факторизації вирішена, а інакше перейти до кроку 7.
8. Видалити інформацію про B – гладке число, що відповідає рядку $j0$ з нульовими значеннями елементів перетвореної матриці. Присвоїти:

a. $Vf[i] = Vf[i] + 1$, де $i \geq VM[t_{j0}]$;

b. $VM[t_{j0}] = VM[nb]$; $VM[nb] = 0$;

c. $Vb[t_{j0}] = Vb[nb]$; $Vb[nb] = 0$;

d. $nb = nb - 1$.

e. Перейти до кроку 2.

3.4.2. Приклади застосування алгоритму MLB.

Приклад 1. Нехай p=23, q= 97, N = 2231, розмір факторної бази: $L^a = 4$, інтервал просіювання: $L^b = 67$, початкове значення X0 = xx = $\sqrt{N}$ = 48.

Елементи факторної бази: 2, 5, 11, 17. Елементи факторної бази подвоєної довжини: 2, 5, 11, 17, 37, 43, 59, 71.

Дані про B-гладкі числа, їх множники та значення елементів векторів *Ve*[] та *Vf*[] по елементах бази наведено в табл. 3.7.

Таблиця 3.7

Дані про B – гладкі числа для N = 2231

Номер B-гладкого	dx	$R = \sqrt{X^2 - N}$	Вектори *Ve* та *Vf*	Знак *R*	Елементи факторної бази							
					2	5	11	17	37	43	59	71
0	-	-	*Ve*	0	0	0	0	0	0	0	0	0
			Vf	2	3	4	5	6	7	8	9	10
1	1	170	*Ve*	0	1	1	0	1	0	0	0	0
			Vf	2	3	4	5	5	6	7	8	9
2	3	370	*Ve*	0	1	1	0	0	1	0	0	0
			Vf	2	3	4	5	5	5	6	7	8

Продовж. табл. 3.7

3	4	473	*Ve*	0	0	0	1	0	0	1	0	0
			Ve	2	3	4	5	5	5	5	6	7
4	5	578	*Ve*	0	1	0	0	2	0	0	0	0
			Vf	2	2	3	4	4	4	4	5	6
5	11	1250	*Ve*	0	1	4	0	0	0	0	0	0
			Vf	2	1	2	3	3	3	3	4	5
6	12	1369	*Ve*	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**0**
			Vf									

Для останнього з B – гладких чисел отримано вектор його множників *Ve*, в якому всі показники степенів парні, що відповідає умовам кроку 5 алгоритму **MLB.**

Отримані множники N: p = X-Y = 60 - 37 = 23; q = X+Y = 60 + 37 = 97.

Приклад 2. p=277, q=2917, N=808009, L^a=8, L^b=555, X0=xx=$\sqrt{N}$ = 899.

Елементи факторної бази: 2, 3, 5, 11, 17, 31, 37, 47.

Елементи факторної бази подвоєної довжини: 2, 3, 5, 11, 17, 31, 37, 47, 53, 59, 61, 67, 71, 79, 83, 89.

Дані про B-гладкі числа, їх множники та значення елементів векторів *Ve*[] та *Vf*[] по елементах бази наведено в табл. 3.8.

Таблиця 3.8

Дані про B – гладкі числа для N = 808009

№	dx	R	*Ve*, *Vf*	Знак *R*	Елементи факторної бази															
					2	5	11	17	37	43	59	71	2	5	11	17	37	43	59	71
0	-	-	*Ve*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			Vf	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	192	*Ve*	**0**	**6**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	2	3792	*Ve*	0	6	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
			Vf	2	3	3	4	5	6	7	8	9	10	11	12	13	14	14	15	16
3	-2	-3400	*Ve*	**1**	**3**	**0**	**2**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Ve	2	3	3	4	5	5	6	7	8	9	10	11	12	13	13	14	15
4	4	7400	*Ve*	0	3	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0
			Ve	2	3	3	4	5	5	6	6	7	8	9	10	11	12	12	13	14
5	5	9207	*Ve*	0	0	3	0	1	0	1	0	0	0	0	0	0	0	0	0	0
			Vf	2	3	3	4	5	5	5	5	6	7	8	9	10	11	11	12	13

Продовж. табл. 3.8

6	6	11016	*Ve*	**0**	**3**	**4**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	4	5	4	4	4	5	6	7	8	9	10	10	11	12
7	-6	-10560	*Ve*	**1**	**6**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	4	4	3	3	3	4	5	6	7	8	9	9	10	11
8	8	14640	*Ve*	0	4	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0
			Vf	2	3	3	4	4	3	3	3	4	5	6	6	7	8	8	9	10
9	-10	-17688	*Ve*	1	3	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0
			Vf	2	3	3	4	4	3	3	3	4	5	6	6	6	7	7	8	9
10	12	21912	*Ve*	0	3	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0
			Vf	2	3	3	4	4	3	3	3	4	5	6	6	6	7	7	7	8
11	-12	-21240	*Ve*	1	3	2	1	0	0	0	0	0	0	1	0	0	0	0	0	0
			Vf	2	3	3	4	4	3	3	3	4	5	5	5	5	6	6	6	7
12	14	25560	*Ve*	0	3	2	1	0	0	0	0	0	0	0	0	0	1	0	0	0
			Vf	2	3	3	4	4	3	3	3	4	5	5	5	5	5	5	5	6
13	16	29216	*Ve*	0	5	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
			Vf	2	3	3	4	4	3	3	3	4	5	5	5	5	5	5	4	5
14	-16	-28320	*Ve*	1	5	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0
			Vf	2	3	3	4	4	3	3	3	4	5	4	4	4	4	4	3	4
15	-21	-37125	*Ve*	**1**	**0**	**3**	**3**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	4	3	2	2	2	3	4	3	3	3	3	3	2	3
16	-22	-38880	*Ve*	**1**	**5**	**5**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	3	2	1	1	1	2	3	2	2	2	2	2	1	2
17	23	42075	*Ve*	**0**	**0**	**2**	**2**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
			Vf	2	3	3	3	2	0	0	0	1	2	1	1	1	1	1	0	1

В результаті знайдено 17 B-гладких, серед яких для 7 (в табл. 3.9 степені дільників для них виділені напівжирним) максимальний за значенням дільник не перевищує 5-го елемента факторної бази, рівного 37. Такі B-гладкі числа формують матрицю M, що містить 6 стовпчиків та 7 рядків, де 4 та 5 рядки співпадають:

Таблиця 3.9 Матриця M

Знак числа	2	5	11	17	37	*B*-гладкі
0	**0**	**1**	**0**	**0**	**0**	192
1	**1**	**0**	**0**	**0**	**1**	-3400
0	**1**	**0**	**0**	**0**	**1**	11016
1	**0**	**1**	**1**	**1**	**0**	-10560
1	**0**	**1**	**1**	**1**	**0**	-37125
1	**1**	**1**	**1**	**0**	**0**	-38880
0	**0**	**0**	**0**	**1**	**1**	42075

Приклад 3. p=509, q=2857, N=1454213, L^a=8, L^b=669, X0=xx=$\sqrt{N}$ = 1206.

Елементи факторної бази: 2, 17, 41, 53, 61, 89, 97, 103.

Елементи факторної бази подвоєної довжини (в дужках вказано порядковий номер елемента): 2(1), 17(2), 41(3), 53(4), 61(5), 89(6), 97(7), 103(8), 107(9), 113(10), 131(11), 163(12), 167(13), 173(14), 179(15), 181(16), 191(17).

При однаковому інтервалі просіювання у випадку факторної бази, що містить 8 елементів, не було знайдено достатньої кількості B-гладких чисел. При використанні розширеної факторної бази було отримано 18 B-гладких чисел, серед яких виявилося чотири, у яких максимальне значення дільника з непарним показником степеня не перевищує 17. Дані про такі B-гладкі числа, їх множники та значення елементів векторів *Ve* та *Vf* по елементах бази наведено в табл. 3.10.

Таблиця 3.10 Дані про B – гладкі числа для N = 1454213.

№	dx	R	*Ve*, *Vf*	Знак *R*	Порядкові номери елементів факторної бази															
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	*Ve*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			Vf	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	-4	-9409	*Ve*	1	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
			Vf	1	2	3	4	5	6	7	8	8	9	10	11	12	13	14	15	16	17
4	-12	-28577	*Ve*	1	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			Vf	1	2	2	3	4	5	6	7	7	8	9	10	11	12	13	13	14	15
16	-216	-474113	*Ve*	1	0	1	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0
			Ve	1	2	1	2	2	3	4	5	5	5	5	6	7	6	4	4	3	3
18	269	721412	*Ve*	**0**	**2**	**1**	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
			Ve	1	2	0	1	1	2	3	4	4	4	4	5	6	5	2	2	1	1

Виділені B-гладкі числа формують матрицю M, що містить 3 стовпчики та 4 рядки, де 2 та 3 рядки співпадають:

Таблиця 3.11 Матриця M

Знак числа	2	17	*B*-гладкі
1	**0**	**0**	-9409
1	**0**	**1**	-28577
1	**0**	**1**	-474113
0	**0**	**1**	721412

При розширенні факторної бази збільшується кількість чисел N, які можна розкласти на множники методом квадратичного решета. Відмічається також, що її збільшення призводить до росту обчислювальної складності, оскільки слід знаходити більшу кількість B-гладких чисел. Проте при проведенні чисельних експериментів, де розмір факторної бази збільшувався двічі, виявилося, що з використанням алгоритму **MLB** час, необхідний для пошуку достатньої кількості B-гладких чисел, навпаки зменшувався.

В чисельних експериментах досліджувалися 10^6 відносно малих чисел, що є добутками простих p в діапазоні p = 277 ÷ 8369 та простих q в діапазоні q = 37811÷ 48589. За результатами розрахунків встановлено, що при використанні розширеної вдвічі факторної бази:

- час, необхідний для формування достатньої кількості елементів факторної бази зменшився на 29%;

- загальна кількість досліджуваних пробних значень із інтервалу просіювання при знаходженні B-гладких чисел (якщо достатню їх кількість не вдалося знайти, вважалося що використано весь інтервал просіювання від $-L^b$ до L^b) зменшилася від 1056931741 до 370331821, тобто в 2,85 рази;

- кількість чисел N, для яких не вдалося знайти їх множники зменшилася із 39612 до 227 тобто на 3.96% від загальної кількості N. У випадку числа елементів факторної бази, рівного L^a, коли не використовувався спосіб вибору достатньої кількості B-гладких чисел, кількість чисел N, для яких не вдалося знайти їх множники, становила 40084;

- розмірність матриці M при збільшенні факторної бази вдвічі зростає в середньому в 1,8 рази.

3.5. Прискорення методу квадратичного решета на основі використання умовно B- гладких чисел.

Основною проблемою для методу квадратичного решета - є пошук достатньої кількості B-гладких чисел. Тому пошук способів отримання додаткових варіантів остач, що можуть розглядатися як B-гладкі числа, є актуальним завданням.

Додатковий аналіз B-гладких чисел згадується в літературі [95, 127, 131]. Пропонується запам'ятовувати $y(x) = y_1(x) \cdot y_2(x)$ такі, що $y_1(x)$ гладке число, а $B < y_2(x) < B^2$. За наявності двох таких чисел y з однаковим y_2 їх добуток стане B-гладким.

Пропонується розглянути не одиничні залишки, які є квадратами простих чисел. Вектори таких чисел можна добавляти до матриці не враховуючи ці залишки. Як квадрати, вони ні як не впливають на рішення. Якщо $y(a) = 7 * 11^2 * 23 * 137^2$ та $y(b) = 7 * 23$, тоді $y(a) * y(b) = 7^2 * 11^2 * 23^2 * 137^2$. При обраному максимальному числі для факторної бази 23, вектор *y(a)* увійде до матриці. Ми можемо не враховувати 137^2 при розв'язанні матриці, тому що 137 має парну степінь. Такі залишки і називаються **умовно *B*-гладкими.**

3.5.1. Застосування аналізу умовно B-гладких чисел.

Розглянемо на прикладі ефективність запропонованої модифікації. Оберемо *p=401* та *q=103*, ці прості числа створюють нам число для факторизації *p*q=N=41303*. Обчислимо за формулою (1) розмір факторної бази *A=6*. За допомогою формули (2) отримаємо інтервал просіювання *M=203*.

Після просіювання варіантів *y(x)* через факторну базу, отримуємо *B*-гладкі числа. Ці числа зображені в таблиці.

Таблиця 3.12 B-гладкі числа.

Знак числа	2	11	19	23	29	37	*B*-гладкі
1	1	1	0	0	2	0	-18502
1	0	1	0	0	0	2	-15059
1	1	0	0	1	0	1	-1702
0	1	0	2	0	0	0	722
0	0	0	0	2	1	0	15341
0	1	1	0	1	0	1	18722

Цих чисел не достатньо для факторизації обраного *N*. Знайдемо умовно *B*-гладкі числа, вони зображені в таблиці.

Таблиця 3.13 Умовно B-гладкі числа

Знак числа	2	11	19	23	29	37	Дільники які не входять до факторної бази	Умовно *B*-гладкі
0	0	0	0	0	0	0	149^2	22201
0	1	0	0	0	0	0	131^2	34322
0	1	0	0	0	0	0	157^2	49298

Число 22201 не увійшло до матриці тому що воно має прості дільники які не потрапили до факторної бази. Число 22201 є квадратом, завдяки йому ми отримуємо рішення. Числа 32322 та 49298 не є квадратами, але разом дають нам ще одне рішення.

Розглянемо інший приклад. Оберемо *p=11* та *q=601*, отримаємо *p*q=N=6611*. Обчислимо розмір факторної бази та інтервал просіювання *A=5*, *M=102*.

Після просіювання варіантів *y(x)* через факторну базу, отримуємо *B*-гладкі числа. Ці числа зображені в таблиці.

Таблиця 3.14

***B*-гладкі числа**

Знак числа	2	5	17	29	31	*B*-гладкі
1	1	1	1	1	0	-4930
1	0	1	0	1	1	-4495
1	1	1	2	0	0	-2890
1	1	0	1	1	0	-986
1	0	0	1	0	1	-527
1	1	2	0	0	0	-50
0	0	1	0	2	0	4205
0	1	1	1	0	1	5270

Обчислюючи матрицю створену з векторів з таблиці 3 ми отримаємо тільки хибні рішення. Знайдемо умовно *B*-гладкі числа, вони зображені в таблиці.

Таблиця 3.15

Умовно *B*-гладкі числа

Знак числа	2	5	17	29	31	Дільники які не входять до факторної бази	Умовно *B*-гладкі
1	1	0	0	0	0	41^2	-3362
0	0	1	0	0	0	37^2	6845

Число -3362 дозволило сформувати рішення з чисел: -4930, -4495, -3362 та -527.

Приклади випадків де умовно *B*-гладкі входять до рішення наведені в таблиці.

Таблиця 3.16

Приклади факторизації з умовно *B*-гладкими числами.

p	q	N	*B*-гладкі, які утворюють квадрат	Умовно *B*-гладкі	Множники Умовно *B*-гладких
27743	41203	1143094829	45292900		$5^2 * 7^2 * 673^2$
89	46411	4130579	-1496450, -5618	-1496450	$-1 * 2 * 5^2 * 173^2,\ -1 * 2 * 53^2$
5647	40577	229138319	-29848630, -2996875, 11514850	-29848630,	$-1 * 2 * 5 * 7653^2,$ $-1 * 5^5 * 7 * 137,$ $2 * 5^2 * 41^2 * 137$
29741	40087	1192227467	26759929		$7^2 * 739^2$
30271	48533	1469142443	83375161		$23^2 * 397^2$
30707	32089	985356923	477481		691^2
31729	32423	1028749367	120409		347^2
32443	45137	1464379691	40284409		$11^2 * 577^2$
32887	39371	1294794077	10510564		$2^2 * 1621^2$
6163	44777	275960651	-22386875, -2107, 23952473	23952473	$-1 * 5^4 * 7^2 * 17 * 43,$ $-1 * 7^2 * 43,$ $17 * 1187^2,$
36353	39511	1436343383	2493241		1579^2
37561	43067	1617639587	7579009		2753^2
38239	45413	1736547707	12866569		$17^2 * 211^2$
39157	45119	1766724683	8886361		$11^2 * 271^2$
40577	46811	1899449947	9715689		$3^2 * 1039^2$
41719	45137	1883070503	2920681		1709^2
6359	43051	273761309	-38568413 -23710340 -6177145 -685684	-38568413 -23710340	$-1 * 13 * 41 * 269^2,$ $-1 * 2^2 * 5 * 37 * 179^2$ $-1 * 5 * 13 * 29^2 * 113$ $-1 * 2^2 * 37 * 41 * 113$
44867	47911	2149622837	2316484		$2^2 * 761^2$
45403	46589	2115280367	351649		593^2
48193	48539	2339240027	29929		173^2

Застосування умовно B-гладких дозволяє знаходити рішення для деяких N, навіть коли розмір факторної бази дорівнює нулю. Тому такий метод можна вважати подальшим розвитком методу Ферма і він є певного роду переходом від методу Ферма до методу квадратичного решета. На відміну від методу Ферма метод використовує не тільки квадрати $X^2 - N$ які є додатними а і від'ємні значення, при цьому для факторизації достатньо не більше двох умовно B-гладких чисел.

3.5.2. Порівнювальна оцінка аналізу умовно B-гладких чисел.

Додаткові вектори у сформованій матриці дозволили отримати рішення без розширення факторної бази або інтервалу просіювання.

Отримати числа у яких залишок є квадратом можна доволі часто. Взявши перших 5000 простих чисел та сформувавши за них $12.5*10^6$ можливих варіантів *N*, ми знайшли додаткові вектори у 99% випадках.

Корисну дію цього методу можна побачити, якщо обрати випадки в яких базовий алгоритм квадратичного решета при рекомендованих [95, 127] розмірах факторної бази та інтервалу просіювання обчислених за формулами (2.1) та (2.2) не зміг знайти рішення, та застосувати аналіз *y(x)*, у яких залишок після просіювання є просте число у парній степені.

У 7% випадках модифікований алгоритм зміг факторизувати число.

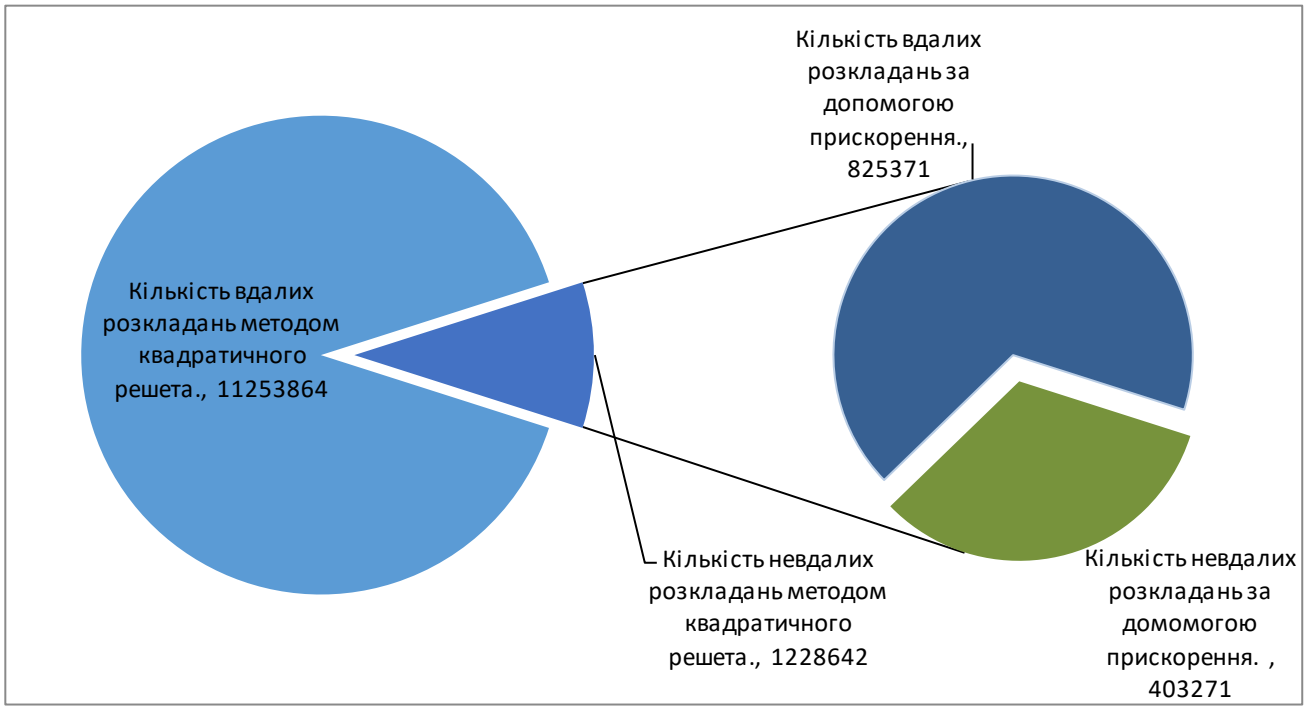


Рисунок 3.4 – Ефективність застосування умовно B-гладких чисел.

Варто зазначити, що якщо для порівнювального аналізу взяти меншу кількість простих чисел, починаючи не з першого простого числа, ми отримаємо кращі результати. Наприклад якщо взяти тисячу простих чисел починаючи з простого числа з порядковим номером 4000 або 5000 ми знайдемо, що модифікований алгоритм зміг факторизувати всі числа.

3.5.3. Оцінка складності та часу виконання.

Складання матриці для стандартного алгоритму квадратичного решета потребує L^{2a} місця [84]. Кількість варіантів y(x) які нам потрібні (B-гладкі) для стандартного квадратичного решета можливо розрахувати за формулою L^{2a+1}.

Всі залишки y(x) вже отримані, їх пошук не потребує додаткової роботи. При застосуванні аналізу умовно B-гладких чисел нам необхідно для кожного варіанта y(x) запам'ятовувати залишок (якщо він є квадратом), тому необхідний нам об'єм пам'яті збільшується і стає рівним L^{2a+1}.

На перший погляд додатковий аналіз варіантів y(x) робить алгоритм складнішим, та збільшує час його роботи – додаткове обчислення квадратного кореню з усіх залишків $y(x)$ більших за одиницю. Однак, при застосуванні додаткового аналізу варіантів $y(x)$, кількість $y(x)$ які нам підходять збільшується (за рахунок умовно B-гладких) на деяке γ і становить $b = a + (4a)^{-1} + \gamma$. Значення b - кількість ітерацій в алгоритмі, обирається таким, щоб кількість варіантів $y(x)$ які нам підходять становила L^a, тому $b = a + (4a)^{-1} + \gamma$. Як ми бачимо ця кількість зменшилась.

Беручи до уваги те що b - показник степені інтервалу просіювання L^b, необхідно відмітити що кожне знайдене умовно B-гладке значення зменшує інтервал просіювання та можливі випадки отримання кореня коли остача $y(x)$ буде повним квадратом, або кілька умовно B-гладких дозволять сформувати достатню кількість остач, на основі яких буде отримано дільники N.

Оцінити швидкість модифікованого алгоритму можна за формулою:

$$L^{\max\{2a+1,\, a+(4a)^{-1}-y,\, 3a\}} \tag{3.2}$$

Швидкість просіювання зменшилась на γ, де γ кількість елементів y(x) доданих умовно B-гладких залишків.

3.6. Висновки до розділу 3

У результаті чисельних експериментів було показано, що для найбільш затратного за часом етапу методу квадратичного решета – пошуку B-гладких чисел, - у випадку збільшення факторної бази вдвічі та формуванні достатнього кількості B-гладких:

- в середньому в 2,85 рази зменшилося кількість використаних пробних значень з інтервалу просіювання при збільшенні розміру матриці M в 1.8 рази;
- на 29% зменшився загальний час на просіювання пробних значень та формування достатньої кількості B-гладких для 10^6 варіантів факторизованих чисел N;
- на 3,96 відсотків від загальної кількості N, що розкладалися на множники, збільшилась кількість вдалих факторизації по відношенню до базового методу квадратичного решета.

Був розроблений алгоритм рішення матриці на ходу, який прискорює базовий метод квадратичного решета. В окремих випадках можливі прискорення в 10, 100 і більше разів. Середнє ж значення величини зниження обчислювальної складності модифікованого методу для чисел порядку до 10^{130}, згідно отриманих оцінок, складає 5.45 відсотка. Такий ефект пов'язаний з можливістю отримання нульового вектора в ряді випадків значно раніше ніж будуть знайдені L^a+2 B-гладких числа, що передбачено в алгоритмі базового методу та проілюстровано в наведених прикладах.

На основі проведених чисельних експериментів показано, що вирішення матриці на ходу дозволяє факторизувати число у деяких випадках, коли базовий алгоритм квадратичного решета (при стандартному інтервалі просіювання та розміру факторної бази) не зміг сформувати матрицю для отримання рішення. А саме, для обраних 10000 чисел розміром 10^{13} модифікований алгоритм зміг зменшити кількість невдалих факторизацій з 686 випадків до 503, відносно базового алгоритму квадратичного решета.

Швидкість методу квадратичного решета залежить від таких евристичних значень як розмір факторної бази та інтервал просіювання. На основі проведених чисельних експериментів показано, що використання умовно B-гладких чисел дозволяє факторизувати число у тих випадках, коли базовий алгоритм квадратичного решета (при стандартному інтервалі просіювання та розміру факторної бази) не зміг сформувати матрицю для отримання рішення.

Застосування умовно B-гладкі дозволяє знаходити рішення для деяких N, навіть коли розмір факторної бази дорівнює нулю. Тому такий метод можна вважати подальшим розвитком методу Ферма і він є певного роду переходом від методу Ферма до методу квадратичного решета.

Модифікований алгоритм зміг зменшити кількість невдалих факторизацій з 11% до 3%, відносно базового алгоритму квадратичного решета.

Швидкість просіювання модифікованого алгоритму зменшилась на γ. Для кожного випадку значення γ є різним і дорівнює кількості елементів y(x) доданих завдяки використанню умовно B-гладких чисел.

Отримані результати є підставою для подальшого дослідження на числах 1024 біт та більше. Даний результат може бути використаний для методів MQkS, QS, та MPQS.

РОЗДІЛ 4. РОЗРОБКА РЕКОМЕНДАЦІЙ З РЕАЛІЗАЦІЇ ЗАПРОПОНОВАНИХ МЕТОДІВ ФАКТОРИЗАЦІЇ БАГАТОРОЗРЯДНИХ ЧИСЕЛ ПРИ КРИПТОАНАЛІЗІ RSA-АЛГОРИТМУ

Попередні розділи дисертаційної роботи було присвячено представленню розробленого методу факторизації MQkS, оцінена його обчислювальна складність і ефективність.

Однак ефективність використання запропонованого методу напряму пов’язана з можливістю його реалізації за допомогою сучасних програмно-апаратних або програмних засобів.

Даний розділ присвячений розробці рекомендацій з апаратно-програмної реалізації розроблених методів, що дозволять знизити часові витрати при проведенні криптографічного аналізу RSA-алгоритму.

В [47], [92] показано, що вирішення задачі факторизації криптомодуля RSA-алгоритму можливо не тільки за рахунок одночасного (паралельного) використання різних методів факторизації, а й шляхом паралельної реалізації етапів факторизації.

Крім того, аналіз останніх досліджень методів факторизації цілих чисел [95, 96] показують, що всі останні експерименти по факторизації багаторозрядних RSA-чисел, порівнянних з довжиною криптомодуля RSA-алгоритму, з метою прискорення виконуваних обчислювальних задач були проведені з використанням розподілених обчислювальних систем.

У зв’язку з цим, необхідно провести дослідження можливості підвищення ефективності використання апаратно-програмних засобів криптографічного аналізу RSA-алгоритму при реалізації розроблених методів факторизації, яке проведене в напрямку використання паралельних обчислювальних систем.

4.1. Реалізація розроблених обчислювальних методів у вигляді програмних макетів.

Під програмним макетом будемо розуміти такий програмний продукт, який розроблений на одній з мов програмування згідно існуючих алгоритмів

(представлених в роботі у вигляді етапів реалізації методів), але не володіє вимогами, що висуваються до ПЗ відповідно до [135]: надійність, інформаційна та програмна сумісності і т.д.

Для розробки програмного макету алгоритму MQkS, була вибрана мова програмування С. В якості середовища розробки програмного макету використовувалася Microsoft Visual Studio 2012, що дозволяє розробляти програмні продукти для ОС Windows. Реалізація програмного макету здійснювалася відповідно до алгоритму вдосконаленого методу А, представленого у 2 розділі дисертаційного дослідження.

Експеримент проводився в середовищі розробки під керуванням 32-розрядної ОС Windows 7 на персональному комп'ютері з наступними характеристиками: тактова частота 2.4 GHz, ОЗУ 4 ГБ, 32-розрядна ОС. При проведенні численних експериментів ставилося завдання оцінки часу факторизації чисел виду $N = p * q$ де p і q – прості, для методу MQkS у однопоточному режимі та методу QS. Результати чисельних експериментів з факторизації ряду чисел наведені в розділі 3.

Оцінимо можливість використання паралельних і розподілених обчислень, а також спеціалізованих апаратно-програмних та апаратних засобів для вирішення завдання підвищення їх продуктивності при реалізації розроблених методів.

4.2. Використання паралельних обчислень при криптоаналізі RSA-алгоритму.

Поряд з розробкою нових ефективних алгоритмів факторизації при розробці чисельних методів криптоаналізу алгоритму RSA, загальним напрямком зниження практичних часових затрат залишається вибір архітектури обчислювальних систем та ефективної моделі обчислень, що потребує адаптації методів, які розробляються до обраної архітектури. У рамках класичної моделі обчислень А. Т’юрінга – Дж. Неймана основним напрямком зниження часових затрат є раціональний розподіл необхідної кількості інформаційно-обчислювальної роботи між вільними в даний момент паралельними ресурсами. Збільшення продуктивності обчислювальних систем породжує нові загрози для алгоритму RSA. Використання паралельних

обчислень на базі графічних процесорів nVidia може суттєво збільшити ефективність атак методом факторизації відкритого ключа.

4.2.1. Аналіз існуючих варіантів організації паралельних обчислень.

В [47], [92] показано, що вирішення задачі факторизації криптомодуля RSA-алгоритму можливо не тільки за рахунок одночасного (паралельного) використання різних методів факторизації, а й шляхом паралельної реалізації етапів факторизації. Результати порівняльного аналізу реалізації основних відомих методів факторизації з використанням розподілених або паралельних обчислювальних систем представлено в [96], [97].

У зв'язку з зазначеним, подальші дослідження можливості підвищення ефективності використання апаратно-програмних засобів криптографічного аналізу RSA-алгоритму при реалізації розроблених методів факторизації, проводилися в напрямку використання паралельних обчислювальних систем.

Згідно [95], [136] варіанти організації паралельних обчислень можна розбити на наступні класи.

Симетричні мультипроцесорні системи (SMP – symmetric multiprocessing) – системи, що складаються з декількох однорідних процесорів і масиву загальної пам'яті, доступ до якої розділяють усі процесори. Хоча загальна пам'ять і спрощує взаємодію процесорів системи, але, з іншого боку, обмежує їх кількість в реальних системах. Програмування здійснюється в рамках моделі загальної пам'яті, як правило, це технологія OpenMP.

Масивно-паралельні системи (MPP – massive parallel processing) – складаються з однорідних обчислювальних модулів, включають один або кілька процесорів і локальну пам'ять, що розділена фізично. Прямий доступ до пам'яті інших модулів неможливий. Модулі системи з'єднуються комунікаційними каналами. Програмування здійснюється в рамках моделі передачі повідомлень (наприклад, MPI).

Кластерні системи – «спрощений» варіант MPP-систем, що складається, як правило, з певної кількості ЕОМ загального призначення, які використовують для взаємодії стандартні мережеві технології. Як у MPP-систем, програмування здійснюється в рамках моделі передачі повідомлень, зазвичай MPI. Можливості масштабованості кластерів дозволяють багаторазово збільшувати продуктивність застосунків для більшого числа користувачів. Основною перевагою таких суперкомп'ютерів є їх вартість, але суттєвим недоліком є затримка обміну даними завдяки стандартним мережевим інтерфейсам.

Grid мережа (від англ. *Grid* – грати) – об'єднання в загальну обчислювальну мережу різнорідних обчислювальних ресурсів. Даний клас реалізує розподілені обчислення, що реалізуються на територіально віддалених пристроях, завдяки чому виникає затримка обміну даними між пристроями.

Паралельні векторні системи (PVP – Parallel Vector Process) складаються зі спеціалізованих векторно-конвеєрних процесорів, в яких передбачені команди типової обробки векторів незалежних даних. Як правило, кілька таких процесорів працюють одночасно із загальною пам'яттю (аналогічно SMP) в рамках багатопроцесорних конфігурацій. Обмін даними в PVP системах здійснюється у векторному форматі, який є набагато швидшим, ніж в скалярний, завдякі чому проблема взаємодії між потоками даних при розпаралелюванні стає несуттєвою.

Системи з неоднорідним доступом до пам'яті (NUMA – nonuniform memory access). Складаються з декількох однорідних об'єднаних між собою базових модулів, кожен з яких має кілька процесорів і блок пам'яті. Суть цієї архітектури полягає в моделі організації пам'яті, яка фізично розподілена по різних частинах системи, але використовується як єдиний адресний простір. Програмування здійснюється аналогічно SMP-систем.

«Хмарні» обчислення, що представляють собою динамічно масштабований спосіб доступу до зовнішніх розподілених обчислювальних ресурсів у вигляді сервісів, що надаються через об'єднану мережу. Перевага даного способу організації обчислень полягає в тому, що користувачеві не потрібно особливих знань про

інфраструктуру «хмарної» технології або навичок її адміністрування. Принципи організації і функціонування «хмарних» ресурсів представлено в працях В.М. Глушкова, Е.А. Якубайтиса, А.Н. Мямліна, А.І. Нікітіна [137-139]. Більшість розгорнутих на серверах центрів обробки даних «хмарних» інфраструктур використовують технології віртуалізації.

Крім розбиття паралельних систем на класи, росповсюдженою є класифікація за архітектурою побудови (наприклад, в [122] [140]). Поняття архітектури високопродуктивної системи є досить широким, оскільки під архітектурою можна розуміти і спосіб паралельної обробки даних, який використовується в системі, і організацію пам'яті, і топологію зв'язку між процесорами, і спосіб виконання системою арифметичних операцій. Загальноприйнятою і найбільш відомою є класифікація архітектур М. Флінна, запропонована в 1966 році, в основу якої покладено поняття потоків інструкцій і потоків даних. Класифікація визначає наступні архітектурні класи: SISD, MISD, SIMD, MIMD.

SISD (Single Instruction Stream / Single Data Stream) – одиночний потік команд і одиночний потік даних. До цього класу належать послідовні комп'ютерні системи, які мають один центральний процесор, здатний обробляти тільки один потік послідовно виконуваних інструкцій.

MISD (Multiple Instruction Stream / Single Data Stream) – множинний потік команд і одиночний потік даних. Теоретично в цьому типі машин множина інструкцій повинна виконуватися над єдиним потоком даних. До сих пір жодної реальної машини, що потрапляє в даний клас, створено не було.

SIMD (Single Instruction Stream / Multiple Data Stream) – одиночний потік команд і множинний потік даних. Ці системи зазвичай мають велику кількість процесорів, які можуть виконувати одну і ту ж інструкцію щодо різних даних в жорсткій конфігурації. Єдина інструкція паралельно виконується над багатьма елементами даних.

MIMD (Multiple Instruction Stream / Multiple Data Stream) – множинний потік команд і множинний потік даних. Архітектура даного класу передбачає наявність множини процесорів, що обробляють кожен свій потік даних.

Більш детальну класифікацію можна представити у вигляді рис. 4.1.

У наведеній класифікації викликає інтерес архітектура MIMD, яка до недавнього часу вважалася основним способом реалізації розподілених обчислень. Даний клас заснований на використанні мультипроцесорних або мультикомпьютерних обчислювальних систем, які можна розділити на дві категорії. Перша категорія містить MPP процесори (Massively Parallel Processors – процесори з масовим паралелізмом) – дорогі суперкомп'ютери, що складаються з великої кількості процесорів, пов'язаних високошвидкісний комунікаційною мережею. Головною перевагою систем з роздільною пам'яттю є хороша масштабованість.

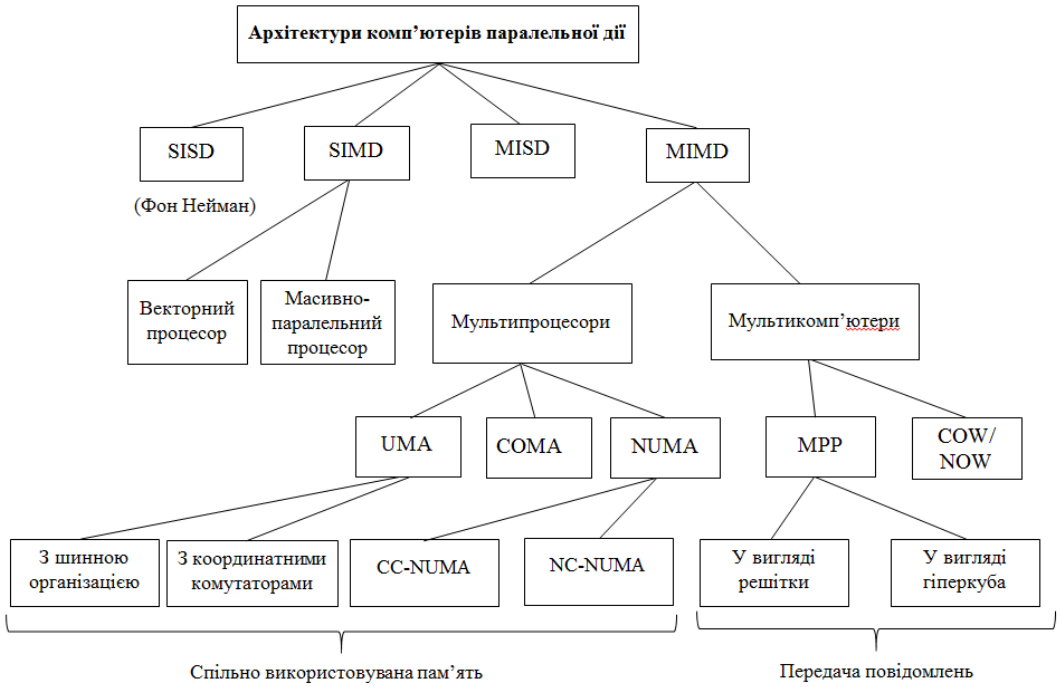


Рисунок. 4.1 – Класифікація архітектур паралельних обчислювальних систем

Друга категорія мультикомпьютерів включає робочі станції, об'єднані за допомогою технологій локальних або глобальних обчислювальних мереж в формі кластера (групи комп'ютерів) і здатних вирішувати складні математичні або наукові завдання в якості єдиного обчислювального ресурсу. Кластери робочих станцій або потужних серверів, об'єднаних в єдину інфраструктуру, отримали назву GRID-систем. З точки зору мережевої організації, GRID-системи являють собою узгоджену, відкриту і стандартизовану середу, яка забезпечує гнучкий, безпечний, скоординований розподіл обчислювальних ресурсів і ресурсів зберігання інформації,

які є частиною цього середовища, в рамках однієї віртуальної організації [141]. В Україні завдяки затвердженою постановою Кабінету Міністрів Державної цільової науково-технічної програми з впровадження та застосування GRID-технологій на 2009-2013 роки [142], GRID-обчислення так само отримали широке поширення [143]. На даний момент існує більш 35 кластерів, об'єднаних в Українську національну GRID-інфраструктуру [143]. До числа найбільш потужних обчислювальних кластерів можна віднести кластер суперкомп’ютерів для інформаційних технологій Інституту кібернетики імені В.М. Глушкова НАН України, кластер Центру суперкомп'ютерних обчислень КПИ ім. Ігоря Сікорського, обчислювальний кластер Інституту теоретичної фізики ім. М.М. Боголюбова НАН України.

Однак до числа основних недоліків ГРІД-систем можна віднести проблему існування затримок при обробці інформації, пов'язаної з необхідністю обміну даними по каналах передачі.

4.3 Застосування технології GPGPU CUDA для організації паралельних обчислень.

На відміну від попередніх парадигм програмування GPU CUDA (Compute Unified Device Architecture) дозволяє реалізовувати прямий доступу до апаратних можливостей графічних карт [124]. Дана особливість технології GPGPU дозволяє забезпечити швидкість обчислень на персональному комп'ютері з відеокартою, що підтримує технологію CUDA, порівняну зі швидкістю обчислень на суперкомп'ютері [144].

Використання GPU для реалізації паралельних обчислень при реалізації методів факторизації Лемана і ρ-методу Полларда представлені в дисертаційній роботі [95]. В рамках проведених досліджень були отримані оцінки обчислювальної складності паралельних алгоритмів зазначених методів, реалізованих в гетерогенних архітектурах з CPU і GPGPU CUDA.

Оцінка результатів проводилася на прикладі метода факторизації QS, програмна реалізація яких була здійснена в класичному варіанті для послідовного виконання на CPU на стандартній мові *C* і модифікованому для паралельних

обчислень на графічних прискорювачах NVIDIA на розширенні мови Сі для CUDA. Для обчислювальних експериментів використовувались апаратні платформи на CPU (Intel Core i5) і GPU компанії NVIDIA (GeForce 9600 (64 ядра)). Моделювання проводилося на гетерогенних обчислювальних машинах під управлінням операційної системи Windows 7.

На малюнку 4.2 наведені значення експериментальної апробації паралельного алгоритму QS при послідовній і паралельній програмних реалізаціях. Розмірністю задачі вважати кількість біт в двійковому записі факторизуємого числа n*N.

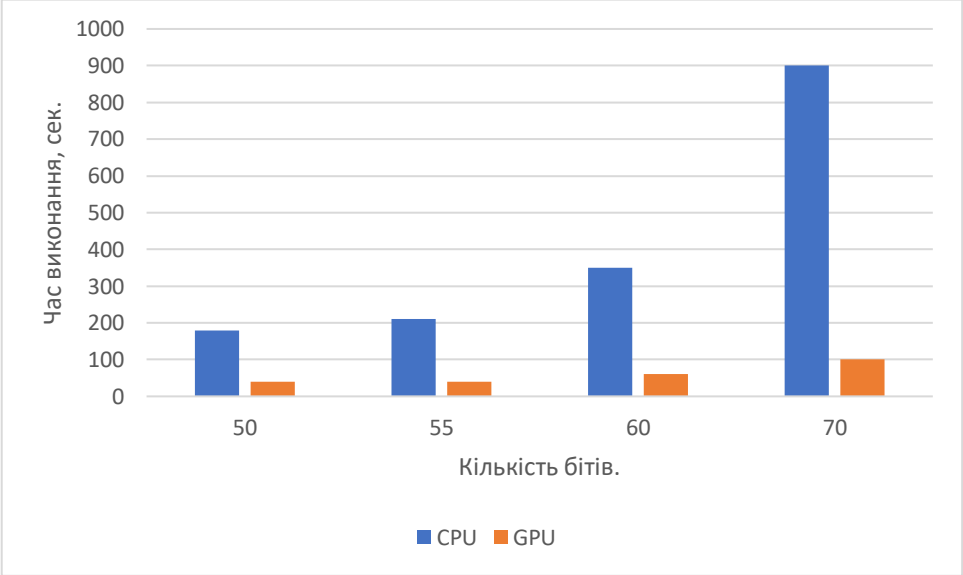


Рисунок. 4.2. – Результати факторизації із застосуванням алгоритму QS.

Аналіз представлених на діаграмі результатів свідчить про те, що використання паралельних технології GPGPU CUDA дозволяє знизити фактичні витрати часу рішення задачі факторизації методом QS до 5-ти разів. Отже, підхід з використанням GPU буде ефективним.

Аналіз представлених в розділі результатів порівняльних оцінок часових характеристик паралельної і послідовної реалізацій метода QS показує, що використання технології неспеціалізованих паралельних обчислень GPGPU CUDA дозволяє знизити фактичні витрати часу на вирішення задачі факторизації до 5-ти разів. Досить широке розповсюдження технології GPGPU, відносно низьке питоме енергоспоживання і вартість в порівнянні з сучасними суперкомп'ютерними системами дозволяє стверджувати, що технологія GPGPU CUDA є перспективним

напрямком реалізації алгоритмів факторизації багаторозрядних чисел, в тому числі і проведення тематичних досліджень сучасних апаратних і програмно-апаратних засобів крипторгафічного захисту інформації, що побудовано з використанням RSA-алгоритму.

4.3.1. Архітектура системи CUDA.

У цьому розділі буде детально розглянута архітектура GPU NVIDIA, що використовується в даному проекті з точки зору розвитку CUDA. CUDA GPU орієнтовано на виконання програм з великим об’ємом даних та розрахунків і представляє собою масив процесорів (Streaming Processor Array), що складаються з кластерів текстурних процесорів (Texture Processor Clusters, TPC). TPC в свою чергу складаються з набору мультипроцесорів (SM – Streaming Multi-processor), в кожному з яких декілька потокових процесорів (SP – Streaming Processors) ябо ядер (в сучасних процесорах кількість ядер перевищує 1024). Кожен GPU в даний час вже має до 4 гігабайт графічної пам’яті з подвійною швидкістю передачі даних (GDDR) DRAM, що має назву глобальна пам'ять.

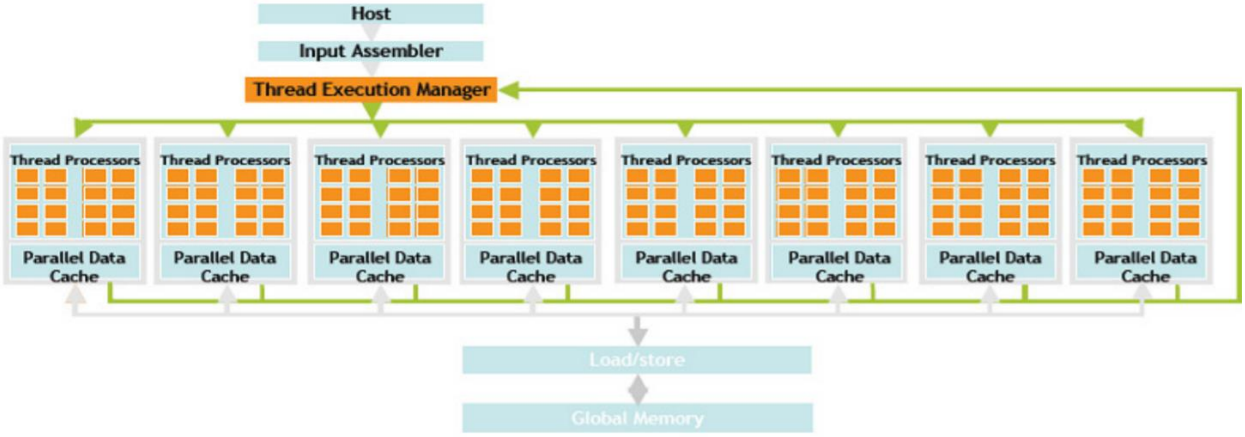


Рисунок 4.3 – Загальна структура CUDA.

4.3.2. Пам’ять CUDA.

CUDA підтримує декілька типів пам’яті які можуть бути використані розробниками програмного забезпечення для досягнення високого співвідношення швидкістю обчислення з швидкістю доступу до глобальної пам’яті (Compute to Global

Memory Access) і тому отримати високі швидкості виконання в ядрах. На малюнку 3 зображені види пам’яті CUDA.

Нитки CUDA можуть отримувати доступ через дані з декількох видів пам'яті під час їх виконання. Кожен потік має свою приватну локальну пам'ять. Кожен Блок потоку має спільну пам'ять, видиму для кожного потоку блоку і з часом життя яке дорівнює часу життя блоку. Кожен потік має доступ до тієї ж глобальної пам'яті. Кожен вид пам'яті включає глобальну, константну та текстуровану види пам'яті які оптимізовані для окремих функцій звернення до цих видів пам'яті.

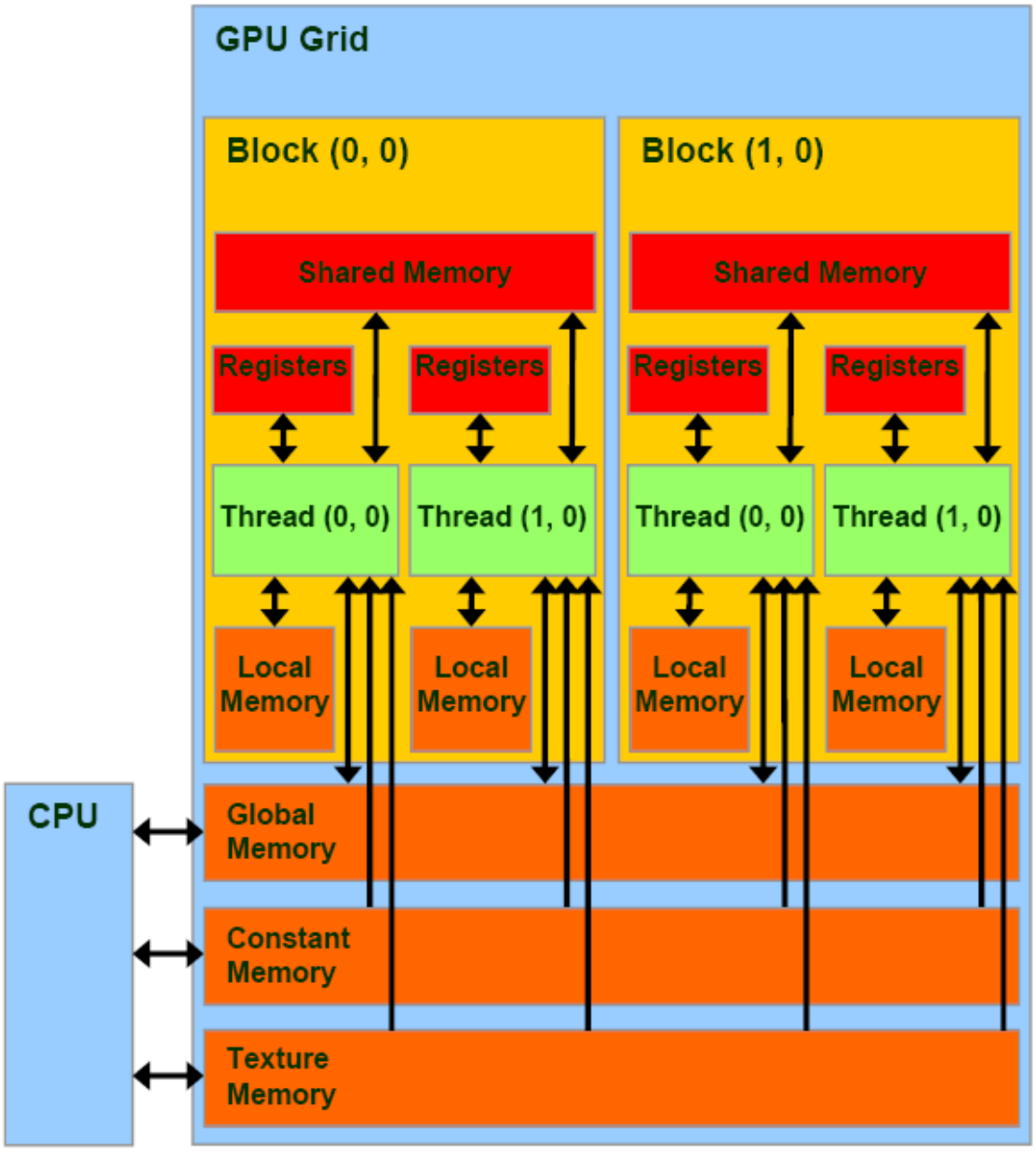


Рисунок 4.4 – Структура пам’яті CUDA.

У структурі пам'яті CUDA кожен вид пам'яті має обов'язки і функції, ось типи і характеристики пам'яті CUDA.

Таблиця 4.1 Типи пам’яті**.**

Тип пам’яті	розташування	Кеш	Вид доступу	область
Регістри	На чіпі	Ні	Запис/Читання	Один потік
Локальна	На чіпі	Так	Запис/Читання	Один потік
Спільна	На чіпі	Ні	Запис/Читання	Всі потоки у блоці
Глобальна	Поза чипом	Так	Запис/Читання	Всі host + потоки
Постійна	Поза чипом	Так	Читання	Всі host + потоки
Текстури	Поза чипом	Так	Запис/Читання	Всі host + потоки

Важливо зрозуміти деталі роботи які стоять за кожним з різних типів пам'яті в системі CUDA і як вони пов'язані з різними одиницями паралелізації.

Використання правильного типу пам'яті в потрібний час дуже важливо. Адже необхідно знати коли компілятор виділяє різні типи пам'яті, і як GPU найкраще виконує доступ до них.

Найвищий рівень доступності для потоків має глобальна пам'ять (global memory), фізично реалізована у вигляді інтегральних мікросхем, запаяних на платі графічного адаптера - та сама відео пам’ять, яка нині обчислюється вже гігабайтами. Розташування поза процесором робить цей тип пам'яті найбільш повільним, в порівнянні з іншими, наданими для обчислень на відео карті. Меншою доступністю наділена колективна пам'ять (shared memory): розташована в кожному SM блоці (дивись Рисунок 4.4), зазвичай розміром в 16KB, доступна тільки тим потокам, які виконуються на ядрах цього SM. Так як для паралельного виконання на одному SM може бути відведено більше одного блоку, то весь доступний в SM обсяг пам'яті розподіляється між цими блоками порівну.

Необхідно згадати, що колективна пам'ять фізично розташована десь дуже близько до ядер SM, тому має високу швидкість доступу, яку можна порівняти з швидкодією реєстрової (registers) - основним видом пам'яті. Саме регістри можуть служити операндами елементарних машинних команд, і є найбільш швидкою пам'яттю. Всі готівкові регістри одного SM порівну розділяються між всіма потоками, запущеними на цьому SM. Група регістрів, виділена в користування якомусь потоку,

доступна йому і тільки йому. На правах ілюстрації CUDA (або, навпаки, масштабів лиха): в тій же Tesla кожен SM надає в користування 16384 штук 32-х розрядних регістрів загального призначення.

Переваги графічних процесорів обертаються основною проблемою при складанні алгоритмів, так як для того, щоб досягти високої ефективності при таких масивно-паралельних обчисленнях, потрібно враховувати безліч чинників: архітектурні особливості GPU, швидкодія і порядок доступу до пам'яті, механізми синхронізації між обчислювальними потоками. Важливу роль відіграє також пристосованість самого алгоритму до виконання в паралельному режимі.

Можна виділити такі основні ключові моменти при реалізації методів перевірки криптостійкості RSA з використанням чисел розміром 10^{100} на GPU:

- швидкість доступу до пам'яті;
- паралельна реалізація алгоритмів простих арифметичних операцій;
- синхронізація між блоками паралельних ниток.

Далі розглянемо більш докладно проблеми, що виникають при роботі з GPU.

4.3.3. Особливості реалізації арифметичних операцій з багаторозрядними числами при використанні технології GPGPU CUDA.

У прикладних задачах криптоаналізу RSA-алгоритму потрібно виконувати обчислювальні операції над «довгими» цілими числами, які не можна зберігати в змінних стандартних типів даних мов програмування. Для зберігання таких чисел можна використовувати різні способи, наприклад, зберігати число у вигляді рядка, вважаючи окремий символ рядка відповідним розрядом числа. Найбільш часто для зберігання чисел використовують масиви, при цьому, як правило, в одну клітинку записується відразу кілька розрядів числа та подальші операції реалізуються над масивом з кожним елементом. Таке представлення вимагає додаткових дій для отримання результату обчислювальних операцій.

Як було зазначено в розділі 1.5 дисертаційного дослідження, організація обчислень з багаторозрядними числами, розмір яких перевищує значення діапазону стандартних типів даних сучасних мов програмування, потребує використання додаткових програмних модулів,

Бібліотеки довгих чисел, на яких заснована реалізація алгоритмів криптографії, як було виявлено в результаті досліджень [145], знижують швидкодію алгоритмів в 6-10 разів внаслідок великого числа звернень до оперативної пам'яті, і збільшують обсяг оперативної пам'яті яка витрачається в 4-5 разів, у порівнянні з реалізацією без використання цих бібліотек. В результаті таких накладних витрат на забезпечення роботи довгої арифметики математична оцінка складності деяких алгоритмів не збігається з одержуваними на практиці результатами їх роботи.

Основні результати алгоритмічного підходу до вирішення даної проблеми докладно представлені в класичній роботі Д. Кнута [50], але і на даний час вона залишається актуальною. Питанням представлення багаторозрядних чисел при організації паралельних обчислень, в тому числі і графічних адаптерах, присвячено багато сучасних робіт, Д. Орлов [146], И. Дзегеленок [147].

З урахуванням особливостей алрхітектури CUDA одним з найбільш зручним способом представлення «великих» чисел є використання масивів.

Число будемо представляти за деякою базою b (наприклад 1000). При цьому в одну клітинку масиву будемо записувати тільки один розряд числа. Це подання може бути більш незручним при виведенні числа, але, як правило, операції введення-виведення разові і не надають істотного впливу на час роботи алгоритму. З урахуванням вище сказаного для збереження десяткового числа з n розрядів необхідно мати масив з t елементів по вісім біт кожен, тобто для зберігання такого числа потрібно 8-м біт пам'яті. Для реалізації арифметичних операцій на графічному пристрої (GPU) з архітектури CUDA необхідно в його пам'яті помістити два числа, тобто 2t байт. В даний час в прикладних задачах захисту інформації використовують числа від 768 до 2048 біт, тобто від 233 до 620 десяткових розрядів. Для зберігання числа записаного в двійковій системі за допомогою 1024 біт потрібно масив з 310 елементів, тобто 310 байт пам'яті, для зберігання числа з 2048 біт достатньо

використовувати масив з 620 байт. Найпростіші з сучасних відеокарт мають об'єм пам'яті 512 Мбайт і вище, відповідно, на них без проблем можуть бути реалізовані операції з великими числами, представлені в такому вигляді.

Таким чином, обраний метод представлення великих чисел дозволяє реалізувати арифметику великих чисел без застосування бібліотеки великих чисел використовуючи тільки стандартні типи даних.

4.4. Технологія GPGPU в задачах криптоаналізу RSA-алгоритму.

Графічний процесор був обраний як цільова платформа з двох причин.

По-перше, паралельне програмування представляє потужний інтерес, і сучасні тенденції прогнозують, що апаратні реалізації які забезпечують паралельні обчислення зростатимуть з часом, розширюючись поза межі традиційної наукової спільноти.

По-друге, реалізація алгоритмів факторизації великих чисел на графічних процесорах може значно прискорити процес факторизації. Як вже було описано у першому розділі графічні процесори позбавлені багатьох недоліків центрального процесора. До недавнього часу відеоприскорювачі розглядалися лише як спеціалізовані пристрої, призначені тільки для обробки графіки.

Технологія NVIDIA CUDA - це єдина середа розробки на C, яка дозволяє програмістам і розробникам писати програмне забезпечення, на порядки прискорює складні обчислювальні завдання завдяки многоядерній обчислювальної потужності графічних процесорів.

4.5. Розпаралелювання MQkS.

Розглянемо кожен крок алгоритму MQkS на можливість застосування паралельних обчислень та подальшого скорочення часових затрат.

Загальний алгоритм був описаний у третьому розділі, на малюнку 3 зобразимо його у вигляді блок-схеми.

Розглянемо кожен крок алгоритму на можливість застосування паралельних обчислень та подальшого скорочення часу виконання алгоритму. Найбільш зручним для розпаралелювання є етап просіювання пробних *X*. Його можливо розбити на незалежні підзадачі які, які будуть виконуватися у паралельному режимі.

При реалізації паралельних алгоритмів треба враховувати апаратні обмеження. При застосуванні графічних карт, першим на найголовнішим обмеженням є існуючий обсяг пам'яті.

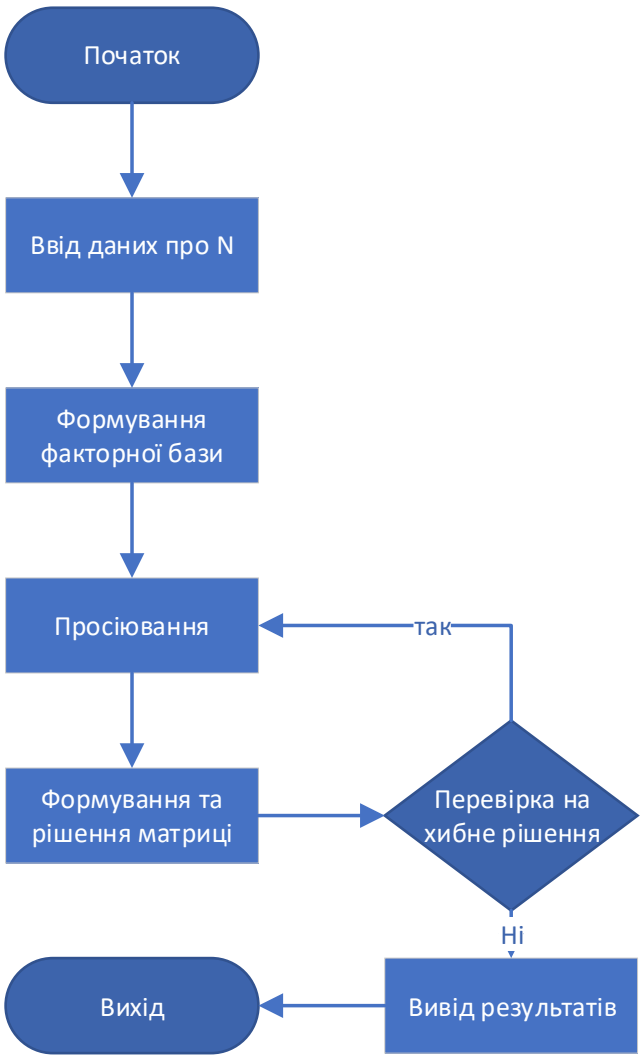


Рисунок 4.5 Алгоритм MQkS.

Кожний SM має доступ до колективної пам'яті (shared memory). Всі потоки з одного SM мають доступ до цієї пам'яті. Цей вид пам'яті розміщений на чіпі що забезпечує високу швидкість операцій зчитування та запису (дуже низька затримка 20-30 тактів, висока пропускна здатність 1000+ GB/s), та значно зменшує кількість

звернень до пам’яті яка розміщена за межами чіпу. Але розмір такої пам’яті становить 48 KB.

Суттєвий влив на необхідний розмір пам’яті має розмір ЗФБ. У методі MQkS розмір ЗФБ регулює параметр pLa. У таблиці 1 наведено залежність розміру факторної бази від параметру pLa для чисел розміру $N=2^{1024}=10^{308}$.

Таблиця 4.2

Залежність розміру ЗФБ від параметру pLa, для чисел $N=2^{1024}$

pLa	Розмір ЗФБ
1	$30*10^9$
0.95	$8*10^9$
0.9	$2*10^9$
0.85	$8*10^8$
0.8	$2*10^8$
0.75	$7*10^7$

Слід зазначити що зменшення параметру pLa призводить до значного збільшення часу факторизації, наприклад для чисел розміром 10^{30} та значенням параметру pLa=0.75 час факторизації в однопоточному режимі збільшився в 44 рази у порівнянні із значенням параметру pLa=1.

Найбільш зручним для розпаралелювання є етап просіювання пробних *X*. Його можливо розбити на незалежні підзадачі які, які будуть виконуватися у паралельному режимі. За одну таку задачу будемо вважати просіювання одного варіанту *kN* за одним підінтервалом. Алгоритм не передбачає зв’язку між різними варіантами *kN* та різними підінтервалами тому немає необхідності обміну даними між різними підзадачами. Для виконання однієї задачі необхідно зберегти інформацію про:

- Поточну факторнну базу масиви mfb, mp, lg_fa;
- Остачі від ділення N на прості p, масиви mpry, mpkr
- Інформація про x_0 та y_0, масиви X0, Y0
- Поточне значення $\sqrt{N}$, масив kr;

- Остача від значення Y, масив ry;
- сума логарифмів всіх простих на яке ділиться X, масив mlg;

Для зменшення необхідної пам’яті оберемо параметр pLa рівним 0.75. Обчислення об’єму необхідної пам’яті зображено у табл. 4.2

Таблиця 4.3

Необхідна інформація для виконання просіювання, для чисел $N=2^{1024}$, pLa=0.75

Опис змінної	кіль-ть змінних	тип (byte)	розмір
X0, Y0	50*2	int (4)	400
kff	1	int (4)	4
mlg - Log(x)	2*1000	double (8)	16000
mp – p прості числа	10^8	int (4)	$4*10^8$
lg_fa - $\log p$ з ФБ	10^8	double (8)	$8*10^8$
mfb(pfa) ПФБ.	10^8	double (8)	$8*10^8$
kr, ry	50*2	int (4)	800
mpry, mpkr остачі від ділення N на прості p	$2*10^3*5$	int (4)	40000
Сума			2 GB

З таблиці 4.3 видно, що необхідний об’єм пам’яті для забезпечення роботи однієї підзадачі дорівнює 2GB. Що є не допустимим для зберігання у колективній пам’яті графічних карт розміром 48 KB.

Тому було запропоновано використати попереднє просіювання із застосуванням сигнальних остач, яке було описано у другому розділі, для зменшення необхідного об’єму пам’яті.

При використанні параметра h на етапі попереднього просіювання використовуються тільки перші степені дільників $y_k(X)$, та немає потреби знаходження квадратних лишків для всіх степенів елементів факторної бази і їх подальшого використання.

Реалізацію попереднього просіювання доцільно проводити на кластерній системі, з об’ємом пам’яті 2 GB для кожної під-задачі.

А пошук дільників B-гладких чисел для відносно малих значень елементів ФБ, показники степеня яких перевищують одиницю, які не перевищують деякої границі - параметр *kff*, поводити на графічних картах.

Параметр *kff* дозволяє явно визначити кількість простих чисел, що будуть використані для пошуку дільників $Y_k(X)$, показник степеня яких перевищує одиницю. За рахунок вибору *kff* можна регулювати обсяг пам'яті який використовується для роботи із процесорами графічних карт.

Для зменшення необхідної пам'яті оберемо параметр *kff* рівним 0.25 тоді кількість простих степеня яких перевищує одиницю становитиме 420. Обчислення об'єму необхідної пам'яті зображено у табл. 4.4.

Таблиця 4.4

Необхідна інформація для виконання просіювання на графічних картах, для чисел N=2^{1024}, kff=0.25

Опис змінної	кіль-ть змінних	тип (byte)	розмір
X0, Y0	50*2	int (4)	400
kff	1	int (4)	4
mlg - Log(x)	2*1000	double (8)	16000
mp – p прості числа	420	int (4)	1680
mpr, mprm – номер простого числа та його степінь	2*420	int (4)	3360
lg_fa - $\log p$ з ФБ	420	int (4)	1680
mfb(pfa) ПФБ.	420	int (4)	1680
kr, ry	50*2	int (4)	400
mpry, mpkr остачі від ділення N на прості p	2*420*5	int (4)	16800
Сума			42 kB

З таблиці 4.4 видно, що необхідний об’єм пам’яті для забезпечення роботи однієї підзадачі дорівнює 42 kB. Що є допустимим для виконання на графічних картах.

Етап рішення матриці потребує великого масиву загальної пам'яті, доступ до якої розділяють усі процесори, тому доцільно застосовувати супер комп’ютер класу SMP, тобто із загальною пам’яттю.

З урахуванням зазначеного було запропоновано схему алгоритму А реалізації методу MQkS з урахуванням особливостей організації проведення розрахунків із застосуванням технології GPGPU CUDA (рис. 4.6).

Основні етапи алгоритму А методу MQkS відповідають представленому в розділі 2.5 опису, з урахуванням особливостей архітектури паралельних обчислень, які були описані вище.

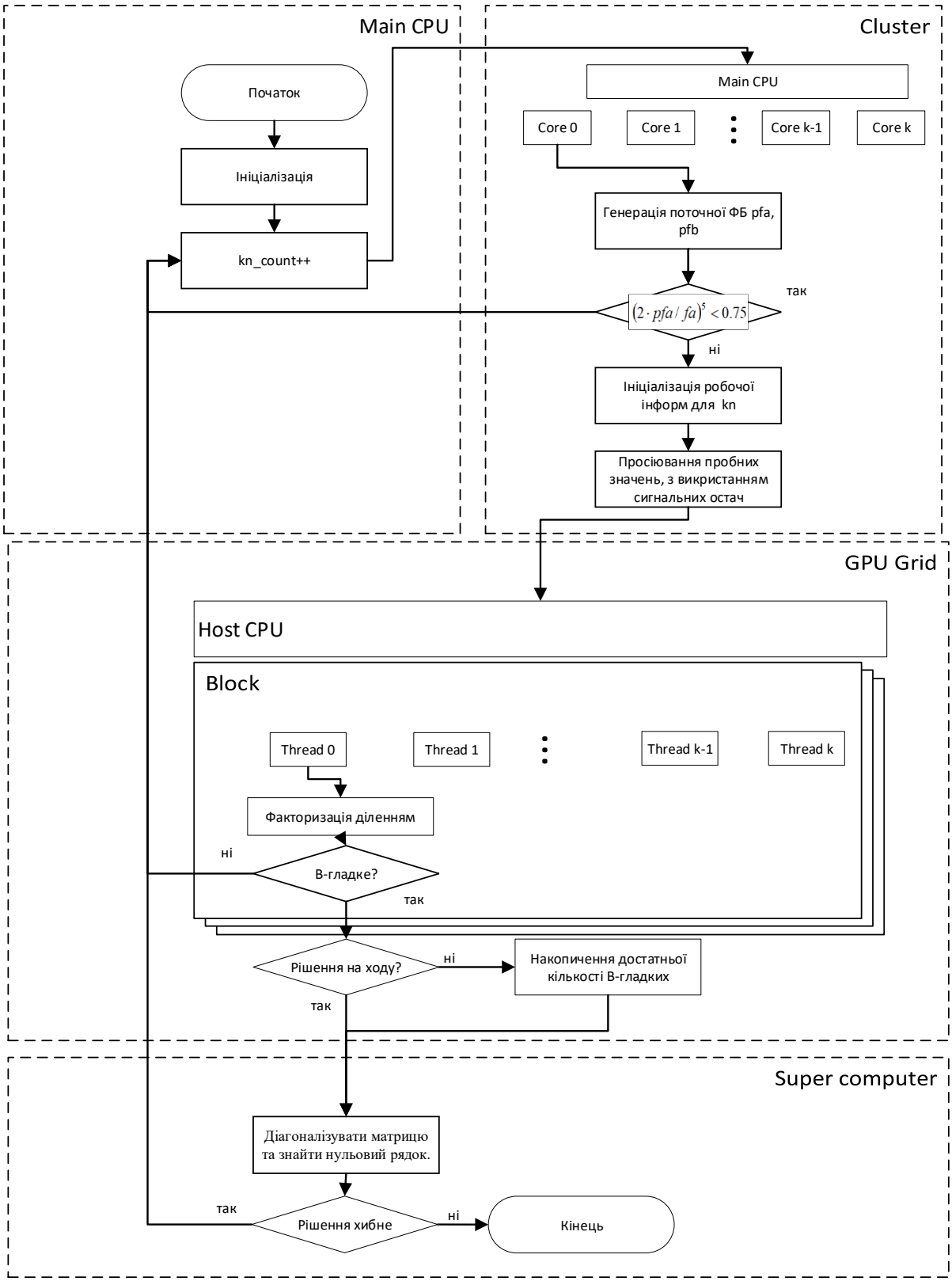


Рисунок 4.6 – Схема алгоритму А методу MQkS для реалізації із використанням технології GPGPU CUDA

4.6. Висновки до розділу 4.

У зв'язку з тим, що сучасні реалізації криптосистем захисту інформації на основі RSA-алгоритму для формування криптомодулю використовують багаторозрядні числа, вирішення завдань факторизації за допустимий проміжок часу вимагає залучення значних обчислювальних ресурсів.

Ідеальна платформа має ряд критеріїв, які необхідно виконати, при реалізації структури алгоритму квадратичного решета: можливість обробки великих цілих чисел, різноманітна швидка пам'ять, швидкі модулі обробки математики, паралелізація, вартість і доступність.

Аналіз існуючих рішень показав, що для реалізації запропонованого модифікованого методу MQkS можливо використання технології неспеціалізованих паралельних обчислень GPGPU на базі апаратно-програмної архітектури CUDA, яка на даний час є безперечним лідером в області високопродуктивних розподілених обчислень на базі GPU. Крім зазначеного, інтерфейс програмування додатків CUDA (CUDA API) заснований на стандартній мові програмування Сі, що в свою чергу дозволяє провести адаптацію існуючих і розроблених на мові Сі застосунків з мінімальними модифікаціями.

Важливу роль грає також пристосування самого алгоритму до виконання в паралельній режимі, та в умовах обмеженого об’єму пам’яті.

Запропонований метод MQkS є гнучким по відношенню до можливостей апартно-прогамних засобів, це може бути реалізовано на основі вибору параметрів які орієнтуються на можливості апартно-програмних засобів такимими параметрами є pLa, kff, та h. Наведено рекомендації використання параметрів pLa, kff, та h для вархування апаратних обмежень.

ВИСНОВКИ.

У дисертації отримано нове рішення наукової задачі, що полягає в розробці методів факторизації на основі методів QS та MPQS, які можна використовувати в сучасних апаратно програмних комплексах, що забезпечить зниження обчислювальної складності в порівнянні з уже існуючими методами QS та MPQS при вирішенні завдань криптоаналізу RSA алгоритму.

Основні наукові і практичні результати роботи полягають у наступному:

1. Розроблено метод множинного квадратичного *k*-решета (MQkS), в якому для пошуку B-гладких остач використовуються остачі $y_k(X) = X^2 - kN$, що при більшості значень *k* забезпечує пошук B-гладких серед всіх пробних $X = X_0 + x = \left\lfloor \sqrt{N} + 1 \right\rfloor + x$ в єдиному інтервалі просіювання, в якому, на відміну від методів QS та MPQS:

- використовується загальна факторна база (ЗФБ), утворена всіма найменшими простими числами починаючи з 2, кількість яких $fa = \left(\exp\left(\frac{\sqrt{2}}{4} \sqrt{\ln N \cdot \ln \ln N} \right) \right)^{pla} = \left(L^a\right)^{pla}$, де, $pla \in [0.5, 1.5]$ – параметр, а при кожному зі значень *k* з елементів ЗФБ формується ПФБ;

- розмір радіусу просіювання $fb = \left(L^a\right)^{plb}$, де $plb \in [0.5, 4]$ – параметр;

- на етапі просіювання реалізується попереднє просіювання пробних *X* на основі використання сигнальних остач $y^*(X)$, що є добутками перших степенів дільників $y_k(X)$ з числа елементів ЗФБ, при якому до множини відсіяних *X* відносяться ті, для яких виконана умова $\log\left(y_k^*(X)\right) < h \cdot \log\left(y_k(X)\right)$, де $h \in [0, 1]$ – параметр;

- при просіюванні пробних *X*, які не були відсіяні, пошук дільників остач $y_k(X)$, показник степеня яких може перевищувати одиницю, здійснюється для простих чисел з поточної факторної бази за умови, що для порядкового номера f_p простого *p* у списку простих чисел виконана умова $f_p \le ff = (L^a)^{kff}$, де $kff \in [0, 1]$ – параметр, при виборі якого можливе врахування даних про обмеження на обсяг пам'яті та доступні стандартні типи даних апаратних засобів;

- при пошуку нульового рядка матриці, елементи якої дорівнюють одиниці для непарних показників степеня дільників B-гладких чисел при рівних нулю інших значеннях, за рахунок перенумерації стовпчиків замість двох матриць з числом стовпчиків, що дорівнює *fa*, використовується одна, за рахунок чого розмір необхідної пам'яті суперкомп'ютера можна скоротити вдвічі.

При значеннях параметрів *pla* 0.9÷0.94, *plb* =1.4, *h*=0.7, *kff* = 0.4÷0.6 для визначеної множини чисел порядку 10^m, де *m*=20÷32, отримано значення коефіцієнту $C < 1$ в оцінці складності методу MQkS виду $O\left(\exp(C\sqrt{\ln N \ln\ln N})\right)$. У відомих оцінках обчислювальної складності методів методів QS та MPQS коефіцієнт $C \geq 1$. Для аналізованої множини чисел N порядку 10^m, де *m*=9÷32 встановлено також, що в порівнянні з методом QS кількість пробних *X*, на основі яких шукають B-гладкі, в 6 та більше разів перевищує їх кількість для аналогічного числа пробних в методі QS та зменшується час пошуку B-гладких.

2. Виявлені випадки отримання нульового стовпчика для матриці, що формується на основі показників степенів дільників B-гладких, та запропоновано метод діагоналізації матриці «на ходу», що в окремих випадках може забезпечити розкладання криптомодуля *N* на множники раніше ніж будуть знайдені *fa* + 2 B-гладкі остачі $y_k(X)$ незалежно від величини числа *N*. Даний результат може бути використаний для методів MQkS, QS, та MPQS.

3. Запропоновано метод визначення достатньої кількості B - гладких чисел, при використанні яких можна сформувати матрицю за показниками степенів дільників B-гладких, де можливими є випадки формування достатньої кількості B - гладких для отримання нульового рядка раніше, ніж буде знайдено *fa* + 2 B-гладких остач $y_k(X)$.

4. Встановлено, що серед остач $y_k(X)$ існуть такі, що $y_k(X) = y1(X) \cdot y2(X)$, де $y1(X)$ є добутком простих чисел з факторної бази, а $y2(X)$ – квадратом цілого числа. Такі числа названо умовно B-гладкими та показано, що існують випадки, коли на основі їх використання можливе скорочення часу отримання достатньої кількості B-гладких, хоча спосіб виявлення умовно B-гладких є дуже затратним в обчислювальному плані та необхідні подальші дослідження стосовно способів їх отримання.

5. Запропоновано алгоритм реалізації методу MQkS на апаратно-програмних засобах, які включають суперкомп’ютер, кластери та графічні процесори, в якому враховуються обмеження на стандартні типи даних та обсяг доступної пам’яті, а виконання арифметичних операцій з багаторозрядними числами, замінюються операціями з числами типу long (чи long long) та double.

6. Розроблено рекомендації стосовно використання розроблених методів в залежності від можливості апаратних засобів та загальна структура апаратно-програмних засобів при оцінці криптографічної стійкості RSA-алгоритму.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Факторизация числа N= pq при простых p и q методом дискретного логарифмирования. / ВН Мисько, СД Винничук, АВ Жилин. // Электронное моделирование . – 2013. – № 5 (35). – С. 3-10.
2. Ускорение метода Ферма методом прореживания с использованием нескольких баз. /В.М. Міський. // Журнал «Безпека інформації». Ukrainian Scientific Journal of Information Security. – 2015. – № 1 (21). – С. 64-68. DOI: 10.18372/2225-5036.21.8310
3. Удосконалення методу квадратичного решета на основі використання розширеної факторної бази та формування достатньої кількості B - гладких чисел. / В.М. Міський, С.Д. Винничук. // Збірник "Information technology and security". – 2017. – том. 5. выпуск. 2 (9). – С. 67-75.
4. Прискорення методу квадратичного решета на основі використання додаткового пошуку B-гладких чисел. / Міський В.М. // Моделювання та інформаційні технології. Збірник наукових праць. – 2017. – №78. – С. 51-57.
5. Acceleration analysis of the quadratic sieve method based on the online matrix solving. / V. Misko, S. Vynnychuk. // Eastern-European journal of Enterprise technologies. – 2018. – №10 (2). – С. 33-38. DOI: 10.15587/1729-4061.2018.127596
6. “Прискорення методу квадратичного решета на основі використання умовно B-гладких чисел” / Міський В.М. // Міжнародний науково–технічний журнал. Системні дослідження та інформаційні технології. – 2018. – №1. – С. 99-106. DOI: 10.20535/SRIT.2308-8893.2018.1.08
7. “Метод множинного квадратичного k-решета цілочисельної факторизації.” / В.М. Міський, С.Д. Винничук. // Електронне моделювання. – 2018. – №5 (40) С. 3-26. DOI: https://doi.org/10.15407/emodel.40.05.003
8. Просіювання пробних значень в методі множинного квадратичного k-решета на основі сигнальних остач. / С.Д. Винничук, В.М. Міський. // Безпека інформації. – 2019. – №1 (25). – С. 45-52. DOI: 10.18372/2225-5036.25.13446
9. “Метод множинного квадратичного k-решета з використанням сигнальних остач при просіюванні пробних значень.” / В.М. Міський, С.Д. Винничук. // Електронне

моделювання. – 2019. – №2 (41) С. 3-22. DOI: https://doi.org/10.15407/emodel.41.02.003

10. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания.” / В.М. Мисько. // Матеріали XXXIV Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 13-14 січня 2015 року. – тези доп. – м. Київ. – С.7.
11. “Ускорение метода Ферма факторизации чисел вида n = pq, где p и q простые, методом прореживания с использованием нескольких баз. ” / С.Д. Винничук, В.М. Місько. // Матеріали XXXVI Науково-технічної конференції «Моделювання» ІПМЕ ім. Г.Є. Пухова НАН України. 12-13 січня 2016 року. – тези доп. – м. Київ. – С.22.
12. “Прискорення методу квадратичного решета на основі використання розширеної факторної бази та формування достатньої кількості B-гладких чисел” / В.М. Місько. // ІІІ Міжнародна науково-практична конференція "Інформаційна безпека та комп’ютерні технології" 19-20 квітня 2018 року.: тези доп. – м. Кропивницький., – С. 106-108.
13. “Прискорення методу квадратичного решета на основі пошуку додаткових B-гладких чисел.” / В.М. Місько // Матеріали VI заочної наукової конференції «Наукові підсумки 2017 р», 13.11.2017. Науковий журнал «ScienceRise». – 2017. – №12(41). м. Харків. – С. 67-71. DOI: 10.15587/2313-8416.2017.118298
14. «Прискорення методу квадратичного решета на основі рішення матриці на ходу.» / В.М. Місько // Програма І Міжнародної науково-практична конференції “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS). 05-06 квітня 2018 р,: тези доп. – К., – С. 272-274.
15. «Прискорення методу квадратичного решета на основі рішення матриці на ходу». / В.М. Місько // Щорічна науково-технічна конференція молодих вчених та спеціалістів ІПМЕ ім. Г.Є. Пухова НАН України. 16 травня 2018 року, м. Київ. ,: тези доп. – К., – С. 29-30.

16. Множинне Квадратичне К-решето (MQKS). / С.Д. Винничук. В.М. Міська // Матеріали VI міжнародної наукової конференції «Моделювання-2018». 12-14 вересня 2018 р.: тези доп. – К., – С. 207-210 .
17. Множинне квадратичне k-решето факторизації чисел. / С.Д. Винничук В.М. Міська // Матеріали науково-практичної конференції «Сучасні інформаційні технології та кібербезпека», 15-16 листопада 2018р.: тези доп. – К., – С. 194-197.
18. Метод множинного квадратичного k-решета з використанням сигнальних остач при просіювання пробних значень. / В.М. Міська, С.Д. Винничук // XII Міжнародна науково-практична конференція «Комп'ютерні системи та мережеві технології» 28 –30 березня 2019 року.: тези доп. – К., – С. 27-28.
19. Верховна Рада України. (1992. Жовт. 02). *№2657-12, Закон України Про інформацію*. [Ел. ресурс]. Доступно: http://zakon2.rada.gov.ua/laws/show/2657-/12. Дата звернення: Лист. 25, 2014.
20. Верховна Рада України. (2011. Січ. 13). *№2939-17, Закон України Про доступ до публічної інформації*. [Ел. ресурс]. Доступно: http://zakon2.rada.gov.ua//laws/show/2939-17. (дата звернення: 24.10, 2018) Назва з екрану.
21. Верховна Рада України. (1994. Лип. 5). *№80/94 - вр, Про захист інформації в інформаційно-телекомунікаційних системах*. [Ел. ресурс]. Доступно: http://zakon2.rada.gov.ua//laws/show/80/94-%D0%B2%D1%80. (дата звернення: 22.06.2018) Назва з екрану.
22. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. М. : Триумф, 2002.
23. С.Коутинхо. Введение в теорию чисел. Алгоритм RSA. Москва: Постмаркет, 2001. - 328 с.
24. Н. Смарт, Криптография. Москва, Россия,: Техносфера, 2005.
25. А.П. Алферов, А.Ю. Зубов, А.С. Кузьмин, и А.В. Черемушкин, Основы криптографии. Москва, Россия: Гелиос АРВ, 2005.
26. А.В. Жилин, А.В. Корнейко, и В.В. Мохор, "Использование RSA алгоритма для обеспечения задач КЗІ в современных информационно-телекоммуникационных системах", *Захист інформації*, Т.15, № 3, с. 225-231, 2013.

27. Правління Національного банку України. (2015. Лист. 26). Постанова v0829500-15, Про затвердження нормативно-правових актів з питань інформаційної безпеки. [Ел. ресурс]. Доступно: http://zakon3.rada.gov.ua/ /laws/show/v0829500-15. (дата звернення: 11.04.2019) Назва з екрану.
28. Програмний додаток Інтернет-банкінгу iBank 2 UA. [Ел. ресурс]. Доступно: https://ibank2.ua/. (дата звернення: 09.03.2019) Назва з екрану.
29. Перелік акредитованих центрів сертифікації ключів [Ел. ресурс]. Доступно: http://www.ukrstat.gov.ua/elektr_zvit/inf_akcen/centr.htm. (дата звернення: 10.03.2019) Назва з екрану.
30. Перелік засобів КЗІ, які мають експертний висновок за результатами державної експертизи у галузі КЗІ. [Ел. ресурс]. Доступно: http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article?art_id=283948&cat_id=72110. (дата звернення: 13.03.2019) Назва з екрану.
31. Президент України. (1998. Трав. 22) Указ № 505/98, *Про Положення про порядок здійснення КЗІ в Україні*. [Ел. ресурс]. Доступно: http://zakon1.rada.gov.ua/laws/show/505/98. (дата звернення: 05.04.2019) Назва з екрану.
32. Щодо ліцензування у галузі криптографічного захисту інформації. [Ел. ресурс]. Доступно: http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article%3Bjsessionid=0131A20F50379E403E2941674E05C6A7?art_id=44645&cat_id=38712. (дата звернення: 12.04.2019) Назва з екрану.
33. Перелік суб'єктів господарювання, які мають ліцензії на провадження господарської діяльності у галузі КЗІ та ТЗІ, за переліком Уряду. [Ел. ресурс]. Доступно:http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article?art_id=284081&cat_id=266373. (дата звернення: 14.04.2019) Назва з екрану.
34. D. Boneh, Twenty years of attacks on the RSA acyptosystem, Notices of the AMS 46 (2) (February 1999) 203–213.
35. Yan S.Y. Cryptanalytic Attacks on RSA .Germany: Springer-Verlag, Heidelberg, 2008.

36. Kocher, P.C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, 1999. Pages 104 – 113.
37. Balasubramanian K., Rajakani M. The Quadratic Sieve Algorithm for Integer Factoring //Algorithmic Strategies for Solving Complex Problems in Cryptography. – IGI Global, 2018. – С. 241-252.
38. Tenenbaum, G.: Introduction to Analytic and Probabilistic Number Theory (3rd edn), Graduate Studies in Mathematics, vol. 163. American Mathematical Society, Providence (2015).
39. Koblitz N. and Menezes J. A. (2004) A Survey of Public- Key Cryptosystem
40. RSA Security (2004) what would it Take to Break the RSA Cryptosystem [online] Available from <http://www.rsasecurity.com/rsalabs/node.asp?id=2216 >.
41. Childs L. N. Factoring by the Quadratic Sieve //Cryptology and Error Correction. – Springer, Cham, 2019. – С. 293-312.
42. С.М. Авдошин, А.А. Савельева, "Криптоанализ и криптография", Бизнес-Информатика, № 2(8), с. 3-11, 2009.
43. Ю.І. Горбенко, Прикладна криптографія. Методи побудування та аналізу криптографічних систем: монографія. Харків, Україна: Форт, 2015.
44. Brown, "Breaking RSA May be as difficult as factoring", Journal of Cryptology, pp.1-20, 2008. [Ел. ресурс]. Доступно: https://eprint.iacr.org/2005/380.pdf . (дата звернення: 03.05.2019) Назва з екрану.
45. D. Aggarwal, and U. Maurer, "Breaking RSA Generically is Equivalent to Factoring", Advances in Cryptology - EUROCRYPT 2009, pp. 36-53, 2009.
46. Abubakar A, Jabaka S, Tijani BI. Cryptanalytic attacks on Rivest, Shamir, and Adleman (RSA) cryptosystem: issues and challenges. Journal of Theoretical and Applied Information Technology, 61 (1). 2014. pp. 37-43. ISSN 1817-3195 (O), 1992-8645 (P)
47. О.Н. Василенко, *Теоретико-числовые алгоритмы в криптографии*. Москва, Россия: МЦНМО, 2003.

48. С. Коутинхо, Введение в теорию чисел. Алгоритм RSA. Москва, Росія: Постмаркет, 2001.
49. Kraft J., Washington L. An introduction to number theory with cryptography. – Chapman and Hall/CRC, 2018.
50. Д. Кнут, Искусство программирования . Москва, Россия: Вильямс, 2013.
51. Указ президента україни. Про рішення Ради національної безпеки і оборони України від 29 грудня 2016 року «Про Доктрину інформаційної безпеки України» від 25 лютого 2017 року № 47/2017. [Електроний ресурс]. [2018]. – Режим доступу : https://www.president.gov.ua/documents/472017-21374
52. Стратегія розвитку інформаційного суспільства в Україні від 15 травня 2013 р. № 386-р. [Електроний ресурс]. [2018]. – Режим доступу : http://zakon.rada.gov.ua/laws/show/386-2013-%D1%80
53. Верховня Рада України. (2006. Лют. 23). Закон України № 3475-IV, *Про ДССЗЗІ України*. [Ел. ресурс]. Доступно: http://zakon2.rada.gov.ua/laws/show/3475-/15/page2. Дата звернення: Квіт. 28, 2015.
54. Granville A. *It is easy to determine whether a given integer is prime II* BULLETIN (New Series) OF THE AMERICAN MATHEMATICAL SOCIETY. Vol. 42. № 1. R 3-38.
55. Коблиц Н. Курс теории чисел и криптографии. — М.: ТВП. 2001.260 с.
56. Черемушкин А. В. Лекции по арифметическим функциям в крипто-графии. - М.: МЦНМО. 2002. 103 с.
57. L. Adleman, R. Rivest, and A.Shamir, "Method for obtaining digital signatures and public-key cryptosystems", *Comm. ACM*, № 21, pp. 120–126,1978.
58. Robinson C. P. The Key to Cryptography: The RSA Algorithm. – 2018.
59. Zhang X. et al. A Review of the Factorization Problem of Large Integers //International Conference on Artificial Intelligence and Security. – Springer, Cham, 2019. – С. 202-213.
60. Childs L. N. Cryptology and Error Correction. – Springer International Publishing, 2019.

61. A. Bortz, D. Boneh, and P. Nangy, Proceedings of the 16th International World Wide Web Conference, Ban Alberta. 2007.
62. D. Brumley, and D. Boneh, "Remote Timing Attacks are Practical", Proceedings of 12th Usenix Security Symposium.Wahsington, USA, 2003, pp.1-14.
63. K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic Analysis: Concrete Results", Gemplus Card International, France, 2001. [Ел. ресурс]. Доступно: http://citeseer.ist.psu.edu. Дата звернення: 24.11.19.
64. New openssl packages fix predictable random number generator. [Ел. ресурс]. Доступно: https://lists.debian.org/debian-security-announce/2008/msg00152.html. Дата звернення: Трав. 03, 2019.
65. A. Pellegrini, V. Bertacco, and T. Austin, "Fault-Based Attack of RSA Authentication", University of Michigan. [Ел. ресурс]. Доступно: http://web.eecs.umich.edu/~taustin/papers/DATE10-rsa.pdf. Дата звернення: Черв. 22, 2019.
66. А. Ализар, "RSA Security наконец признала компрометацию SecurID". [Ел. ресурс]. Доступно: https://xakep.ru/2011/06/07/55875/. Дата звернення: Лип. 20, 2019.
67. A. Shamir, and E. Tromer, "Acoustic cryptanalysis" [Ел. ресурс]. Доступно: http://www.wisdom./weizmann./ ac.il/~tromer/acoustic/. Дата звернення: Квіт. 23, 2019.
68. FREAK: Factoring RSA Export Keys. [Ел. ресурс]. Доступно: https://mitls.org/pages/ /attacks/SMACK#freak. Дата звернення: Жовт. 16, 2015.
69. The DROWN Attack. [Ел. ресурс]. Доступно: https://drownattack.com/. Дата звернення: Груд. 19, 2019.
70. Benne de Weger Cryptanalysis of RSA with Small Prime Difference de Weger Benne AAECC, The Netherlands. 2001. № 13. P. 17-28.
71. Pomerance C. Analysis and comparison of some integer factoring algorithms. / Carl Pomerance // Mathematisch Centrum Computational Methods in Number Theory, Pt. 1. / Carl Pomerance., 1982. – С. 89–139.

72. Kleinjung T., Aoki K., Franke J., Lenstra A., Thome E., Bos J., Gaudry P., Kruppa A., Montgomery P., Osvik D. A., te Riele H., Timofeev A., Zimmermann P. Factorization of a 768-bit RSA modulus. Online report. 18 Feb 2010.
73. Cavallar S., Lioen W. M., te Riele H. J., Dodson B., Lenstra A. K., Montgomery P .L., Murphy B. et al. Factorization of 512-bit RSA-modulus 11 CWI Report MAS-R0007, Feb 2000.
74. Lenstra A. et al Factoring Estimates for a 1024-Bit RSA Modulus // Advances in cryptology - Asiacrypt 2003. LNCS. 2003. Vol. 2894/2003. R 55-74.
75. Shiu P. Fermat's method of factorisation //The Mathematical Gazette. – 2015. – Т. 99. – №. 544. – С. 97-103.
76. Robert Erra, Christophe Grenier, The Fermat factorization method revisited, 2009, epreprint.org
77. Н.А. Калеников, и В.А. Минаев, "Улучшение метода ферма: новый алгоритм факторизации*", Российский новый университет*, № 4, с. 31-41, 2011.
78. J.M. Pollard, "Theorems on factorization and primality testing", *Proc. Cambridge Phil. Soc*., № 76, pp. 521-528, 1974.
79. Wolfram Math World. Mathematics resource. Number Field Sieve. [Ел. ресурс]. Доступно: http://mathworld.wolfram.com/NumberFieldSieve.html. Дата звернення: Жовт. 13, 2014.
80. D. Coppersmith, "Modifications to the Number Field Sieve", Cryptology, №6, pp.169-180, 1993.
81. V.P. Hoang, Integer Factorization with the General Number Field Sieve, Rovaniemi University of Applied Sciences. [Ел. ресурс]. Доступно: http://www.ramk.fi/loader./aspx?id=84041118-a526-414f-847f-5ceb0afec3b1. Дата звернення: Лист. 13, 2018.
82. Pomerance C. The quadratic sieve factoring algorithm // Advances in cryptology (Paris, 1984). 1985. (Lecture Notes in Comput. Sci.; V. 209). P. 169—183.
83. C. Pomerance, and A Pipeline, Architecture for Factoring Large Integers with the Quadratic Sieve Method, № 17, pp. 387-403, 1988.

84. J. Gerver, "Factoring Large Numbers with a Quadratic Sieve", Math. Comput, № 41, pp. 287-294, 1983.
85. Wolfram Math World. Mathematics resource. Quadratic Sieve. [Ел. ресурс]. Доступно: bhttp://mathworld.wolfram.com/QuadraticSieve.html. Дата звернення: 13.10.2015.
86. H.W. Lenstra, "Factoring integers with elliptic curves", Annals of Mathematics, №126, pp.649-673, 1987.
87. H.W. Lenstra, "Elliptic curves and number-theoretic algorithms", International Congress of Mathematicians,pp. 99-120, 1986.
88. C. Pomerance, H. Lenstra, "Analysis and comparision of some integer factoring algorithms", Computational methods in number theory, №1, pp. 89-139, 1982.
89. R. S. Lehman, "Factoring large integers", *Math. Comp*., № 78, pp. 637-646, 1976-77.
90. Ш.Т. Ишмухаметов, и Р.Г. Рубцова, *Математические основы защиты информации*. Казань, Россия, 2012.
91. А. Бойко, Д. Зиятдинов, и Ш. Ишмухаметов, "Об одном подходе к проблеме факторизации натуральных чисел". *Russian Math*, № 55, с. 12-17, 2011.
92. Я.І. Кінах, "Методи паралельних обчислень та обґрунтування рівня криптографічного захисту інформації в комп'ютерних мережах". Дисертація на здобуття наукового ступеня к.т.н.: 05.13.13, Тернопіль, Україна. 2007.
93. S.Maitra, S. Sarkar and S. Gupta, "Factoring RSA modulus using prime reconstruction from random known bits", *Lecture Notes in Computer Science*, № 6055, pp.82-99, 2010.
94. K.A. Kanabar, "A Comparison Of Integer Factoring Algorithm", *The University of Bath*. 2007. [Ел. ресурс]. Доступно: http://www.cs.bath.ac.uk/~mdv/courses/ /CM30082/projects.bho/2006-7/Kanabar-KA-dissertation-2006-7.pdf. Дата звернення: Трав. 05, 2019.
95. А.Е. Баранович, Эффективные вычисления в архитектуре CUDA в приложениях информационной безопасности. Москва, Россия, 2014.
96. А.В. Макаренко, А.В. Пыхтеев, и С.С. Ефимов, "Параллельная реализация и сравнительный анализ алгоритмов факторизации в системах с распределенной памятью". Мат. структуры и моделирование, №26, с. 94-109, 2012.

97. L.M. Itu, F. Moldoveanu, and C. Suciu, A. Postelnicu, "Comparison of single and double floating point precision performance for Tesla architecture GPUs", Bulletin of the Transilv. Univ.of Braşov: Engin. Sciences, Vol. 4 (53), №. 2, pp. 131-138, 2011.
98. Crandall R., Pomerance C. Prime Numbers A Computational Perspective / New York, 2005. C. 597.
99. Granville A. *Smooth numbers: computational number theory and beyond II* Algorithmic Number Theory MSRI Pubhcations. 2008. Vol. 44.
100. Wilhams C. P., Clearwater S. H. *Ultimate zero and one: computing at the quantum frontier.* — Springer-Verlag New York. Oct 1999.
101. Dixon J.D. Asymptotically fast factorization of integers / J.D. Dixon.– Math. Comp. 36, 1981, p. 255–260.
102. Pomerance C.Factoring// Proc. of Symp. Appl. Math. 1990. V. 42. P. 2 4 —47.
103. Sieve_of_Eratosthenes. [Ел. ресурс]. Доступно: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes Дата звернення: 13.08.2019.
104. Boender H., te Riele H. J. J. Factoring integers with large prime variations of the quadratic sieve / CWI Report NM-R9513. 1995.
105. Caron T. R., Silverman R. D. Parallel implementation of the quadratic sieve. J. Supercomputing. 1988. V. 1. P. 273—290.
106. Contini S. Factoring integers with the self initializing quadratic sieve / Master's thesis. Univ. Georgia, 1997.
107. Peralta R. Implementation of the hypercube multiple polynomial sieve. Preprint.
108. Pomerance C., Smith J. W., Tuler R. A pipeline architecture for factoring large integers with the quadratic sieve algorithm // SIAM J. Comput. 1988. V. 17 (2). P. 387—403.
109. Silverman R. D. The multiple polynomial quadratic sieve // Math. Comp. 1987. V. 48 (177). P. 329—339.
110. te Riele H. J. J., Lioen W., Winter D. Factoring with the quadratic sieve on large vector computers // Belgian J. Comp. Appl. Math. 1989. V. 27. P. 267—278.
111. Cohen H. A course in computational algebraic number theory. Springer-Verlag, 1993.

112. Landquist E. The Quadratic Sieve Factoring Algorithm // MATH 488: Cryptographic Algorithms. 2001. URL: http://www.cs.virginia.edu/crab/QFS_Simple.pdf (дата звернення: 18.02.2018).
113. Arjen K. Lenstra. General purpose integer factoring. EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland. pp 23-44. 2017.
114. Pomerance C. Smooth Numbers and the Quadratic Sieve / C. Pomerance. – MSRI publications, v.44 – 2008, p. 69–82.)
115. Mark Janeba (1994), Factoring Challenge Conquered. [Ел. ресурс]. Доступно: http://www.willamette.edu/~mjaneba/rsa129.html . Дата звернення: Бер. 13, 2018.
116. Pomerance, Carl. Cryptology and Computational Number Theory; Factoring. AMS, Providence, RI, 1990.
117. T. Denny, B. Dodson, A. K. Lenstra, M. S. Manasse. On the factorization of RSA-120. Advances in Cryptology - CRYPT0 '93, LNCS 773, pp. 166-174, 1994
118. Song Y. Quadratic Sieve / Yan Song // Primality Testing and Integer Factorization in Public-Key Cryptography Second Edition / Yan Y. Song., 2008. – С. 234–239.
119. MSDN Archive. (2006). Factoring large numbers with quadratic sieve. . [Ел. ресурс]. Доступно: https://blogs.msdn.microsoft.com/devdev/2006/06/19/factoring-large-numbers-with-quadratic-sieve (дата звернення: 18.02.2018).
120. Бабенко Л.К., Ищукова Е.А., Сидоров И.Д. Параллельные алгоритмы факторизации для анализа асимметричных криптосистем // Материалы международной конференции «Моделирование устойчивого регионального развития». – Нальчик: Издательство КБНЦ РАН, 2011. – с. 78-84.
121. Бабенко Л.К., Сидоров И.Д. Параллельные алгоритмы криптоанализа асимметричных систем // Актуальные аспекты защиты информации в Южном федеральном университете. Монография. – Таганрог: Изд-во ТТИ ЮФУ, 2011. – с. 207-252
122. GPGPU.RU Использование видеокарт для вычислений [Электронный ресурс]. – Электрон. дан. - [2011]. - Режим доступа свобод.: http://www.gpgpu.ru/

123. General-purpose computing on graphics processing units. [Электронный ресурс]. – Электрон. дан. – [2011]. – Режим доступа свобод.: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

124. nVidia CUDA — неграфические вычисления на графических процессорах [Электронный ресурс]. – Электрон. дан. – [2010]. – Режим доступа свобод.: http://www.ixbt.com/video3/cuda-1.shtml

125. Баранович А.Е., Желтов С.А. Гетерогенные архитектуры массовых вычислений и новые угрозы кибербезопасности // Системы высокой доступности. – 2012. – № 2. – т.8. – с. 16-22

126. Graphics Add-in Board Shipments Decline 15.2% from Last Quarter [Электрон. ресурс]. - Электрон. дан. – [2011]. – Режим доступа свобод.: http://jonpeddie.com/publications /add-in-board-report/

127. Обчислювальний комплекс «СКИТ» (. [Электронный ресурс]. – Электрон. дан. – [2018]. – Режим доступа свобод.: http://icybcluster.org.ua/index.php?lang_id=2&menu_id=5

128. Victor Zhenyu Guo; William David Banks (May 2017). Exponential sums, character sums, sieve methods and distribution of prime numbers. [Columbia, Missouri] : [University of Missouri].

129. Anthony Vazzana; David Garth; Martin J Erickson. Introduction to number theory. Boca Raton : Chapman & Hall/CRC, 2015.

130. D Breitenbacher, I Homoliak, J Jaros, P Hanacek. Impact of Optimization and Parallelism on Factorization Speed of SIQS. Journal of Systemics 2016; 4 (3), 51-58.

131. B.A. LaMacchia and A.M. Odlyzko, Solving large sparse systems over finite fields, Advances in Cryptology, CRYPT0 '90 (A.J. Menezes and S.A. Vanstone, eds.), Lecture Notes in Computer Science, vol. 537, Springer-Verlag, pp. 109-133.

132. Douglas H. Wiedemann, Solving sparse linear equations over finite fields, IEEE Trans. Inform. Theory 32 (1986), no. 1, 54-62.

133. Montgomery P.L. A block Lanczos algorithm for finding dependences over GF(2)/ P.L. Montgomery.– in Advances in Cryptology: Eurocrypt’95, Lect.Notes in Comp.Sci. 921, Springer–Verlag, Berlin, p. 106–120.

134. Elkenbracht-Huising M. An implementation of the Number Field Sieve / M. Elkenbracht-Huising.– Experimental Mathematics, 1996, v.5, p. 231-253.

135. Brian Berenbach, Daniel Paulish, Juergen Katzmeier, Arnold Rudorfer (2009). Software & Systems Requirements Engineering: In Practice. New York: McGraw-Hill Professional. ISBN 0-07-1605479.

136. А.В. Богданов, В.В. Корхов, В.В. Мареев, и Е.Н. Станкова, Архитектуры и топологии многопроцессорных вычислительных систем. Курс лекций. Учебное пособие. Москва, Россия: ИНТУИТ.РУ. 2004.

137. В.М. Глушков, и др., "Функциональная структура и элементы сетей ЭВМ", УСиМ, № 3, с. 1-15, 1975.

138. В.М. Глушков, А.И. Никитин, Сети с коммутацией пакетов (состояние проблемы и пути ее решения), Вычислительные сети коммутации пакетов, Рига, 1979, с. 12-21.

139. В. Глушков, А. Никитин, Некоторые проблемы создания и развития сетей ЭВМ, ПО вычислительных сетей и систем реального времени, Киев, 1981, с. 3-6.

140. Quarter-to-quarter graphics board shipments decreased 29.8%, and decreased 19.2% year-to-year. [Ел. ресурс]. Доступно: http://jonpeddie.com/publications/ /add-in-board-report/. Дата звернення: Лип.12,2019.

141. А. Барський, Grid-вычисления:органызацыя, методы, планирование. LAP LAMBERT Academic Publishing, 2012.

142. Кабінет Міністрів України. (2009. Вер. 23). Постанова № 1020 Про затвердження Державної цільової науково-технічної програми впровадження і застосування грід-технологій на 2009-2013 роки. [Ел. ресурс]. Доступно: http://zakon5.rada.gov.ua/laws/show/1020-2009-%D0%BF. Дата звернення: Лют. 22, 2019.

143. Список ресурсів УНГ. [Ел. ресурс]. Доступно: http://ung.in.ua/ua/resourses_arc/?p=1. Дата звернення: Серп. 08, 2018.

144. Применение технологии CUDA при компьютерном моделировании тепловых процессов. [Ел. ресурс]. Доступно: http://www.nvidia.ru/object/cuda-modeling-/thermal-processes-tesla-case-study-ru.html/. Дата звернення: Вер. 21, 2016.

145. Неласая А.В. Оценка эффективности вычислений в библиотеках длинной арифметики / А.В. Неласая, М.И. Верещак // Проблеми і перспективи розвитку ІТіндустрії: ІІ Міжнародна науково-практична конференція, 18 – 19 листопада 2010 р. – Харків. – С.226-227.
146. Д.А. Орлов, "Реализация арифметики повышенной разрядности на графических процессорах", Программная инженерия, №4, с.33-43, 2012.
147. И.И. Дзегеленок, и Ш.А. Оцопков, "Алгебраизация числовых представлений для обеспечения высокоточных суперкомпьютерных вычислений", Вестник Московского энергетического института, № 3, с. 107–116, 2010

ДОДАТОК А

Впровадження результатів дисертації

ЗАТВЕРДЖУЮ
Начальник ІСЗЗІ КПІ
ім. Ігоря Сікорського
полковник О.О.Пучков
04 06 2019 р.

АКТ
про використання результатів дисертаційної роботи
Міська Віталія Миколайовича
на тему «Обчислювальні методи на основі квадратичного решета при криптоаналізі RSA алгоритму апаратно-програмними засобами»

Комісія у складі завідуючого спеціальної кафедри №1 Роми О.М., професора спеціальної кафедри №1 Олексійчука А.М., доцента спеціальної кафедри №1 Самойлова І.В. підтверджує, що результати дисертаційної роботи В.М. Міська на тему «Обчислювальні методи на основі квадратичного решета при криптоаналізі RSA алгоритму апаратно-програмними засобами» використовуються у навчальному процесі з 2018 р. на спеціальної кафедри №1 при викладанні навчальних дисциплін «Теоретична криптологія», «Математичні методи побудови та аналізу асиметричних криптосистем».

В навчальному процесі використовуються наступні результати, що отримані в дисертаційній роботі В.М. Міська:

1. Метод множинного квадратичного *k*-решета (MQkS), в якому для пошуку B-гладких остач використовуються остачі $yk(X) = X^2 - kN$, що при більшості значень k забезпечує пошук B-гладких серед всіх пробних $X = X_0 + x = \lfloor \sqrt{N} + 1 \rfloor + x$, на відміну від методів MPQS. В алгоритмі методу передбачено використання чотирьох параметрів, на основі яких можливе налаштування методу, при якому стає можливим використання сучасних апаратно-програмних засобів для факторизації криптомодуля *N*.

2. Метод діагоналізації матриці «на ходу», що в окремих випадках може забезпечити розкладання криптомодуля *N* на множники раніше ніж буде знайдено кількість B-гладких остач $y_k(X)$, яка перевищує число елементів факторної бази незалежно від величини числа *N*.

3. Метод визначення достатньої кількості B-гладких чисел, при використанні яких можна сформувати матрицю за показниками степенів дільників B-гладких, де можливими є випадки формування достатньої кількості B-гладких для отримання нульового рядка раніше, ніж буде знайдено кількість B-гладких остач $y_k(X)$, яка перевищує число елементів факторної бази незалежно від величини числа *N*.

Обчислювальні методи, запропоновані у дисертації В.М. Міська, можуть використовуватися як метод факторизації крипто модуля RSA *N*, для якого забезпечується можливість використання сучасних апаратно-програмних засобів з урахуванням обмежень апаратних реалізацій при вирішенні завдань криптоаналізу RSA алгоритму, де принципово можливе існування чисел *N*, для яких даний метод характеризується нижчою обчислювальною складністю у порівнянні з уже існуючими методами факторизації.

Завідувач спеціальної кафедри №1
д.т.н., с.н.с. О.М. Рома

Професор спеціальної кафедри №1
д.т.н., доцент А.М. Олексійчук

Доцент спеціальної кафедри №1
к.т.н., доцент І.В. Самойлов

ДОДАТОК В

Програмний додаток mpks.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define N_B    12   // Число блоків
#define N_F_B  842  // Число простих для загальної факторної бази
#define NFB2   842  // Корінь з числа простих загальної факторної бази

long mp[14684] = {  1,  2,  ... };

     // Це елементи факторних баз

int mpr[32] = { .... };
int mprc[32] = { ... };
long mpb[49][6] = { .... };

long m0pb[49][7] = { .... };
 //int nfb[105];  // інформація про перенумерацію елементів ФБ при діагоналізації матриці

 int nn[(4*N_B+5)];  // Число, що розкладається
 int nk[(4*N_B+5)];  // Число, що розкладається домножене на k
 int rc[(4*N_B+5)];  // Робочий масив для числа, з якого шукаємо корінь
 int kr[(2*N_B+5)];  // Робочий масив для кореня з числа RC
 int rx[(2*N_B+5)];  // Корінь X з числа C в робочому масиві RC[]
 int ry[(2*N_B+5)];  // Остача RY=X*X - C
 int yy[(2*N_B+5)];  // Початкова остача RY=X*X - C
 int r0[(2*N_B+5)];  // Остача RY=X*X - C
 int rr[(2*N_B+5)];  // Остача RY=X*X - C

 long kb,nb4,nb2,i0,i,i1,i2,j,jj,j2,j3,j4,zz,k,k0,k1,k2,k3,k4,k5,n,p,q,ac,ac1,r,t,t1,t2,x1,x2;
 long m,m1,m2,m3,m4,m9,m0,f,f1,f2,f3,f4,f5,fb,fa,lla,f0,f00,fa00,s,s1,s2,kn;
 long c,c0,c1,c2,c3,c4, ff, cff;
 int mb[(N_F_B+1)],kfb[(N_F_B+1)],b,bb,b0,bc,bc0;
 long p0,q0,pa,qa,np,nq,ch_bh,c_bh,f_kn,ff_kn,rf_kn,p_r,k_r,t_r,nfb;

 double b00, a, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a00, a20, z, z1, ll10, logn, lg1, logkr, kff, klkr;
 //double lg_fa[(N_F_B+1)],mlg[1001],lg_b0,lg_ry,lg_yy,csum, pfa, pfb, afb, mc1,mc0,lg_kr,lg_ry,lg_yy,ry_kr,yy_kr;

 //int ml2[(N_F_B+1)], mln[(N_F_B+1)], mlkn[(N_F_B+1)],beta[(N_F_B+10)][2], mcp[1001], mcm[1001];
```

```
 double lg_fa[(N_F_B+1)],mlg[1001],lg_b0,lg_ry,lg_yy,csum, pfa, pfb, afb,
mc1,mc0,lg_kr,lg_ry,lg_yy,ry_kr,yy_kr;
 double mxd[12], mrd[12], mxc[12];

 int ml2[(N_F_B+1)], mln[(N_F_B+1)], mlkn[(N_F_B+1)], beta[(N_F_B+1)][2],
mcp[1001], mcm[1001], mrc[12];

 int mpn[(N_F_B+1)];
 int mpnk[(N_F_B+1)];
 int mx1[(N_F_B+1)];
 int mxkr[(N_F_B+1)];
 int mxrr[(N_F_B+1)];
 int mpkr[NFB2][11];
 int mprr[NFB2][11];
 int mpry[NFB2][11];
 int mpyy[NFB2][11];


 int rrx[(2*N_B+5)];
 int rry[(2*N_B+5)];
 long rcl[51], mmc[12];

 int sp[1001][20], sm[1001][20], mfb[NFB2]; //
 int shp[1001], shm[1001];
 int mnm[51],mm0[51],mm1[51],nkl,p0max,c00;
 FILE *fr, *frp, *fp, *flt;

main()
{
 n=0;  p=0;  q=0;
 k=0; i=0; c=0;
 b0=1000; b00=1000.0; ll10=log(b00);
 nfb=(long)(sqrt(0.5+N_F_B)+2.5);

 fr =fopen("f_r77.txt","w");
 frp=fopen("f_b77.txt","w");
 flt=fopen("f_t77-100-14-k101-h07(18).txt","w");


 nb2=2*N_B+2; nb4=4*N_B+2;
 a=0.0;
 a0=0.0; a1=0.0; a2=0.0; a3=0.0; a4=0.0;
 a5=0.0; a6=0.0; a7=0.0; a8=0.0; a9=0.0;
```

```
 if ( (fp=fopen("param_qs77.txt","r") ) == NULL )
 {
  printf("\n Nevozmojno otkritj file dannyx param_qs77.txt  ");
  exit(0);
 } // файл параметрів: p0, q0, pa, qa, p0max, kfa, kfb
 if ( fscanf(fp," %ld %ld %ld %ld %ld %lf %lf %lf %lf ",&p0, &q0, &pa, &qa, &p0max,
&pfa, &pfb, &kff, &klkr)==9 ) k=1;
 else
 { //kff,klkr,kmmc
  fprintf(fr,"\n Parametry  p0, q0, pa, qa, p0max, pLa, pLb, kff, klkr ne vvedeno \n");
  exit(0);
 }

 k=p0max-p0;
 fprintf(frp,"\n qs_5pu.exe: Параметри розрахунку              :  \n");
 fprintf(frp,"\n Початковий номер опорного P                    : P0  = %ld ", p0);
 fprintf(frp,"\n Початковий номер опорного Q                    : Q0  = %ld ", q0);
 fprintf(frp,"\n Число значень простих для опорного P0          : PA  = %ld ", pa);
 fprintf(frp,"\n Число значень простих для опорного Q0          : QA  = %ld ", qa);
 fprintf(frp,"\n Максимальний номер опорного P0                 : P0max = %ld ", p0max);
 fprintf(frp,"\n Коефіцієнт розміру факторної бази FA = La^pLa     : pLa = %g ", pfa);
 fprintf(frp,"\n Коефіцієнт розміру радіуса просіювання FB = La^pLb: pLb = %g ",
pfb);
 fprintf(frp,"\n Коефіцієнт обмеження на першу степінь  FF = La^kff: kff = %g ", kff);
 fprintf(frp,"\n Коефіцієнт відсіювання      KLKR = 0.5*logN/logYY : klkr =%g ",klkr);

  fprintf(flt,"\n Параметри розрахунку              :  \n");
 fprintf(flt,"\n Початковий номер опорного P                    : P0  = %ld ", p0);
 fprintf(flt,"\n Початковий номер опорного Q                    : Q0  = %ld ", q0);
 fprintf(flt,"\n Число значень простих для опорного P0          : PA  = %ld ", pa);
 fprintf(flt,"\n Число значень простих для опорного Q0          : QA  = %ld ", qa);
 fprintf(flt,"\n Максимальний номер опорного P0                 : P0max = %ld ", p0max);
 fprintf(flt,"\n Коефіцієнт розміру факторної бази FA = La^pLa     : pLa = %g ", pfa);
 fprintf(flt,"\n Коефіцієнт розміру радіуса просіювання FB = La^pLb: pLb = %g ", pfb);
 fprintf(flt,"\n Коефіцієнт обмеження на першу степінь  FF = La^kff: kff = %g ", kff);
 fprintf(flt,"\n Коефіцієнт відсіювання      KLKR = 0.5*logN/logYY : klkr =%g ",klkr);

 if ( p0<0 || q0<0 || p0==q0 || pa<1 || pa>5 || qa<1 || qa>5 || p0max<0 || p0max>48 ||
k+q0>48 || klkr>0.99 )
 {
  fprintf(fr,"\n p0<0 || q0<0 || p0==q0 || pa<1 || pa>5 || qa<1 || qa>5 || p0max<0 || p0max>48
|| k+q0>48 || klkr>0.99");
  fprintf(fr,"\n Порушені обмеження для параметрів p0, q0, pa, qa, p0max, klkr. ");
```

```
 fprintf(fr,"\n p0=%ld  q0=%ld  pa=%ld  qa=%ld   p0max=%ld  ",p0, q0, pa, qa, p0max );
 exit(0);
}
if ( pfa<0.5 || pfa>2.0 || pfb<0.5 || pfb>3.0 || kff>1.5 )
{
 fprintf(fr,"\n pLa<0.5 || pLa>2.0 || pLb<0.5 || pLb>3.0 || pLb>3.0  || kff>1.5 ");
 fprintf(fr,"\n Порушені обмеження для параметрів pLa, pLb чи kff. ");
 fprintf(fr,"\n pLa=%g  pLb=%g  kff=%g  ",pfa, pfb, kff);
 exit(0);
}
p0=p0-2; q0=q0-2;
met_p0: p0=p0+2; q0=q0+2;
c_bh=0; // число B-гладких
//for (i=0; i<12; i++) { mmc[i]=0; mxd[i]=0.0; mrd[i]=0.0; mxc[i]=0.0; }

p_r = clock();
t_r = p_r;

k=0; rrx[0]=m0pb[p0][0]; rry[0]=m0pb[q0][0];
for (f=1; f<nb2; f++) { rrx[f]=0; rry[f]=0; rx[f]=0; ry[f]=0; }
fprintf(fr,"\n\n  !!!!!!!!!! p0=%ld    q0=%ld  !!!!!!!!!!!! ",p0,q0);

for (f=1; f<=rrx[0]; f++) rrx[f]=m0pb[p0][f];
for (f=1; f<=rry[0]; f++) rry[f]=m0pb[q0][f];

np=1; nq=1;
for (f=1; f<nb4; f++) { nn[f]=0; nk[f]=0; rc[f]=0; }
for (np=1; np<=pa; np++)
{
 k=mpb[p0][np];

 j=k; rx[0]=rrx[0];
 for (i=1; i<rrx[0]+2; i++)
 {
  jj=rrx[i]+j;
  if ( jj>=b0 ) { j=jj/b0; rx[i]=jj%b0; } else { j=0; rx[i]=jj; }

 }
 k=rx[0]; if ( rx[k+1]>0 ) rx[0]=k+1;

 csum = 0.0;
 for (nq=1; nq<=qa; nq++)
 {
  k=mpb[q0][nq]; ry[0]=rry[0];
```

```
fprintf(fr,"\n\n q0=%3ld  nq=%3ld  mpb[q0][nq]=%4ld  ",q0,nq,k);
fprintf(frp,"\n\n q0=%3ld  nq=%3ld  mpb[q0][nq]=%4ld  ",q0,nq,k);
j=k;
for (i=1; i<rry[0]+2; i++)
{
 jj=rry[i]+j;
 if ( jj>=b0 ) { j=jj/b0; ry[i]=jj%b0; } else { j=0; ry[i]=jj; }
 //if ( j==0 ) break;
}
k=ry[0]; if ( ry[k+1]>0 ) ry[0]=k+1;
for (j=0; j<=rx[0]; j++) fprintf(frp," rx[%2ld]=%4d ",j,rx[j]);

fprintf(frp,"\n Masyv  ry[].  ");
for (j=0; j<=ry[0]; j++) fprintf(frp," ry[%2ld]=%4d ",j,ry[j]);

for (j=0; j<nb4; j++) nn[j]=0;

for (i=1; i<=rx[0]; i++)
{ k=i-1;
      for (j=1; j<=ry[0]; j++)
 {
  jj=rx[i]*ry[j];
  nn[k+j]=nn[k+j]+jj;
 }
}
nn[0]=rx[0]+ry[0];

j=0; k=nn[0]+1;
for (i=1; i<=k; i++)
{
 jj=nn[i]+j;
 j=jj/b0; nn[i]=jj%b0;
}
if ( nn[k]==0 ) k--;
if ( nn[k]==0 ) k--;
if ( nn[k]==0 ) k--;
nn[0]=k;  // Сформовано поточне значення N
ch_bh=0; // число B-гладких
fprintf(frp,"\n Masyv  nn[]. np=%3ld  nq=%4d ",np,nq);
for (j=nn[0]; j>=0; j--) fprintf(frp," nn[%2ld]=%4d",j,nn[j]);
fprintf(fr,"\n\n\n Masyv  nn[]. np=%3ld  nq=%4d ",np,nq);
for (j=nn[0]; j>=0; j--) fprintf(fr," nn[%2ld]=%4d",j,nn[j]);
lg_b0=log(b00);
a=0.0; m=0;
```

```
j=nn[0]-4; if ( j<1 ) j=1;
a=0.0; for (i=nn[0]; i>j; i--) a=a*b0+nn[i];

a1=log(a)+ll10*j; // logN
logn = a1/log(10.0);
a2=log(a1); // loglogN
a3=sqrt(2.0*a1*a2)/4.0; // Показник степеня експоненти
fa=(long)(exp(pfa*a3)+0.5); // розмір факторної бази
lla=(long)(exp(a3)+0.5); // La - базовий розмір факторної бази
fb=(long)(exp(a3*pfb)+0.5); // радіус просіювання
ff=(long)(exp(kff*a3)+0.5); // La - базовий розмір факторної бази

if ( fa>N_F_B )
{
 fprintf(fr,"\n\n  fa>N_F_B. fa=%ld   N_F_B=%ld ",fa,N_F_B);
 exit(0);
}
k=nn[0];
for (i=1; i<=fa+1; i++)
{
 p=mp[i];
       k=nn[0]; m2=k; m=0; q=0; a=(double)(mp[i]); kfb[i]=0;
       lg_fa[i]=log(a);
 for (j=0; j<=k; j++) rc[j]=nn[j];

       c=m2; // **1**
       {
        m2=c;
  m1=0;
  for (j=m2; j>=1; j--)
  {
   m=m1*b0+rc[j];
         m1=m%p;
        }
        mpn[i]=m1; // остачі від ділення N на прості p
        c=m2;
        for (j=m2; j>=1; j--)
        {
         if ( rc[j]==0 ) c=j;
         if ( rc[j]>0  ) break;
        }
  if ( c<2 ) break;
       }
```

```
} //


// Символи Лежандра (n/p)  масив mln[]
mln[0]=1; mln[1]=1;

for (i=2; i<=fa+1; i++)
{
      jj=mp[i];
 k=1; m3=1;
      m=mpn[i]; m2=jj; m1=m%jj;
      if ( m1>1 )
      {
       while(1)
       {
        if ( m1==1 ) break;
  if ( (m1%4)==0 ) m1=m1/4;
        else
        {
         c=m1%4;
         if ( c==2 )
         {
          if ( ((m2-1)%8)==0 || ((m2+1)%8)==0  ) m3=1; else m3=-1; k=k*m3; m1=m1/2;
         }
         else
         {
          if ( m1>m2 ) m1=m1%m2;
    else
               {
         if ( ((m1-1)%4)==0 || ((m2-1)%4)==0 ) m3=1; else m3=-1;
         m=m2;
              m2=m1;
              m1=m%m2;
         k=k*m3;
              }
        }
       }
      }
      mln[i]=k;
}


for(m=0; m<nb4; m++) nk[m]=0;
```

```
   for(m=nn[0]+1; m<nb4; m++) nn[m]=0;
   for (i=1; i<=fa; i++) mpnk[i]=0;

   kn=0;
met_kn: kn++;
   {

    j=0; k=nn[0]+1; if ( k<=nk[0] ) k=nk[0]+1;
    for (i=1; i<=fa; i++)
          {
           p=mp[i]; mpnk[i] = mpnk[i] + mpn[i];
           if ( mpnk[i]>=p ) mpnk[i]=mpnk[i]-p;
          } // визначено остачі від ділення kN на p
    for (i=1; i<=k; i++)
    {
     jj=nn[i]+nk[i]+j;
          if ( jj<b0 ) { j=0; nk[i]=jj; } else { j=1; nk[i]=jj-b0; }
     j=jj/b0; nk[i]=jj%b0;
    }
    if ( nk[k]==0 ) { k--; if ( nk[k]==0 ) { k--;  if ( nk[k]==0 ) { k--;  if ( nk[k]==0 ) k--; } }
}
    nk[0]=k;

    // Аналіз допустимих kn
    kb=0; f=0; f_kn=-1;
    for (i=1; i<=fa; i++)
    {
       j=mp[i];
       m=j*j;
       if ( m > kn ) break;
       if ( kn%m == 0 ) { f++; kb=i; }
       if ( f > 1 ) break;
    }
    if ( f>1 ) goto met_kn;
    if ( f==1 )
    f_kn=kb;


    // Символи Лежандра (k/p) і масив mlkn[]
    mlkn[0]=1; mlkn[1]=1;
    if ( kn<2 )
    {
       for (f=0; f<=fa+1; f++)
          mlkn[f]=mln[f];
```

```
    }
    else
    {

       for (f=2; f<=fa; f++)
       {
          jj=mp[f];
          k=1;
          m=kn; m2=jj; m1=m%jj;
          if ( m1<2 || f==f_kn )
          {
             if ( m1==0 ) mlkn[f]=1; else mlkn[f]=mln[f];
             if ( f==f_kn ) mlkn[f]=-1;
          }
          else
          {
             while(1)
             {
                if ( m1<2 ) break;
                if ( (m1%4)==0 ) m1=m1/4;
                else
                {
                   c=m1%4;
                   if ( c==2 )
                   {
                      if ( ((m2-1)%8)==0 || ((m2+1)%8)==0  ) m3=1; else m3=-1; k=k*m3;
m1=m1/2;
                   }
                   else
                   {
                      if ( m1>m2 ) m1=m1%m2;
                      else
                      {
                         if ( ((m1-1)%4)==0 || ((m2-1)%4)==0 ) m3=1; else m3=-1;
                         m=m2;
                         m2=m1;
                         m1=m%m2;
                         k=k*m3;
                      }
                   }
                }
                mlkn[f]=mln[f]*k;
             }
          }
```

```
   }
}// Лежандр ( закрито дужку { - номер 5 )
f=f_kn;
for (i=1; i<=fa; i++)
{
   if ( i==f_kn ) continue;
   j=mp[i];
   if ( j > kn ) break;
   if ( kn%j == 0 ) mlkn[i]=1;
}
if ( f_kn>0 ) mlkn[f_kn]=-1;
a1=0.0;  for (f=0; f<=fa; f++) { if ( mlkn[f]>0 ) a1=a1+1.0; }
a2 = 2.0*a1/fa;

if ( a2<0.75 ) goto met_kn;    afb = fb*a2*a2*a2*a2*a2*a2;
a=(double)(kn);
logkr = 0.01 + 0.5*(logn * log(10.0) + log(a)) * klkr;
k=0; cff=1;
for (f=1; f<=fa; f++)
{
   mx1[f]=-1;
   if ( mlkn[f]>0 )
   {
      k++; mfb[k]=f;
      if ( f<=ff ) cff++;
   }
}//
mfb[0]=k;
//cff=k+1;
for (f1=1; f1<=mfb[0]; f1++)
{
   f=mfb[f1];
   p=mp[f];
   ac=mpnk[f];
   a=0.1+ac;
   a1=sqrt(a);
   k=(long)(a1);
   f00=p%8;
   if ( ac==(k*k) ) mx1[f]=k;
   else
   { //6
      mx1[f]=-1;
      b=0; r=0; ac1=0;
      if ( f00==3 || f00==7 )
```

```
        { //7
           k = p/4 + 1;
           i1=0; k1=k;
           while(1)
           {
              k=k1; i1++; rcl[i1] = k%2; k1=k/2;
              if ( k1<1 ) break;
           }
           k2=ac;
           for(i=i1-1; i>0; i--)
           {
              k2=(k2*k2)%p;
              if ( rcl[i]==1 ) k2=(k2*ac)%p;
           } // обчислено значення x1 при ( c1==3 || c1==7 )
           k=(k2*k2)%p;
           if ( k==ac )
           {
              if ( k2>(p/2) ) mx1[f]=p-k2; else mx1[f]=k2;
           } // обчислено значення x1 при ( c1==5 ) де корінь визначено за першою
формулою
           else
           {
              fprintf(fr,"\n Dlja p=%6ld ta kvadratnoho lyshku a=%6ld   k2=%6ld
c2=%6ld",p,ac,k2,c2);
              fprintf(fr,"\n ne vdalosja znaity X1. ");
              exit(0);
           }
        }

        if ( f00==5 )
        { //7
           k = p/8;
           i1=0; k1=k;
           while(1)
           {
              k=k1; i1++; rcl[i1] = k%2; k1=k/2;
              if ( k1<1 ) break;
           }
              k2=ac; k3=ac; //
              for(i=i1-1; i>0; i--)
              {
                 k2=(k2*k2)%p;
                 if ( rcl[i]==1 ) k2=(k2*ac)%p;
              }
```

```
            k2=(k2*ac)%p;  k=(k2*k2)%p;
            if ( k==ac )
            {
               if ( k2>(p/2) ) mx1[f]=p-k2; else mx1[f]=k2;
            }
            else
            {
               c3=2;
               c2=4;
               for(i=i1-1; i>0; i--)
               {
                  c2=(c2*c2)%p;
                  if ( rcl[i]==1 ) c2=(c2*4)%p;
               }
               c2=(c2*2*k2)%p;
               k=(c2*c2)%p;
               if ( k==ac )
               {
                  if ( c2>(p/2) ) mx1[f]=p-c2; else mx1[f]=c2;
               } // обчислено значення x1 при ( c1==5 ) де корінь визначено за
першою формулою
               else
               {
                  fprintf(fr,"\n Для p=%6ld та квадратного лишку a=%6ld   k2=%6ld
c2=%6ld",p,ac,k2,c2);
                  fprintf(fr,"\n не вдалося знайти X1. ",p,ac);
                  exit(0);
               }
            }
         }
         if ( f00==1 )
         { //7
            k = p-1; t=0;
            i1=0; s=k; // rcl[]; Записуємо остачі від ділення K на степені двійки
            while(1)
            {
               k=s; i1++; k2 = k%2;
               if ( k2==0 ) t++;
               if ( k2>0 ) break;
               s=k/2;
            }
            for (i=0; i<=40; i++  ) { mm0[i]=0; mm1[i]=0; }
            i=1; mm0[0]=p; mm0[1]=ac;
            while(1)
```

```
            {
               i++;
               k1 = mm0[i-2] / mm0[i-1]; mm1[i]=k1; mm0[i]=mm0[i-2] % mm0[i-1];
               if ( mm0[i] <2 ) break;
            }
            t1=-mm1[i]; s1=1;
            for (j=i-1; j>1; j-- )
            {
               k=t1; t1=s1-k*mm1[j]; s1=k;
            }
            ac1=t1; if ( t1<0 ) ac1=p+t1;
            if ( ((ac*ac1)%p)!=1 )
            {
               fprintf(fr,"\n Для p=%6ld та квадратного лишку a=%6ld d",p,ac);
               fprintf(fr,"\n Обернене до ac = ac1=%6ld знайдено некоректно",ac1);
               fprintf(fr,"\n (ac*ac1)%p)!=1");
               exit(0);
            }
            for (j2=2; j2<100; j2++ )
            { //8
               m=mp[j2]; // далі обчислення символа Лежандра (m/p)
               k=1; jj=p;
               m2=jj; m1=m%jj;
               while(1)
               { //9
                  if ( m1<2 ) break;
                  if ( (m1%4)==0 ) m1=m1/4;
                  else
                  { //10
                     c=m1%4;
                     if ( c==2 )
                     {
                        if ( ((m2-1)%8)==0 || ((m2+1)%8)==0  ) m3=1; else m3=-1;
k=k*m3; m1=m1/2;
                     }
                     else
                     { //11
                        if ( m1>m2 ) m1=m1%m2;
                        else
                        { //12
                           if ( ((m1-1)%4)==0 || ((m2-1)%4)==0 ) m3=1; else m3=-1;
                           jj=m2;
                           m2=m1;
                           m1=jj%m2;
```

```
                k=k*m3;
              } //12
            } //11
          } //10
      } //9
      if ( k<0 ) break;
} //8

zz = m;
k = s;
i1=0; k1=k; // rcl[]; Записуємо остачі від ділення К на степені двійки
while(1)
{
   k=k1; i1++; rcl[i1] = k%2; k1=k/2;
   if ( k1<1 ) break;
}
k2=m;
for(i=i1-1; i>0; i--)
{
   k2=(k2*k2)%p;
   if ( rcl[i]==1 ) k2=(k2*m)%p;
}
b=k2;  // обчислено значення z^s mod(p) = b

k = (s+1)/2;
i1=0; k1=k; // rcl[]; Записуємо остачі від ділення К на степені двійки
while(1)
{
   k=k1; i1++; rcl[i1] = k%2; k1=k/2;
   if ( k1<1 ) break;
}
k2=ac;
for(i=i1-1; i>0; i--)
{
   k2=(k2*k2)%p;
   if ( rcl[i]==1 ) k2=(k2*ac)%p;
}
r=k2;  // обчислено значення (ac^((s+1)/2))mod(p) = r

c2 = r; mnm[0]=b;
for(i=1; i<t; i++)
{
   c1=mnm[i-1];
   c2=(c1*c1)%p;
```

```
            mnm[i]=c2;
         }
         c2 = (r*r)%p;
         c1 = (c2*ac1)%p;
         c3=c1; // обчислено (r*r*a^(-1)). Далі підносимо його до степеніf(k,0) =
2^(t-2).
         for(i=1; i<t-1; i++)
         {
            c2=(c1*c1)%p;
            c1=c2;
         }
         if ( c2==1 ) f0=0;
         else
         {
            if ( c2==(p-1) )
               f0=1;
            else
            {
               fprintf(fr,"\n Обчислення j0: 1<c2<(p-1). p=%6ld   ac=%6ld   r=%6ld
a^(-1)=ac1=%6ld   t=%6ld   c3=(r*r*a^(-1))=%6ld   c2=(c3^f(0))=%6ld  ",
                  p,ac,r,ac1,t,c3,c2);
               exit(0);
            }
         }
         rcl[0]=f0;
         bb=1; // bb = b ^ (2^(t-1)) = 1
         for(m0=t-3; m0>=0; m0--)
         {
            k=(t-2)-m0; // Обчислення j(k). f => f(k)=2^f.
            if ( rcl[(k-1)]==1 ) bb = (bb * mnm[(k-1)])%p;
            c1 = (bb*r)%p;
            c2=(c1*c1)%p;
            c3=(c2*ac1)%p;
            c1=c3;
            c2=c3;
            for(i=1; i<=m0; i++)
            {
               c2=(c1*c1)%p;
               c1=c2;
            }
            if ( c2==1 ) f0=0;
            else
            {
               if ( c2==(p-1) ) f0=1;
```

```
                    else
                    {
                       fprintf(fr,"\n Обчислення j(k): 1<c2<(p-1). p=%6ld   a=%6ld
r=%6ld   a^(-1)=%6ld   c3=%6ld   c2=(c3^f(k))=%6ld  ",p,ac,r,ac1,c3,c2);
                       exit(0);
                    }
                 }
                 rcl[k]=f0;
              }
              if ( f0==1 ) bb = (bb * mnm[k])%p;

              x1=(bb*r)%p;
              if ( x1>p/2 )  mx1[f]=p-x1; else mx1[f]=x1;
           } //7
        } //6

        c2=mx1[f];
        k=(c2*c2)%p;
        if ( k>ac || k<ac )
        {
           fprintf(fr,"\n Для p=%6ld та квадратного лишку ac=%6ld
(mx1[f]*mx1[f])modp!=ac  mx1[f]=%6ld  ",p,ac,c2);
           fprintf(fr,"\n не вдалося знайти X1. ",p,ac);
           exit(0);
        }
     } //5;

     a=0.0;
     for(m=0; m<nb4; m++) rc[m]=0;
     for (i=0; i<=nk[0]; i++) rc[i]=nk[i];

     f0=rc[0];     // число блоків у числі
     m0=(f0+1)/2;  // число блоків у корені з числа
     m9=f0%2;      // Ознака парності числа блоків у масиві RC[]
     f00 = m0-2;   // Число блоків, які потрібно обробити
     for(m=0; m<nb2; m++) { kr[m]=0; rr[m]=0; r0[m]=0;  ry[m]=0; yy[m]=0; }
     kr[0]=m0;
     if ( f0<5 )
     {
        a=0.0;
        for (i=f0; i>0; i--) a=a*b00+rc[i];
        a=a+0.01;
        a2=sqrt(a);
        s=(long)(a2)+1;
```

```
   kr[m0]=s/b0;
   kr[m0-1]=s%b0;
   rr[m0]=(s-1)/b0;
   rr[m0-1]=(s-1)%b0;
   a1=(double)(s-1);
   a3=a1*a1-a-0.21;
   i1=(long)(-a3);
   yy[0]=1; yy[1]=i1%b0; i2=i1/b0;
   if ( i2>0  )  { yy[2]=i2; yy[0]=2; }
   a1=a1+1.0;
   a3=a1*a1-a+0.21;
   i1=(long)(a3);
   ry[0]=1; ry[1]=i1%b0; i2=i1/b0;
   if ( i2>0  )  { ry[2]=i2; ry[0]=2; }
   r0[0]=1; if ( rr[2]>0  ) rr[0]=2;
}

if ( f0>4 )
{
   if ( m9==1 ) a = 0.999+(rc[f0-2]+b00*(rc[f0-1]+b00*rc[f0]));
   else a = 0.999+(rc[f0-3]+b00*(rc[f0-2]+b00*(rc[f0-1]+b00*rc[f0])));
   a2=sqrt(a);
   s=(long)(a2);
   a0=a2*2.0; // число на яке ділимо (максимальне)
   a00=2.0*s; // число на яке ділимо (мінімальне)
   kr[m0]=s/b0;
   kr[m0-1]=s%b0;

   k5=kn;
   k5=0;//
   s1=(long)(a-s*s); // Остачі після віднімання квадрата
   f=2*m0-3;
   rc[f]=s1%b0; i=s1/b0;
   f++; s1=i;           // 2*m0-3
   rc[f]=s1%b0; i=s1/b0;
   f++; s1=i;           // 2*m0-2
   rc[f]=s1%b0;
   i=s1/b0; f++;        // 2*m0-1
   rc[f]=i;             // 2*m0
   if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) f--; } } }
   rc[0]=f;
   for(m=f00; m>0; m--)
   {
      m3=f00+m; // номер блоку що додається
```

```
      m1=m3+4; k5++; //число А формується в границях від M1 до M3
      a = 0.0;
      for(f=rc[0]+1; f>=m3; f--) a=a*b00+rc[f];
      i1=(long)(a/a0);
      kr[m]=i1;
      r0[m]=i1*i1;
      j4=2*i1;
      for(f=m+1; f<=kr[0]+1; f++)
      {
        r0[f]=j4*kr[f];
      }
      f2=0; jj=0; // 2kr*b0 + kr[m]^2 у вигляді блоків
      for(f=m; f<kr[0]+3; f++)
      {
        m1=r0[f]+jj;  j2=m1%b0;
        r0[f]=j2; jj=m1/b0;
      }

      f2=2*m-2; jj=0; // Нове  RC[]
      for(f=m; f<kr[0]+3; f++)
      {
        f2++;
        m1=rc[f2]-r0[f]-jj;
        if ( m1<0 ) { m1=m1+b0; jj=1; } else jj=0;
        rc[f2]=m1; r0[f]=0;
      }
      if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) { f--; if ( rc[f]==0 ) f--; } }
}
      rc[0]=f;
      if ( i1>=b0 )
      {
        jj=1; kr[m]=i1-b0;
        for(j=m+1; j<=kr[0]; j++)
        {
          m3=kr[j]+jj;
          if ( m3>=b0 ) { jj=1; kr[j]=m3-b0; } else { jj=0; kr[j]=m3; }
          if ( jj==0 ) break;
        }
      }
    }
    jj=1; c=kr[0]+1; kr[c+1]=0; r0[0]=0; yy[0]=rc[0]; if ( rc[0]>c ) c = rc[0];
    for(f=1; f<=c+1; f++)
    {
      yy[f]=rc[f];
```

```
   m3=kr[f]+kr[f]+jj;
   if ( m3>=b0 ) { jj=1; r0[f]=m3-b0; } else { jj=0; r0[f]=m3; }
   rr[f]=kr[f];
}
if ( r0[c+1]==0 ) r0[0]=c; else r0[0]=c+1; // r0[](0)

jj=1;
for(j=1; j<=c+1; j++)
{
   rr[j]=kr[j];
   if ( jj>0 )
   {
      m3=kr[j]+jj;
      if ( m3>=b0 ) { jj=1; kr[j]=m3-b0; } else { jj=0; kr[j]=m3; }
      }
   } // До кореня добавили 1: KR[](+1)

   f3=0; c1=rc[0]; if ( c1<r0[0] ) c1=r0[0];
   c1=c1+2;
   for(f=c1+1; f>0; f--)
   {
      if ( f3>0 ) break;
      if ( r0[f]==rc[f] && f3==0 ) continue;
      if ( r0[f]>rc[f] ) f3=2;
      if ( r0[f]<rc[f] ) f3=4;
   }

   if ( f3>3 )
   { // обчислення при збільшеному X на 1 (RR і KR)
      jj=0;
      for(j=1; j<=rc[0]+1; j++)
      {
         m3=rc[j]-r0[j]-jj;
         if ( m3<0 ) { jj=1; rc[j]=m3+b0; } else { jj=0; rc[j]=m3; }
      } // змінили RC[] і  YY[]
      jj=2;
      for(j=1; j<=c+1; j++)
      {
         m3=r0[j]+jj;
         if ( m3>=b0 ) { jj=1; r0[j]=m3-b0; } else { jj=0; r0[j]=m3; }
         if ( jj==0 ) break;
      } // Знайшли R0[] для KR+1

      f4=0; c1=rc[0]; if ( c1<r0[0] ) c1=r0[0];
```

```
c1=c1+2;
for(f=c1+1; f>0; f--)
{
   if ( f4>0 ) break;
   if ( r0[f]==rc[f] && f4==0 ) continue;
   if ( r0[f]>rc[f] ) f4=2;
   if ( r0[f]<rc[f] ) f4=4;
}

jj=1;
for(j=1; j<=c+1; j++)
{
   rr[j]=kr[j];
   if ( jj>0 )
   {
      m3=kr[j]+jj;
      if ( m3>=b0 ) { jj=1; kr[j]=m3-b0; } else { jj=0; kr[j]=m3; }
   }
} // До кореня добавили 2

if ( f4>3 )
{  //     kr^2 < kN
   jj=0;
   for(j=1; j<=rc[0]; j++)
   {
      m3=rc[j]-r0[j]-jj;
      if ( m3<0 ) { jj=1; rc[j]=m3+b0; } else { jj=0; rc[j]=m3; }
   } // змінили RC[] і  YY[]
   jj=2;
   for(j=1; j<=c+1; j++)
   {
      m3=r0[j]+jj;
      if ( m3>=b0 ) { jj=1; r0[j]=m3-b0; } else { jj=0; r0[j]=m3; }
      if ( jj==0 ) break;
   } // Знайшли R0[] для KR+2

   f5=0; c1=rc[0]; if ( c1<r0[0] ) c1=r0[0];
   c1=c1+2;
   for(f=c1+1; f>0; f--)
   {
      if ( f3>0 ) break;
      if ( r0[f]==rc[f] && f3==0 ) continue;
      if ( r0[f]>rc[f] ) f3=2;
      if ( r0[f]<rc[f] ) f3=4;
```

```
                }
                if ( f5>3 )
                {
                   fprintf(fr,"\n  f5>3. Помилка при обчисленні кореня. Процес рішення
зупинено ");
                   exit(0);
                }
             }
          }
          // обчислюється різниця r0-rc
          jj=0; rr[0]=kr[0]; yy[0]=kr[0]+1; ry[0]=yy[0];
          for(f=1; f<r0[0]+2; f++)
          {
             m1=r0[f]-rc[f]-jj; yy[f]=rc[f];
             if ( m1<0 ) { m1=m1+b0; jj=1; }
             else
             {
                if ( m1>=b0 ) { m1=m1-b0; jj=-1; }
                else jj=0;
             }
             ry[f]=m1;
          }
          jj=0; j=0;
          for(f=r0[0]+3; f>0; f--)
          {
             if ( ry[f]>0 && jj==0 ) jj=f;
             if ( yy[f]>0 && j==0  ) j=f;
          }
          yy[0]=j+1; ry[0]=jj+1;
       }

       if ( f_kn>0 )
       {
          m=0; p=mp[f_kn];
          for(j=kr[0]; j>0; j--)
          {
             m=m*b0+kr[j];
             m1=m%p;
             m=m1;
          }
          rf_kn=m; // остача від ділення X на p=mp[f_kn].
       }

       for (i=1; i<cff; i++)
```

```
        {
           f=mfb[i];
           if ( f<32 ) p=mpr[f]; else p=mp[f];
           k=kr[0]; m2=k; m=0; q=0;
           for (jj=0; jj<10; jj++) { mpkr[i][jj]=0; mprr[i][jj]=0; mpry[i][jj]=0;
mpyy[i][jj]=0; }
           for (j=0; j<=k; j++) rc[j]=kr[j];
           c=m2; // **1**
           for (jj=1; jj<10; jj++)
           {
              m2=c;
              m=0;
              for (j=m2; j>=1; j--)
              {
                 m=m*b0+rc[j];
                 m1=m%p; rc[j]=m/p;
                 m=m1;
              }
              mpkr[i][jj]=m;
              c=m2;
              for (j=m2; j>=1; j--)
              {
                 if ( rc[j]==0 ) c=j;
                 if ( rc[j]>0  ) break;
              }
              if ( c<2 ) break;
           }
           mpkr[i][0]=jj+1; mpkr[i][jj+1]=0;
           k=rr[0]; m2=k; m=0; q=0;
           for (j=0; j<=k; j++) rc[j]=rr[j];
           c=m2; // **1**
           for (jj=1; jj<10; jj++)
           {
              m2=c;
              m=0;
              for (j=m2; j>=1; j--)
              {
                 m=m*b0+rc[j];
                 m1=m%p; rc[j]=m/p;
                 m=m1;
              }
              mprr[i][jj]=m;
              c=m2;
              for (j=m2; j>=1; j--)
```

```
    {
      if ( rc[j]==0 ) c=j;
      if ( rc[j]>0  ) break;
    }
    if ( c<2 ) break;
}
mprr[i][0]=jj+1;

k=yy[0]+2; if ( k>9 ) k=9; m2=k; m=0; q=0;
for (j=0; j<=k; j++) rc[j]=yy[j];
c=m2; // **1**
mpyy[i][0]=1; for (jj=1; jj<10; jj++) mpyy[i][jj]=0;
for (jj=1; jj<10; jj++)
{
    m2=c;
    m=0;
    for (j=m2; j>=1; j--)
    {
      m=m*b0+rc[j];
      m1=m%p; rc[j]=m/p;
      m=m1;
    }
    mpyy[i][jj]=m;
    c=m2;
    for (j=m2; j>=1; j--)
    {
      if ( rc[j]==0 ) c=j;
      if ( rc[j]>0  ) break;
    }
    if ( c<2 ) break;
}
mpyy[i][0]=jj+1;

k=ry[0]; m2=k; m=0; q=0;
for (j=0; j<=k; j++) rc[j]=ry[j];
c=m2; // **1**
mpry[i][0]=1; for (jj=1; jj<10; jj++) mpry[i][jj]=0;
for (jj=1; jj<10; jj++)
{
    m2=c;
    m=0;
    for (j=m2; j>=1; j--)
    {
      m=m*b0+rc[j];
```

```
            m1=m%p; rc[j]=m/p;
            m=m1;
        }
        mpry[i][jj]=m;
        c=m2;
        for (j=m2; j>=1; j--)
        {
            if ( rc[j]==0 ) c=j;
            if ( rc[j]>0  ) break;
        }
        if ( c<2 ) break;
    }
    mpry[f][0]=jj+1;
}


for (i=1; i<=mfb[0]; i++)
{
    m2=kr[0];
    f=mfb[i];
    p=mp[f];
    if ( i<cff ) { mxkr[i]=(mpkr[i][1])%p; mxrr[i]=(mprr[i][1])%p;  }
    else
    {
        k=kr[0]; m2=k; m=0; q=0;
        mxkr[i]=0;
        for (j=0; j<=k; j++) rc[j]=kr[j];
        m=0;
        for (j=m2; j>=1; j--)
        {
            m=m*b0+rc[j];
            m1=m%p; rc[j]=m/p;
            m=m1;
        }
        mxkr[i]=m;
        for (j=0; j<=k; j++) rc[j]=rr[j];
        m=0;
        for (j=rr[0]; j>=1; j--)
        {
            m=m*b0+rc[j];
            m1=m%p; rc[j]=m/p;
            m=m1;
        }
        mxrr[i]=m;
```

```
        }
    }

    for (f=kr[0]+1; f<nb2; f++) kr[f]=0;
    for (f=rr[0]+1; f<nb2; f++) rr[f]=0;
    for (f=ry[0]+1; f<nb2; f++) ry[f]=0;
    for (f=yy[0]+1; f<nb2; f++) yy[f]=0;
    a=0;
    for (f=ry[0]; f>0; f--)
    {
      if ( (ry[0]-f)<5 ) a = a*b00+ry[f];
      else a = a*b00;
    }

    a1=0;
    for (f=kr[0]; f>0; f--)
    {
      if ( (kr[0]-f)<5 ) a1 = a1*b00+kr[f];
      else a1 = a1*b00;
    }
    ry_kr=0.5*a/a1; //   RY[]/KR[]/2;
    lg_kr=log(a1)+log(2.0); //if ( kr[0]>4 ) lg_kr=lg_kr+(kr[0]-4.0)*lg_b0;

    a=0;
    for (f=yy[0]; f>0; f--)
    {
      if ( (yy[0]-f)<5 ) a = a*b00+yy[f];
      else a = a*b00;
    }
    yy_kr=0.5*a/a1; //   YY[]/KR[]/2;

    mc1=0.0;

met_bc: k=0;
    mc0=mc1;
    if ( (afb-mc0)>999.5 ) bc=1000; else bc=(int)(afb-mc0+0.1);
    mc1=mc1+bc; if ( bc<1 ) goto met_kn;

    for (f=0; f<bc; f++) { shp[f]=0; shm[f]=0;  for (c=0; c<20; c++)  { sp[f][c]=0;
sm[f][c]=0; } }
    for (c=0; c<bc; c++) mlg[c]=0.0;
    if ( f_kn>0 )
    {
      p=mp[f_kn]; if ( rf_kn>0 ) c1=p-rf_kn; else c1=0;
```

```
   for (c=c1; c<bc; c=c+p) mlg[c]=-10000.0;
}
for (f1=1; f1<=mfb[0]; f1++)
{ // побудови для (c+)
   f=mfb[f1]; // Номер простого з факторної бази
   p=mp[f];  // просте з факторної бази

   x1=mx1[f]; x2=p-x1;
   if ( x1==0 ) x2=0;
   m=mxkr[f1]%p; // остача від ділення kr[]%p;

   if ( m<=x1 ) { m1=x1-m; m2=x2-m; } // ( m - x1 - x2 )
   else
   {
      if ( x1<m && m<=x2 ) { m1=x2-m; m2=p-m+x1; } // ( x1 - m - x2 )
      else
      {
         if ( x2<m ) { m1=p-m+x1; m2=p-m+x2; } // ( x1 - x2 - m )
      }
   }

   s=-1;
   while(1)
   {
      if ( m1<bc )
      {
         t=sp[m1][0]+1;
         sp[m1][t]=f1;
         mlg[m1]=mlg[m1]+lg_fa[f];
         sp[m1][0]=t;
         m1=m1+p;
      }
      else s=m1-bc;
      if ( s>=0 ) break;
   }
      if ( x1>0 && p>2 )
      { s=-1;
         while(1)
         {
            if ( m2<bc )
            {
               t=sp[m2][0]+1;
               sp[m2][t]=f1;
               mlg[m2]=mlg[m2]+lg_fa[f];
```

```
            sp[m2][0]=t;
            m2=m2+p;
          }
          else s=m2-bc;
          if ( s>=0 ) break;
        }
    }
}
shp[0]=0; t=0;

for (c=1; c<bc; c++)
{
    if ( mlg[c]>logkr )
    {
        t++; shp[t]=c;
    }
}
t++; shp[t]=1001;

for (c=0; c<bc; c++)  mlg[c]=0.0;
if ( f_kn>0 )
{
    p=mp[f_kn]; j=1-rf_kn;
    if ( j<0 ) c1=p+j; else c1=j;
    for (c=c1; c<bc; c=c+p) mlg[c]=-10000.0;
}

for (f1=1; f1<=mfb[0]; f1++)
{ // побудови для (c-)
    f=mfb[f1]; // Номер простого з факторної бази
    p=mp[f];  // просте з факторної бази
    m=mxrr[f1]%p; // остача від ділення kr[]%p;
    x1=mx1[f]; x2=p-x1;
    if ( x1==0 ) x2=0;

    if ( m<=x1 ) { m1=m+p-x2; m2=m+p-x1; } // ( x2 > x1 > m )
    else
    {
        if ( x1<m && m<=x2 ) { m1=m-x1; m2=m+p-x2; } // ( x2 > m > x1 )
        else
        {
            if ( x2<m ) { m1=m-x2; m2=m-x1; } // ( m > x2 > x1 )
        }
    }
```

```
   s=-1;
   while(1)
   {
      if ( m1<bc )
      {
         t=sm[m1][0]+1;
         sm[m1][t]=f1;
         mlg[m1]=mlg[m1]+lg_fa[f];
         sm[m1][0]=t;
         m1=m1+p;
      }
      else s=m1-bc;
      if ( s>=0 ) break;
   }
   if ( x1>0 && p>2 )
   {
      s=-1;
      while(1)
      {
         if ( m2<bc )
         {
            t=sm[m2][0]+1;
            sm[m2][t]=f1;
            mlg[m2]=mlg[m2]+lg_fa[f];
            sm[m2][0]=t;
            m2=m2+p;
         }
         else s=m2-bc;
         if ( s>=0 ) break;
      }
   }
}
shm[0]=0; t=0;
for (c=1; c<bc; c++)
{

   if ( mlg[c]>logkr )
   {
      t++; shm[t]=c;
   }
}
t++; shm[t]=1002;
```

```
t1=0;
t2=0;

while(1)
{
   t=-1;
   c1=shp[t1];
   c2=shm[t2];
   if ( c1<=c2 )
   { // обробка C(+)
      if ( c1<bc )
      {
         c0=sp[c1][0];
         k2=0;
         t=1; t1++;
         lg_ry = lg_kr + log(ry_kr+mc0+c1);
         a=lg_ry;
         for (c=1; c<=c0; c++)
         {
            f1=sp[c1][c];
            f=mfb[f1];
            if ( f<32 ) bb=mpr[f]; else bb=mp[f];
            b=mp[f];
            a1 = lg_fa[f];
            if ( f>ff ) k1=1;
            else
            {
               k1=0; // показник степеня b для ry[]
               if ( f>31 ) k0=1; else k0=mprc[f]; // показник степеня b для bb
               m1=mpry[f1][1]; // (X0*X0 - N)(mod bb)
               m2 = c1*c1 + 2*mpkr[f1][1]*c1 + m1;
               m3=m2%b;
               k1=0;
               for (s=1; s<10; s++)
               {
                  if ( (m2%bb) == 0 )
                  {
                     k1=k1+k0;
                     m3 = m2/bb ;
                     m2 = m3 +2*mpkr[f1][s+1]*c1 + mpry[f1][s+1];
                  }
                  else break;
               }
               if ( f<32 )
```

```
                    {
                        m4=m2%bb;
                        while(1)
                        {
                            if ( (m4%b)==0 ) { k1++; m4=m4/b; }
                            else break;
                        }
                    }
                }
                mb[f]=k1;
                lg1=lg1+a1;
                a=a-a1*k1;
            }

            if ( a<0.1 )
            {

                if ( a<-0.1 )
                {
                    fprintf(fr,"\n a<-0.1.  c(+)=%g.  kN = %4ld.  a=%g
",(mc0+c1),kn,a);
                    fprintf(fr,"\n Некоректна робота алгоритму визначення B-
гладких. ");
                }
                else ch_bh++;
                fprintf(fr,"\n  B_gl %4ld.    kN=%6ld  x(+)=%7.0f.  M:
",ch_bh,kn,(mc0+c1));
                if ( a>-0.1 ) fprintf(frp,"\n B_gl %4ld.    kN=%6ld  x(+)=%7.0f.  M:
",ch_bh,kn,(mc0+c1));
                j=kr[0]; f=0; jj=c1*c1;
                while(1)
                {
                    f++;
                    m1 = ry[f] + 2*kr[f]*c1 + jj;
                    if ( m1>b0 ) { jj=m1/b0; ml2[f]=m1%b0; } else { jj=0; ml2[f]=m1;
}
                    if ( f>=kr[0] && jj==0 ) break;
                }
                ml2[0]=f;
                for (j=ml2[0]; j>0; j--)
                {
                    fprintf(fr," %3d",ml2[j]);
                    if ( a>-0.1 ) fprintf(frp," %3d",ml2[j]);
                }
```

```
            a7=0.0; a8=0.0; f4=0;
            for (c=1; c<=c0; c++)
            {
               f1=sp[c1][c];
               f=mfb[f1];
               k=mb[f];
               if ( k%2==1 ) kfb[f]=kfb[f]+1;
               //if ( k>1 && f4==0 ) f4=f;
               if ( a>-0.1 ) fprintf(frp," %2ld(%3ld) ",mb[f],mp[f]);
               fprintf(fr," %2ld(%3ld) ",mb[f],mp[f]);
            }
         }
      }
   }
   else
   {
      if ( c2<bc )
      {
         c0=sm[c2][0];
         t=1; t2++;
         k2=0;
         if ( kn==3 && c2==3 )
         t=1;
         lg_ry = lg_kr + log(yy_kr+mc0+c2);
         a=lg_ry;
         for (c=1; c<=c0; c++)
         {
            f1=sm[c2][c];
            f=mfb[f1];
            if ( f<32 ) bb=mpr[f]; else bb=mp[f];
            b=mp[f];
            a1 = lg_fa[f];
            if ( f>31 ) k0=1; else k0=mprc[f]; // показник степеня b для bb
            if ( f>ff ) k1=1;
            else
            {
               m1=mpyy[f1][1]; // (X0*X0 - N)(mod bb)
               m2 =  m1 + 2*mprr[f1][1]*c2 - c2*c2;
               m3=m2%b;
               k1=0;
               for (s=1; s<10; s++)
               {
                  if ( (m2%bb) == 0 )
                  {
```

```
                        k1=k1+k0;
                        m3 = m2/bb ;
                        m2 = m3 +2*mprr[f1][s+1]*c2 + mpyy[f1][s+1];
                     }
                     else break;
                  }
                  if ( f<32 )
                  {
                     m4=m2%bb;
                     while(1)
                     {
                        if ( (m4%b)==0 ) { k1++; m4=m4/b; }
                        else break;
                     }
                  }
               }
               mb[f]=k1;
               lg1=lg1+a1;
               a=a-a1*k1;
            }

            if ( a<0.1 )
            {
               if ( a<-0.1 )
               {
               fprintf(fr,"\n a<-0.1.  c(+)=%g.  kN = %4ld.  a=%g ",(mc0+c1),kn,a);
               fprintf(fr,"\n Некоректна робота алгоритму визначення B-гладких.
");
               }
               else ch_bh++;

               fprintf(fr,"\n  B_gl %4ld.    kN=%6ld  x(-)=%7.0f.  M:
",ch_bh,kn,(mc0+c2));
               if ( a>-0.1 ) fprintf(frp,"\n B_gl %4ld.    kN=%6ld  x(-)=%7.0f.  M:
",ch_bh,kn,(mc0+c2));
               j=rr[0]; f=0; jj=-c2*c2;
               while(1)
               {
                  f++;
                  m1 = yy[f] + 2*rr[f]*c2 + jj;
                  if ( m1>b0 ) { jj=m1/b0; ml2[f]=m1%b0; }
                  else
                  {
                     if ( m1>=0 ) { jj=0; ml2[f]=m1; }
```

```
                    else
                    {
                    m2=-m1;
                    jj=-1-(m2/b0);
                    ml2[f]=b0-m2%b0;
                    }
                 }
                 if ( f>=kr[0] && jj==0 ) break;
              }
              ml2[0]=f;

              for (j=f; j>0; j--) fprintf(fr," %3d",ml2[j]);
              if ( a>-0.1 ) for (j=f; j>0; j--) fprintf(frp," %3d",ml2[j]);

              a7=0.0; a8=0.0; f4=0;
              for (c=1; c<=c0; c++)
              {
                 f1=sm[c2][c];
                 f=mfb[f1];
                 k=mb[f];
                 if ( k%2==1 ) kfb[f]=kfb[f]+1;
                 //if ( k>1 && f4==0 ) f4=f;
                 if ( a>-0.1 ) fprintf(frp," %2ld(%3ld) ",mb[f],mp[f]);
                 fprintf(fr," %2ld(%3ld) ",mb[f],mp[f]);
              }

           }
        }
     }
     if ( t<0 ) break;
     if ( ch_bh>=3+fa ) goto met_fa;
  }

  if ( bc==1000 )
  {
     c=p0; m=np; j=nq; m1=kn;
     jj=1; c=0; m=kr[0]+3; if ( m < (ry[0]+2) ) m=ry[0]+2;
     for (j=2; j<m; j++)
     {
        m1=jj+kr[j];
        if ( m1>=b0 ) { kr[j]=m1-b0; jj=1; } else { jj=0; kr[j]=m1; }
        if ( jj==0 ) break;
     }
     jj=1;
```

```
            for (j=2; j<m; j++)
            {
               m1=-jj+rr[j];
               if ( m1<0 ) { rr[j]=b0+m1; jj=1; } else { jj=0; rr[j]=m1; }
               if ( jj==0 ) break;
            }
            jj=0;
            for (f=2; f<=m; f++)
            {
               m1 = ry[f] + 2*kr[f-1] + jj; if ( f==3 ) m1--;
               if ( m1>b0 ) { jj=m1/b0; ry[f]=m1%b0; } else { jj=0; ry[f]=m1; }
            }
            jj=0;
            for (f=2; f<=m; f++)
            {
               m1 = yy[f] + 2*rr[f-1] + jj; if ( f==3 ) m1++;
               if ( m1>b0 ) { jj=m1/b0; yy[f]=m1%b0; } else { if ( m1<0 ) {
yy[f]=m1+b0; jj=-1; } else { jj=0; yy[f]=m1; } }
            }
            f1=-1; f2=-1;        f3=-1;        f4=-1;
            for (f=m; f>0; f--)
            {
               if ( f1<0 && kr[f]>0 ) f1=f;
               if ( f2<0 && rr[f]>0 ) f2=f;
               if ( f3<0 && ry[f]>0 ) f3=f;
               if ( f4<0 && yy[f]>0 ) f4=f;
            }
            kr[0]=f1+1; rr[0]=f2+1; ry[0]=f3+1; yy[0]=f4+1;
            for (f1=1; f1<cff; f1++)
            { // побудови для (c-)
               f=mfb[f1]; // Номер простого з факторної бази
               p=mp[f];  // просте з факторної бази
               if ( f<32 ) bb=mpr[f]; else bb=p;
               j2=mpkr[f][0]+3; if ( j2>10 ) j2=10;
               jj=b0*b0;
               for (j=1; j<j2; j++)
               {
                  rcl[j]=mpry[f1][j] + 2*b0*mpkr[f1][j]+jj;
                  jj=0;
               }
               jj=0;
               for (j=1; j<j2; j++)
               {
                  m1 = rcl[j] + jj;
```

```
            if ( m1>=bb ) { jj=m1/bb; mpry[f1][j]=m1%bb; }
            else          { jj=0; mpry[f1][j]=m1; }
         }
      }


      for (f1=1; f1<cff; f1++)
      { // побудови для (c-)
         f=mfb[f1]; // Номер простого з факторної бази
         p=mp[f];  // просте з факторної бази
         jj=b0;
         if ( f<32 ) bb=mpr[f]; else bb=p;
         for (j=1; j<10; j++)
         {
            m1 = jj + mpkr[f1][j];
            if ( m1>bb ) { jj=m1/bb; mpkr[f1][j]=m1%bb; } else { jj=0;
mpkr[f1][j]=m1; }
            if ( jj==0 ) break;
         }
      }

      for (f1=1; f1<cff; f1++)
      { // побудови для (c-)
         f=mfb[f1]; // Номер простого з факторної бази
         p=mp[f];  // просте з факторної бази
         jj=-b0;
         if ( f<32 ) bb=mpr[f]; else bb=p;
         j2=mprr[f1][0]+1; if ( j2>10 ) j2=10;
         for (j=1; j<j2; j++)
         {
            m1 = mprr[f1][j] + jj;
            if ( m1>0 )
            {
               if ( m1>bb ) { jj=m1/bb; mprr[f1][j]=m1%bb; } else { jj=0;
mprr[f1][j]=m1; }
            }
            else
            {
               m2=-m1; m3=m2%bb; if ( m3==0 ) { jj=-m2/bb; mprr[f1][j]=0; } else
{ jj=-1-m2/bb; mprr[f1][j]=bb-m3; }
            }
            if ( jj==0 ) break;
         }
      }
```

```
            for (f1=1; f1<cff; f1++)
            { // побудови для (c-)
               f=mfb[f1]; // Номер простого з факторної бази
               p=mp[f];  // просте з факторної бази
               jj=b0*b0;
               if ( f<32 ) bb=mpr[f]; else bb=p;
               j2=mprr[f1][0]+3; if ( j2>10 ) j2=10;
               for (j=1; j<j2; j++)
               {
                  rcl[j]=mpyy[f1][j] + 2*b0*mprr[f1][j]+jj;
                  jj=0;
               }
               for (j=1; j<j2; j++)
               {
                  m1 = rcl[j] + jj;
                  if ( m1>bb ) { jj=m1/bb; mpyy[f1][j]=m1%bb; } else { jj=0;
mpyy[f1][j]=m1; }
               }
            }

            for (f1=1; f1<=mfb[0]; f1++)
            { // побудови для (c-)
               f=mfb[f1]; // Номер простого з факторної базиF
               p=mp[f];  // просте з факторної бази
               if ( f>ff )
               {
                  m1=mxkr[f1] + b0;
                  if ( m1>=p ) { if ( p<b0 ) mxkr[f1]=m1%p; else mxkr[f1]=m1-p; } else
mxkr[f1]=m1;
                  if ( p<b0 ) { m=b0%p; m2=mxrr[f1] - m; if ( m2<0 ) mxrr[f1]=m2+p;
else mxrr[f1]=m2; }
                  else { m2=mxrr[f1] - b0; if ( m2<0 ) mxrr[f1]=m2+p; else mxrr[f1]=m2;
}
               }
               else { mxkr[f1]=mpkr[f1][1]; mxrr[f1]=mprr[f1][1]; }
            }
         }
         if ( mc1<(afb-0.99999) ) goto met_bc;
         if ( ch_bh<3+fa && mc1<afb ) goto met_bc;

met_fa: k=0;

      // Закінчено роботу в інтервалі просіювання
        }
```

```
        if ( ch_bh<3+fa ) goto met_kn;

        fprintf(frp,"\n kN=%5ld.   Визначено ch_bh=%ld  B-гладких  \n",kn,ch_bh);
        c_bh=c_bh+ch_bh; // число B-гладких

        if ( np>=1 && nq>=1  )
        {
           fprintf(frp,"\n Частота появи простих з непарним показником в B-гладких.
Masyv KFB: \n ");
           for (f=1; f<=fa; f++)
           {
              fprintf(frp," %2ld(%3ld) ",kfb[f],mp[f]);
              if (  (f%10) == 0  ) fprintf(frp," \n ");
           }
           fprintf(frp,"\n \n");
        }
        k_r = clock();
        a=0.001*(k_r-t_r);
        fprintf(frp,"\n np=%3ld nq=%3ld log(N)=%7.4f La=%4ld Lb=%6ld   Time:%8.3f
",np,nq,logn,lla,fb,a);
        printf(" p0=%3ld  q0=%3ld  log(N)=%7.4f  Time: %8.3f ",p0,q0,logn,a);
        fprintf(flt,"\n np=%3ld nq=%3ld log(N)=%7.4f La=%4ld Lb=%6ld   Time:%8.3f
",np,nq,logn,lla,fb,a);
        t_r = k_r;

      }
   }
   a1=(double)(pa*qa); a2=c_bh/a1;
   fprintf(frp,"\n Середнє значення числа B-гладких %7.2f  ",a2);

   k_r = clock();
   a=0.001*(k_r-p_r);
   fprintf(frp,"\n p0=%5ld  q0=%5ld  log(N)=%7.4f  La=%4ld  fa=%4ld  Lb=%6ld
Time(sum):%8.3f ",p0,q0,logn,lla,fa,fb,a);
   printf("\n p0=%3ld  q0=%3ld  log(N)=%7.4f  Time(sum):  %8.3f \n",p0,q0,logn,a);
   fprintf(flt,"\n p0=%5ld  q0=%5ld  log(N)=%7.4f  La=%4ld  fa=%4ld  Lb=%6ld
Time(sum):%8.3f ",p0,q0,logn,lla,fa,fb,a);

   if ( p0<p0max ) goto met_p0;

   fcloseall();
}
```

ДОДАТОК С

Програмний додаток online_matrix.c

```
//#define BOOST_TEST_MODULE QS unit maytrix Test
//#include <boost/test/included/unit_test.hpp>
#include <iostream>
#include "dynamic_bin_matrix.h"
// #include "primes.h"
#include "quadratic_sieve_big.h"
#include "log.h"

#include <fstream>
#include "greatest_common_divisor_big.h"

#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include "primes_10_8.h"
//#include "primes.h"
#include "big_2.h"

#include <math.h>
#include <time.h>

int showDebugMsg = 4;

//BOOST_AUTO_TEST_CASE(test_2)
int main (void)
{
   int iter_1 = 5122300;
   int iter = 5561457;


   time_t start;
   time_t start_gen;
   time_t finish;

   FILE * pFile;
   char mystring [100];
   pFile = fopen ("prime_1","r");
   if (pFile == NULL) perror ("Error opening file");
   else {
      if ( fgets (mystring , 100 , pFile) != NULL ) {
         //puts (mystring);
      } else {
         std::cout << "error read prime numbers 1\n";
```

```
        return 1;
    }
    fclose (pFile);
}

exit;
// remove new line character
char *pos;
if ((pos=strchr(mystring, '\n')) != NULL)
        *pos = '\0';

big p(mystring);

//std::cout << "p " << p << "\n";

pFile = fopen ("prime_2","r");
if (pFile == NULL) perror ("Error opening file");
else {
    if ( fgets (mystring , 100 , pFile) != NULL ) {
        //puts (mystring);
    } else {
        std::cout << "error read prime numbers 2\n";
        return 1;
    }
    fclose (pFile);
}

// remove new line character
if ((pos=strchr(mystring, '\n')) != NULL)
        *pos = '\0';

big q(mystring);
//std:cout << "q " << q << "\n";

big one = 1;
big null = 0;


big N = p * q;
big sqrt_N = 0;
LOG(1) std::cout << "iter = " << iter
                 << "\titer_1 " << iter_1
                 << "\tp=" << p
                 << "\tq=" << q
```

```
            << "\tN=" << N << "\n";
//DEBUG (1, "p=%" PRIu64 "\tq=%" PRIu64 "\tp*q=N=%" PRIu64 "\n", p, q, N);
sqrt_N = squareRoot(N);

sqrt_N = sqrt_N + one;
LOG(2) std::cout << "n=" << N << "\tsqrt=" << sqrt_N <<  "\n";

// selecting the size of the factor base
double size_B;
//DEBUG (2,"log _B=%f\n", ln(N));
//size_B = exp (0.5 * sqrt (ln(N) * log(ln(N))) );
size_B = exp (sqrt (ln(N) * log(ln(N))) );

size_B = pow(size_B , sqrt(2)/4);

DEBUG (2,"size of factor base size_B=%f\n", size_B);

std::vector<long long > p_smooth;
DEBUG (2, "smooth numbers\n");
make_smooth_numbers_1(p_smooth, size_B, N);


if ((p_smooth.size() < size_B))
{
   DEBUG (0, "Fail solution i=%d\tj=%d p=%lu\tq=%lu\t", iter, iter_1, p, q);
   finish = clock();

   return 1;
}
// selecting the sieving interval
long long  M;
M = exp (sqrt (ln(N) * log(ln(N))) );
M = pow(M , 3*sqrt(2)/4);

DEBUG (2, "The sieving interval M=%li\n", M);

// *** construct our sieve *** //
std::vector<big_2> X;
//std::vector<big_2> X_sm;
std::vector<big_2> Y;
//std::vector<big_2> Y_sm;
std::vector<big_2> V;

// simple sieve
```

```
    std::vector<long> solution_candidates_number;
    std::vector< std::vector<uint64_t> > v_exp;
    std::vector< std::vector<uint64_t> > v_exp_sm;
    //std::vector<int> smooth_num;

    bin_matrix_t m_all(p_smooth.size() + 1);
    bin_matrix_t m(p_smooth.size() + 1);

    long long x_count = 0;
    long long smooth_count = 0;
    long long eucl_count = 0;
    int exit_flag = 0;
    start = clock();
    start_gen = clock();
    for (long  j = 0, y_number = -1; j < M/2; j++){
        for (int  d = 0; d < 2; d++){
            big_2 tmp_x;
            if(d == 1 && j == 0)
                continue;
            if(d == 0 )
                tmp_x = sqrt_N -j;
                //X.push_back(sqrt_N - j);
            else
                //X.push_back(sqrt_N + j);
                tmp_x = sqrt_N + j;


            y_number++;
            x_count++;

            big_2 tmp_y = tmp_x*tmp_x;
            big_2 tmp_v;

            if(tmp_y < N) {

                tmp_y = N - tmp_y;
                tmp_y.sign = 1;

            } else
                tmp_y = tmp_y % N;
                //Y.push_back(tmp % N);


#define NEGATIVE_SIGN    0
```

```
#define FIRST_VALUE     1

        std::vector<uint64_t> v_exp_tmp(p_smooth.size() + 1);
        //V.push_back(Y[y_number]);

        if(tmp_y < null )
           v_exp_tmp[NEGATIVE_SIGN] = 1;

        tmp_v = prime_factorisation(tmp_y, p_smooth, v_exp_tmp);


        LOG(3) std::cout << "X = " << tmp_x << "\tY = " << tmp_y << " tmp_v " << tmp_v << " pers " << (j * 100) / (M/2) << "\n";

        big_2 one(1);
        big_2 min_one(-1);

        if(tmp_v == min_one || tmp_v == one){

           int null_flag = 1;
           //exit(0);
           null_flag = zero_vector_mod2_check(v_exp_tmp);

              DEBUG(3, "%s %d  try to add \n",__func__, __LINE__);
           if (null_flag && tmp_v > 0) { // sign check is extra !!!!
              continue; // special case need to work
              big_2 found = 0;
              std::vector<int64_t> tmp;
              tmp.push_back(y_number);
              found = euclid_gcd_big( X, Y, tmp, p, q, N,v_exp, p_smooth);
              if (! (found == 0)) {
                 exit_flag = 1;
                 break;
              }
           } else {
              Y.push_back(tmp_y);
              X.push_back(tmp_x);
              v_exp.push_back(v_exp_tmp);
              smooth_count++;

              finish = clock();
              ofstream myfile;
              LOG(1) std::cout << tmp_x << "\t" << tmp_y  << "\t" << (double)(finish - start) / CLOCKS_PER_SEC << "\n";
```

```
            start = clock();

            if (m_all.add_row(v_exp_tmp) == 1){
               m.add_row(v_exp_tmp);
               int exponent_num = (v_exp_tmp.size() - 1);
               // ERROR("exp %d exp_num %d\n", v_exp[y_number][exponent_num],
exponent_num);
               int count_flag = 0;
               //add_counter_row(m_counter ,counter ,exponent_num);
               // DEBUG (2,"size num = %d\t", smooth_num.size());
               m_all.show();
               int null_line = m_all.make_upper_triangular_static();

               if (null_line > -1) {
                  std::vector<int64_t> XYiters;
                  DEBUG(3, "line %d NUll line  %d=============", __LINE__ ,
null_line);
                  for (uint64_t col = 0; col <  m_all.filled; ++col) {
                     DEBUG (3,"matrix[%d][%d] = %ld\n",null_line,col,
m_all.unit_matrix[null_line][col]);
                     if( m_all.unit_matrix[null_line][col] > 0) {
                        DEBUG (3,"num = %d\t",col);
                        LOG(3) std::cout << "Y  = " <<  Y[col] << "\n";
                        XYiters.push_back(col);
                     }
                  }
                  DEBUG (2,"\n");

                  eucl_count++;
                  big found = 0;
                  found = euclid_gcd_big( X, Y, XYiters, p, q, N, v_exp, p_smooth);
                  // printf("found %lu\n", found);
                  // m_all.show();
                  if (found.size != 0) {
                     std::cout << "Found solution i=" << iter << "\tj=" << iter_1 << " p="
<< p << " q=" << q << "\n";
                     // exit( null_line);
                     exit_flag=1;
                     //break_flag = 1;
                     break;
                  } else {
                     //m_all.delete_row(m_all.filled -1);
                     //m.show();
```

```
                        LOG(2) std::cout << "filled =" << m_all.filled <<  "\n";
                        if (m_all.filled > m_all.collumn_size) {
                           m_all.show();
                           LOG(2) std::cout << "trunagular size " << m_all.triangular_v.size()
<< "\n";
                           for (int j = 0; j < m_all.triangular_v.size(); j++)
                              LOG(2) std::cout << m_all.triangular_v[j] << "\n";
                           LOG(2) std::cout << "\n";
                        }
                        m_all.delete_row(m_all.filled - 1 );
                        m.delete_row(m_all.filled - 1 );
                        LOG(2) std::cout << "filled1 =" << m_all.filled <<  "\n";
                        if (m_all.filled > m_all.collumn_size) {
                           m_all.show();
                           ERROR("matrix to big aaaaa\n");
                           exit(0);
                        }

                        Y.pop_back();
                        X.pop_back();
                        v_exp.pop_back();
                        //exit(0);
                     }
                  }

               }
               else
               {
                  ERROR("cant add aaaaa\n");
                  exit(0);
               }

            }
            DEBUG (3, "\n");
         }
      }
      if (exit_flag)
         break;
   }
   if(!exit_flag)
      LOG(0) std::cout << "Fail solution i=" << iter << "\tj=" << iter_1 << " p=" << p <<
"\tq=" << q << "\n";

               finish = clock();
```

```
    ofstream myfile;
                myfile.open ("results.txt", ios::app);
    myfile        << "" << exit_flag
                << "\t" << p
                << "\t" << q
                << "\t" << N
                << "\t" <<  M
                << "\t" << size_B
                << "\t" << x_count
                << "\t" << smooth_count
                << "\t" << eucl_count
                << "\t" << (double)(finish - start_gen) / CLOCKS_PER_SEC
                << "\n";

                myfile.close();

    return 0;
}

//file quadratic_sieve_big.cpp

#include <vector>
#include <stdint.h>
#include <stdio.h>
#include <math.h>

#include "bin_matrix.h"
#include "big_2.h"
#include "greatest_common_divisor_big.h"
#include "quadratic_sieve_big.h"

#include <gmp.h>
#include <gmpxx.h>



int make_exp_array_condBsmooth(std::vector< std::vector<uint64_t> > &v_exp,
std::vector<int> &smooth_num, std::vector<long> Y, std::vector<long> &p_smooth,
double size_B, uint32_t M,
    std::vector<long> &solution_candidates_number, std::vector<uint64_t> &v_extra_exp)
{

        std::vector<long> V;
        V = Y;
```

```
// add sign to exponent matrix
#define NEGATIVE_SIGN    0
#define FIRST_VALUE    1
// v_exp[i].size()-1

int print_flag = 1;
for (int y_number = 0; y_number < v_exp.size(); ++y_number)
{

   if(V[y_number] < 0 )
      v_exp[y_number][NEGATIVE_SIGN] = 1;

   for (   int smooth_iter = 0, exponent_num = FIRST_VALUE ;
         smooth_iter < p_smooth.size();
         smooth_iter++, exponent_num++)
   {
      long int tmp;
      do{
         tmp = V[y_number] % p_smooth[smooth_iter];
         DEBUG (4, "v = %10li\t",V[y_number]);
         DEBUG (4, "p_smooth = %li\t",p_smooth[smooth_iter]);
         DEBUG (4, "tmp = %li\n",tmp);
         if(tmp == 0){
            V[y_number] = V[y_number] / p_smooth[smooth_iter];
            v_exp[y_number][exponent_num] += 1;
         }
      } while (tmp == 0);

      if(V[y_number] == -1 || V[y_number] == 1){
         int null_flag = 1;
         // printf("V = %" PRIu64 "\t",V[i]);
         for (   int exp_num = 0;
                  exp_num < v_exp[y_number].size();
                  exp_num++ )
         {
            DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
            if ((v_exp[y_number][exp_num] % 2 )!= 0)
               null_flag = 0;
         }
         DEBUG (3, "%ld\n", Y[y_number]);
         // skip negative value !!!!
         if (null_flag && V[y_number] > 0) {
            solution_candidates_number.push_back(y_number);
         } else {
```

```
                smooth_num.push_back(y_number);
            }
            DEBUG (3, "\n");

            break;
        }
    }

    // DEBUG (0, "V ===== %li \n",V[y_number]);
    if(V[y_number] != -1 && V[y_number] != 1){
        int sign_flag = 0;
        if(V[y_number] < 0 ){
            sign_flag = 1;
            V[y_number] *= -1;
        }

        double res = sqrt(V[y_number]);
        if(res == trunc(res)) {
            v_extra_exp[y_number] = res;
            V[y_number] /= V[y_number];
            if(print_flag){
                DEBUG (0, "found
========================================= %f  number %d\n",res,
y_number);
                print_flag = 0;
            }
            DEBUG (2, "Y = %li\tV = %li\n",Y[y_number], V[y_number]);
            // DEBUG (0, "iter %d\n",);

            int null_flag = 1;
            // printf("V = %" PRIu64 "\t",V[i]);
            for (   int exp_num = 0;
                        exp_num < v_exp[y_number].size();
                        exp_num++ )
            {
                DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
                if ((v_exp[y_number][exp_num] % 2 )!= 0)
                    null_flag = 0;
            }
            DEBUG (3, "\n");

            if (null_flag && V[y_number] > 0) {
                solution_candidates_number.push_back(y_number);
                // DEBUG (2, "sol cand %d\n",solution_candidates_number.size() );
```

```
                } else {
                    smooth_num.push_back(y_number);
                }
                // smooth_num.push_back(y_number);
            }

            if(sign_flag)
                V[y_number] *= -1;
        }

    }

    if (smooth_num.size()  < size_B + 1)
    {
        //ERROR( "to small number of smooth numbbers\n");
        return 0;
    }
  return 1;
}

// additional check by mod 8
int make_exp_array_condBsmooth_1(std::vector< std::vector<uint64_t> > &v_exp,
std::vector<int> &smooth_num, std::vector<long> Y, std::vector<long> &p_smooth,
double size_B, uint32_t M,
  std::vector<long> &solution_candidates_number, std::vector<uint64_t> &v_extra_exp)
{

    std::vector<long> V;
    V = Y;
    // add sign to exponent matrix
    #define NEGATIVE_SIGN    0
    #define FIRST_VALUE    1
    // v_exp[i].size()-1

    int print_flag = 1;
    for (int y_number = 0; y_number < v_exp.size(); ++y_number)
    {

        if(V[y_number] < 0 )
            v_exp[y_number][NEGATIVE_SIGN] = 1;

        for (   int smooth_iter = 0, exponent_num = FIRST_VALUE ;
                smooth_iter < p_smooth.size();
                smooth_iter++, exponent_num++)
```

```
{
   long int tmp;
   do{
      tmp = V[y_number] % p_smooth[smooth_iter];
      DEBUG (4, "v = %10li\t",V[y_number]);
      DEBUG (4, "p_smooth = %li\t",p_smooth[smooth_iter]);
      DEBUG (4, "tmp = %li\n",tmp);
      if(tmp == 0){
         V[y_number] = V[y_number] / p_smooth[smooth_iter];
         v_exp[y_number][exponent_num] += 1;
      }
   } while (tmp == 0);

   if(V[y_number] == -1 || V[y_number] == 1){
      int null_flag = 1;
      // printf("V = %" PRIu64 "\t",V[i]);
      for (   int exp_num = 0;
              exp_num < v_exp[y_number].size();
              exp_num++ )
      {
         DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
         if ((v_exp[y_number][exp_num] % 2 )!= 0)
            null_flag = 0;
      }
      DEBUG (3, "%ld\n", Y[y_number]);
      // skip negative value !!!!
      if (null_flag && V[y_number] > 0) {
         solution_candidates_number.push_back(y_number);
      } else {
         smooth_num.push_back(y_number);
      }
      DEBUG (3, "\n");

      break;
   }
}

// show();

// DEBUG (0, "V ===== %li \n",V[y_number]);
if(V[y_number] != -1 && V[y_number] != 1){
   int sign_flag = 0;
   if(V[y_number] < 0 ){
      sign_flag = 1;
```

```
            V[y_number] *= -1;
        }

        if (V[y_number] % 8 == 1 )
        {
            double res = sqrt(V[y_number]);
            if(res == trunc(res)) {
                v_extra_exp[y_number] = res;
                V[y_number] /= V[y_number];
                if(print_flag){
                    DEBUG (0, "found
========================================= %f  number %d\n",res,
y_number);
                    print_flag = 0;
                }
                DEBUG (2, "Y = %li\tV = %li\n",Y[y_number], V[y_number]);
                // DEBUG (0, "iter %d\n",);

                int null_flag = 1;
                // printf("V = %" PRIu64 "\t",V[i]);
                for (   int exp_num = 0;
                        exp_num < v_exp[y_number].size();
                        exp_num++ )
                {
                    DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
                    if ((v_exp[y_number][exp_num] % 2 )!= 0)
                        null_flag = 0;
                }
                DEBUG (3, "\n");

                if (null_flag && V[y_number] > 0) {
                    solution_candidates_number.push_back(y_number);
                    // DEBUG (2, "sol cand %d\n",solution_candidates_number.size() );
                } else {
                    smooth_num.push_back(y_number);
                }
                // smooth_num.push_back(y_number);
            }
        }

        if(sign_flag)
            V[y_number] *= -1;
    }
```

```
        }

       if (smooth_num.size()  < size_B + 1)
        {
           //ERROR( "to small number of smooth numbbers\n");
           return 0;
        }
    return 1;
}

int make_exp_array(std::vector< std::vector<uint64_t> > &v_exp, std::vector<int>
&smooth_num, std::vector<big_2> Y, std::vector<long long> &p_smooth, double size_B,
long long  M,
    std::vector<long long > &solution_candidates_number)
{

        std::vector<big_2> V;
        V = Y;
          big_2 null = 0;
        // add sign to exponent matrix
        #define NEGATIVE_SIGN    0
        #define FIRST_VALUE    1
        // v_exp[i].size()-1
        big_2 one(1);
        big_2 min_one(-1);

           // show();

        for (int y_number = 0; y_number < v_exp.size(); ++y_number)
        {
           if(V[y_number] < null )
              v_exp[y_number][NEGATIVE_SIGN] = 1;

           V[y_number] = prime_factorisation(Y[y_number], p_smooth, v_exp[y_number]);

           if(V[y_number] == min_one || V[y_number] == one){

               int null_flag = 1;
               null_flag = zero_vector_mod2_check(v_exp[y_number]);

               if (null_flag && V[y_number] > 0) {
                  solution_candidates_number.push_back(y_number);
               } else {
                     smooth_num.push_back(y_number);
```

```
            }

        }

    }

    if (smooth_num.size()  < size_B + 1)
    {
       // show();
       //ERROR( "to small number of smooth numbbers\n");
       return 0;
    }
  return 1;
}

void construct_xy(std::vector<big_2> &X, std::vector<big_2> &Y, big_2 sqrt_N, big_2
N, long long  M)
{
      long y_number = 0;
      for (long  j = 0 ; j < M/2; j++){
            for (int  d = 0; d < 2; d++){
                  if(d == 1 && j == 0)
                        continue;
                  if(d == 0 )
                        X.push_back(sqrt_N - j);
                  else
                        X.push_back(sqrt_N + j);

                  big_2 tmp = X[y_number]*X[y_number];
                  //std::cout <<   "tmp " << tmp << "\n";
                  //std::cout <<   "N " << N << "\n";
                  if(tmp < N) {
                        //std::cout <<   "tmp sign " << tmp << "\n";
                        //std::cout <<   "N sign " <<  N << "\n";
                        tmp = N - tmp ;
                        tmp.sign = 1;
                        //std::cout <<   "tmp1 " << tmp << "\n";
                        //std::cout <<   "N " << N << "\n";
                        Y.push_back(tmp);
                        //Y.push_back(N-tmp);
                  } else
                        Y.push_back(tmp % N);
                  LOG(2) std::cout << "X = " << X[y_number] << "\tY = " <<
Y[y_number] << "\n";
```

```
                        y_number++;
                        //exit(0);
                    }
                }
            }
```

```
// uint64_t max value  10^19
//  N max 10^9
// p max 10^4
/*
void make_smooth_numbers(std::vector<big> &p_smooth, double size_B, const big N)
{
   //prime is 2 - special case
   // Modulo 2, every integer is a quadratic residue.
   p_smooth.push_back(prime[2]);
   DEBUG(2, "%llu\n", prime[2]);

   for (uint64_t i = 3; (p_smooth.size() < size_B) && (i < prime_size); ++i)
   {
      long long  tmp;
      //tmp  = N;
      big one(1);
         // show();
       big tmp_p( prime[i]);
       //std::cout << "N=" << N << " % ";
       //std::cout << "N=" << N << "\n";
       //std::cout << tmp_p << "\n";
     big N_mod;
     N_mod = N % tmp_p;
      long long N_mod_l = N_mod.to_long();
     //tmp.number[0] = 0;
       //std::cout << "N1=" << N << "\n";
       //std::cout << "=" << N_mod << "\n";
      tmp = N_mod_l;
       //std::cout << "tmp1=" << tmp << "\n";
      for (int j = 1; j < (prime[i]-1)/2; ++j)
      {
      tmp = tmp * N_mod_l;
      //std::cout << "tmp1= " << tmp << "\t";
      //std::cout << "N23=" << N << "\n";
      //std::cout << "N32=" << N.number[0] << "\n";
      //tmp = tmp % tmp_p;
      tmp = tmp % prime[i];
```

```
      //std::cout << "tmp2= " << tmp << "\n";

        //std::cout << "N2=" << N << "\n";
          //tmp = tmp * N_mod;
          // show();
        //std::cout << "N23=" << N << "\n";
        //std::cout << "N32=" << N.number[0] << "\n";
          //tmp = tmp % tmp_p;
          //tmp % tmp_p;
          //tmp = (tmp * N_mod) % tmp_p;
        //std::cout << "N3=" << N << "\n";
        //std::cout << "tmp=" << tmp << "\n";
      }
        //std::cout << "N7=" << N << "\n";
      //exit(0);
     // tmp = tmp % tmp_p;

        //std::cout << "N4=" << N << "\n";
      //if( tmp == one)
      if( tmp == 1)
      {
         p_smooth.push_back(tmp_p);
         //std::cout << "tmp " << tmp << "added " << tmp_p << "\n";
         DEBUG(2, "%llu\n", prime[i]);
      }
        //std::cout << "N5=" << N << "\n";
   }
}
*/

void make_smooth_numbers_1(std::vector<long long > &p_smooth, double size_B, const
big N)
{
   //prime is 2 - special case
   // Modulo 2, every integer is a quadratic residue.
   p_smooth.push_back(2);
   DEBUG(2, "%llu\n", 2);

   for (uint64_t i = 3; (p_smooth.size() < size_B) && (i < prime_size); ++i)
   {
      long long  tmp;
      //tmp  = N;
      big one(1);
         // show();
```

```
        big tmp_p( prime[i]);
        big N_mod;
        N_mod = N % tmp_p;
        long long N_mod_l = N_mod.to_long();
        tmp = N_mod_l;

        if(mp_legendre_1(N_mod_l, prime[i]) == 1)
           //if( tmp == one)
           //if( tmp == 1)
        {
           p_smooth.push_back(prime[i]);
           //std::cout << "tmp " << tmp << "added " << tmp_p << "\n";
           DEBUG(2, "%llu\n", prime[i]);
        }
        //std::cout << "N5=" << N << "\n";
    }
}

void make_smooth_numbers_2(std::vector<long long > &p_smooth, double size_B, const
big N)
{
   //prime is 2 - special case
   // Modulo 2, every integer is a quadratic residue.
   p_smooth.push_back(2);
   DEBUG(2, "%llu\n", 2);
        mpz_t prime;
        long int p;
        mpz_t prime_next;
        mpz_init(prime);
        mpz_init(prime_next);
        mpz_set_ui (prime, 2);

   for (uint64_t i = 3; (p_smooth.size() < size_B) ; ++i)
   {
      long long  tmp;
      //tmp  = N;
      big one(1);
      mpz_nextprime(prime, prime);
      gmp_printf ("%Zd\n",prime);
      p = mpz_get_si(prime);
      big tmp_p( p);
      big N_mod;
      N_mod = N % tmp_p;
      long long N_mod_l = N_mod.to_long();
```

```
        tmp = N_mod_l;

        if(mp_legendre_1(N_mod_l, p) == 1)
           //if( tmp == one)
           //if( tmp == 1)
        {
           p_smooth.push_back(p);
           //std::cout << "tmp " << tmp << "added " << tmp_p << "\n";
           DEBUG(2, "%llu\n", p);
        }
        //std::cout << "N5=" << N << "\n";
     }
}


/*----------------------------------------------------------------*/
long long  mp_legendre_1(long long   a, long long  p) {

   long long  x, y, tmp;
   long long  out = 1;

   x = a;
   y = p;
   while (x) {
      while ((x & 1) == 0) {
         x = x / 2;
         if ( (y & 7) == 3 || (y & 7) == 5 )
            out = -out;
      }

      tmp = x;
      x = y;
      y = tmp;

      if ( (x & 3) == 3 && (y & 3) == 3 )
         out = -out;

      x = x % y;
   }
   if (y == 1)
      return out;
   return 0;
}
```

```
int fill_matrix(bin_matrix_t &m1, std::vector<int> &smooth_num, std::vector<
std::vector<uint64_t> > &v_exp)
{
   int count = 0;

   // DEBUG(3, "%s %d \n",__func__, __LINE__);
   // DEBUG(3, "%s %d filled \n",__func__, m1.filled);
   // DEBUG(3, "%s %d size \n",__func__, m1.size);
   if ((m1.row_size) == m1.filled)
      return count;

   DEBUG(4, "%s %d \n",__func__, __LINE__);
   for (int i = 0; i < smooth_num.size() && ((m1.row_size) != m1.filled); ++i) {
      // DEBUG(3, "%s %d iteration\n",__func__, i);
      // DEBUG(3, "%s %d filled \n",__func__, m1.filled);
      // DEBUG(3, "%s %d size \n",__func__, m1.size);
      // DEBUG(3, "%s %d smooth_num \n",__func__, smooth_num[i]);
      if (smooth_num[i] != -1) {
         DEBUG(3, "%s %d  try to add %d \n",__func__, __LINE__, smooth_num[i]);
         m1.add_row(v_exp[ smooth_num[i]]);
         smooth_num[i] = -1;
         // DEBUG(3, "%s %d =added\n",__func__, smooth_num[i]);
         count++;
      }
   }
   return count;
}


void add_counter_row(bin_matrix_t &m2 ,std::vector<uint64_t> &counter ,int
exponent_num)
{
   std::vector<uint64_t> tmp(counter.size());
   // for (int i = exponent_num; i < v_exp[y_number].size(); ++i)
   for (int i = exponent_num; i < counter.size(); ++i)
   {
         // show();
      tmp[i] = 1;
      counter[i]++;
   }
   m2.add_row(tmp);
   // ERROR("added num %d\n", exponent_num);
/*
   int count_flag = 0;
```

```
   for (exponent_num = 1; exponent_num < counter.size(); ++exponent_num)
   {
      DEBUG(2, "counter[%d]=%lu\t", exponent_num, counter[exponent_num] );
      if (counter[exponent_num] == exponent_num + 2)
      {
         DEBUG(2, "flag is set \n");
         count_flag = 1;
         break;
      }
   }
      DEBUG(2, "\n");
   if (count_flag)
      return exponent_num;
   else
      return -1;*/
}

int is_counter_full(std::vector<uint64_t> &counter)
{
   int count_flag = 0;
   int exponent_num;
   for (exponent_num = 1; exponent_num < counter.size(); ++exponent_num)
   {
      DEBUG(2, "counter[%d]=%lu\t", exponent_num, counter[exponent_num] );
      if (counter[exponent_num] == exponent_num + 2)
      {
         DEBUG(2, "flag is set \n");
         count_flag = 1;
         break;
      }
   }
      DEBUG(2, "\n");
   if (count_flag)
      return exponent_num;
   else
      return -1;

}

big prime_factorisation(big Y, std::vector<long long> p_smooth, std::vector<uint64_t>
&v_exp)
{
        big null(0);
        big one(1);
```

```
        big min_one(-1);
    for (   int smooth_iter = 0, exponent_num = FIRST_VALUE ;
                    smooth_iter < p_smooth.size();
                    smooth_iter++, exponent_num++)
    {
        big tmp;
        long long  reminder;
        do{
            big quotient;


            div_rem_l(Y, p_smooth[smooth_iter], quotient, reminder);
            if(reminder == 0){
                //Y = Y / p_smooth[smooth_iter];
                Y = quotient;
                LOG(3) std::cout << "y = " << Y << "\ttmp = " << tmp << "\tp_smooth " <<
p_smooth[smooth_iter] << " \n";
                LOG(3) std::cout << "expon num " << exponent_num <<" \n";
                v_exp[exponent_num] += 1;
            }
                        //exit(0);
        } while (reminder == 0);

        if(Y == one || Y == min_one){
            break;
                }
    }
    return Y;
}

int zero_vector_mod2_check(std::vector<uint64_t> v_exp) {
    // modulo-2 division
    int null_flag = 1;
    for (   int exponent_num = 0;
                exponent_num < v_exp.size();
                exponent_num++ )
    {
        DEBUG (4, "%ld\t", v_exp[exponent_num]);
        v_exp[exponent_num] %= 2;
        if (v_exp[exponent_num] != 0)
            null_flag = 0;
    }
    DEBUG (4, "\n");
    return null_flag;
```

```
}

#if 0
int find_solution_big (bin_matrix_t & m2,
                       std::vector<int> &smooth_num_back,
                       std::vector<int> &smooth_num,
                       std::vector< std::vector<uint64_t> > &v_exp,
                       std::vector<long long> p_smooth,
                       const std::vector<big_2>& X,
                       const std::vector<big_2>& Y,
                       const big_2 &p,
                       const big_2 &q,
                       const big_2 &N)
{

   std::vector<int64_t> P11;
   int retval = fill_matrix(m2, smooth_num_back, v_exp);
   m2.show();
   DEBUG(2, "%d 1added\n", retval);
   if (retval == 0 || m2.filled != m2.row_size)
      return -1;

   bin_matrix_t m1 = m2;

   int null_line = m1.resolve_matrix();
   // m1.show();
   // return 0;
   //WARN(1, "it should be -1 or greater. null =  %d\n", null_line);

   if (null_line > -1)
   {
         // show();
      // DEBUG(2, "collumn size %d\n", m1.collumn_size);
      for (uint64_t col = 0; col <  m1.collumn_size; ++col)
      {
         DEBUG (2,"matrix[%d][%lu] = %lu\n",null_line,col,
m1.unit_matrix[null_line][col]);
         if( m1.unit_matrix[null_line][col] > 0)
         {
            DEBUG (2,"num = %d\t", smooth_num[col]);
               DEBUG (2,"Y  = %ld\n", Y[smooth_num[col]]);
               P11.push_back(smooth_num[col]);
         }
      }
```

```
        DEBUG (2,"\n");

        big_2 found = 0;
        found = euclid_gcd_big( X, Y, P11, p, q, N, v_exp, p_smooth);
        // printf("found %lu\n", found);
        m1.show();
        if (! (found == 0)) {
           return null_line;
        }
        else {

           int max_i = 0;
           max_i = m2.max_unit_num(m1.unit_matrix[null_line]);

           DEBUG (3," iter %d\n",max_i );
           m2.show();
           m2.delete_row(max_i);
           DEBUG (4,"===== %d \n", smooth_num_back[max_i]);
           smooth_num_back.erase(smooth_num_back.begin() + max_i);
           smooth_num.erase(smooth_num.begin() + max_i);
           m2.show();


           // return find_solution(m2, smooth_num_back);
           null_line = find_solution_big(m2, smooth_num_back, smooth_num, v_exp,
p_smooth, X, Y, p, q, N);
           // DEBUG(3, "finish %d\n", null_line);
           if (null_line == -1)
              WARN(1, "failed\n");

        }
     } else {
           int max_i = 0;
           m2.show();
           m2.delete_row(max_i);
           DEBUG (4,"===== %d \n", smooth_num_back[max_i]);
           smooth_num_back.erase(smooth_num_back.begin() + max_i);
           smooth_num.erase(smooth_num.begin() + max_i);
           m2.show();


           // return find_solution(m2, smooth_num_back);
           null_line = find_solution_big(m2, smooth_num_back, smooth_num, v_exp,
p_smooth, X, Y, p, q, N);
```

```
        // DEBUG(3, "finish %d\n", null_line);
        if (null_line == -1)
            WARN(1, "failed\n");
    }
    return null_line;
}
int find_solution_condBsmooth (bin_matrix_t m2,
                std::vector<int> &smooth_num_back,
                std::vector<int> &smooth_num,
                std::vector< std::vector<uint64_t> > &v_exp,
                std::vector<long> p_smooth,
                const std::vector<long>& X,
                const std::vector<long>& Y,
                const uint64_t &p,
                const uint64_t &q,
                const uint64_t &N,
                std::vector<uint64_t> &v_extra_exp)
{

    std::vector<int64_t> P11;
    int retval = fill_matrix(m2, smooth_num_back, v_exp);
    m2.show();
    DEBUG(2, "%d 1added\n", retval);
    if (retval == 0 || m2.filled != m2.row_size)
        return -1;

    // for (int i = 0; i < smooth_num_back.size() ; ++i) {
    //  DEBUG(3, "%s %d smooth_num \n",__func__, smooth_num_back[i]);
    // }
        // show();

    bin_matrix_t m1 = m2;

    // DEBUG(1, "--------------------------");
    int null_line = m1.resolve_matrix();
    // m1.show();
    // return 0;

    WARN(1, "it should be -1 or greater. null =  %d\n", null_line);

    if (null_line > -1)
    {
        // DEBUG(2, "collumn size %d\n", m1.collumn_size);
        for (uint64_t col = 0; col <  m1.collumn_size; ++col)
```

```
      {
         DEBUG (2,"matrix[%d][%lu] = %ld\n",null_line,col,
m1.unit_matrix[null_line][col]);
         if( m1.unit_matrix[null_line][col] > 0)
         {
            DEBUG (2,"num = %d\t", smooth_num[col]);
               DEBUG (2,"Y  = %ld\n", Y[smooth_num[col]]);
               P11.push_back(smooth_num[col]);
         }
      }
      DEBUG (2,"\n");

      int found = 0;
      found = euclid_gcd_m_big( X, Y, P11, p, q, N, v_exp, p_smooth, v_extra_exp);
      // printf("found %lu\n", found);
      m1.show();
      if (found) {
         return null_line;
      }
      else {

         int max_i = 0;
         //max_i = m2.max_unit_num(m1.unit_matrix[null_line]); // temp need to return
back
         max_i = smooth_num_back.size() -1;

         DEBUG (3," iter %d\n",max_i );
         m2.show();
         m2.delete_row(max_i);
         DEBUG (4,"===== %d \n", smooth_num_back[max_i]);
         smooth_num_back.erase(smooth_num_back.begin() + max_i);
         smooth_num.erase(smooth_num.begin() + max_i);
         m2.show();


         // return find_solution(m2, smooth_num_back);
         null_line = find_solution_condBsmooth(m2, smooth_num_back, smooth_num,
v_exp, p_smooth, X, Y, p, q, N, v_extra_exp);
         DEBUG(3, "finish %d\n", null_line);
         if (null_line == -1)
            WARN(1, "failed\n");
         // show();
```

```
        }
    } else {
        int max_i = 0;
        m2.show();
        m2.delete_row(max_i);
        DEBUG (4,"===== %d \n", smooth_num_back[max_i]);
        smooth_num_back.erase(smooth_num_back.begin() + max_i);
        smooth_num.erase(smooth_num.begin() + max_i);
        m2.show();


        // return find_solution(m2, smooth_num_back);
        null_line = find_solution_condBsmooth(m2, smooth_num_back, smooth_num,
v_exp, p_smooth, X, Y, p, q, N, v_extra_exp);
        DEBUG(3, "finish %d\n", null_line);
        if (null_line == -1)
            WARN(1, "failed\n");
    }

    return null_line;
}
#endif


//file  bin_matrix.cpp

#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <inttypes.h>
#include <vector>
#include <algorithm>
#include <unistd.h>

#include "bin_matrix.h"

bin_matrix_t::bin_matrix_t(){}

bin_matrix_t::bin_matrix_t(int size) : matrix(size + 1 , std::vector<uint64_t> (size)),
                        unit_matrix(size, std::vector<uint64_t> (size)),
                        unit_num(size + 1),
                        collumn_size(size),
                        row_size(size + 1),
                        unit_matrix_size(size),
```

```
                        filled(0)
{
   init_unit();
}

void bin_matrix_t::init_unit(void)
{
   // don't feel the last row
   for (int row = 0; row < unit_matrix_size; ++row)
   {
      unit_num[row] = 0;
      for (int col = 0; col < unit_matrix_size; ++col)
      {
         if(row == col)
            unit_matrix[row][col] = 1;
         else
            unit_matrix[row][col] = 0;

         DEBUG (4,"%ld\t", unit_matrix[row][col]);
      }
      DEBUG (4,"\n");
   }
}


void bin_matrix_t::show(void){
      // just print exponent mod 2 array
   DEBUG (3,"matrix \n");
   for (int row = 0; row < row_size; ++row)
   {
      for (int col = 0; col < collumn_size; ++col)
      {
         DEBUG (3,"%lu\t", matrix[row][col]);
      }
      DEBUG(3, "\n");
   }

         // show();
   DEBUG (3,"\n");
   DEBUG (3,"unit matrix\n");
   for (int row = 0; row < unit_matrix_size; ++row)
   {
      for (uint64_t col = 0; col <  unit_matrix_size; ++col)
      {
```

```
            DEBUG (3,"%"PRIu64"\t", unit_matrix[row][col]);
        }
        DEBUG (3,"\n");
    }
    DEBUG (3,"\n");
}

int bin_matrix_t::add_row(std::vector<uint64_t> row_v)
{
    // DEBUG(4, "%s %d \n",__func__, __LINE__);
    if (filled < row_size){
        DEBUG(3, "%s %d \n",__func__, __LINE__);
        for (int col = 0; col < row_v.size() && col < collumn_size; ++col) {
            DEBUG(4, "%s %d \n",__func__, __LINE__);
            matrix[filled][col] = row_v[col] % 2;
            DEBUG(4, "%s %d \n",__func__, __LINE__);
        }
        // DEBUG(4, "%s %d \n",__func__, __LINE__);
        filled++;
        return 1;
    }
    else
        return 0;
    // DEBUG(4, "%s %d filled %d\n",__func__, __LINE__, filled);
}

int bin_matrix_t::delete_row(unsigned int row_number)
{
    if (filled > row_number){
        DEBUG(3, "%s %d row_number: %d filled: %d\n",__func__, __LINE__,
row_number, filled);
            // show();
        for (int row = row_number; row < filled  ; ++row)
        {
            DEBUG(4, "%s %d \n",__func__, __LINE__);
            for (int col = 0; col < unit_matrix_size; ++col)
            {
                DEBUG(4, "%s %d row=|%d| col=|%d|\n",__func__, __LINE__, row, col);
                if (row == (filled - 1))
                    matrix[row][col] = 0;
                else
                    matrix[row][col] = matrix[row+1][col];
            }
```

```
        }
        DEBUG(4, "%s %d \n",__func__, __LINE__);
        filled--;
        DEBUG(4, "%s %d filled %d\n",__func__, __LINE__, filled);
        return 1;
    } else
        return 0;
}

int  bin_matrix_t::make_upper_triangular(void)
{
    int current_col = 0;
    int current_row = 0;
    int null_line = -1;
    for (int i = 0; i < unit_matrix_size ; ++i)
    {

        int row_nonnull;
        // find first non zero value. it will be in row.
        for (row_nonnull = current_row; row_nonnull < unit_matrix_size; ++row_nonnull)
        {
            if (matrix[row_nonnull][current_col] == 1)
                break;
        }

        DEBUG (3,"row_nonnull=%d current_row=%d unit_matrix_size=%d
current_col=%d\n",
        row_nonnull, current_row, unit_matrix_size, current_col);

        // row should be less then unit_matrix_size , otherwise all value are zero
        if(row_nonnull == unit_matrix_size ){
            WARN (1, "We havn't find values in column %d \n", current_col);
            current_col++;
            continue;
        } else if( row_nonnull > current_row && row_nonnull < unit_matrix_size) {
            DEBUG(3, "i = %d row_nonnull = %d\n", i, row_nonnull);
            matrix[row_nonnull].swap(matrix[current_row]);
            unit_matrix[row_nonnull].swap(unit_matrix[current_row]);
        }

        // remove all value under the first one
        for (int row = current_row + 1; row < unit_matrix_size; ++row) {
            if(matrix[row][current_col] != 0) {
                // DEBUG(3, "CHAECK r = %d row=%d\n", r, row);
```

```
            // show();
            for (int col = 0; col < unit_matrix_size; ++col) {

                // DEBUG(3, "matrix r = %d c=%d m=%ld\n", r, c, matrix[r][c]);
                matrix[row][col] += matrix[current_row][col];
                matrix[row][col] %= 2;
                DEBUG(4, "matrix row = %d col=%d m=%ld\n", row, col, matrix[row][col]);

                unit_matrix[row][col] += unit_matrix[current_row][col];
                unit_matrix[row][col] %= 2; // remove unnecessary moves
                                // if we use two the same lines they remove each other.
            }
          }
          // show();
          if ( std::find(matrix[row].begin(), matrix[row].end(), 1) == matrix[row].end() ) {
                DEBUG (1,"nULLL upper j=%d coll=%d\n", row, current_col);
                // DEBUG (3," smooth_num \n");
                null_line = row;
                break;
          }
          // show();
       }

       if (null_line != -1)
          break;

       // show();
       current_col++;
       current_row++;
    }
    show();
    return null_line;
}

void bin_matrix_t::count_unit_num( void )
{
    for (int row = 0; row < row_size; ++row)
    {
       unit_num[row] = 0;
       for (int cal = 0; cal < collumn_size; ++cal)
       {
          if (matrix[row][cal] == 1){
             unit_num[row] += 1;
          }
```

```
        }
    }
}

int bin_matrix_t::max_unit_num(std::vector<uint64_t> selected_row)
{
    count_unit_num();
    int max_unit = 0;
    int max_iter = 0;
    for (int iter = 0; iter < unit_matrix_size; ++iter)
    {
        DEBUG(3, "%lu\t", selected_row[iter]);
        if (selected_row[iter] == 1){
            if (unit_num[iter] > max_unit){
                max_unit = unit_num[iter];
                max_iter = iter;
            }
        }
    }
    // printf("max %d\n", max_unit);
    DEBUG(3, "\n");
    return max_iter;
}



int  bin_matrix_t::make_lower_triangular(void)
{
    // printf("tpp1 size %d\n", tpp1.size());
    int null_line = -1;
    for (int i = (unit_matrix_size -1) ; i > 0; --i)
    {
        if (matrix[i][i] != 1) {
            WARN(1, "we have zero value in main diagonal, line: %d\n", i);
            continue;
        }

        for (int row = i-1; row >= 0; --row)
        {
            DEBUG(3, "check row=%d\n", row);
            if(matrix[row][i] != 0)
            {
                DEBUG(2, "change row=%d\n", row);
                for (int col = 0; col < collumn_size; ++col) {
```

```
                matrix[row][col] += matrix[i][col];
                matrix[row][col] %= 2;

                unit_matrix[row][col] += unit_matrix[i][col];
                // unit_matrix[r][c] %= 2;  // ???????
            }
        }

        if ( std::find(matrix[row].begin(), matrix[row].end(), 1) == matrix[row].end() ) {
            DEBUG (3,"nULLL lower %d\n", row);
            null_line = row;
        }
    }

    if (null_line != -1)
        break;

  show();

  }
  return null_line;

}


int bin_matrix_t::resolve_matrix()
{
  std::vector<uint64_t> P11;
  uint64_t found = 0;

  show();
  int null_line = make_upper_triangular();
  DEBUG(2, "upper triangular\n");
  // show();


  if (null_line != -1) {
    return null_line;
  }

  // exit(0);

  null_line = make_lower_triangular();
  DEBUG(2, "lower triangular\n");
```

```
    show();

    if (null_line != -1) {
        return null_line;
    }

    DEBUG(1, " last \n");
    for (int c = 0; c < unit_matrix_size; ++c) {
        if (matrix[row_size - 1][c]  == 1) {
            for (int r = 0; r < row_size; ++r) {
                matrix[row_size - 1][r] += matrix[c][r];
                matrix[row_size - 1][r] %= 2;

                unit_matrix[row_size - 1][c] += unit_matrix[c][r];
                unit_matrix[row_size - 1][c] %= 2;
            }
        }
    }
    if(matrix[row_size-1][row_size-1] != 0 ) {
        ERROR("no resolv\n");
    } else{
        null_line = row_size -1;
    }
    DEBUG(3, " last \n");

    return null_line;

}

//file quadratic_sieve.cpp
int make_exp_array_condBsmooth(std::vector< std::vector<uint64_t> > &v_exp,
std::vector<int> &smooth_num, std::vector<long> Y, std::vector<long> &p_smooth,
double size_B, uint32_t M,
    std::vector<long> &solution_candidates_number, std::vector<uint64_t> &v_extra_exp)
{

        std::vector<long> V;
        V = Y;
        // add sign to exponent matrix
        #define NEGATIVE_SIGN    0
        #define FIRST_VALUE    1
        // v_exp[i].size()-1

        int print_flag = 1;
```

```
for (int y_number = 0; y_number < v_exp.size(); ++y_number)
{

   if(V[y_number] < 0 )
      v_exp[y_number][NEGATIVE_SIGN] = 1;

   for (   int smooth_iter = 0, exponent_num = FIRST_VALUE ;
         smooth_iter < p_smooth.size();
         smooth_iter++, exponent_num++)
   {
      long int tmp;
      do{
         tmp = V[y_number] % p_smooth[smooth_iter];
         DEBUG (4, "v = %10li\t",V[y_number]);
         DEBUG (4, "p_smooth = %li\t",p_smooth[smooth_iter]);
         DEBUG (4, "tmp = %li\n",tmp);
         if(tmp == 0){
            V[y_number] = V[y_number] / p_smooth[smooth_iter];
            v_exp[y_number][exponent_num] += 1;
         }
      } while (tmp == 0);

      if(V[y_number] == -1 || V[y_number] == 1){
         int null_flag = 1;
         // printf("V = %" PRIu64 "\t",V[i]);
         for (   int exp_num = 0;
                  exp_num < v_exp[y_number].size();
                  exp_num++ )
         {
            DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
            if ((v_exp[y_number][exp_num] % 2 )!= 0)
               null_flag = 0;
         }
         DEBUG (3, "%ld\n", Y[y_number]);
         // skip negative value !!!!
         if (null_flag && V[y_number] > 0) {
            solution_candidates_number.push_back(y_number);
         } else {
            smooth_num.push_back(y_number);
         }
         DEBUG (3, "\n");

         break;
      }
```

```
        }

        // DEBUG (0, "V ===== %li \n",V[y_number]);
        if(V[y_number] != -1 && V[y_number] != 1){
           int sign_flag = 0;
           if(V[y_number] < 0 ){
              sign_flag = 1;
              V[y_number] *= -1;
           }

           double res = sqrt(V[y_number]);
           if(res == trunc(res)) {
              v_extra_exp[y_number] = res;
              V[y_number] /= V[y_number];
              if(print_flag){
                 DEBUG (0, "found
========================================= %f  number %d\n",res,
y_number);
                 print_flag = 0;
              }
              DEBUG (2, "Y = %li\tV = %li\n",Y[y_number], V[y_number]);
              // DEBUG (0, "iter %d\n",);

              int null_flag = 1;
              // printf("V = %" PRIu64 "\t",V[i]);
              for (   int exp_num = 0;
                      exp_num < v_exp[y_number].size();
                      exp_num++ )
              {
                 DEBUG (3, "%ld\t", v_exp[y_number][exp_num]);
                 if ((v_exp[y_number][exp_num] % 2 )!= 0)
                    null_flag = 0;
              }
              DEBUG (3, "\n");

              if (null_flag && V[y_number] > 0) {
                 solution_candidates_number.push_back(y_number);
                 // DEBUG (2, "sol cand %d\n",solution_candidates_number.size() );
              } else {
                 smooth_num.push_back(y_number);
              }
              // smooth_num.push_back(y_number);
           }
```

```
            if(sign_flag)
               V[y_number] *= -1;
         }

      }

     if (smooth_num.size()  < size_B + 1)
      {
         //ERROR( "to small number of smooth numbbers\n");
         return 0;
      }
   return 1;
}

void construct_xy(std::vector<long> &X, std::vector<long> &Y, long sqrt_N, long long
N, long M)
{
      for (long  i = -M/2; i < M/2; i++)
      {
         X.push_back(sqrt_N + i);
         DEBUG (4, "X%lu =%lu\n",i, sqrt_N - i );
      }

      DEBUG (2,"\n");
      for (uint64_t i = 0; i < X.size(); ++i)
      {
         DEBUG (2, "X = %lu\t",X[i]);
         long long tmp = X[i]*X[i];
         if(X[i]*X[i] < N)
            Y.push_back(tmp - N);
         else
            Y.push_back(tmp % N);
         DEBUG (2, "Y = %li\t",Y[i]);
         DEBUG (2, "\n");
      }

}
void make_smooth_numbers(std::vector<long> &p_smooth, double size_B, uint64_t N)
{
   p_smooth.push_back(prime[2]);
   DEBUG(2, "%llu\n", prime[2]);

   for (uint64_t i = 3; (p_smooth.size() < size_B) && (i < prime_size); ++i)
   {
```

```
        uint64_t tmp = N;
        uint64_t N_mod = N % prime[i];
        tmp %= prime[i];
        for (int j = 1; j < (prime[i]-1)/2; ++j)
        {
           tmp = (tmp * N_mod) % prime[i];
        }
        tmp %= prime[i];

        if( tmp == 1)
        {
           p_smooth.push_back(prime[i]);
           DEBUG(2, "%llu\n", prime[i]);
        }
     }
}

int fill_matrix(bin_matrix_t &m1, std::vector<int> &smooth_num, std::vector<
std::vector<uint64_t> > &v_exp)
{
   int count = 0;

   if ((m1.row_size) == m1.filled)
      return count;
   DEBUG(4, "%s %d \n",__func__, __LINE__);
   for (int i = 0; i < smooth_num.size() && ((m1.row_size) != m1.filled); ++i) {
      if (smooth_num[i] != -1) {
         DEBUG(3, "%s %d  try to add %d \n",__func__, __LINE__, smooth_num[i]);
         m1.add_row(v_exp[ smooth_num[i]]);
         smooth_num[i] = -1;
         count++;
      }
   }
   return count;
}
```