Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Вычислительной математики и информационных технологий

Отчет

учебной практики (научно-исследовательская работа (получение первичных навыков научно-исследовательской работы))

Обучающийся Гусев Виталий Евгеньевич гр.09-335 \_\_\_\_\_\_\_

(ФИО студента) (Группа) (Подпись)

Научный руководитель:

канд. техн. наук, доцент Мубараков Б.Г. \_\_\_\_\_\_\_\_\_

(Подпись)

Руководитель практики от КФУ:

Доцент кафедры САИТ Андрианова А.А.

Оценка за практику \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Подпись)

Дата сдачи отчета «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2024 г.

Казань – 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3

1. Разработка методов дискретного логарифмирования 4

2. Вспомогательные математические функции 6

3. Алгоритм Шенкса 10

4. Алгоритм Полига-Хеллмана 12

5. Ро-метод Полларда 14

6. Алгоритм Адлемана 16

7. Алгоритм COS 18

ЗАКЛЮЧЕНИЕ 21

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 23

ПРИЛОЖЕНИЕ 1. СТАТЬЯ 25

Приложение 2. Алгоритм Полига-Хеллмана 27

Приложение 3. Ро-метод Полларда 32

Приложение 4. Алгоритм Шенкса 34

Приложение 5. Математические функции 37

Приложение 6. Алгоритм Адлемана 47

ВВЕДЕНИЕ

Учебная практика (научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)) проходила на кафедре системного анализа и информационных технологий Института вычислительной математики и информационных технологий КФУ с 09 февраля 2024 по 07 июня 2024 года.

Целью практики является разработка алгоритмов дискретного логарифмирования с экспоненциальной и субэкспоненциальной сложностью.

В процессе практики будут реализованы и протестированы алгоритмы: алгоритм Шенкса, алгоритм Полига-Хеллмана, ро-метод Полларда, алгоритм Адлемана, алгоритм COS, решето числового поля. Для тестирования данных алгоритмов будет реализован и использован генератор параметров Диффи-Хеллмана и возведение в степень по модулю.

1. Разработка методов дискретного логарифмирования

В процессе практики были изучены и реализованы методы дискретного логарифмирования на языке программирования C# на .NET8 в Windows Forms (рисунок 1). Также для тестирования данных алгоритмов был реализован генератор параметров Диффи-Хеллмана и возведение числа в степень по модулю [1].

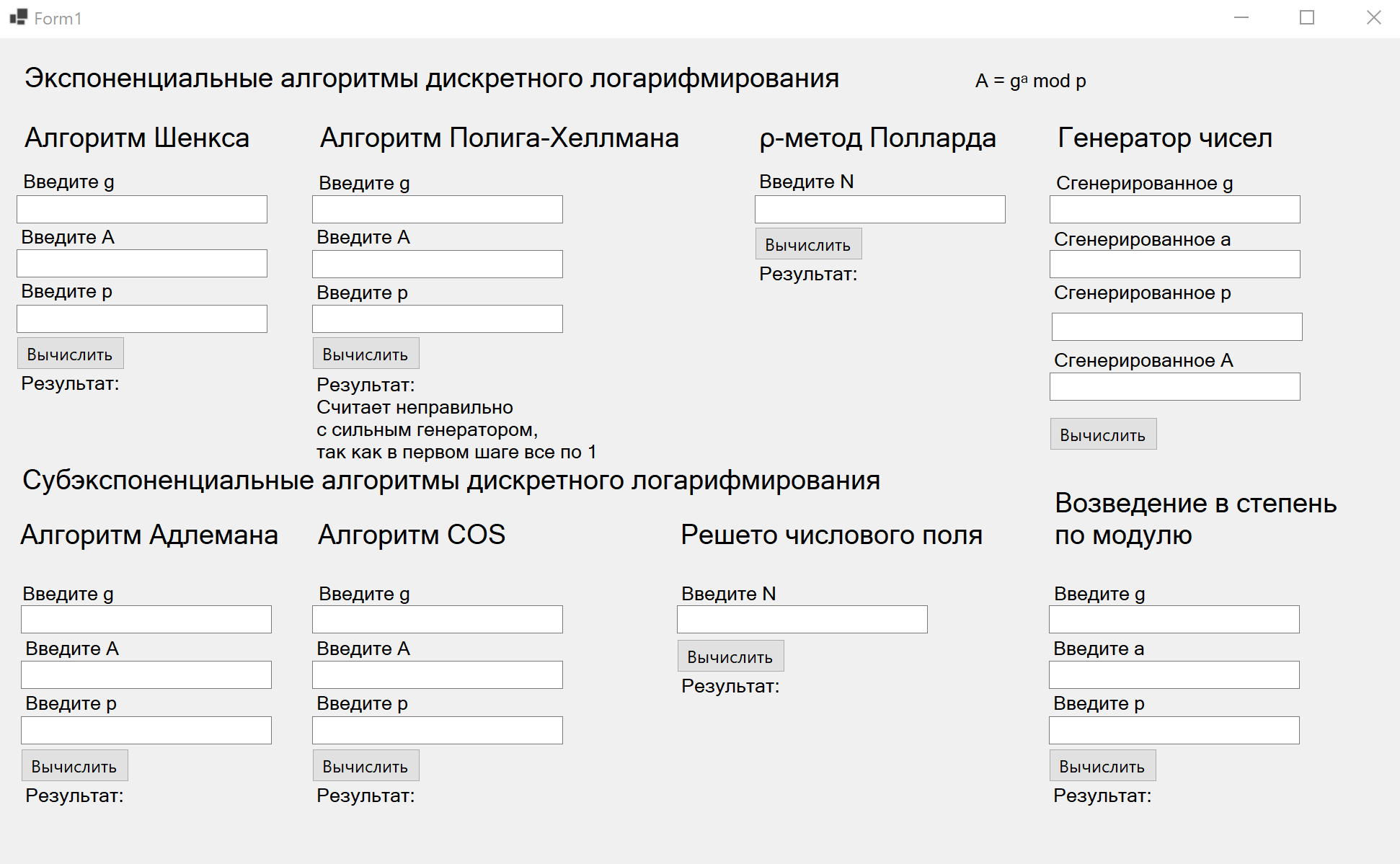


Рисунок 1 - Реализованная программа

Были реализованы экспоненциальные алгоритмы дискретного логарифмирования: алгоритм Шенкса [2], алгоритм Полига-Хеллмана [3], ро-метод Полларда [4], а также субэкспоненциальные алгоритмы дискретного логарифмирования: алгоритм Адлемана [5], алгоритм COS [6], решето числового поля [7]. Для разработки всех алгоритмов были реализованы вспомогательные математические функции.

Разработанная программа позволяет вносить в текстовые поля необходимые значения параметров возведения чисел в степень по модулю: g, a, p, A, либо целых чисел N для разложения на простые множители и выводить результат вычисления. Для корректности работы программы была реализована проверка на корректность ввода параметров для вычисления результатов алгоритмов. При помощи встроенного метода TryParse() в платформе для разработки программного обеспечения .NET идёт попытка конвертировать введённые значения в BigInteger. Если конвертация проходит успешно, то идёт проверка отрицательность конвертированных значений. Если проверка введённых значений проходит успешно, то идёт вычисление алгоритма (рисунок 2).

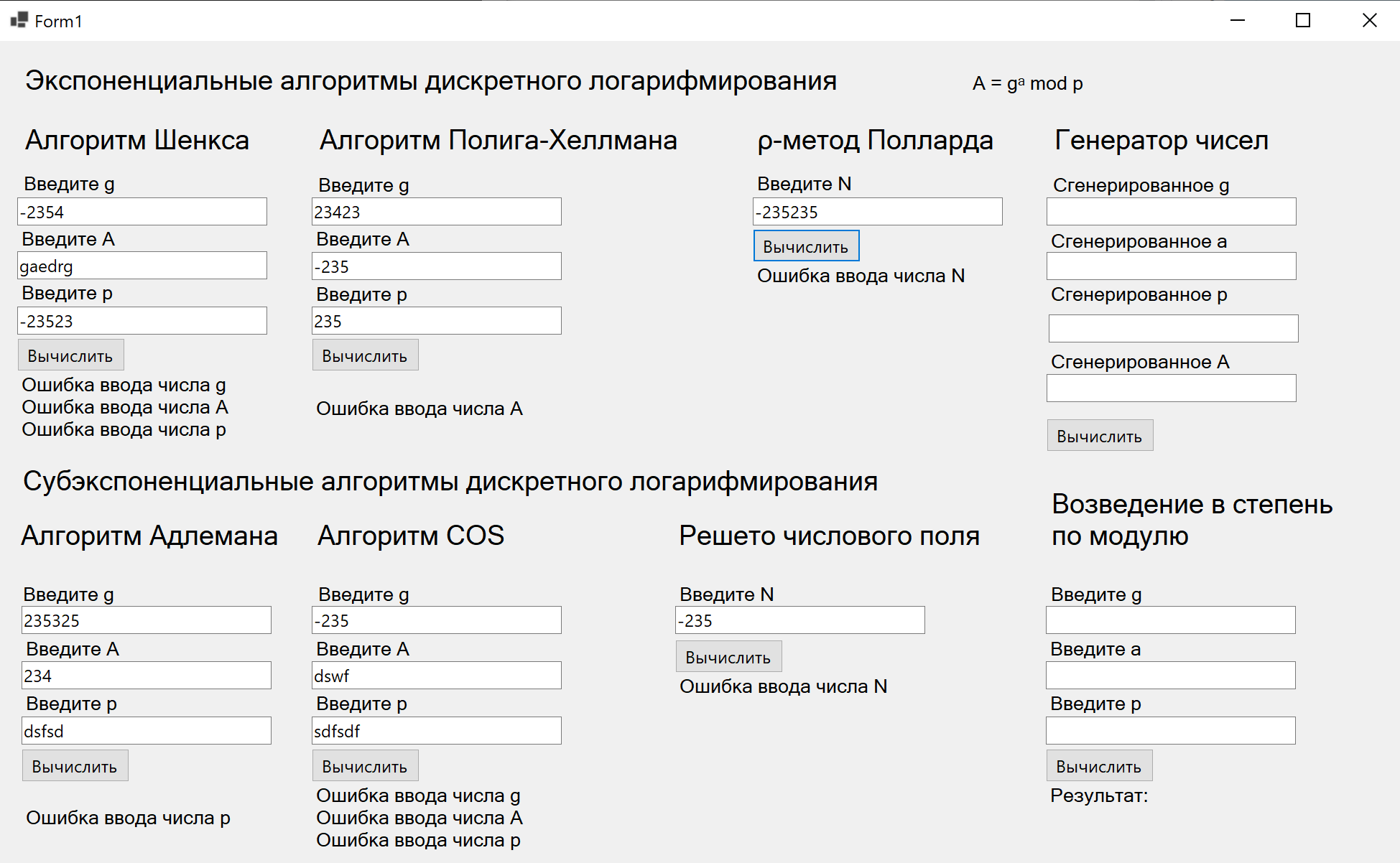


Рисунок 2 - Валидация введённых значений

2. Вспомогательные математические функции

Для реализации алгоритмов дискретного логарифмирования были реализованы вспомогательные математические функции.

Была реализована функция быстрого возведения в степень по модулю. Возведение в степень по модулю — это операция над натуральными числами возведения в степень, выполняемая по модулю. Находит применение в информатике, особенно, в области криптографии с открытым ключом. Возведение в степень по модулю — это вычисление остатка от деления натурального числа (основание), возведенного в степень (показатель степени), на натуральное число (модуль). Обозначается: .

Если , и неотрицательны и , то единственное решение существует, причем . Возведение в степень по модулю может быть выполнено и с отрицательным показателем степени . Для этого необходимо найти число , обратное числу по модулю . Это легко сделать с помощью алгоритма Евклида. Таким образом,

.

Возвести в степень по модулю довольно легко, даже при больших входных значениях. А вычисление дискретного логарифма, то есть нахождение показателя степени при заданных , и , намного сложнее. Такое одностороннее поведение функции делает её кандидатом для использования в криптографических алгоритмах.

Шаги алгоритма быстрого возведения в степень по модулю:

1) Показатель степени переводится в двоичный вид.

2) Создаётся список для чисел, первый элементом которого является .

3) В созданный список циклично добавляются остатки от деления квадратов последних элементов на .

4) Все элементы списка, индексы которых совпадают с позициями 1 в двоичном числе из первого пункта, перемножаются.

5) Вычисляется остаток от деления перемноженных чисел списка на .

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: . В результате работы алгоритма был корректно вычислен результат (рисунок 3).

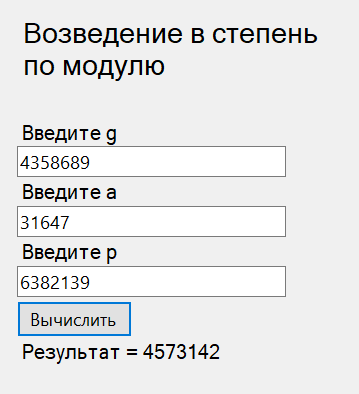


Рисунок 3 - Результат возведения в степень по модулю

Для проверки чисел на простоту был реализован тест Миллера-Рабина [8]. Данный тест является вероятностным полиномиальным тестом простоты. Тест Миллера-Рабина, наряду с тестом Ферма [9] и тестом Соловея-Штрассена [10], позволяет эффективно определить, является ли данное число составным. Однако, с его помощью нельзя строго доказать простоту числа. Тем не менее тест Миллера-Рабина часто используется в криптографии для получения больших случайных простых чисел.

Так как криптостойкость многих алгоритмов шифрования основывается на секретных ключах, для создания которых необходимы простые числа (например, так работает шифр RSA), то при создании таких ключей важно уметь достаточно быстро проверять большие числа на простоту. Вероятностные тесты простоты, такие как тест Миллера-Рабина и тест Соловея-Штрассена, показывают большую эффективность использования и простоту выражения по сравнению с детерминированными тестами. Алгоритм Миллера-Рабина позволяет выполнять проверку за малое время и давать при этом достаточно малую вероятность того, что число на самом деле является составным.

Как и тесты Ферма и Соловея-Штрассена, тест Миллера-Рабина опирается на проверку ряда равенств, которые выполняются для простых чисел. Если хотя бы одно такое равенство не выполняется, это доказывает, что число составное.

Для теста Миллера-Рабина используется следующее утверждение. Пусть – простое число и , где – нечётно. Тогда для любого из выполняется хотя бы одно из условий:

1) .

2) Существует целое число такое, что .

Если это утверждение (условие 1 или 2) выполняется для некоторых чисел и (не обязательно простого), то число называют свидетелем простоты числа по Миллеру, а само число — вероятно простым. (При случайно выбранном вероятность ошибочно принять составное число за простое составляет 25 %, но её можно уменьшить, выполнив проверки для других .)

В случае, когда выполняется контрапозиция доказанного утверждения, то есть если найдётся число такое, что:

и

,

то число не является простым. В этом случае число называют свидетелем того, что число составное.

У нечётных составных чисел существует, согласно теореме Рабина, не более свидетелей простоты, где — функция Эйлера, таким образом вероятность того, что случайно выбранное число окажется свидетелем простоты, меньше 1/4. Идея теста заключается в том, чтобы проверять для случайно выбранных чисел , являются ли они свидетелями простоты числа . Если найдётся свидетель того, что число составное, то число действительно является составным. Если было проверено чисел, и все они оказались свидетелями простоты, то число считается простым. Для такого алгоритма вероятность принять составное число за простое будет меньше .

Для проверки больших чисел принято выбирать числа случайными, так как распределение свидетелей простоты и свидетелей составного числа среди чисел заранее неизвестно.

Алгоритм Миллера-Рабина параметризуется количеством раундов . Рекомендуется брать порядка величины , где — проверяемое число.

Для данного находятся такие целое число и целое нечётное число , что . Выбирается случайное число . Если не является свидетелем простоты числа , то выдаётся ответ « — составное», и алгоритм завершается. Иначе, выбирается новое случайное число и процедура проверки повторяется. После нахождения свидетелей простоты, выдаётся ответ « — вероятно простое», и алгоритм завершается.

3. Алгоритм Шенкса

Был реализован экспоненциальный алгоритм «Шаг младенца — шаг великана» — в теории групп детерминированный алгоритм дискретного логарифмирования в мультипликативной группе кольца вычетов по модулю простого числа. Был предложен советским математиком Александром Гельфондом в 1962 году и Дэниелом Шенксом в 1972 году. Метод теоретически упрощает решение задачи дискретного логарифмирования, на вычислительной сложности которой построены многие криптосистемы с открытым ключом. Относится к методам встречи посередине. Это был один из первых методов, который показал, что задача вычисления дискретного логарифма может быть решена значительно быстрее, чем методом перебора. Идея алгоритма состоит в выборе оптимального соотношения времени и памяти, а именно в усовершенствованном поиске показателя степени.

Пусть задано сравнение , необходимо найти натуральное число , удовлетворяющее данному сравнению.

Алгоритм реализован следующим образом:

1) Сначала берутся два целых числа и , такие, что . Как правило .

2) Вычисляются два ряда чисел:

,

.

Все вычисления проводятся по модулю .

3) Идёт поиск таких и , для которых выполняется равенство . То есть ищется во втором ряду такое число, которое присутствует и в первом ряду. Запоминаются показатели степени и , при которых данные числа получались.

4) В результате работы алгоритма неизвестная степень вычисляется по формуле .

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: . В результате работы алгоритма был корректно вычислен результат (рисунок 4).

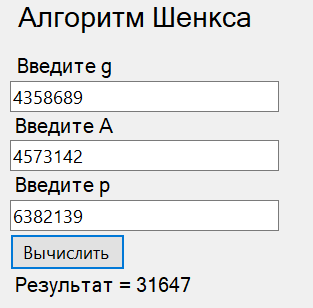


Рисунок 4 - Алгоритм Шенкса

4. Алгоритм Полига-Хеллмана

Был реализован детерминированный алгоритм дискретного логарифмирования с экспоненциальной сложностью Полига-Хеллмана в кольце вычетов по модулю простого числа. Одной из особенностей алгоритма является то, что для простых чисел специального вида можно находить дискретный логарифм за полиномиальное время. Данный алгоритм был придуман американским математиком Роландом Сильвером, но впервые был опубликован другими двумя американскими математиками Стивеном Полигом и Мартином Хеллманом в 1978 году в статье «An improved algorithm for computing logarithms over GF(p) and its cryptographic significance», которые независимо от Роланда Сильвера разработали данный алгоритм.

Пусть задано сравнение , необходимо найти натуральное число , удовлетворяющее данному сравнению.

Шаги выполнения алгоритма:

1) Идёт разложение числа на простые множители.

2) Составляется таблица значений ,

где .

3) Вычисляется .

Для от 1 до :

Пусть ,

где .

Тогда верно сравнение:

.

С помощью таблицы, составленной на шаге 1, находится .

Для от 0 до рассматривается сравнение

.

Решение находится по таблице

Конец цикла по .

Конец цикла по .

4) Найдя для всех , происходит поиск

по китайской теореме об остатках.

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: . В результате работы алгоритма был корректно вычислен результат (рисунок 5).

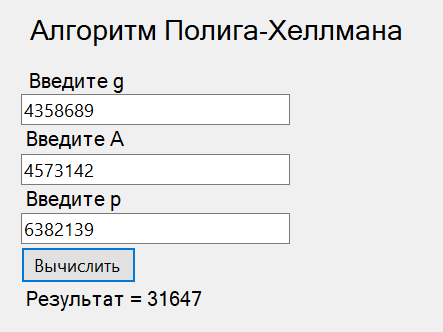


Рисунок 5 - Алгоритм Полига-Хеллмана

5. Ро-метод Полларда

Был реализован экспоненциальный алгоритм дискретного логарифмирования ро-метод Полларда для факторизации (разложения на множители) целых чисел. Данный алгоритм основывается на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Ро-метод Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера , что может быть проиллюстрировано, расположением чисел в виде греческой буквы ρ, что послужило названием семейству алгоритмов.

Шаги выполнения алгоритма:

1) Вычисляются тройки чисел

.

Причём каждая такая тройка получается из предыдущей.

2) Каждый раз, когда число кратно , вычисляется наибольший общий делитель любым известным способом.

3) Если , то частичное разложение числа найдено, причём . Найденный делитель может быть составным, поэтому его также необходимо факторизовать. Если число составное, то продолжаем алгоритм с модулем .

4) Вычисления повторяются раз. Если при этом число не было до конца факторизовано, выбирается, например, другое начальное число .

Для проверки работы разработанного алгоритма был сгенерирован параметр: . В результате работы алгоритма был корректно вычислен результат (рисунок 6).

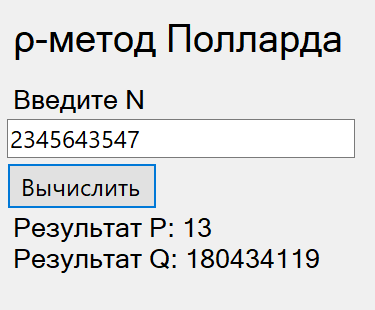


Рисунок 6 - Ро-метод Полларда

6. Алгоритм Адлемана

Был реализован алгоритм Адлемана, который является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа. Алгоритм был предложен Леонардом Максом Адлеманом в 1979 году. Леонард Макс Адлеман — американский учёный-теоретик в области компьютерных наук, профессор компьютерных наук и молекулярной биологии в Университете Южной Калифорнии. Он известен как соавтор системы шифрования RSA и ДНК-вычислений. RSA широко используется в приложениях компьютерной безопасности, включая протокол HTTPS.

Пусть задано сравнение , необходимо найти натуральное число , удовлетворяющее данному сравнению.

Описание алгоритма:

1) Сформировывается факторная база, состоящая из всех простых чисел :

.

2) С помощью перебора идёт поиск натуральных чисел таких, что

,

то есть раскладывается по факторной базе. Отсюда следует, что .

3) Набрав достаточно много соотношений из 2 шага, решается получившаяся система линейных уравнений относительно неизвестных дискретных логарифмов элементов факторной базы .

4) С помощью некоторого перебора ищется одно значение , для которого , где – простые числа «средней» величины, то есть , где – также некоторая субэкспоненциальная граница, .

5) С помощью вычислений, аналогичных этапам 2 и 3 ищутся дискретные логарифмы .

6) Определяется искомый дискретный логарифм:

.

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: . В результате работы алгоритма был корректно вычислен результат (рисунок 7).

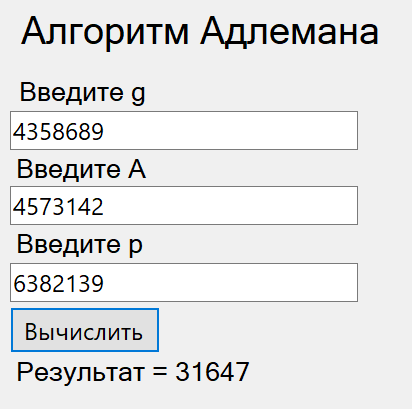


Рисунок 7 - Алгоритм Адлемана

7. Алгоритм COS

Был реализован алгоритм COS (Копперсмит, Одлыжко, Шреппель), который является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа.

Пусть задано сравнение , необходимо найти натуральное число , удовлетворяющее данному сравнению.

Описание алгоритма:

1) Задаётся . Сформировывается множество , где и – простые величины, .

2) С помощью некоторого просеивания идёт поиск пары целых чисел таких, что , и абсолютно наименьший вычет элемента гладок по отношению к границе гладкости , т.е.

.

При этом, поскольку , то

, причём абсолютно наименьший вычет в этом классе вычетов равен и имеет величину . Поэтому вероятность его гладкости выше, чем для произвольных чисел на отрезке . Логарифмируя по основанию , получается соотношение

*.*

Это однородное уравнение относительно неизвестных величин . Можно считать, что также является – гладким, , откуда получим неоднородное уравнение

*.*

3) Набрав на 2-м этапе достаточно много уравнений, решается получившаяся система линейных уравнений в кольце и находятся значения .

4) Для нахождения конкретного логарифма мы введём новую границу гладкости . Случайным перебором находим одно значение такое, что

.

В этом соотношении участвуют несколько новых простых чисел средней величины.

5) С помощью методов, аналогичных 2 и 3 этапам, мы находим логарифмы нескольких простых чисел средней величины, возникших на 4 этапе.

6) Находим ответ

.

Конец алгоритма.

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: . В результате работы алгоритма был корректно вычислен результат .

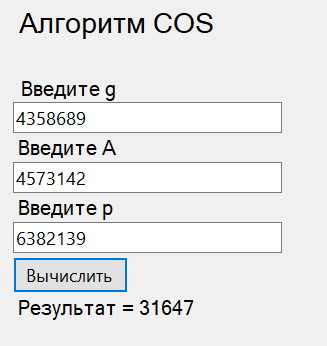


Рисунок 8 - Алгоритм COS

ЗАКЛЮЧЕНИЕ

В результате практики были изучены и реализованы алгоритмы дискретного логарифмирования с экспоненциальной и субэкспоненциальной сложностью. Также была протестирована корректность работы реализованных алгоритмов. Понимание алгоритмов позволит придумать и изучить модификацию для увеличения скорости и эффективности работы программы.

За время практики были реализованы следующие компетенции:

| Шифр компетенции | Расшифровка приобретаемой компетенции | Расшифровка освоения компетенции |
| --- | --- | --- |
| УК-1 | Способен осуществлять критический анализ проблемных ситуаций на основе системного подхода, вырабатывать стратегию действий | Получен навык осуществлять критический анализ для корректной работы методов дискретного логарифмирования с большими числами на основе системного подхода |
| УК-4 | Способен применять современные коммуникативные технологии, в том числе на иностранном(ых) языке(ах), для академического и профессионального взаимодействия | Получена способность применять современные коммуникативные технологии на иностранном языке для изучения методов дискретного логарифмирования |
| УК-6 | Способен определять и реализовывать приоритеты собственной деятельности и способы ее совершенствования на основе самооценки | Приобретена способность реализовывать приоритеты собственной деятельности для разработки методов дискретного логарифмирования на основе самооценки |
| ОПК-1 | Способен находить, формулировать и решать актуальные проблемы прикладной математики, фундаментальной информатики и информационных технологий | Приобретён навык нахождения и формулирования актуальных проблем в области информационной безопасности и фундаментальной информатике |
| ОПК-2 | Способен применять компьютерные/суперкомпьютерные методы, современное программное обеспечение (в том числе отечественного производства) для решения задач профессиональной деятельности | Получен навык применения компьютерных методов, современного программного обеспечения для решения задач профессиональной деятельности в области информационной безопасности |
| ОПК-3 | Способен проводить анализ математических моделей, создавать инновационные методы решения прикладных задач профессиональной деятельности в области информатики и математического моделирования | Получена способность проводить анализ математических алгоритмов для решения задач дискретного логарифмирования |

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Теоретический минимум и алгоритмы цифровой подписи / Молдовян Н. А. – Текст: непосредственный // Книжный Дом «ЛИБРОКОМ», 2010. — С. 304.

2. The infrastructure of a real quadratic field and its applications. Proceedings of the Number Theory Conference. / D. Shanks. – Текст: непосредственный // University of Colorado, Boulder, 1972. — С. 217-224.

3. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance (англ.) / S. C. Pohlig and M. E. Hellman. - Текст: непосредственный // IEEE Transactions on Information Theory. — 1978. — Vol. 1, no. 24. — С. 106-110.

4. Theorems on factorization and primality testing / Pollard J.M. - Текст: непосредственный // Mathematical Proceedings of the Cambridge Philosophical Society. — 1974. — Т. 76, вып. 03. — С. 521–528.

5. A subexponential algorithm for discrete logarithms over all finite fields / Adleman L. M., Demarrais J. - Текст: непосредственный // Mathematics of computation. — 1993.

6. Теоретико-числовые алгоритмы в криптографии. / Василенко О.Н. - Текст: непосредственный // N— М.: МЦНМО, 2003. — C. 328.

7. Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — C. 190.

8. Riemann's Hypothesis and Tests for Primality. / Miller G. - Текст: непосредственный // Proceedings of seventh annual ACM symposium on Theory of computing — New York City: ACM, 1975. — С. 234—239.

9. Теоретико-числовые алгоритмы в криптографии. / Василенко О.Н. - Текст: непосредственный // N— М.: МЦНМО, 2003. — C. 57.

10. Теоретико-числовые алгоритмы в криптографии. / Василенко О.Н. - Текст: непосредственный // N— М.: МЦНМО, 2003. — C. 37.

ПРИЛОЖЕНИЕ 1

**МЕТОДЫ ДИСКРЕТНОГО ЛОГАРИФМИРОВАНИЯ**

***Гусев В.Е.***

*Научный руководитель - канд. техн. наук Мубараков Б.Г.*

*Казанский (Приволжский) федеральный университет, Институт Вычислительной математики и информационных технологий*

*viegusev@stud.kpfu.ru*

В процессе практики были изучены и реализованы методы дискретного логарифмирования на языке программирования C# на .NET8 в Windows Forms. Также для тестирования данных алгоритмов был реализован генератор параметров Диффи-Хеллмана и возведение числа в степень по модулю.

Были реализованы экспоненциальные алгоритмы дискретного логарифмирования: алгоритм Шенкса, алгоритм Полига-Хеллмана, ро-метод Полларда, а также субэкспоненциальные алгоритмы дискретного логарифмирования: алгоритм Адлемана, алгоритм COS, решето числового поля. Для разработки всех алгоритмов были реализованы вспомогательные математические функции.

Была реализована функция быстрого возведения в степень по модулю. Возведение в степень по модулю — это операция над натуральными числами возведения в степень, выполняемая по модулю. Находит применение в информатике, особенно, в области криптографии с открытым ключом. Возведение в степень по модулю — это вычисление остатка от деления натурального числа (основание), возведенного в степень (показатель степени), на натуральное число (модуль). Обозначается: .

Для проверки чисел на простоту был реализован тест Миллера-Рабина. Данный тест является вероятностным полиномиальным тестом простоты. Тест Миллера-Рабина, наряду с тестом Ферма и тестом Соловея-Штрассена, позволяет эффективно определить, является ли данное число составным. Однако, с его помощью нельзя строго доказать простоту числа. Тем не менее тест Миллера-Рабина часто используется в криптографии для получения больших случайных простых чисел.

Был реализован экспоненциальный алгоритм «Шаг младенца — шаг великана» — в теории групп детерминированный алгоритм дискретного логарифмирования в мультипликативной группе кольца вычетов по модулю простого числа. Был предложен советским математиком Александром Гельфондом в 1962 году и Дэниелом Шенксом в 1972 году. Метод теоретически упрощает решение задачи дискретного логарифмирования, на вычислительной сложности которой построены многие криптосистемы с открытым ключом. Относится к методам встречи посередине. Это был один из первых методов, который показал, что задача вычисления дискретного логарифма может быть решена значительно быстрее, чем методом перебора. Идея алгоритма состоит в выборе оптимального соотношения времени и памяти, а именно в усовершенствованном поиске показателя степени.

Был реализован детерминированный алгоритм дискретного логарифмирования с экспоненциальной сложностью Полига-Хеллмана в кольце вычетов по модулю простого числа. Одной из особенностей алгоритма является то, что для простых чисел специального вида можно находить дискретный логарифм за полиномиальное время. Данный алгоритм был придуман американским математиком Роландом Сильвером, но впервые был опубликован другими двумя американскими математиками Стивеном Полигом и Мартином Хеллманом в 1978 году в статье «An improved algorithm for computing logarithms over GF(p) and its cryptographic significance», которые независимо от Роланда Сильвера разработали данный алгоритм.

Был реализован экспоненциальный алгоритм дискретного логарифмирования ро-метод Полларда для факторизации (разложения на множители) целых чисел. Данный алгоритм основывается на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Ро-метод Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера , что может быть проиллюстрировано, расположением чисел в виде греческой буквы ρ, что послужило названием семейству алгоритмов.

Был реализован алгоритм Адлемана, который является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа. Алгоритм был предложен Леонардом Максом Адлеманом в 1979 году. Леонард Макс Адлеман — американский учёный-теоретик в области компьютерных наук, профессор компьютерных наук и молекулярной биологии в Университете Южной Калифорнии. Он известен как соавтор системы шифрования RSA и ДНК-вычислений. RSA широко используется в приложениях компьютерной безопасности, включая протокол HTTPS.

Был реализован алгоритм COS (Копперсмит, Одлыжко, Шреппель), который является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа.

Приложение 2. Алгоритм Полига-Хеллмана

using DiscreteLogarithm.MathFunctionsForCalculation;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Runtime.Serialization.Formatters;

using System.Text;

using System.Threading.Tasks;

using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ExponentialAlgorithms

{

public class ListGroupedValues

{

public ListGroupedValues(BigInteger Key, int degree\_number, BigInteger key\_degree)

{

this.Key = Key;

this.degree\_number = degree\_number;

this.key\_degree = key\_degree;

}

public BigInteger Key { get; set; }

public int degree\_number { get; set; }

public BigInteger key\_degree { get; set; }

}

public class PoligHellman

{

MathFunctions mathFunctions;

public PoligHellman()

{

mathFunctions = new MathFunctions();

}

public void CheckingTheInputValues(

string input\_g,

string input\_A,

string input\_p,

Label inputLabel,

ref bool theValuesAreCorrect,

out BigInteger g,

out BigInteger A,

out BigInteger p)

{

inputLabel.Text = "";

if (!BigInteger.TryParse(input\_g, out g) || g <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text = "Ошибка ввода числа g";

};

if (!BigInteger.TryParse(input\_A, out A) || A <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа A";

};

if (!BigInteger.TryParse(input\_p, out p) || p <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа p";

};

}

public void CalculatePoligHellman(BigInteger g, BigInteger A, BigInteger p, Label inputLabel)

{

BigInteger fi\_p = p - 1;

List<BigInteger> p\_dividers = mathFunctions.Factorization(fi\_p);

List<ListGroupedValues> fi\_p\_dividers\_grouped = p\_dividers.GroupBy(x => x).Select(group => new ListGroupedValues(group.Key, group.Count(), BigInteger.Pow(group.Key, group.Count()))).ToList();

List<List<BigInteger>> step1\_result = Step1(fi\_p\_dividers\_grouped, g, fi\_p, p);

List<List<BigInteger>> step2\_result = Step2(fi\_p\_dividers\_grouped, step1\_result, g, A, p);

BigInteger step3\_result = Step3(fi\_p\_dividers\_grouped, step2\_result);

inputLabel.Text = string.Format("Результат = {0}", step3\_result);

}

private List<List<BigInteger>> Step1(List<ListGroupedValues> fi\_p\_dividers\_grouped, BigInteger g, BigInteger fi\_p, BigInteger p)

{

List<List<BigInteger>> step1\_result = new List<List<BigInteger>>();

for (int i = 0; i < fi\_p\_dividers\_grouped.Count; i++)

{

List<BigInteger> step1\_result\_i = new List<BigInteger>();

for (int j = 0; j < fi\_p\_dividers\_grouped[i].Key; j++)

{

step1\_result\_i.Add(mathFunctions.ExponentiationModulo(g, j \* fi\_p / fi\_p\_dividers\_grouped[i].Key, p));

}

step1\_result.Add(step1\_result\_i);

}

return step1\_result;

}

private List<List<BigInteger>> Step2(List<ListGroupedValues> fi\_p\_dividers\_grouped, List<List<BigInteger>> step1\_result, BigInteger g, BigInteger A, BigInteger p)

{

BigInteger Agmodp;

BigInteger p\_1\_q\_degree;

List<List<BigInteger>> x\_list = new List<List<BigInteger>>();

for (int i = 0; i < fi\_p\_dividers\_grouped.Count; i++)

{

List<BigInteger> x\_list\_i = new List<BigInteger>() { 0 };

for (int j = 0; j < fi\_p\_dividers\_grouped[i].degree\_number; j++)

{

p\_1\_q\_degree = (p - 1) / BigInteger.Pow(fi\_p\_dividers\_grouped[i].Key, j + 1);

Agmodp = mathFunctions.ExponentiationModulo(A / BigInteger.Pow(g, CalculateDegreeStep2(fi\_p\_dividers\_grouped[i].Key, x\_list\_i)), p\_1\_q\_degree, p);

Find\_x\_j\_Step2(Agmodp, step1\_result[i], x\_list\_i);

}

x\_list\_i.RemoveAt(0);

x\_list.Add(x\_list\_i);

}

return x\_list;

}

private int CalculateDegreeStep2(BigInteger q\_i, List<BigInteger> x\_list\_i)

{

BigInteger result = 0;

for (int j = 1; j < x\_list\_i.Count; j++)

{

result += x\_list\_i[j] \* BigInteger.Pow(q\_i, j - 1);

}

return (int)result;

}

private void Find\_x\_j\_Step2(BigInteger Agmodp, List<BigInteger> step1\_result\_i, List<BigInteger> x\_list\_i)

{

for (int i = 0; i < step1\_result\_i.Count; i++)

{

if (Agmodp == step1\_result\_i[i])

{

x\_list\_i.Add(i);

break;

}

}

}

private BigInteger Step3(List<ListGroupedValues> fi\_p\_dividers\_grouped, List<List<BigInteger>> step2\_result)

{

List<BigInteger> x\_q = new List<BigInteger>();

for (int i = 0; i < fi\_p\_dividers\_grouped.Count; i++)

{

BigInteger x\_q\_i = 0;

for (int x\_i = 0; x\_i < step2\_result[i].Count; x\_i++)

{

x\_q\_i += step2\_result[i][x\_i] \* BigInteger.Pow(fi\_p\_dividers\_grouped[i].Key, x\_i);

}

x\_q\_i %= fi\_p\_dividers\_grouped[i].key\_degree;

x\_q.Add(x\_q\_i);

}

bool x\_result\_true = true;

BigInteger x\_result = 0;

BigInteger max\_key\_degree = 0;

for (int i = 0; i < fi\_p\_dividers\_grouped.Count; i++)

{

if (max\_key\_degree < fi\_p\_dividers\_grouped[i].key\_degree)

{

x\_result = x\_q[i];

max\_key\_degree = fi\_p\_dividers\_grouped[i].key\_degree;

}

}

while (true)

{

for (int i = 0; i < fi\_p\_dividers\_grouped.Count; i++)

{

if (x\_result % fi\_p\_dividers\_grouped[i].key\_degree != x\_q[i])

{

x\_result\_true = false;

break;

}

}

if (x\_result\_true == false)

{

x\_result += max\_key\_degree;

x\_result\_true = true;

continue;

}

return x\_result;

}

}

}

}

Приложение 3. Ро-метод Полларда

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Numerics;

using System.Text;

using System.Threading.Tasks;

using static System.Windows.Forms.VisualStyles.VisualStyleElement;

using System.Xml.Linq;

using Label = System.Windows.Forms.Label;

using System.Security.Cryptography;

namespace DiscreteLogarithm.ExponentialAlgorithms

{

public class RoPollard

{

public void CheckingTheInputValues(

string input\_N,

Label inputLabel,

ref bool theValuesAreCorrect,

out BigInteger a)

{

inputLabel.Text = "";

if (!BigInteger.TryParse(input\_N, out a) || a < 5)

{

theValuesAreCorrect = false;

inputLabel.Text = "Ошибка ввода числа N";

};

}

public BigInteger ro\_Pollard(BigInteger n)

{

Random random = new Random();

byte[] data = new byte[n.ToByteArray().Length];

random.NextBytes(data);

BigInteger x = new BigInteger(data);

x = x < 0 ? -x - 2 : x - 2;

BigInteger y = 1;

BigInteger i = 0;

BigInteger stage = 2;

while (BigInteger.GreatestCommonDivisor(n, BigInteger.Abs(x - y)) == 1)

{

if (i == stage)

{

y = x;

stage = stage \* 2;

}

x = (x \* x - 1) % n;

i = i + 1;

}

return BigInteger.GreatestCommonDivisor(n, BigInteger.Abs(x - y));

}

public void CalculateRoPollard(BigInteger N, Label inputLabel)

{

BigInteger p = ro\_Pollard(N);

BigInteger q = N / p;

inputLabel.Text = string.Format("Результат P: {0} \nРезультат Q: {1}", p, q);

}

}

}

Приложение 4. Алгоритм Шенкса

using DiscreteLogarithm.MathFunctionsForCalculation;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Reflection.Emit;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ExponentialAlgorithms

{

// Алгоритм https://ilovecalc.com/calcs/maths/baby-step-giant-step/1382/

public class Shenks

{

MathFunctions mathFunctions;

public Shenks()

{

mathFunctions = new MathFunctions();

}

public void CheckingTheInputValues(

string input\_g,

string input\_A,

string input\_p,

Label inputLabel,

ref bool theValuesAreCorrect,

out BigInteger g,

out BigInteger A,

out BigInteger p)

{

inputLabel.Text = "";

if (!BigInteger.TryParse(input\_g, out g) || g <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text = "Ошибка ввода числа g";

};

if (!BigInteger.TryParse(input\_A, out A) || A <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа A";

};

if (!BigInteger.TryParse(input\_p, out p) || p <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа p";

};

}

public void CalculateShenks(BigInteger g, BigInteger A, BigInteger p, Label inputLabel)

{

BigInteger m, k;

Step1(p, out m, out k);

List<BigInteger> g\_km\_degree = new List<BigInteger>();

List<BigInteger> Ag\_m\_degree = new List<BigInteger>();

Step2(g\_km\_degree, Ag\_m\_degree, g, A, p, m, k);

int i, j;

Step3(g\_km\_degree, Ag\_m\_degree, out i, out j);

BigInteger result = BigInteger.Multiply(i, m) - j;

inputLabel.Text = "Результат = " + result.ToString();

}

private BigInteger Sqrt(BigInteger number)

{

BigInteger n = 0, p = 0;

if (number == BigInteger.Zero)

{

return BigInteger.Zero;

}

var high = number >> 1;

var low = BigInteger.Zero;

while (high > low + 1)

{

n = (high + low) >> 1;

p = n \* n;

if (number < p)

{

high = n;

}

else

{

if (number > p)

{

low = n;

}

else

{

break;

}

}

}

return number == p ? n : low;

}

private void Step1(BigInteger p, out BigInteger m, out BigInteger k)

{

m = k = Sqrt(p) + 1;

}

private void Step2(List<BigInteger> g\_km\_degree, List<BigInteger> Ag\_m\_degree, BigInteger g, BigInteger A, BigInteger p, BigInteger m, BigInteger k)

{

for (BigInteger k\_i = 1; k\_i <= k; k\_i++)

{

g\_km\_degree.Add(mathFunctions.ExponentiationModulo(g, k\_i \* m, p));

}

for (int m\_i = 0; m\_i <= m - 1; m\_i++)

{

Ag\_m\_degree.Add(mathFunctions.ExponentiationModulo(A \* BigInteger.Pow(g, m\_i), 1, p));

}

}

private void Step3(List<BigInteger> g\_km\_degree, List<BigInteger> Ag\_m\_degree, out int ind\_i, out int ind\_j)

{

for(int i = 0; i < g\_km\_degree.Count; i++)

{

for (int j = 0; j < Ag\_m\_degree.Count; j++)

{

if (g\_km\_degree[i] == Ag\_m\_degree[j])

{

ind\_i = i + 1;

ind\_j = j;

return;

}

}

}

ind\_i = 0;

ind\_j = 0;

}

}

}

Приложение 5. Математические функции

using DiscreteLogarithm.ExponentialAlgorithms;

using System.Numerics;

using System.Security.Cryptography;

using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.MathFunctionsForCalculation

{

public class MathFunctions

{

RoPollard roPollard;

public MathFunctions()

{

roPollard = new RoPollard();

}

public void CheckingTheInputValues(

string input\_g,

string input\_a,

string input\_p,

Label inputLabel,

ref bool theValuesAreCorrect,

out BigInteger g,

out BigInteger a,

out BigInteger p)

{

inputLabel.Text = "";

if (!BigInteger.TryParse(input\_g, out g))

{

theValuesAreCorrect = false;

inputLabel.Text = "Ошибка ввода числа g";

};

if (!BigInteger.TryParse(input\_a, out a))

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа a";

};

if (!BigInteger.TryParse(input\_p, out p))

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа p";

};

}

private string ConvertToBinaty(BigInteger number)

{

string binary\_letter = "";

if (number == 0)

{

return "0";

}

while (number >= 1)

{

binary\_letter += Convert.ToString(number % 2);

number /= 2;

}

return binary\_letter;

}

public BigInteger ExponentiationModulo(BigInteger g, BigInteger a, BigInteger n)

{

//перевод alpha в двоичный вид

string binary\_a = ConvertToBinaty(a);

List<BigInteger> number = new List<BigInteger>() { g };

for (int i = 1; i < binary\_a.Length; i++)

{

number.Add((number[i - 1] \* number[i - 1]) % n);

}

BigInteger result = 1;

for (int i = 0; i < binary\_a.Length; i++)

{

if (binary\_a[i] == '1')

{

result \*= number[i];

}

}

result %= n;

return result;

}

public void ExponentiationModuloWin(BigInteger g, BigInteger a, BigInteger n, Label inputLabel)

{

BigInteger result = ExponentiationModulo(g, a, n);

inputLabel.Text = string.Format("Результат = {0}", result);

}

public BigInteger Generate\_a()

{

// число a 16 бит

int byteCount = 16 / 8;

BigInteger a;

while (true)

{

a = new BigInteger(RandomNumberGenerator.GetBytes(byteCount));

if (a > 1)

{

return a;

}

}

}

public BigInteger Generate\_p()

{

// число p 64 бит

int byteCount = 24 / 8;

BigInteger p;

while (true)

{

p = new BigInteger(RandomNumberGenerator.GetBytes(byteCount));

if (p < 3)

{

continue;

}

if (TestMillerRabin(p) == "Вероятно простое")

{

return p;

}

}

}

public List<BigInteger> Factorization(BigInteger fi\_p)

{

List<BigInteger> p\_dividers = new List<BigInteger>();

BigInteger p\_factorized\_new;

BigInteger q\_factorized\_new;

BigInteger fi\_p\_initial = fi\_p;

BigInteger fi\_p\_help = fi\_p;

BigInteger check\_factorization;

List<BigInteger> dividers = new List<BigInteger>() { 2, 3, 5, 7, 11, 13, 17 };

bool сontinue\_cycle;

while (true)

{

while (true)

{

сontinue\_cycle = true;

if (TestMillerRabin(fi\_p\_help) == "Вероятно простое")

{

p\_dividers.Add(fi\_p\_help);

break;

}

for (int i = 0; i < dividers.Count; i++)

{

if (fi\_p\_help % dividers[i] == 0)

{

fi\_p\_help /= dividers[i];

fi\_p = fi\_p\_help;

p\_dividers.Add(dividers[i]);

сontinue\_cycle = false;

break;

}

}

if (сontinue\_cycle == false)

{

continue;

}

p\_factorized\_new = roPollard.ro\_Pollard(fi\_p\_help);

q\_factorized\_new = fi\_p\_help / p\_factorized\_new;

string p\_factorized\_new\_miller\_rabin = p\_factorized\_new > 1 ? TestMillerRabin(p\_factorized\_new) : "Меньше 2";

string q\_factorized\_new\_miller\_rabin = q\_factorized\_new > 1 ? TestMillerRabin(q\_factorized\_new) : "Меньше 2";

if (p\_factorized\_new != 1 && q\_factorized\_new != 1 && p\_factorized\_new\_miller\_rabin == "Вероятно простое")

{

p\_dividers.Add(p\_factorized\_new);

fi\_p /= p\_factorized\_new;

fi\_p\_help = fi\_p;

}

else if (p\_factorized\_new != 1 && q\_factorized\_new != 1 && q\_factorized\_new\_miller\_rabin == "Вероятно простое")

{

p\_dividers.Add(q\_factorized\_new);

fi\_p /= q\_factorized\_new;

fi\_p\_help = fi\_p;

}

else if (p\_factorized\_new > q\_factorized\_new)

{

fi\_p\_help /= p\_factorized\_new;

}

else

{

fi\_p\_help /= q\_factorized\_new;

}

}

check\_factorization = 1;

for (int i = 0; i < p\_dividers.Count; i++)

{

check\_factorization \*= p\_dividers[i];

}

if (check\_factorization != fi\_p\_initial)

{

p\_dividers.Clear();

fi\_p = fi\_p\_initial;

fi\_p\_help = fi\_p\_initial;

continue;

}

else

{

return p\_dividers;

}

}

}

public void FindAllDivisors(List<BigInteger> p\_dividers, BigInteger p\_factorized)

{

if (TestMillerRabin(p\_factorized) == "Вероятно простое")

{

return;

}

BigInteger p\_factorized\_new;

BigInteger q\_factorized\_new;

while (true)

{

p\_factorized\_new = roPollard.ro\_Pollard(p\_factorized);

q\_factorized\_new = p\_factorized / p\_factorized\_new;

if (p\_factorized\_new != 1 && q\_factorized\_new != 1)

{

break;

}

}

p\_dividers.Add(p\_factorized\_new);

p\_dividers.Add(q\_factorized\_new);

if (p\_factorized\_new == 1 || q\_factorized\_new == 1)

{

return;

}

FindAllDivisors(p\_dividers, p\_factorized\_new);

FindAllDivisors(p\_dividers, q\_factorized\_new);

}

public List<BigInteger> Generate\_p\_g()

{

BigInteger p;

BigInteger fi\_p;

List<BigInteger> p\_dividers;

// число g 64 бит

int byteCount = 24 / 8;

BigInteger g;

bool true\_p;

while (true)

{

p = Generate\_p();

fi\_p = p - 1;

p\_dividers = Factorization(fi\_p);

p\_dividers = p\_dividers.Distinct().ToList();

for (int step = 0; step < 1000; step++)

{

true\_p = true;

do

{

g = new BigInteger(RandomNumberGenerator.GetBytes(byteCount));

}

while (g < 2 || g >= p - 2);

if (BigInteger.GreatestCommonDivisor(g, p) != 1)

{

continue;

}

//for (int i = 0; i < p\_dividers.Count; i++) // для усиления генератора

//{

// if (ExponentiationModulo(g, fi\_p / p\_dividers[i], p) != 1)

// {

// true\_p = false;

// break;

// };

//}

if (true\_p)

{

return new List<BigInteger> { p, g };

}

}

}

}

public string TestMillerRabin(BigInteger n)

{

if (n == 1 || n == 2 || n == 3) // 1 не является простым числом. Это для того, чтобы программа не падала

{

return "Вероятно простое";

}

if (n % 2 == 0)

{

return "Составное";

}

double k = BigInteger.Log(n);

// представим n − 1 в виде (2^s)·t, где t нечётно, это можно сделать последовательным делением n - 1 на 2

BigInteger t = n - 1;

int s = 0;

while (t % 2 == 0)

{

t /= 2;

s += 1;

}

for (int i = 0; i < k; i++)

{

BigInteger a;

do

{

a = new BigInteger(RandomNumberGenerator.GetBytes(n.GetByteCount()));

}

while (a < 2 || a >= n - 2);

BigInteger x = ExponentiationModulo(a, t, n);

if (x == 1 || x == n - 1)

continue;

for (int r = 1; r < s; r++)

{

x = ExponentiationModulo(x, 2, n);

if (x == 1)

return "Составное";

if (x == n - 1)

break;

}

if (x != n - 1)

return "Составное";

}

return "Вероятно простое";

}

public BigInteger FindInvertibleNumberModulo(BigInteger n, BigInteger p)

{

for (BigInteger i = 1; i < 10000; i++)

{

if (ExponentiationModulo(n \* i, 1, p) == 1)

{

return i;

}

}

return -1;

}

public BigInteger[] FindMultipliersModulo\_x\_y(BigInteger number1, BigInteger number2, BigInteger p, BigInteger sum)

{

for (BigInteger x = 1; x < 10000; x++)

{

for (BigInteger y = 1; y < 10000; y++)

{

if (ExponentiationModulo(number1 \* x + number2 \* y, 1, p) == sum)

{

return [x, y];

}

else if (ExponentiationModulo(number1 \* -x + number2 \* y, 1, p) == sum)

{

return [p - x, y];

}

else if (ExponentiationModulo(number1 \* x + number2 \* -y, 1, p) == sum)

{

return [x, p - y];

}

else if (ExponentiationModulo(number1 \* -x + number2 \* -y, 1, p) == sum)

{

return [p - x, p - y];

}

}

}

return [0, 0];

}

}

}

Приложение 6. Алгоритм Адлемана

using DiscreteLogarithm.ExponentialAlgorithms;

using DiscreteLogarithm.MathFunctionsForCalculation;

using ExtendedNumerics;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Text;

using System.Threading.Tasks;

using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.SubExponentialAlgorithms

{

public class ListGroupedValuesIndex

{

public ListGroupedValuesIndex(int index, List<ListGroupedValues> listGroupedValues)

{

this.index = index;

this.listGroupedValues = listGroupedValues;

}

public int index { get; set; }

public List<ListGroupedValues> listGroupedValues { get; set; }

}

public class Log\_g\_NUM\_result

{

public Log\_g\_NUM\_result(BigInteger input\_num, BigInteger input\_result)

{

num = input\_num;

result = input\_result;

}

public BigInteger num { get; set; }

public BigInteger result { get; set; }

}

public class Adleman

{

MathFunctions mathFunctions;

BigRational expNumber;

FactorBase primeFactorBase { get; set; }

BigInteger B;

List<ListGroupedValuesIndex> exponentiationModuloDividersGroupedList;

List<BigInteger> log\_g\_NUM;

List<List<BigInteger>> SLAU;

List<Log\_g\_NUM\_result> log\_g\_NUM\_result;

List<BigInteger[,]> slauArrayResults;

BigInteger g;

BigInteger p;

BigInteger A;

bool numbersSwaped;

public Adleman()

{

mathFunctions = new MathFunctions();

expNumber = new BigRational(2, 718, 1000); // 2.718

primeFactorBase = new FactorBase();

B = 0;

exponentiationModuloDividersGroupedList = new List<ListGroupedValuesIndex>();

log\_g\_NUM = new List<BigInteger>();

log\_g\_NUM\_result = new List<Log\_g\_NUM\_result>();

SLAU = new List<List<BigInteger>>();

slauArrayResults = new List<BigInteger[,]>();

numbersSwaped = false;

}

public void CheckingTheInputValues(

string input\_g,

string input\_A,

string input\_p,

Label inputLabel,

ref bool theValuesAreCorrect,

out BigInteger g,

out BigInteger A,

out BigInteger p)

{

inputLabel.Text = "";

if (!BigInteger.TryParse(input\_g, out g) || g <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text = "Ошибка ввода числа g";

};

if (!BigInteger.TryParse(input\_A, out A) || A <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа A";

};

if (!BigInteger.TryParse(input\_p, out p) || p <= 0)

{

theValuesAreCorrect = false;

inputLabel.Text += "\nОшибка ввода числа p";

};

}

public void CalculateAdleman(BigInteger input\_g, BigInteger input\_A, BigInteger input\_p, Label inputLabel)

{

//g = 21;

//p = 127;

//A = 34;

g = input\_g;

p = input\_p;

A = input\_A;

Step1();

Step2();

Step3();

BigInteger result = Step4();

inputLabel.Text = string.Format("Результат = {0}", result);

}

private void Step1()

{

BigInteger degree = BigRational.Sqrt(BigInteger.Log2(p) \* BigInteger.Log2(BigInteger.Log2(p))).WholePart;

B = (BigInteger)BigRational.Pow(expNumber, degree).FractionalPart;

primeFactorBase.RationalFactorBase = PrimeFactory.GetPrimesTo(B);

}

private void Step2()

{

BigInteger exponentiationModuloResult = 0;

List<BigInteger> exponentiationModuloList = new List<BigInteger>();

List<ListGroupedValues> exponentiationModuloDividersGrouped = new List<ListGroupedValues>();

bool isSmooth = true;

for (int i = 4; i < B; i++)

{

exponentiationModuloResult = mathFunctions.ExponentiationModulo(g, i, p);

exponentiationModuloList = mathFunctions.Factorization(exponentiationModuloResult);

exponentiationModuloDividersGrouped = exponentiationModuloList

.GroupBy(x => x)

.Select(group => new ListGroupedValues(group.Key, group.Count(), BigInteger.Pow(group.Key, group.Count())))

.ToList();

for (int j = 0; j < exponentiationModuloDividersGrouped.Count; j++)

{

if (exponentiationModuloDividersGrouped[j].Key > B)

{

isSmooth = false;

break;

}

}

if (isSmooth)

{

ListGroupedValuesIndex listGroupedValuesIndex = new ListGroupedValuesIndex(i, exponentiationModuloDividersGrouped);

exponentiationModuloDividersGroupedList.Add(listGroupedValuesIndex);

}

isSmooth = true;

}

}

private void Step3()

{

CreateSLAU();

CalculateSLAU();

PrintSLAU();

}

private BigInteger Step4()

{

List<BigInteger> exponentiationModuloList = new List<BigInteger>();

List<ListGroupedValues> exponentiationModuloDividersGrouped = new List<ListGroupedValues>();

BigInteger x;

bool isContains;

for (int i = 2; i < 100; i++)

{

x = 0;

exponentiationModuloList = mathFunctions.Factorization(mathFunctions.ExponentiationModulo(A \* BigInteger.Pow(g, i), 1, p));

exponentiationModuloDividersGrouped = exponentiationModuloList

.GroupBy(x => x)

.Select(group => new ListGroupedValues(group.Key, group.Count(), BigInteger.Pow(group.Key, group.Count())))

.ToList();

isContains = false;

foreach(var exponentiationModuloDivider in exponentiationModuloDividersGrouped)

{

foreach(var log\_g\_NUM\_element in log\_g\_NUM\_result)

{

if (exponentiationModuloDivider.Key == log\_g\_NUM[(int)log\_g\_NUM\_element.num])

{

isContains = true;

x += exponentiationModuloDivider.degree\_number \* log\_g\_NUM\_element.result;

break;

}

}

if(isContains == false)

{

break;

}

}

if (isContains == true)

{

x -= i;

if (x != 0 && mathFunctions.ExponentiationModulo(g, x, p) == A)

{

return x;

}

}

}

return 0;

}

private void CreateSLAU()

{

for (int i = 0; i < exponentiationModuloDividersGroupedList.Count; i++)

{

for (int j = 0; j < exponentiationModuloDividersGroupedList[i].listGroupedValues.Count; j++)

{

log\_g\_NUM.Add(exponentiationModuloDividersGroupedList[i].listGroupedValues[j].Key);

}

}

log\_g\_NUM.Sort();

log\_g\_NUM = log\_g\_NUM.Distinct().ToList();

// создание СЛАУ

int rowSLAUindex;

for (int i = 0; i < exponentiationModuloDividersGroupedList.Count; i++)

{

List<BigInteger> rowSLAU = new List<BigInteger>();

for (int j = 0; j < log\_g\_NUM.Count; j++)

{

rowSLAU.Add(0);

}

rowSLAU.Add(exponentiationModuloDividersGroupedList[i].index);

for (int j = 0; j < exponentiationModuloDividersGroupedList[i].listGroupedValues.Count; j++)

{

rowSLAUindex = log\_g\_NUM.IndexOf(exponentiationModuloDividersGroupedList[i].listGroupedValues[j].Key);

rowSLAU[rowSLAUindex] = exponentiationModuloDividersGroupedList[i].listGroupedValues[j].degree\_number;

}

SLAU.Add(rowSLAU);

}

PrintSLAU();

}

private void CalculateSLAU()

{

for (int i = 0; i < SLAU.Count - 1; i++)

{

if (NonZeroValuesCount(SLAU[i]) > 2)

{

continue;

}

for (int j = i + 1; j < SLAU.Count; j++)

{

if (NonZeroValuesCount(SLAU[j]) > 2)

{

continue;

}

BigInteger[,] slauArray = new BigInteger[3, 3];

if (SlauMatrixCreated(slauArray, SLAU[i], SLAU[j]))

{

if (CalculateCreatedSlauMatrix(slauArray, i, j))

{

slauArrayResults.Add(slauArray);

}

numbersSwaped = false;

}

}

}

CreateLog\_g\_NUM\_result();

}

private void CreateLog\_g\_NUM\_result()

{

bool isСontainsList\_0\_0 = false;

bool isСontainsList\_0\_1 = false;

for (int i = 0; i < slauArrayResults.Count; i++)

{

for (int j = 0; j < log\_g\_NUM\_result.Count; j++)

{

if (log\_g\_NUM\_result[j].num == slauArrayResults[i][0, 0] && log\_g\_NUM\_result[j].result == slauArrayResults[i][1, 2])

{

isСontainsList\_0\_0 = true;

}

if (log\_g\_NUM\_result[j].num == slauArrayResults[i][0, 1] && log\_g\_NUM\_result[j].result == slauArrayResults[i][2, 2])

{

isСontainsList\_0\_1 = true;

}

}

if (isСontainsList\_0\_0 == false)

{

log\_g\_NUM\_result.Add(new Log\_g\_NUM\_result(slauArrayResults[i][0, 0], slauArrayResults[i][1, 2]));

}

if (isСontainsList\_0\_1 == false)

{

log\_g\_NUM\_result.Add(new Log\_g\_NUM\_result(slauArrayResults[i][0, 1], slauArrayResults[i][2, 2]));

}

isСontainsList\_0\_0 = false;

isСontainsList\_0\_1 = false;

}

log\_g\_NUM\_result = log\_g\_NUM\_result.OrderBy(x => x.num).ToList();

}

private int NonZeroValuesCount(List<BigInteger> slauRow)

{

int nonZeroValuesCount = 0;

for (int i = 0; i < slauRow.Count - 1; i++)

{

if (slauRow[i] != 0)

{

nonZeroValuesCount++;

}

}

return nonZeroValuesCount;

}

private bool SlauMatrixCreated(BigInteger[,] slauArray, List<BigInteger> slauRow\_i, List<BigInteger> slauRow\_j)

{

int slauArrayIndex\_i\_0 = -1;

int slauArrayIndex\_i\_1 = -1;

int slauArrayIndex\_j\_0 = -1;

int slauArrayIndex\_j\_1 = -1;

int slauRow\_i\_NonZeroValuesCount = NonZeroValuesCount(slauRow\_i);

int slauRow\_j\_NonZeroValuesCount = NonZeroValuesCount(slauRow\_j);

for (int q = 0; q < slauRow\_i.Count - 1; q++)

{

if (slauRow\_i[q] > 0)

{

if (slauArrayIndex\_i\_0 == -1)

{

slauArrayIndex\_i\_0 = q;

}

else

{

slauArrayIndex\_i\_1 = q;

}

}

if (slauRow\_j[q] > 0)

{

if (slauArrayIndex\_j\_0 == -1)

{

slauArrayIndex\_j\_0 = q;

}

else

{

slauArrayIndex\_j\_1 = q;

}

}

}

bool result = false;

if (slauArrayIndex\_i\_0 == slauArrayIndex\_j\_0

&& slauArrayIndex\_i\_1 == slauArrayIndex\_j\_1

&& slauArrayIndex\_i\_0 != -1 && slauArrayIndex\_i\_1 != -1)

{

result = true;

}

else if (slauArrayIndex\_i\_0 == slauArrayIndex\_j\_0

&& slauRow\_i\_NonZeroValuesCount == 2

&& slauRow\_j\_NonZeroValuesCount == 1)

{

slauArrayIndex\_j\_1 = slauArrayIndex\_i\_1;

result = true;

}

else if (slauArrayIndex\_i\_1 == slauArrayIndex\_j\_1

&& slauRow\_i\_NonZeroValuesCount == 2

&& slauRow\_j\_NonZeroValuesCount == 1)

{

slauArrayIndex\_j\_0 = slauArrayIndex\_i\_0;

result = true;

}

else if (slauArrayIndex\_i\_0 == slauArrayIndex\_j\_0

&& slauRow\_i\_NonZeroValuesCount == 1

&& slauRow\_j\_NonZeroValuesCount == 2)

{

slauArrayIndex\_i\_1 = slauArrayIndex\_j\_1;

result = true;

}

else if (slauArrayIndex\_i\_1 == slauArrayIndex\_j\_1

&& slauRow\_i\_NonZeroValuesCount == 1

&& slauRow\_j\_NonZeroValuesCount == 2)

{

slauArrayIndex\_i\_0 = slauArrayIndex\_j\_0;

result = true;

}

if (result)

{

slauArray[0, 0] = slauArrayIndex\_i\_0;

slauArray[0, 1] = slauArrayIndex\_i\_1;

slauArray[1, 0] = slauRow\_i[slauArrayIndex\_i\_0];

slauArray[1, 1] = slauRow\_i[slauArrayIndex\_i\_1];

slauArray[1, 2] = slauRow\_i[slauRow\_i.Count - 1];

slauArray[2, 0] = slauRow\_j[slauArrayIndex\_j\_0];

slauArray[2, 1] = slauRow\_j[slauArrayIndex\_j\_1];

slauArray[2, 2] = slauRow\_j[slauRow\_j.Count - 1];

BigInteger swapNumber;

if (slauRow\_j\_NonZeroValuesCount == 1 && slauArray[2, 1] == 0

|| slauRow\_i\_NonZeroValuesCount == 1 && slauArray[1, 0] == 0)

{

for (int i = 0; i < 3; i++)

{

swapNumber = slauArray[1, i];

slauArray[1, i] = slauArray[2, i];

slauArray[2, i] = swapNumber;

}

numbersSwaped = true;

}

PrintSlauArray(slauArray);

}

return result;

}

private bool CalculateCreatedSlauMatrix(BigInteger[,] slauArray, int i, int j)

{

BigInteger invertibleNumberModulo;

BigInteger[] multipliersModulo\_x\_y;

BigInteger p\_1 = p - 1;

if (slauArray[1, 1] != 0)

{

multipliersModulo\_x\_y = mathFunctions.FindMultipliersModulo\_x\_y(slauArray[1, 1], slauArray[2, 1], p\_1, 0);

if (multipliersModulo\_x\_y[0] == 0 && multipliersModulo\_x\_y[1] == 0)

{

return false;

}

slauArray[1, 0] = mathFunctions.ExponentiationModulo(slauArray[1, 0] \* multipliersModulo\_x\_y[0] + slauArray[2, 0] \* multipliersModulo\_x\_y[1], 1, p\_1);

slauArray[1, 1] = 0;

slauArray[1, 2] = mathFunctions.ExponentiationModulo(slauArray[1, 2] \* multipliersModulo\_x\_y[0] + slauArray[2, 2] \* multipliersModulo\_x\_y[1], 1, p\_1);

}

if (slauArray[1, 0] != 1)

{

invertibleNumberModulo = mathFunctions.FindInvertibleNumberModulo(slauArray[1, 0], p\_1);

if (invertibleNumberModulo == -1)

{

return false;

}

slauArray[1, 0] = 1;

slauArray[1, 2] = mathFunctions.ExponentiationModulo(slauArray[1, 2] \* invertibleNumberModulo, 1, p\_1);

}

if (slauArray[2, 0] != 0)

{

multipliersModulo\_x\_y = mathFunctions.FindMultipliersModulo\_x\_y(slauArray[1, 0], slauArray[2, 0], p\_1, 0);

if (multipliersModulo\_x\_y[0] == 0 && multipliersModulo\_x\_y[1] == 0)

{

return false;

}

slauArray[2, 0] = 0;

slauArray[2, 1] = mathFunctions.ExponentiationModulo(slauArray[1, 1] \* multipliersModulo\_x\_y[0] + slauArray[2, 1] \* multipliersModulo\_x\_y[1], 1, p\_1);

slauArray[2, 2] = mathFunctions.ExponentiationModulo(slauArray[1, 2] \* multipliersModulo\_x\_y[0] + slauArray[2, 2] \* multipliersModulo\_x\_y[1], 1, p\_1);

}

if (slauArray[2, 1] != 0)

{

invertibleNumberModulo = mathFunctions.FindInvertibleNumberModulo(slauArray[2, 1], p\_1);

if (invertibleNumberModulo == -1)

{

return false;

}

slauArray[2, 1] = 1;

slauArray[2, 2] = mathFunctions.ExponentiationModulo(slauArray[2, 2] \* invertibleNumberModulo, 1, p\_1);

}

if (numbersSwaped)

{

int swapNumber;

swapNumber = i;

i = j;

j = swapNumber;

}

SLAU[i][(int)slauArray[0, 0]] = slauArray[1, 0];

SLAU[i][(int)slauArray[0, 1]] = slauArray[1, 1];

SLAU[i][SLAU[i].Count - 1] = slauArray[1, 2];

SLAU[j][(int)slauArray[0, 0]] = slauArray[2, 0];

SLAU[j][(int)slauArray[0, 1]] = slauArray[2, 1];

SLAU[j][SLAU[j].Count - 1] = slauArray[2, 2];

PrintSlauArray(slauArray, "Преобразованная СЛАУ");

return true;

}

private void PrintSlauArray(BigInteger[,] slauArray, string inputText = "")

{

Console.WriteLine(inputText);

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

Console.Write(string.Format("{0} ", slauArray[i, j]));

}

Console.WriteLine();

}

}

private void PrintSLAU()

{

for (int i = 0; i < log\_g\_NUM.Count; i++)

{

Console.Write(string.Format("{0} ", log\_g\_NUM[i]));

}

Console.WriteLine();

for (int i = 0; i < SLAU.Count; i++)

{

for (int j = 0; j < SLAU[0].Count; j++)

{

Console.Write(string.Format("{0} ", SLAU[i][j]));

}

Console.WriteLine();

}

}

}

}