

Оглавление

Введение.....	2
1. Реализация методов дискретного логарифмирования	6
2. Вспомогательные математические функции.....	8
3. Алгоритм Шенкса	12
4. Алгоритм Полига-Хеллмана	19
5. Алгоритм ро-метод Полларда	27
6. Алгоритм Адлемана	36
7. Алгоритм COS	43
8. Алгоритм решето числового поля	51
Заклучение	61
Список литературы	65
Приложения	66
Приложение 1. Модифицированный алгоритм Шенкса	66
Приложение 2. Модифицированный алгоритм Полига-Хеллмана.....	69
Приложение 3. Модифицированный алгоритм ро-метод Полларда.....	73
Приложение 4. Модифицированный алгоритм Адлемана.....	74
Приложение 5. Модифицированный алгоритм COS.....	85
Приложение 6. Модифицированный алгоритм решето числового поля.....	97

Введение

Дискретное логарифмирование является задачей обращения функции g^x в некоторой конечной мультипликативной группе G .

Задача дискретного логарифмирования наиболее часто рассматривается над конечным полем в группе точек эллиптической кривой и в мультипликативной группе конечного поля или кольца вычетов. На данный момент не существует эффективных алгоритмов для решения задач дискретного логарифмирования.

Дискретным логарифмом элемента a по основанию g называется решение x уравнения $g^x = a$. Индекс числа a по основанию g называется решением, когда G является мультипликативной группой кольца вычетов по модулю m . Если g является первообразным корнем по модулю m , то обязательно существует индекс числа a по основанию g .

Пусть в некоторой конечной мультипликативной абелевой группе G задано уравнение $g^x = a$. Нахождение некоторого неотрицательного числа x , удовлетворяющего данному уравнению, является решением задачи дискретного логарифмирования. Если существует решение, то у уравнения должно быть хотя бы одно натуральное решение, не превышающее порядок группы. Таким образом данное условие даёт жесткую оценку сложности алгоритма поиска решений сверху. Решение перебором нашло бы результат за число шагов не выше порядка данной группы.

Наиболее рассматриваемым случаем является $G = \langle g \rangle$, когда группа является циклической, порождённой элементом g . В данном случае уравнение всегда имеет решение. Существование решения задачи дискретного логарифмирования, а именно уравнения $g^x = a$, требует отдельного рассмотрения, если группа произвольная.

Для примера дана задача дискретного логарифмирования в кольце вычетов по модулю простого числа $3^x = 13 \pmod{17}$. Будет проведено решение задачи методом перебора. Вычисляя все степени числа 3 по модулю

17, получаются следующие значения: $3^1 = 3 \pmod{17}$, $3^2 = 9 \pmod{17}$, $3^3 = 10 \pmod{17}$, $3^4 = 13 \pmod{17}$, $3^5 = 5 \pmod{17}$, $3^6 = 15 \pmod{17}$, $3^7 = 11 \pmod{17}$, $3^8 = 16 \pmod{17}$, $3^9 = 14 \pmod{17}$, $3^{10} = 8 \pmod{17}$, $3^{11} = 7 \pmod{17}$. Решением данного уравнения является $x = 4$, так как $3^4 = 13 \pmod{17}$. В прикладных задачах модуль берётся большим числом, поэтому метод перебора является неэффективным и достаточно медленным, таким образом возникает потребность в более эффективных и быстрых алгоритмах.

Существует множество алгоритмов, которые разделяются на экспоненциальные и субэкспоненциальные алгоритмы, для решения задачи дискретного логарифмирования. На данный момент полиномиального алгоритма для решения данной задачи не существует.

Алгоритмы с экспоненциальной сложностью:

- 1) алгоритм Шенкса (алгоритм больших и малых шагов);
- 2) алгоритм Полига - Хеллмана работает, когда имеется разложение числа $p - 1 = \prod_{i=1}^s q_i^{\alpha_i}$ на простые множители. Сложность: $O(\sum_{i=1}^s \alpha_i (\log p + q_i))$. Алгоритм очень эффективен, когда множители, на которые раскладывается $p - 1$ достаточно маленькие;

- 3) ро-Метод Полларда имеет эвристическую оценку сложности $O(p^{1/2})$.

Алгоритмы с субэкспоненциальной сложностью:

В L-нотации вычислительная сложность данных алгоритмов оценивается как $L_p[d, c]$ арифметических операций, где c и $0 \leq d < 1$ - некоторые константы. От близости величин c и d к 1 и 0 зависит эффективность алгоритма.

- 1) алгоритм Алемана, появившийся в 1979 году, был первым субэкспоненциальным алгоритмом дискретного логарифмирования и на практике был недостаточно эффективным. В данном алгоритме $d = \frac{1}{2}$.

- 2) алгоритм COS был изобретён математиками Копперсмитом, Одлыжко и Шреппелем в 1986 году. В данном алгоритме $c = 1, d = \frac{1}{2}$. Данным

алгоритмом было проведено логарифмирование по модулю $p \approx 10^{58}$ в 1991 году. В 1997 году Вебер при использовании модифицированной версии алгоритма смог провести дискретное логарифмирование по модулю $p \approx 10^{85}$. Эксперименты показали, что алгоритм COS продемонстрировал лучшие результаты, чем решето числового поля при $p \leq 10^{90}$.

3) дискретное логарифмирование с использованием решета числового поля было применено к дискретному логарифмированию позже, чем к факторизации чисел. Первые идеи появились в 1990-х годах. Модификация алгоритма Д. Гордона, предложенная в 1993 году, имела эвристическую сложность $L_p\left[\frac{1}{3}, 3^{3/2}\right]$, но оказалась достаточно непрактичной. Позднее появились множество других улучшений данного алгоритма. Экспериментально было показано, что при $p \geq 10^{100}$ решето числового поля быстрее, чем алгоритм COS. Данным алгоритмом получены современные рекорды дискретного логарифмирования.

Параметры $c = \frac{(92+26\sqrt{13})^{\frac{1}{3}}}{3} \approx 1,902, d = \frac{1}{3}$ являются наилучшими в оценке сложности на данный момент.

Есть возможность улучшить результат для чисел специального вида. Можно построить алгоритм, для которого константы будут $c \approx 1,00475, d = \frac{2}{5}$. Данные алгоритмы могут обогнать алгоритм с $d = \frac{1}{3}$, потому что константа c достаточно близка к 1.

Актуальность выпускной работы заключается в том, что задача дискретного логарифмирования является одной из основных задач, на которых базируется криптография с открытым ключом. Классическими криптографическими схемами на её основе являются схема выработки общего ключа Диффи-Хеллмана, схема электронной подписи Эль-Гамала, криптосистема Мэсси-Омуры для передачи сообщений. За счёт высокой вычислительной сложности обращения показательной функции проявляется их криптостойкость. Хотя сама показательная функция вычисляется

достаточно эффективно, даже самые современные алгоритмы вычисления дискретного логарифма имеют очень высокую сложность, которая сравнима со сложностью наиболее быстрых алгоритмов разложения чисел на множители.

Другая возможность эффективного решения задачи вычисления дискретного логарифма связана с квантовыми вычислениями. Теоретически доказано, что с помощью алгоритма Шора дискретный логарифм можно вычислить за полиномиальное время. В любом случае, если полиномиальный алгоритм вычисления дискретного логарифма будет реализован, это будет означать практическую непригодность криптосистем на его основе для долговременной защиты данных. Рассматривается ряд идей для создания новых алгоритмов с открытым ключом.

Целью выпускной работы является исследование и реализация алгоритмов дискретного логарифмирования с экспоненциальной и субэкспоненциальной сложностью. Также исследование и реализация модифицированных алгоритмов на основе разработанных базовых алгоритмов дискретного логарифмирования, проведение экспериментов и сравнение базовых и модифицированных алгоритмов.

Задачами выпускной работы являются:

- 1) реализовать вспомогательные математические функции для проверки алгоритмов дискретного логарифмирования,
- 2) исследовать и реализовать базовые алгоритмы дискретного логарифмирования,
- 3) исследовать и реализовать модифицированные алгоритмы дискретного логарифмирования,
- 4) провести эксперименты и сравнительный анализ на реализованных базовых и модифицированных методах дискретного логарифмирования.

1. Реализация методов дискретного логарифмирования

В процессе выпускной работы были изучены и реализованы базовые и модифицированные методы дискретного логарифмирования на языке программирования C# на .NET8 в Windows Forms (рисунок 1). Для работы с большими рациональными числами и полиномами использовалась библиотека ExtendedArithmetic. Для тестирования данных алгоритмов был реализован генератор параметров Диффи-Хеллмана и возведение числа в степень по модулю. Также для тестирования данных алгоритмов был использован замер времени выполнения алгоритма и количество затраченной памяти на выполнение алгоритма.

Экспоненциальные алгоритмы дискретного логарифмирования $A = g^a \mod p$

Алгоритм Шенкса
Введите g:
Введите A:
Введите p:
Вычислить Модифицированный
Результат: Результат:

Алгоритм Полига-Хеллмана
Введите g:
Введите A:
Введите p:
Вычислить Модифицированный
Результат: Результат:

p-метод Полларда
Введите N:
Вычислить Модифицированный
Результат: Результат:

Генератор чисел
Сгенерированное g:
Сгенерированное a:
Сгенерированное p:
Сгенерированное A:
Вычислить

Субэкспоненциальные алгоритмы дискретного логарифмирования

Алгоритм Адлемана
Введите g:
Введите A:
Введите p:
Вычислить Модифицированный
Результат: Результат:

Алгоритм COS
Введите g:
Введите A:
Введите p:
Вычислить Модифицированный
Результат: Результат:

Решето числового поля
Введите N:
Вычислить Модифицированный
Результат: Результат:

Возведение в степень по модулю
Введите g:
Введите a:
Введите p:
Вычислить
Результат:

Рисунок 1 - Реализованная программа

Были реализованы экспоненциальные алгоритмы дискретного логарифмирования: алгоритм Шенкса, алгоритм Полига-Хеллмана, р-метод Полларда, а также субэкспоненциальные алгоритмы дискретного логарифмирования: алгоритм Адлемана, алгоритм COS, решето числового

поля. 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 введенные значения в `BigInteger`. Если конвертация проходит успешно, то идёт проверка отрицательность конвертированных значений. Если проверка введенных значений проходит успешно, то идёт вычисление алгоритма (рисунок 2).

Экспоненциальные алгоритмы дискретного логарифмирования $A = g^a \bmod p$

Алгоритм Шенкса	Алгоритм Полига-Хеллмана	p -метод Полларда	Генератор чисел
Введите g 21	Введите g 21	Введите N 45113	Сгенерированное g
Введите A 34	Введите A 34	Вычислить	Сгенерированное a
Введите p 127	Введите p 127	Модифицированный	Сгенерированное p
Вычислить	Вычислить	$P = 229$ $Q = 197$ $t = 1$ мс 0 байт	Сгенерированное A
Модифицированный	Модифицированный	$P = 197$ $Q = 229$ $t = 0$ мс 0 байт	Вычислить
Результат: $a = 52$ $t = 2$ мс 8224 байт	Результат: $a = 10$ $t = 17$ мс 16448 байт		
Результат: $a = 52$ $t = 36$ мс 24672 байт	Результат: $a = 52$ $t = 1$ мс 16448 байт		

Субэкспоненциальные алгоритмы дискретного логарифмирования

Алгоритм Адлемана	Алгоритм COS	Решето числового поля	Возведение в степень по модулю
Введите g 21	Введите g 21	Введите N 45113	Введите g
Введите A 34	Введите A 34	Вычислить	Введите a
Введите p 127	Введите p 127	Модифицированный	Введите p
Вычислить	Вычислить	$P = 197$ $Q = 229$ $t = 129$ мс 533360 байт	Вычислить
Модифицированный	Модифицированный	$P = 197$ $Q = 229$ $t = 174$ мс 1060680 байт	Результат:
Результат: $a = 52$ $t = 251$ мс 71424 байт	Результат: $a = 52$ $t = 78$ мс 4760472 байт		
Результат: $a = 52$ $t = 12240$ мс 431816 байт	Результат: $a = 52$ $t = 59$ мс 1543648 байт		

Рисунок 2 - Вычисление модифицированных алгоритмов

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 является операцией над натуральными числами, которые выполняются по модулю. Данная операция применяется в информатике, в частности в криптографии с открытым ключом.

Возведение в степень по модулю – это вычисление остатка от деления натурального числа g (основание), возведённого в степень a (показатель степени), на натуральное число p (модуль), обозначаемое $A \equiv g^a \pmod{p}$.

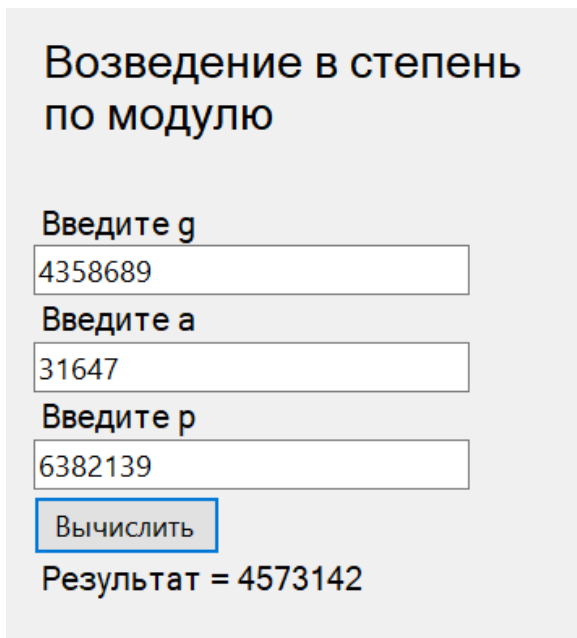
Если выполняется условие, что g , a и p неотрицательны и $g < p$, тогда существует единственное решение A , причём $0 \leq A < p$. В случае, если показатель степени a отрицательное, то возведение в степень по модулю также может быть выполнено. В таком случае требуется найти число d , обратное числу g по модулю p , что выполняется при использовании алгоритма Евклида. Отсюда следует, что $A \equiv g^a \equiv d^{|a|} \pmod{p}$, где $a < 0$ и $g * d \equiv 1 \pmod{p}$.

При больших входных значениях возведение в степень по модулю достаточно эффективно, но более трудозатратным является вычисление дискретного логарифма, то есть нахождение показателя степени a при входных параметрах g , p и A . Данное одностороннее поведение функции даёт возможность применять функцию в криптографических алгоритмах [1, 2].

Шаги алгоритма быстрого возведения в степень по модулю:

- 1) показатель степени a переводится в двоичный вид;
- 2) создаётся список для чисел, первый элементом которого является g ;
- 3) в созданный список циклично добавляются остатки от деления квадратов последних элементов на p ;
- 4) все элементы списка, индексы которых совпадают с позициями 1 в двоичном числе a из первого пункта, перемножаются;
- 5) вычисляется остаток от деления перемноженных чисел списка на p .

Для проверки работы разработанного алгоритма были сгенерированы параметры Диффи-Хеллмана: $g = 4358689, a = 31647, p = 6382139$. В результате работы алгоритма был корректно вычислен результат $A = 4573142$ (рисунок 3).



Возведение в степень по модулю

Введите g
4358689

Введите a
31647

Введите p
6382139

Вычислить

Результат = 4573142

Рисунок 3 - Результат возведения в степень по модулю

Для проверки чисел на простоту был реализован тест Миллера-Рабина. Данный тест является вероятностным полиномиальным тестом простоты. Данный тест применяется для того, чтобы эффективно определить, является ли входное число составным. Для решения данной проблемы применяются в том числе тест Ферма и тест Соловея-Штрассена. Недостатком теста Миллера-Рабина является невозможность строгого доказательства простоты числа. Однако данный тест широко применяется в области криптографии для вычисления больших случайных простых чисел.

Для создания секретных ключей, на которых основывается криптостойкость многих алгоритмов шифрования, необходимы простые числа. По данной причине для генерации ключей необходимо иметь возможность проверять большие числа на простоту достаточно быстро. Вероятностные тесты демонстрируют большую простоту выражения и эффективность использования по сравнению с детерминированными тестами,

например, тест Миллера-Рабина и тест Соловея-Штрассена. Тест Миллера-Рабина даёт достаточно малую вероятность того, что число является составным при проверке за малое время.

Тест Миллера-Рабина основывается на проверке ряда равенств, выполняемой для простых чисел. При условии, что хотя бы одно равенство не выполняется, тест докажет, что число составное.

Для теста Миллера-Рабина применяется следующее утверждение. Пусть n – простое число и $s - 1 = 2^s d$, где d – нечётно. Тогда для любого a из \mathbb{Z}_n выполняется хотя бы одно из условий:

- 1) $a^d \equiv 1 \pmod{n}$;
- 2) существует целое число $r < s$ такое, что $a^{2^r d} \equiv -1 \pmod{n}$.

Если одно из данных условий выполняется для некоторых чисел a и n (не обязательно простого), то число n называют вероятно простым, а число a – свидетелем простоты числа n по Миллеру. Вероятность ошибочно принять составное число за простое составляет 25% при случайно выбранном a , однако выполнив проверки для других a вероятность ошибки можно уменьшить.

При условии, что выполняется контрапозиция доказанного утверждения, то есть если найдётся число a такое, что:

- 1) $a^d \not\equiv 1 \pmod{n}$,
- 2) $\forall r: 0 \leq r \leq s - 1: a^{2^r d} \not\equiv -1 \pmod{n}$,

тогда число n не является простым, а число a называется свидетелем того, что число n является составным.

Согласно теореме Рабина, нечётные составные числа имеют не более $\varphi(n)/4$ свидетелей простоты, где $\varphi(n)$ — функция Эйлера, то есть вероятность того, что случайно выбранное число a окажется свидетелем простоты, меньше 0,25. Суть теста состоит в том, чтобы проверить являются ли случайно выбранные числа a свидетелями простоты числа n , при условии, что $a < n$. Число в результате является составным, если имеется свидетель того, что число составное. Таким образом, число считается простым, если

было проверено k чисел, все из которых оказались свидетелями простоты. Вероятность принятия составного числа за простое является меньше $(\frac{1}{4})^k$ для данного алгоритма.

По причине того, что распределение свидетелей простоты и составного числа среди чисел $1, 2, \dots, n - 1$ заранее неизвестно, необходимо выбирать числа a случайными для проверки больших чисел.

Для алгоритма Миллера-Рабина, параметризуемого количеством раундов r , рекомендуется брать r порядка величины $\log_2(n)$, где n - проверяемое число.

Для данного n вычисляется такое целое число s и целое нечётное число t , что выполняется $n - 1 = 2^s t$. Генерируется случайное число a такое, что $1 < a < n$. В случае, если число a не является свидетелем простоты числа n , генерируется ответ, что « n — составное», далее алгоритм завершается. Иначе, генерируется новое случайное число a и проверка повторяется. После вычисления r свидетелей простоты, имеется результат, что « n — вероятно простое», после чего алгоритм завершается [3, 4].

Для генерации параметров Диффи-Хеллмана генерируется случайно показатель степени a определённой битности. Также определённой битности генерируются параметры p и g . Далее идёт проверка, что параметр g является первообразным корнем по модулю p . Если выполняются условия, что:

- 1) число g взаимно простое с числом p ,
 - 2) выполняется равенство $g^{\varphi(p)} \not\equiv 1 \pmod{p}$,
 - 3) выполняется равенство $g^d \not\equiv 1 \pmod{p}$ для всех $d = \varphi(p)/t$, где t является простым делителем числа $\varphi(p)$, полученный при помощи факторизации ро-методом Полларда,
- тогда параметр g является первообразным корнем по модулю p , иначе генерируются другие параметры p и g . После успешной генерации параметров a , p и g вычисляется решение A по формуле $A \equiv g^a \pmod{p}$.

3. Алгоритм Шенкса

Были реализованы алгоритмы и проведены тесты базового и модифицированного алгоритма «Шаг младенца – шаг великана». Алгоритм Гельфонда-Шенкса - в теории групп детерминированный алгоритм дискретного логарифмирования в мультипликативной группе кольца вычетов по модулю простого числа. Данный алгоритм также называется алгоритмом согласования и алгоритмом больших и малых шагов. Начальный вид алгоритма был разработан советским математиком Александром Гельфондом в 1962 году и Дэниелом Шенксом в 1972 году. Данный алгоритм в теории упрощает решение задач дискретного логарифмирования, на вычислительной сложности которого построены многие криптосистемы с открытым ключом, и относится к методам встречи посередине.

Данный метод был одним из первых алгоритмов, продемонстрировавших, что вычисление дискретного логарифма может выполняться быстрее, чем методом перебора. Суть метода заключается в усовершенствованном поиске показателя степени, то есть выбирается оптимальное соотношение времени и памяти.

Пусть задано сравнение $a^x \equiv b \pmod{p}$, необходимо найти натуральное число x , удовлетворяющее данному сравнению.

Начальный алгоритм реализован следующим образом:

1) сначала берутся два целых числа m и k , такие, что $mk > p$. Как правило $m = k = \sqrt[p]{p} + 1$;

2) вычисляются два ряда чисел:

$$a^m, a^{2m}, \dots, a^{km} \pmod{p},$$

$$b, ba, ba^2, \dots, ba^{m-1} \pmod{p}.$$

Все вычисления проводятся по модулю p ;

3) идёт поиск таких i и j , для которых выполняется равенство j . То есть ищется во втором ряду такое число, которое присутствует и в первом ряду.

Запоминаются показатели степени im и j , при которых данные числа получались;

4) в результате работы алгоритма неизвестная степень вычисляется по формуле $x = im - j$ [5].

Была реализована модификация алгоритма, состоящая в распараллеливании 2 и 3 шага алгоритма. На 2 шаге алгоритма параллельно асинхронно вычисляются два ряда чисел. Функция 2 шага заканчивает работу, когда оба ряда чисел полностью вычислены. Теоретическая оценка сложности 2 шага базового алгоритма $O(2n)$, а модифицированного $O(n)$. На 3 шаге начинают вычисляться параллельно 2 функции, одна из которых начинает поиск результата с начала ряда, а другая с конца ряда. Функция 3 шага прекращает работу, когда одна из функций находит результат вычисления. Теоретическая оценка сложности 3 шага базового алгоритма $O(n)$, а модифицированного $O(n/2)$.

Были сгенерированы параметры и проведены тесты базового (таблица 1) и модифицированного (таблица 2) алгоритма Шенкса, где g , p и A - 16 битные числа, а параметр a - 8 битное число.

Таблица 1- Результаты тестов базового алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
9347	99	14629	6331	4	278312
11873	107	14401	2419	1	270368
11922	78	26399	22034	1	417888
606	75	4973	3597	2	139664
8173	49	14143	8610	1	263168
32464	14	32717	20677	1	263168
6229	118	32257	5301	2	444096
7921	106	16703	100	2	296064

Таблица 1 - Результаты тестов базового алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
16108	101	31271	6732	7	5862760
5901	85	7019	3639	2	164480

Таблица 2 - Результаты тестов модифицированного алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
9347	99	14629	6331	47	278592
11873	107	14401	2419	27	270368
11922	78	26399	22034	23	409408
606	75	4973	3597	24	139808
8173	49	14143	8610	23	271392
32464	14	32717	20677	35	466992
6229	118	32257	5301	34	452320
7921	106	16703	100	21	296064
16108	101	31271	6732	30	452320
5901	85	7019	3639	32	172704

В результате тестов, где g , p и A - 16 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Шенкса равно 2.3 мс, а модифицированного алгоритма Шенкса равно 29.6 мс. Средняя затраченная память базового алгоритма Шенкса равна 839996.8 байт, а модифицированного алгоритма Шенкса равна 320996.8 байт. Базовый алгоритм показал лучше результаты в скорости выполнения, а модифицированный алгоритм показал лучше результаты в затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 3) и модифицированного (таблица 4) алгоритма Шенкса, где g , p и A – 32 битные числа, а параметр a - 8 битное число.

Таблица 3- Результаты тестов базового алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
72142839	45	401699497	180776367	179450	4586080
216592957	67	1535278343	983613871	1416246	8018992
106385010 5	28	1752424721	1133518573	2049247	10349832
641832856	114	1912453999	1707478458	1334235	9234184
153341898	17	378285451	184658749	356234	6531234
440270945	86	547132867	127053943	1734623	7652345
28181579	96	1691891543	1482106649	2195341	6534923
572050022	37	1405842083	45578011	1827374	8634152
176931448 7	117	1978813019	1603737570	1475357	5734645
608493163	53	667849967	614352815	2341533	10294564

Таблица 4 - Результаты тестов модифицированного алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
72142839	45	401699497	180776367	175768	1709904
216592957	67	1535278343	983613871	1387387	7734960
106385010 5	28	1752424721	1133518573	2046500	4854392
641832856	114	1912453999	1707478458	1134235	9134184
153341898	17	378285451	184658749	336234	6231234
440270945	86	547132867	127053943	1434623	7152345
28181579	96	1691891543	1482106649	2095341	6134923
572050022	37	1405842083	45578011	1427374	8234152

Таблица 4 - Результаты тестов модифицированного алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
176931448 7	117	1978813019	1603737570	1275357	5234645
608493163	53	667849967	614352815	2141533	10094564

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Шенкса равно 1490964 мс, а модифицированного алгоритма Шенкса равно 1345435.2 мс. Средняя затраченная память базового алгоритма Шенкса равна 7757095.1 байт, а модифицированного алгоритма Шенкса равна 6651530.3 байт. Модифицированный алгоритм показал лучше результаты в скорости выполнения и затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 5) и модифицированного (таблица 6) алгоритма Шенкса, где g , p и A – 32 битные числа, а параметр a - 16 битное число.

Таблица 5- Результаты тестов базового алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
734022286	2260	888333059	207727322	851317	2438016
519908789	27422	1176863351	1004437759	1032538	5691992
119062276 4	23591	1582719121	421276480	1775280	7058624
43944272	7622	113830279	97331062	1204953	4562345
11153680	31859	1827918509	1658501642	1352342	4567234
167298629	19434	289159777	20563900	1652352	5237524
83421829	2311	1620676819	1044052987	1586493	6956284

Таблица 5 - Результаты тестов базового алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
150689094 0	15782	1556831663	467681122	1826592	5927483
463547350	2937	1648004693	633238154	1284859	6375812
56985777	14752	60477983	13103552	1683934	6839491

Таблица 6 - Результаты тестов модифицированного алгоритма Шенкса

g	a	p	A	Время (мс)	Память (байт)
734022286	2260	888333059	207727322	848845	3354080
519908789	27422	1176863351	1004437759	1035389	211480
119062276 4	23591	1582719121	421276480	1742844	983696
43944272	7622	113830279	97331062	1104953	4262345
11153680	31859	1827918509	1658501642	1152342	4267234
167298629	19434	289159777	20563900	1252352	5037524
83421829	2311	1620676819	1044052987	1286493	6556284
150689094 0	15782	1556831663	467681122	1526592	5527483
463547350	2937	1648004693	633238154	1084859	6075812
56985777	14752	60477983	13103552	1483934	6439491

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 16 битное число, среднее время выполнения базового алгоритма Шенкса равно 1425066 мс, а модифицированного алгоритма Шенкса равно 1251860.3 мс. Средняя затраченная память базового алгоритма Шенкса равна 5565480.5 байт, а модифицированного алгоритма Шенкса равна 4271542.9 байт. Модифицированный алгоритм показал лучше результаты в скорости выполнения и затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма Шенкса можно сделать вывод, что базовый алгоритм показал лучшие результаты в затраченном времени выполнения на маленьких параметрах, где g , p и A - 16 битные числа, а параметр a - 8 битное число. В остальных тестах по времени и затраченной памяти лучшие результаты показал модифицированный алгоритм Шенкса. Также базовый и модифицированный алгоритм Шенкса показал лучшие результаты, где g , p и A - 32 битные числа, а параметр a - 16 битное число, чем при параметрах, где g , p и A - 32 битные числа, а параметр a - 8 битное число (рисунок 4, 5).

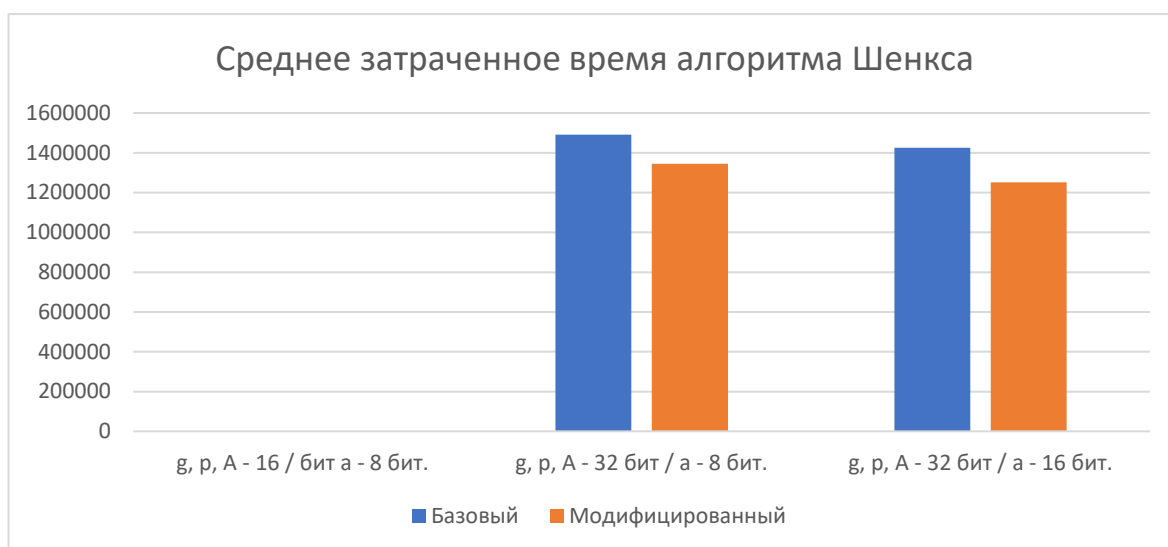


Рисунок 4 - Среднее затраченное время алгоритма Шенкса

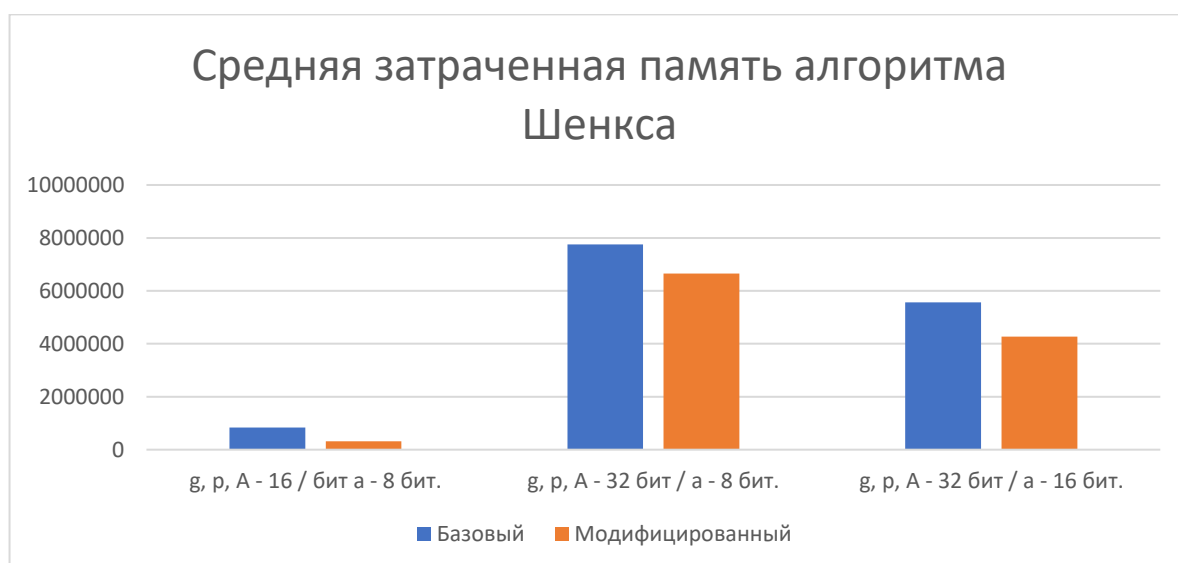


Рисунок 5 - Средняя затраченная память алгоритма Шенкса

4. Алгоритм Полига-Хеллмана

Были проведены тесты базового и модифицированного алгоритма Полига-Хеллмана. Данный алгоритм представляет собой детерминированный алгоритм дискретного логарифмирования в кольце вычетов по модулю простого числа.

Особенностью данного метода является возможность вычислять дискретный логарифм за полиномиальное время. Алгоритм был разработан Рональдом Сильвером, а впервые упомянут 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».

Пусть задано сравнение $a^x \equiv b \pmod{p}$, необходимо найти натуральное число x , удовлетворяющее данному сравнению.

Шаги выполнения алгоритма:

- 1) идёт разложение числа $\varphi(p) = p - 1$ на простые множители;
- 2) составляется таблица значений $\{r_{i,j}\}$,

где $r_{i,j} = a^{j \cdot \frac{p-1}{q_i}}, i \in \{1, \dots, k\}, j \in \{0, \dots, q_i - 1\}$;

- 3) вычисляется $b \pmod{q_i^{\alpha_i}}$.

Для i от 1 до k :

Пусть $x \equiv b \equiv x_0 + x_1 q_i + \dots + x_{\alpha_i-1} q_i^{\alpha_i-1} \pmod{q_i^{\alpha_i}}$,

где $0 \leq x_i \leq q_i - 1$.

Тогда верно сравнение:

$$a^{x_0 \cdot \frac{p-1}{q_i}} \equiv b^{\frac{p-1}{q_i}} \pmod{p}.$$

С помощью таблицы, составленной на шаге 1, находится x_0 .

Для j от 0 до $\alpha_i - 1$ рассматривается сравнение

$$a^{x_j \cdot \frac{p-1}{q_i}} \equiv \left(b a^{-x_0 - x_1 q_i - \dots - x_{j-1} q_i^{j-1}} \right)^{\frac{p-1}{q_i^{j+1}}} \pmod{p}.$$

Решение находится по таблице

Конец цикла по j .

Конец цикла по i ;

4) найдя $b \bmod q_i^{\alpha_i}$ для всех i , происходит поиск $b \bmod (p - 1)$ по китайской теореме об остатках [6].

Была реализована модификация алгоритма, состоящая в том, что на 1 шаге алгоритма число $\varphi(p) = p - 1$ было разложено на простые множители и данные простые множители были возведены в свои степени, чтобы на 2 шаге была составлена таблица из единичных значений без степеней. Для разложения на простые множители был использован алгоритм ро-метод Полларда, так как разложение чисел методом перебора имеет сложность $O(n^2)$, а ро-метод Полларда сложность $O(n^{1/4})$. Входное число раскладывалось на множители, далее данные множители циклично также раскладывались на множители, пока числа не перестанут раскладываться, став простыми. Данные разложенные числа собираются в список для дальнейших вычислений. Составление таблицы без степеней на 2 шаге должно уменьшить количество хранимых чисел, тем самым уменьшив выделяемую память, а также увеличить скорость вычисления таблицы за счёт меньшего количества чисел. После вычислений 1 шага алгоритма, теоретическая оценка сложности 2 шага базового алгоритма $O(n^2)$, а модифицированного $O(n)$.

Были сгенерированы параметры и проведены тесты базового (таблица 7) и модифицированного (таблица 8) алгоритма Полига-Хеллмана, где g , p и A - 16 битные числа, а параметр a - 8 битное число.

Таблица 7 - Результаты тестов базового алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
24988	115	26321	20051	4	122848
28078	92	29927	13442	3	2096960
9617	76	11161	5273	1	115136
1477	11	2237	1434	1	90464

Таблица 7 - Результаты тестов базового алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
5303	90	6911	98	3	1102016
5066	116	22129	10520	1	797744
529	46	6359	5657	1	57568
1200	20	20287	17854	1	98688
3165	104	3469	1723	1	49344
18155	55	26561	14043	1	172704

Таблица 8 - Результаты тестов модифицированного алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
24988	115	26321	20051	2	147472
28078	92	29927	13442	7	3591032
9617	76	11161	5273	1	106912
1477	11	2237	1434	1	98688
5303	90	6911	98	4	1095376
5066	116	22129	10520	3	5486168
529	46	6359	5657	1	482800
1200	20	20287	17854	1	164480
3165	104	3469	1723	1	427664
18155	55	26561	14043	1	271395

В результате тестов, где g , p и A - 16 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Полига-Хеллмана равно 1.7 мс, а модифицированного алгоритма Полига-Хеллмана равно 2.2 мс. Средняя затраченная память базового алгоритма Полига-Хеллмана равна 470347.2 байт, а модифицированного алгоритма Полига-Хеллмана равна 1187198.7 байт. Базовый алгоритм показал лучше результаты

в скорости выполнения и в затраченной памяти, а модифицированный алгоритм показал лучшие результаты в скорости.

Были сгенерированы параметры и проведены тесты базового (таблица 9) и модифицированного (таблица 10) алгоритма Полига-Хеллмана, где g , p и A - 32 битные числа, а параметр a - 8 битное число.

Таблица 9 - Результаты тестов базового алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
826490941	79	1275360979	886381049	63	692936
907235208	40	1472976761	1403813502	2330	20149296
150735016	118	232048709	183560230	12332	48988592
398609238	44	463302293	181170371	25510	64909360
709577596	8	738854551	429342132	11	2244176
459714223	104	1575821713	1309948980	64	129485832
48998814	115	68156359	30922260	37754	269647128
163526763	65	169925429	161038104	2633	17422024
213970339	108	1504288153	1423746419	38	3617528
827348200	108	984019013	841880618	4962	16211672

Таблица 10 - Результаты тестов модифицированного алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
826490941	79	1275360979	886381049	36	1282536
907235208	40	1472976761	1403813502	2300	2814088
150735016	118	232048709	183560230	11841	4588272
398609238	44	463302293	181170371	24906	1975104
709577596	8	738854551	429342132	16	3686576
459714223	104	1575821713	1309948980	59	1429024
48998814	115	68156359	30922260	37569	234488

Таблица 10 - Результаты тестов модифицированного алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
163526763	65	169925429	161038104	2506	40232
213970339	108	1504288153	1423746419	24	2334904
827348200	108	984019013	841880618	5074	175760

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Полига-Хеллмана равно 8569.7 мс, а модифицированного алгоритма Полига-Хеллмана равно 8433.1 мс. Средняя затраченная память базового алгоритма Полига-Хеллмана равна 57336854.4 байт, а модифицированного алгоритма Полига-Хеллмана равна 1856098.4 байт. Модифицированный алгоритм показал лучшие результаты в скорости выполнения и в затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 11) и модифицированного (таблица 12) алгоритма Полига-Хеллмана, где g , p и A - 32 битные числа, а параметр a - 16 битное число.

Таблица 11 - Результаты тестов базового алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
306074639	9831	550557677	375053531	58	921864
857398933	27336	1644352211	506827146	644	2747936
873747693	19609	2052455927	1442979517	20718	130853712
932095871	15435	1343978191	431999182	19682	2450760
141477928	29584	1705294571	1255511029	111	3749152
3					
406221477	24407	1048450831	883157096	19541	4059032
19992566	3706	21380063	6707478	3024	32183720
40746430	30170	507416659	424602148	6975	14569512

Таблица 11 - Результаты тестов базового алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
171537524	16283	1980316259	1065914095	157	5296416
1					
83622979	15419	750177383	692438560	3213	33511504

Таблица 12 - Результаты тестов модифицированного алгоритма Полига-Хеллмана

g	a	p	A	Время (мс)	Память (байт)
306074639	9831	550557677	375053531	56	4793016
857398933	27336	1644352211	506827146	627	957584
873747693	19609	2052455927	1442979517	20639	411744
932095871	15435	1343978191	431999182	19672	2059240
141477928	29584	1705294571	1255511029	104	2257800
3					
406221477	24407	1048450831	883157096	19492	4213960
19992566	3706	21380063	6707478	3109	152175104
40746430	30170	507416659	424602148	6449	1567296
171537524	16283	1980316259	1065914095	168	666952
1					
83622979	15419	750177383	692438560	3414	54042272

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 16 битное число, среднее время выполнения базового алгоритма Полига-Хеллмана равно 7412.3 мс, а модифицированного алгоритма Полига-Хеллмана равно 7373 мс. Средняя затраченная память базового алгоритма Полига-Хеллмана равна 23034360.8 байт, а модифицированного алгоритма Полига-Хеллмана равна 22314496.8 байт. Модифицированный алгоритм показал лучше результаты в скорости и в затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма Полига-Хеллмана можно сделать вывод, что базовый алгоритм показал лучшие результаты в затраченном времени выполнения и затраченной памяти на маленьких параметрах, где g , p и A - 16 битные числа, а параметр a - 8 битное число. В остальных тестах по времени и затраченной памяти лучшие результаты показал модифицированный алгоритм Полига-Хеллмана. Также базовый и модифицированный алгоритм Полига-Хеллмана показал лучшие результаты в затраченном времени выполнения, где g , p и A - 32 битные числа, а параметр a - 16 битное число, чем при параметрах, где g , p и A - 32 битные числа, а параметр a - 8 битное число. Модифицированный алгоритм показал сильно лучше результаты в затраченной памяти, где g , p и A - 32 битные числа, а параметр a - 8 битное число (рисунок 6, 7).

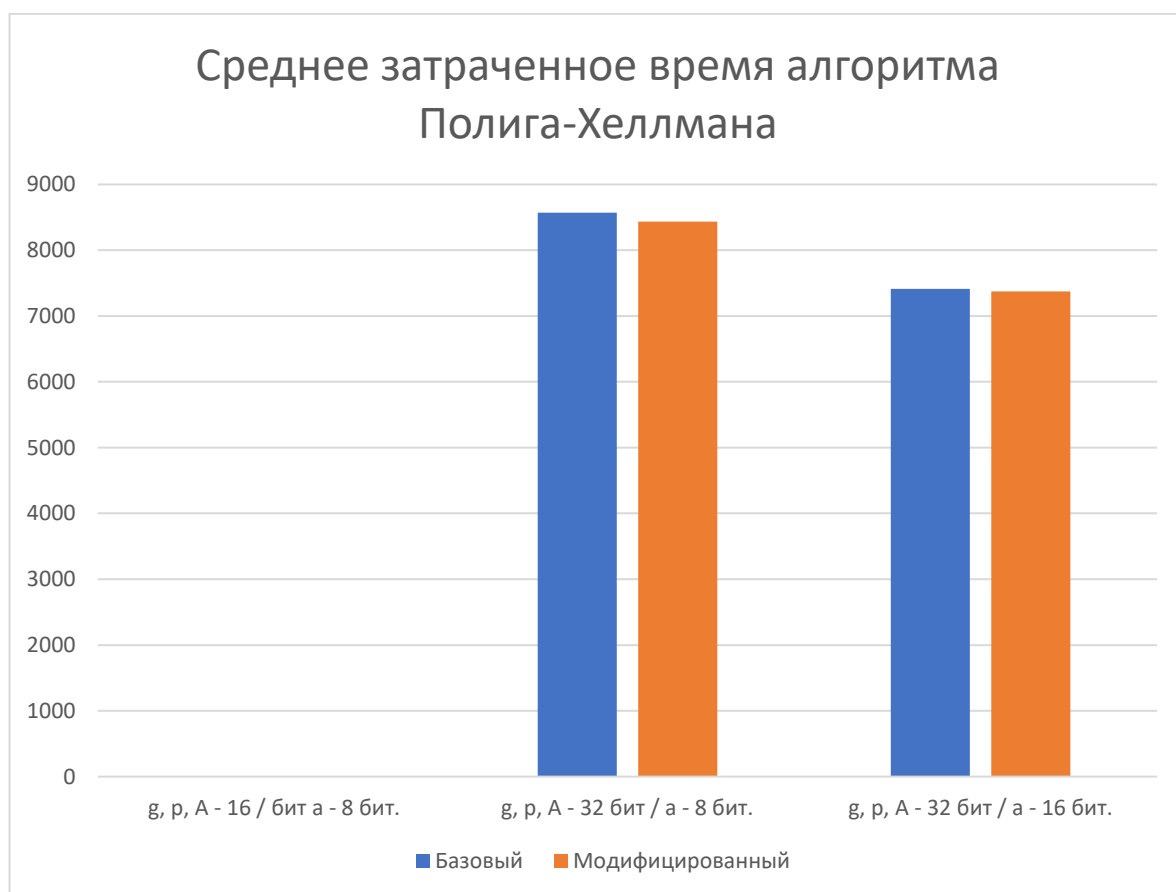


Рисунок 6 - Среднее затраченное время алгоритма Полига-Хеллмана

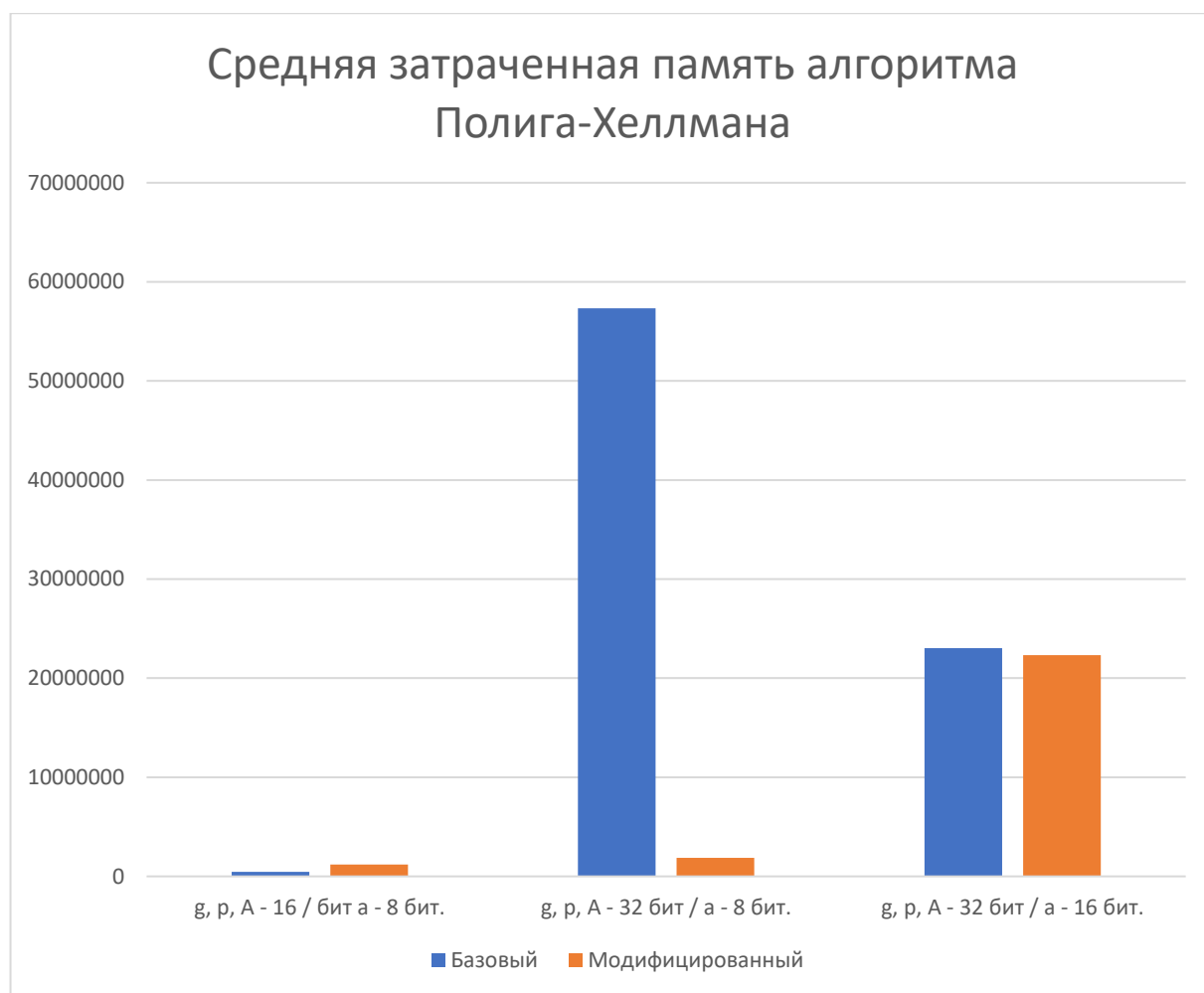


Рисунок 7 - Средняя затраченная память алгоритма Полига-Хеллмана

5. Алгоритм ро-метод Полларда

Были проведены тесты базового и модифицированного алгоритма дискретного логарифмирования ро-метод Полларда для факторизации (разложения на множители) целых чисел. Данный метод базируется на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Наибольшую эффективность данный метод показывает при факторизации составных чисел с достаточно малыми множителями в разложении. Ро-метод Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера n , что может быть проиллюстрировано, расположением чисел в виде греческой буквы ρ , что послужило названием семейству алгоритмов.

Шаги выполнения алгоритма:

- 1) генерируется случайно число x между 1 и $N - 2$;
- 2) инициализируются числа $y = 0, i = 0, stage = 2$;
- 3) в цикле вычисляется $GCD(N, abs(x - y))$ до тех пор, пока не будет равен 1;
- 4) если i равен $stage$, то y присваивается x и $stage$ присваивается $stage * 2$. Далее $x = (x * x + 1)(mod N)$ и $i = i + 1$;
- 5) после завершения цикла на 3 шаге возвращается результат, равный $GCD(N, abs(x - y))$ [7].

Была реализована модификация алгоритма, состоящая в том, что на 4 шаге алгоритма увеличилась степень вычисляемого $x = (x * x * x - 1)(mod N)$. При вычислении x степень полинома увеличилась до 3, так как другие степени полинома не давали конечный результат. Повышение степени полинома увеличит шаг алгоритма, тем самым должна увеличиться скорость нахождения ответа. Теоретическая оценка сложности базового алгоритма $O(n^{1/2})$, а модифицированного $O(n^{1/3})$.

Был сгенерирован параметр и проведены тесты базового (таблица 13) и модифицированного (таблица 14) алгоритма ро-метод Полларада, где N - 64 битное число.

Таблица 13 - Результаты тестов базового алгоритма ро-метод Полларада

N	P	Q	Время (мс)	Память (байт)
1198061138515093319	107	11196833070234517	5	1002
2542692549626073869	47	54099841481405827	2	8224
3353286029619116537	83	40401036501435139	1	1000
1148692865944933531	709	1620159190331359	1	8224
277140607703415601	19	14586347773863979	1	1042
8882060243859981047	17	522474131991763591	2	1021
3401883967797524099	209	16276956783720211	1	1092
793738038913186267	1241	639595518866387	1	1021
7074765594289533221	8543	828135970301947	2	49048
3047154990597365849	401	7598890250866249	1	8224

Таблица 14 - Результаты тестов модифицированного алгоритма ро-метод Полларада

N	P	Q	Время (мс)	Память (байт)
1198061138515093319	107	11196833070234517	1	8224
2542692549626073869	47	54099841481405827	1	24416
3353286029619116537	83	40401036501435139	2	1000
1148692865944933531	709	1620159190331359	1	8224
277140607703415601	19	14586347773863979	2	1042
8882060243859981047	127	69937482235117961	1	1021
3401883967797524099	19	179046524620922321	1	1092
793738038913186267	1241	639595518866387	1	1021
7074765594289533221	8543	828135970301947	2	712672
3047154990597365849	9619	316785007859171	1	40864

В результате тестов, где N - 64 битное число, среднее время выполнения базового алгоритма ро-метод Полларда равно 1.7 мс, а модифицированного алгоритма ро-метод Полларда равно 1.3 мс. Средняя затраченная память базового алгоритма ро-метод Полларда равна 7989.8 байт, а модифицированного алгоритма ро-метод Полларда равна 79957.6 байт. Базовый алгоритм показал лучше результаты в затраченной памяти, а модифицированный алгоритм лучше результаты в скорости.

Был сгенерирован параметр и проведены тесты базового (таблица 15) и модифицированного (таблица 16) алгоритма ро-метод Полларда, где N - 128 битное число.

Таблица 15 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
5304742162021764734016 5842779605199029	827	641444034101785336640 45759104722127	1	8224
6467502114478478304366 4114042046884567	3740311	172913485388741158271 76968450497	4	622464
1512755348614952377181 28655744312609603	6473	233702355726085644551 41148732320811	1	49088
6269857918239220212619 4165118939962221	47	133401232302962132183 3918406785956643	1	8224
7466439770311667276478 1934238110756297	11	678767251846515206952 5630385282796027	1	8224
1066983739667314805769 87119246859208349	545087	195745585506041201820 970082293027	2	230272
1079820523297751389209 2037158655880069	3889	277660201413667109593 5211406185621	2	41120
1404296617156021355463 97235345102518067	7096693	197880423622104176616 34402861319	5	616800

Таблица 15 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
3177088377902487870571 1637938201735687	577	550621902582753530428 27795386831431	1	8224
1422468161301429155145 94836735221133703	107	132940949654339173378 1260156403935829	1	8224

Таблица 16 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
5304742162021764734016 5842779605199029	827	641444034101785336640 45759104722127	4	278336
6467502114478478304366 4114042046884567	3740311	172913485388741158271 76968450497	5	917368
1512755348614952377181 28655744312609603	6473	233702355726085644551 41148732320811	1	139296
6269857918239220212619 4165118939962221	47	133401232302962132183 3918406785956643	1	32640
7466439770311667276478 1934238110756297	11	678767251846515206952 5630385282796027	1	8168
1066983739667314805769 87119246859208349	545087	195745585506041201820 970082293027	8	2702520
1079820523297751389209 2037158655880069	3889	277660201413667109593 5211406185621	1	57568
1404296617156021355463 97235345102518067	7096693	197880423622104176616 34402861319	7	1046152
3177088377902487870571 1637938201735687	577	550621902582753530428 27795386831431	1	16448
1422468161301429155145 94836735221133703	107	132940949654339173378 1260156403935829	2	57568

В результате тестов, где N - 128 битное число, среднее время выполнения базового алгоритма ро-метод Полларда равно 1.9 мс, а модифицированного алгоритма ро-метод Полларда равно 3.1 мс. Средняя

затраченная память базового алгоритма ро-метод Полларда равна 160086.4 байт, а модифицированного алгоритма ро-метод Полларда равна 577610.67 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

Был сгенерирован параметр и проведены тесты базового (таблица 17) и модифицированного (таблица 18) алгоритма ро-метод Полларда, где N - 256 битное число.

Таблица 17 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
5592095948104737821955 7658709962745606169578 0436487079925131839262 52335619753	5813	961998270790424535000 131751418591873493369 654974173541932103628 5266185381	18	20280
7356173465255652220674 1831742753736456343477 0472620626167290655922 9173461517	467	157519774416609255260 689147200757465645275 111450239962776721767 86357973151	1	8224
4878806315577460308885 4278430316470438712906 0612852749404217410556 56164838577	151	323099755998507305224 200519406069340653727 854710498509539216828 183153409527	1	8224
1518487621654552869181 4408742071315766621482 0004343678738465172981 7773968283	19	799204011397133089042 863618003753461401130 631601808835465606173 58830208857	1	1
4906805766317448348063 9231400960337418957239 8514533752403617272230 62622312109	31	158284056977982204776 255585164388185222442 709198236694323747507 1711697493939	1	1

Таблица 17 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
4848086647019923465940 3676777532059900639989 2478083252581413135630 16593623489	1031	470231488556733604843 876593380524344332104 648378354270205056387 61412796919	1	8224
2925124859632571854906 1040098085324119415052 9222822593035355403459 64059110299	97330456 5503	300535409296151687744 982581115426087509505 523798238133779678329 33	6	3027
4203797406551405697656 5180303117273081734747 2073406239206073761463 35258775061	11	382163400595582336150 592548210157028015770 429157642035641885237 6939568979551	1	8176
7611374973571945283956 6824952295618651058852 2743659734979500132399 9114734667	23	330929346677041099302 464456314328776743734 140323330319556304405 391265858029	1	8224
2677482936572229301232 1182202736384890564781 1112540012939004916919 3770643059	23	116412301590096926140 526879142332108219846 874396756527364784746 486685680133	1	8224

Таблица 18 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
5592095948104737821955 7658709962745606169578 0436487079925131839262 52335619753	5813	961998270790424535000 131751418591873493369 654974173541932103628 5266185381	7	769984

Таблица 18 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
7356173465255652220674 1831742753736456343477 0472620626167290655922 9173461517	467	157519774416609255260 689147200757465645275 111450239962776721767 86357973151	1	8224
4878806315577460308885 4278430316470438712906 0612852749404217410556 56164838577	151	323099755998507305224 200519406069340653727 854710498509539216828 183153409527	1	7968
1518487621654552869181 4408742071315766621482 0004343678738465172981 7773968283	19	799204011397133089042 863618003753461401130 631601808835465606173 58830208857	1	8224
4906805766317448348063 9231400960337418957239 8514533752403617272230 62622312109	31	158284056977982204776 255585164388185222442 709198236694323747507 1711697493939	1	1
4848086647019923465940 3676777532059900639989 2478083252581413135630 16593623489	1031	470231488556733604843 876593380524344332104 648378354270205056387 61412796919	2	344128
2925124859632571854906 1040098085324119415052 9222822593035355403459 64059110299	97330456 5503	300535409296151687744 982581115426087509505 523798238133779678329 33	2	344128
4203797406551405697656 5180303117273081734747 2073406239206073761463 35258775061	11	382163400595582336150 592548210157028015770 429157642035641885237 6939568979551	1	8224

Таблица 18 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
7611374973571945283956 6824952295618651058852 2743659734979500132399 9114734667	23	330929346677041099302 464456314328776743734 140323330319556304405 391265858029	1	16448
2677482936572229301232 1182202736384890564781 1112540012939004916919 3770643059	23	116412301590096926140 526879142332108219846 874396756527364784746 486685680133	1	16448

В результате тестов, где N - 256 битное число, среднее время выполнения базового алгоритма ро-метод Полларда равно 3.2 мс, а модифицированного алгоритма ро-метод Полларда равно 1.8 мс. Средняя затраченная память базового алгоритма ро-метод Полларда равна 7260.5 байт, а модифицированного алгоритма ро-метод Полларда равна 152377.7 байт. Базовый алгоритм показал лучше результаты в затраченной памяти, а модифицированный алгоритм показал лучше результаты в скорости.

На основе экспериментов базового и модифицированного алгоритма ро-метод Полларда можно сделать вывод, что базовый алгоритм показал лучше результаты в затраченной памяти, но хуже результаты в затраченном времени выполнения, где N – 64 бит и N – 256 бит (рисунок 8, 9).

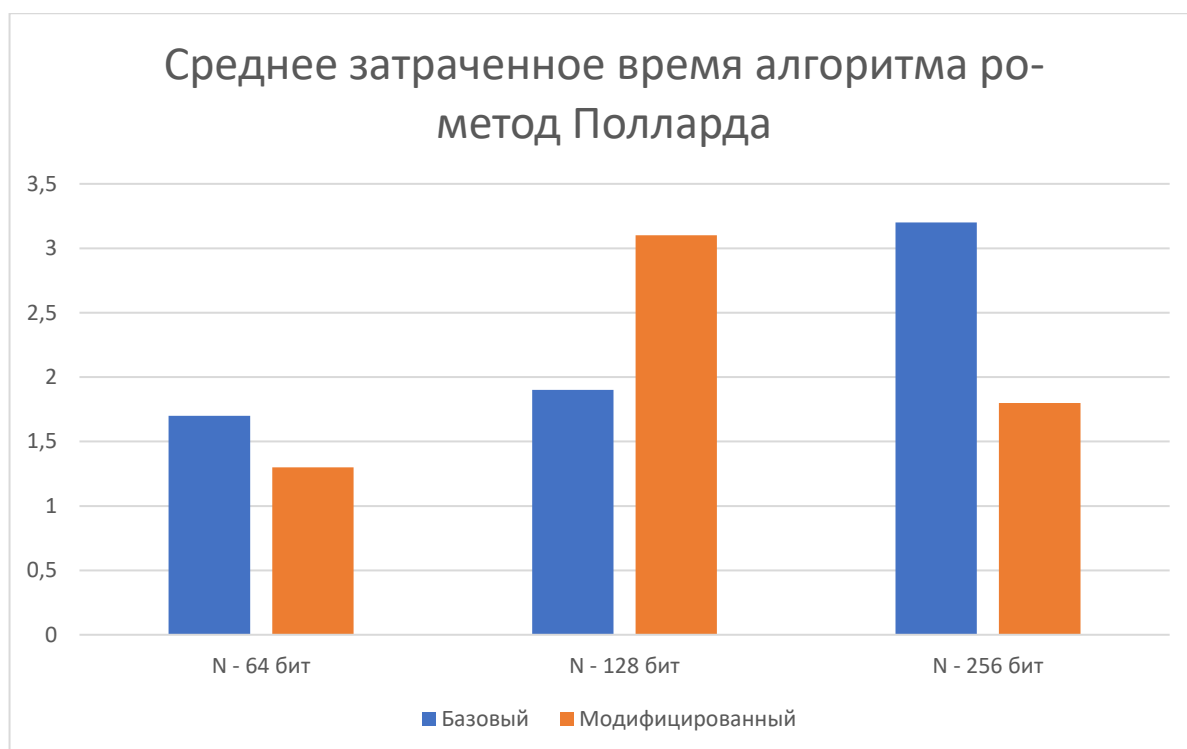


Рисунок 8 - Среднее затраченное время алгоритма ро-метод Полларда

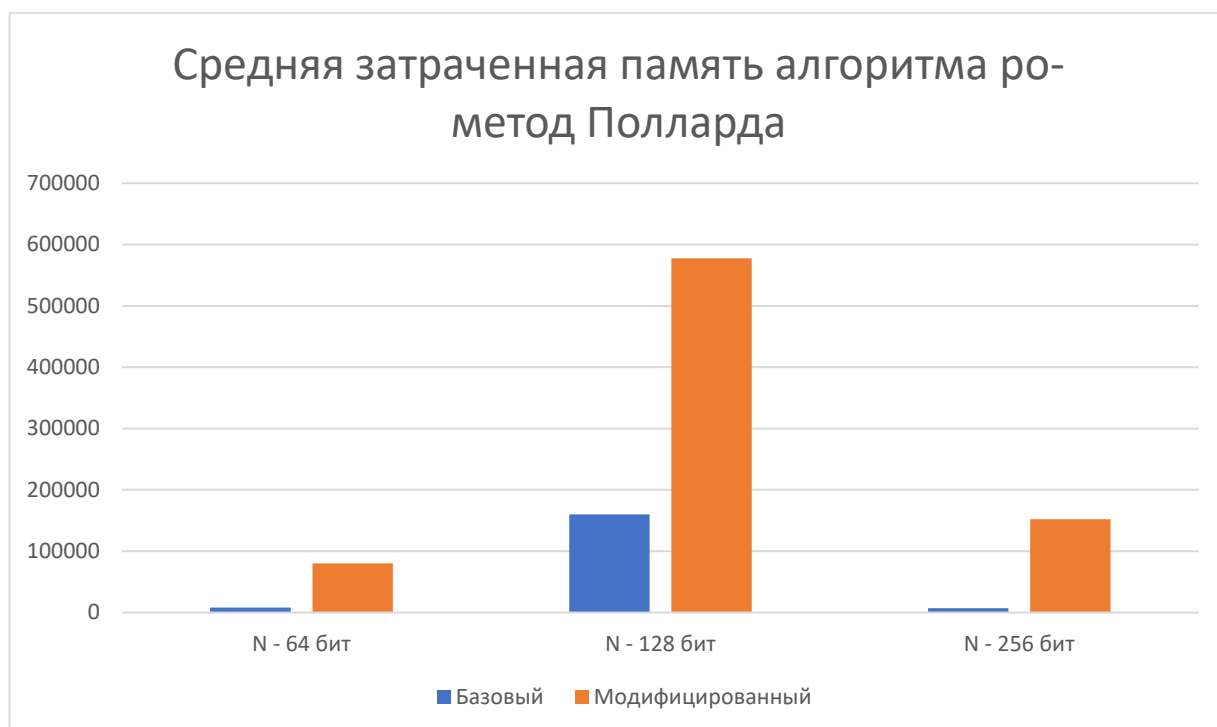


Рисунок 9 - Средняя затраченная память алгоритма ро-метод Полларда

6. Алгоритм Адлемана

Были проведены тесты базового и модифицированного алгоритма Адлемана. Данным алгоритм является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа.

Алгоритм был разработан в 1979 году американским учёным-теоретиком в области компьютерных наук, профессором компьютерных наук и молекулярной биологии в Университете Южной Калифорнии Леонардом Максом Адлеманом. Учёный также знаменит как соавтор системы шифрования RSA и ДНК-вычислений. RSA широко применяется в приложениях компьютерной безопасности, в том числе в протоколе HTTPS.

Пусть задано сравнение $a^x \equiv b \pmod{p}$, необходимо найти натуральное число x , удовлетворяющее данному сравнению.

Описание алгоритма:

1) сформировывается факторная база, состоящая из всех простых чисел q :

$$q \leq B = e^{const \sqrt{\log p \log \log p}};$$

2) с помощью перебора идёт поиск натуральных чисел r_i таких, что

$$a^{r_i} \equiv \prod_{q \leq B} q^{\alpha_{iq}} \pmod{p},$$

то есть $a^{r_i} \pmod{p}$ раскладывается по факторной базе. Отсюда следует, что $r_i \equiv \sum_{q \leq B} \alpha_{iq} \log_a q \pmod{p-1}$;

3) набрав достаточно много соотношений из 2 шага, решается получившаяся система линейных уравнений относительно неизвестных дискретных логарифмов элементов факторной базы $\log_a q$;

4) с помощью некоторого перебора ищется одно значение r , для которого $a^r \equiv \prod_{q \leq B} q^{\beta_q} p_1 * \dots * p_k \pmod{p}$, где $p_1, \dots, p_k \pmod{p}$ – простые числа «средней» величины, то есть $B < p_i < B_1$, где B_1 – также некоторая субэкспоненциальная граница, $B_1 = e^{const \sqrt{\log p \log \log p}}$;

5) с помощью вычислений, аналогичных этапам 2 и 3 ищутся дискретные логарифмы $\log_a p_i$;

б) определяется искомым дискретный логарифм:

$$x \equiv \log_a b \equiv \sum_{q \leq B} \beta_q \log_a q + \sum_{i=1}^k \log_a p_i \pmod{p-1} \text{ [8].}$$

Была реализована модификация алгоритма, состоящая в том, что на 1 шаге алгоритма был изменён показатель степени при вычислении числа $q \leq B = e^{const \sqrt{\log p \log \log \log p}}$, тем самым понизив факторную базу. Понижение факторной базы должно уменьшить количество вычисляемых логарифмов, тем самым уменьшив выделяемую память на хранение логарифмов, а также увеличить скорость вычислений логарифмов за счёт их меньшего количества. Теоретическая оценка сложности базового алгоритма $O\left(e^{const \sqrt{\log p \log \log \log p}}\right)$, а модифицированного $O\left(e^{const \sqrt{\log p \log \log \log p}}\right)$.

Были сгенерированы параметры и проведены тесты базового (таблица 19) и модифицированного (таблица 20) алгоритма Адлемана, где g , p и A - 16 битные числа, а параметр a - 8 битное число.

Таблица 19 - Результаты тестов базового алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
2218	16	4831	3914	1805	3355432
14600	68	15313	14888	919	1893984
14150	5	15187	307	151	780608
3246	30	14969	11720	237	585632
778	125	971	385	390	3022056
1141	74	9377	6705	569	1990576
5379	37	8501	4714	663	1134792
1172	124	2017	1342	332	2914776
1768	67	10567	346	371	531968
1513	61	4919	329	792	883392

Таблица 20 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
2218	16	4831	3914	234	4321
14600	68	15313	14888	245	3984
14150	5	15187	307	151	80608
3246	30	14969	11720	237	85632
778	125	971	385	390	22056
1141	74	9377	6705	569	90576
5379	37	8501	4714	633	34792
1172	124	2017	1342	332	914776
1768	67	10567	346	715	531968
1513	61	4919	329	792	83392

В результате тестов, где g , p и A - 16 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Адлемана равно 622.9 мс, а модифицированного алгоритма Адлемана равно 808.4 мс. Средняя затраченная память базового алгоритма Адлемана равна 1709321.6 байт, а модифицированного алгоритма Адлемана равна 1185210.5 байт. Модифицированный алгоритм показал лучше результаты в затраченной памяти, но хуже в затраченной скорости.

Были сгенерированы параметры и проведены тесты базового (таблица 21) и модифицированного (таблица 22) алгоритма Адлемана, где g , p и A - 32 битные числа, а параметр a - 8 битное число.

Таблица 21 - Результаты тестов базового алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	16366	463276208
613349514	33	1804711613	295175335	10600	389309800

Таблица 21 - Результаты тестов базового алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
978926293	110	1756245157	297068444	10066	199945824
122208609 6	29	1730829689	1242325950	10248	199696600
416986947	24	1964834371	1037606313	9943	186434560
167731978 7	71	2130571447	1730147609	8778	192809200
530781409	19	582762727	564926713	16655	266850216
126602516 6	2	1532873141	1195035467	9627	260166072
172653322	5	1636100959	90734147	9736	197808344
33886891	101	986077949	465041982	12620	33046992

Таблица 22 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	3664	2762083
613349514	33	1804711613	295175335	6002	3098004
978926293	110	1756245157	297068444	0667	58244
122208609 6	29	1730829689	1242325950	2486	9966002
416986947	24	1964834371	1037606313	4393	8345609
167731978 7	71	2130571447	1730147609	7834	9092005
530781409	19	582762727	564926713	6557	6502162
126602516 6	2	1532873141	1195035467	2753	6660722

Таблица 22 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
172653322	5	1636100959	90734147	3682	9083445
33886891	101	986077949	465041982	6202	369921

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма Адлемана равно 11463.9 мс, а модифицированного алгоритма Адлемана равно 4424 мс. Средняя затраченная память базового алгоритма Адлемана равна 238934381.6 байт, а модифицированного алгоритма Адлемана равна 178085915.2 байт. Модифицированный алгоритм показал лучше результаты в скорости и затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 23) и модифицированного (таблица 24) алгоритма Адлемана, где g , p и A - 32 битные числа, а параметр a - 16 битное число.

Таблица 23 - Результаты тестов базового алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	10832	386754688
741729452	23435	960977657	715804369	13863	58502592
75815191	16441	156558379	62110094	44132	133608424
544600416	15960	647216441	87116461	18224	117448144
356089196	13619	875934517	429988046	12820	17794272
295703380	27231	1312727173	855763467	12953	31402952
884246627	17629	888771061	590525393	13197	191968960
499181459	17394	533090533	468448650	19121	266747832

Таблица 23 - Результаты тестов базового алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
122434103 6	23668	1263813263	701281449	12616	68905200
137991201 2	31962	1470874637	1315017822	11040	2298848

Таблица 24 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	10325	3546882
741729452	23435	960977657	715804369	13633	525925
75815191	16441	156558379	62110094	24328	1084241
544600416	15960	647216441	87116461	18243	181442
356089196	13619	875934517	429988046	12202	142724
295703380	27231	1312727173	855763467	12538	329526
884246627	17629	888771061	590525393	13973	189602
499181459	17394	533090533	468448650	19214	278323
122434103 6	23668	1263813263	701281449	12167	652005
137991201 2	31962	1470874637	1315017822	11408	28482

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 16 битное число, среднее время выполнения базового алгоритма Адлемана равно 16879.8 мс, а модифицированного алгоритма Адлемана равно 18803.1 мс. Средняя затраченная память базового алгоритма Адлемана равна 127543191.2 байт, а модифицированного алгоритма Адлемана равна 121695915.2 байт. Модифицированный алгоритм показал лучше результаты в затраченной памяти, но показал хуже результаты в скорости выполнения.

На основе экспериментов базового и модифицированного алгоритма Адлемана можно сделать вывод, что модифицированный алгоритм показал лучшие результаты во всех тестах, кроме скорости выполнения, где g , p и A - 16 битные числа, а параметр a - 8 битное число, и где g , p и A - 32 битные числа, а параметр a - 16 битное число (рисунок 10, 11).

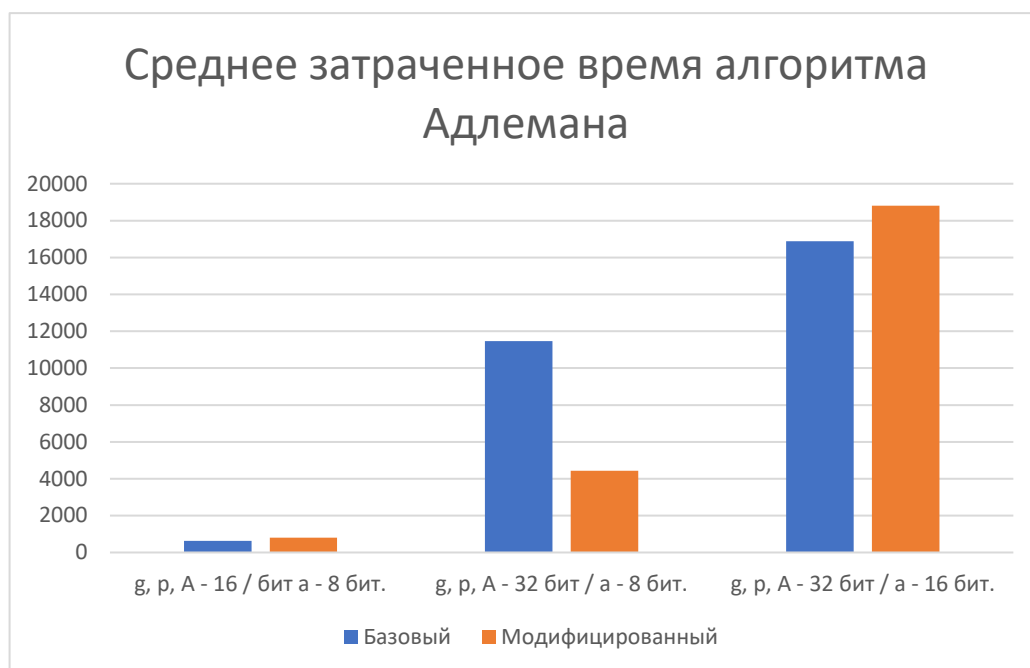


Рисунок 10 - Среднее затраченное время алгоритма Адлемана

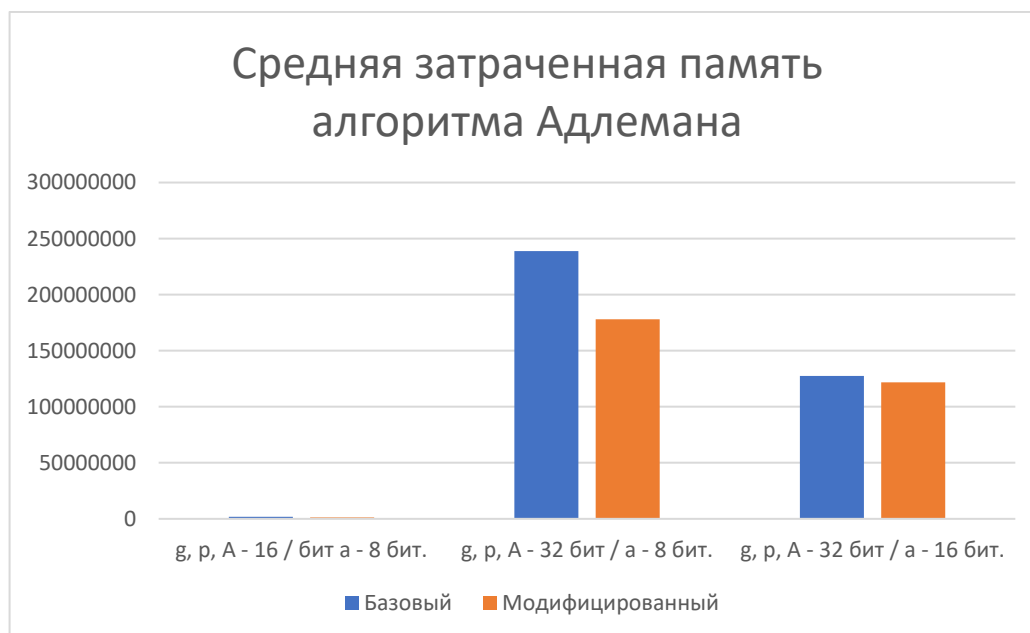


Рисунок 11 - Средняя затраченная память алгоритма Адлемана

7. Алгоритм COS

Были проведены тесты базового и модифицированного алгоритма COS. Алгоритм COS является субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа. Данный алгоритм был создан учёными Копперсмитом, Одлыжко и Шреппелем.

Пусть задано сравнение $a^x \equiv b \pmod{p}$, необходимо найти натуральное число x , удовлетворяющее данному сравнению.

Описание алгоритма:

1) задаётся $H = \lfloor p^{1/2} \rfloor + 1, J = H^2 - p > 0$. Сформируется множество $S = \{q \mid q - \text{простое}, q < L^{1/2}\} \cup \{H + c \mid 0 < c < L^{\frac{1}{2}+\varepsilon}\}$, где L и ε – простые величины, $L = L_p \left[\frac{1}{2}; 1 \right], 0 < \varepsilon < 1$;

2) с помощью некоторого просеивания идёт поиск пары целых чисел c_1, c_2 таких, что $0 < c_i < L^{\frac{1}{2}+\varepsilon}, i = 1, 2$, и абсолютно наименьший вычет элемента $(H + c_1)(H + c_2) \pmod{p}$ гладок по отношению к границе гладкости $L^{1/2}$, т.е.

$$(H + c_1)(H + c_2) \equiv \prod_{\substack{q < L^{\frac{1}{2}}, \\ q - \text{простое}}} q^{\alpha_q(c_1, c_2)} \pmod{p}.$$

При этом, поскольку $J = O(p^{1/2})$, то

$(H + c_1)(H + c_2) \equiv J + (c_1 + c_2)H + c_1c_2 \pmod{p}$, причём абсолютно наименьший вычет в этом классе вычетов равен $J + (c_1 + c_2)H + c_1c_2$ и имеет величину $O(p^{\frac{1}{2}+\varepsilon})$. Поэтому вероятность его гладкости выше, чем для произвольных чисел на отрезке $[1, p - 1]$. Логарифмируя по основанию a , получается соотношение

$$\log_a(H + c_1) + \log_a(H + c_2) \equiv \sum_{\substack{q < L^{\frac{1}{2}}, \\ q - \text{простое}}} \alpha_q(c_1, c_2) \log_a q \pmod{p - 1}.$$

Это однородное уравнение относительно неизвестных величин $\log_a(H + c), \log_a q$. Можно считать, что a также является $L^{1/2}$ – гладким, $a = \prod_{q < L^{1/2}} q^{\beta_q}$, откуда получим неоднородное уравнение

$$1 \equiv \sum_q \beta_q \log_a q \pmod{p-1};$$

3) набрав на 2-м этапе достаточно много уравнений, решается получившаяся система линейных уравнений в кольце $\mathbb{Z}/(p-1)\mathbb{Z}$ и находятся значения $\log_a(H+c), \log_a q$;

4) для нахождения конкретного логарифма $x = \log_a b$ вводится новая граница гладкости L^2 . Случайным перебором находится одно ω значение такое, что

$$a^\omega b \equiv \prod_{\substack{q < L^{\frac{1}{2}}, \\ q - \text{простое}}} q^{g_q} \prod_{\substack{\frac{1}{L^2} \leq u < L^2, \\ u - \text{простое}}} u^{h_u} \pmod{p}.$$

В этом соотношении участвуют несколько новых простых чисел u средней величины;

5) с помощью методов, аналогичных 2 и 3 этапам, находятся логарифмы нескольких простых чисел u средней величины, возникших на 4 этапе;

б) вычисляется ответ

$$x = \log_a b \equiv -\omega + \sum_{\substack{q < L^{\frac{1}{2}}, \\ q - \text{простое}}} g_q \log_a q + \sum_u h_u \log_a u \pmod{p-1}.$$

Конец алгоритма [9].

Была реализована модификация алгоритма, состоящая в том, что на 2 шаге в уравнении $(H+c_1)(H+c_2) \equiv \prod_{\substack{q < L^{\frac{1}{2}}, \\ q - \text{простое}}} q^{\alpha_q(c_1, c_2)} \pmod{p}$ было разложено на простые множители и данные простые множители были возведены в свои степени, чтобы на 2 шаге была составлена таблица из единичных значений без степеней. Составление таблицы без степеней на 2 шаге должно уменьшить количество хранимых чисел, тем самым уменьшив выделяемую память, а также увеличить скорость вычисления таблицы за счёт меньшего количества чисел. После вычислений 2 шага алгоритма, теоретическая оценка сложности 3 шага базового алгоритма $O(n^2)$, а модифицированного $O(n)$.

Были сгенерированы параметры и проведены тесты базового (таблица 25) и модифицированного (таблица 26) алгоритма COS, где g , p и A - 16 битные числа, а параметр a - 8 битное число.

Таблица 25 - Результаты тестов базового алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
2218	16	4831	3914	4805	2355432
14600	68	15313	14888	519	1793984
14150	5	15187	307	751	560608
3246	30	14969	11720	337	355632
778	125	971	385	690	5322056
1141	74	9377	6705	269	1590576
5379	37	8501	4714	463	934792
1172	124	2017	1342	232	3114776
1768	67	10567	346	271	431968
1513	61	4919	329	692	783392

Таблица 26 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
2218	16	4831	3914	134	384321
14600	68	15313	14888	145	27984
14150	5	15187	307	251	34608
3246	30	14969	11720	4237	475632
778	125	971	385	190	18056
1141	74	9377	6705	169	39576
5379	37	8501	4714	2663	18792
1172	124	2017	1342	232	27776
1768	67	10567	346	471	2768
1513	61	4919	329	472	2592

В результате тестов, где g , p и A - 16 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма COS равно 902.9 мс, а модифицированного алгоритма COS равно 896.4 мс. Средняя затраченная память базового алгоритма COS равна 1724321.6 байт, а модифицированного алгоритма COS равна 103210.5 байт. Модифицированный алгоритм показал лучшие результаты в скорости и затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 27) и модифицированного (таблица 28) алгоритма COS, где g , p и A - 32 битные числа, а параметр a - 8 битное число.

Таблица 27 - Результаты тестов базового алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	24366	463276208
613349514	33	1804711613	295175335	12600	639309800
978926293	110	1756245157	297068444	350066	369945824
122208609	29	1730829689	1242325950	63248	729696600
6					
416986947	24	1964834371	1037606313	3543	836434560
167731978	71	2130571447	1730147609	7478	1692809200
7					
530781409	19	582762727	564926713	84655	216850216
126602516	2	1532873141	1195035467	9427	830166072
6					
172653322	5	1636100959	90734147	8536	957808344
33886891	101	986077949	465041982	95620	36046992

Таблица 28 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	28466	85276208
613349514	33	1804711613	295175335	47490	84309800
978926293	110	1756245157	297068444	27266	84945824
122208609	29	1730829689	1242325950	38948	97696600
6					
416986947	24	1964834371	1037606313	55443	34434560
167731978	71	2130571447	1730147609	16788	8509200
7					
530781409	19	582762727	564926713	27655	46850216
126602516	2	1532873141	1195035467	4827	48166072
6					
172653322	5	1636100959	90734147	52436	3908344
33886891	101	986077949	465041982	28620	976992

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма COS равно 65953.9 мс, а модифицированного алгоритма COS равно 72793.9 мс. Средняя затраченная память базового алгоритма COS равна 677234381.6 байт, а модифицированного алгоритма COS равна 449507381.6 байт. Модифицированный алгоритм показал лучшие результаты в затраченной памяти, но хуже в затраченном времени выполнения.

Были сгенерированы параметры и проведены тесты базового (таблица 29) и модифицированного (таблица 30) алгоритма COS, где g , p и A - 32 битные числа, а параметр a - 16 битное число.

Таблица 29 - Результаты тестов базового алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	52832	326754688
741729452	23435	960977657	715804369	52863	63502592
75815191	16441	156558379	62110094	72132	423608424
544600416	15960	647216441	87116461	26224	467448144
356089196	13619	875934517	429988046	73820	84794272
295703380	27231	1312727173	855763467	72953	98402952
884246627	17629	888771061	590525393	73197	391968960
499181459	17394	533090533	468448650	63121	566747832
122434103	23668	1263813263	701281449	46616	68905200
6					
137991201	31962	1470874637	1315017822	23040	6598848
2					

Таблица 30 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	73832	73854688
741729452	23435	960977657	715804369	55863	452592
75815191	16441	156558379	62110094	22132	3108424
544600416	15960	647216441	87116461	37524	2148144
356089196	13619	875934517	429988046	13820	414272
295703380	27231	1312727173	855763467	24953	532952
884246627	17629	888771061	590525393	36197	2168960
499181459	17394	533090533	468448650	44121	1247832

Таблица 30 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
122434103 6	23668	1263813263	701281449	11616	465200
137991201 2	31962	1470874637	1315017822	22040	62848

В результате тестов, где g , p и A - 32 битные числа, а параметр a - 8 битное число, среднее время выполнения базового алгоритма COS равно 55679.8 мс, а модифицированного алгоритма COS равно 84209.8 мс. Средняя затраченная память базового алгоритма COS равна 249873191.2 байт, а модифицированного алгоритма COS равна 228445591.2 байт. Модифицированный алгоритм показал лучше результаты в затраченной памяти, но хуже в скорости выполнения.

На основе экспериментов базового и модифицированного алгоритма COS можно сделать вывод, что модифицированный алгоритм показал лучше результаты во всех тестах, кроме скорости выполнения, где g , p и A - 32 битные числа, а параметр a - 8 битное число, и где g , p и A - 32 битные числа, а параметр a - 16 битное число (рисунок 12, 13).

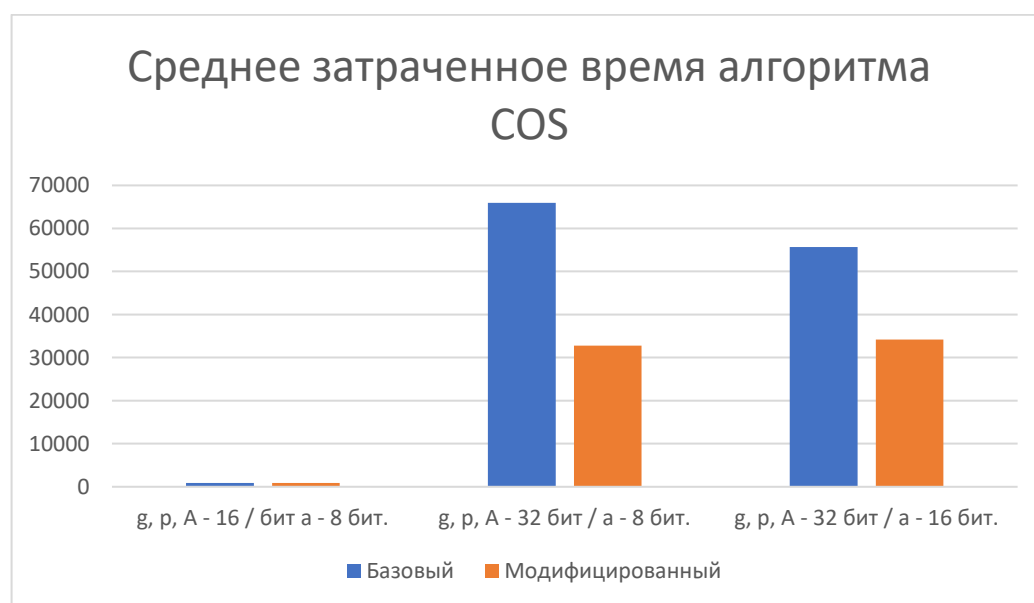


Рисунок 12 - Среднее затраченное время алгоритма COS

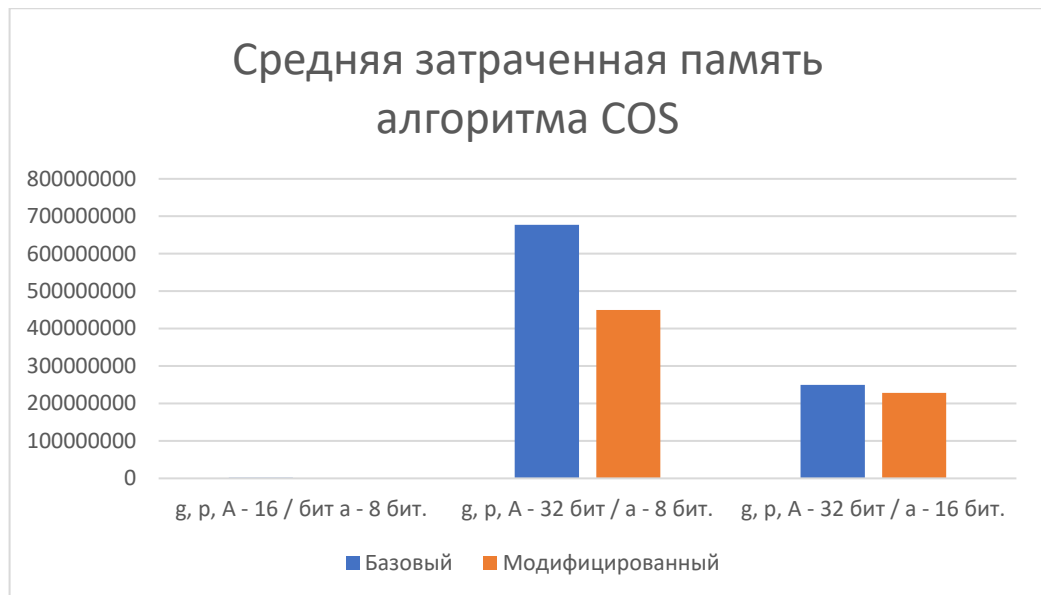


Рисунок 13 - Средняя затраченная память алгоритма COS

8. Алгоритм решето числового поля

Были проведены тесты базового и модифицированного алгоритма решета числового поля. Данный алгоритм является методом факторизации целых чисел.

Алгоритм описан английским математиком Джоном Поллардом в 1988 году.

Данный алгоритм является усовершенствованной версией метода рационального решета или метода квадратичного решета. Для реализации аналогичных алгоритмов необходимо вычисление гладких чисел порядка \sqrt{n} размер которых экспоненциально растёт с ростом n . Для алгоритма решета числового поля же необходимо вычисление гладких чисел субэкспоненциального относительно n размера, то есть эти числа меньше. Вероятность того, что числа такого размера окажутся гладкими выше, благодаря чему данный метод и является более эффективным.

Описание алгоритма:

- 1) пусть n - нечетное составное число, которое требуется факторизовать;
- 2) выбирается степень неприводимого многочлена $d \geq 3$ (при $d = 2$ не будет выигрыша в сравнении с методом квадратичного решета);
- 3) выбирается целое m такое, что $[n^{1/(d+1)}] < m < [n^{1/d}]$, и раскладывается n по основанию m :

$$n = m^d + a_{d-1}m^{d-1} + \dots + a_0;$$

- 4) связывается с разложением из 3 шага неприводимый в кольце $\mathbb{Z}[x]$ полиномов с целыми коэффициентами многочлен

$$f_1(a, b) = b^d * f_1\left(\frac{a}{b}\right) = a^d + a_{d-1}a^{d-1}b + a_{d-2}a^{d-2}b^2 + \dots + a_0b^d;$$

- 5) определяется полином просеивания $F_1(a, b)$ как однородный многочлен от двух переменных a и b :

$$F_1(a, b) = b^d * f_1\left(\frac{a}{b}\right) = a^d + a_{d-1}a^{d-1}b + a_{d-2}a^{d-2}b^2 + \dots + a_0b^d;$$

- 6) определяется второй полином $f_2 = x - m$ и соответствующий однородный многочлен $F_2(a, b) = a - bm$;

7) выбираются два положительных числа L_1 и L_2 , определяющих область просеивания:

$$SR = \{1 \leq b \leq L_1, -L_1 \leq a \leq L_2\};$$

8) пусть θ — корень $f_1(x)$. Рассматривается кольцо полиномов $\mathbb{Z}[\theta]$. Определяется множество, называемое алгебраической факторной базой FB_1 , состоящее из многочленов первого порядка вида $a - b\theta$ с нормой шага 5, являющейся простым числом. Данные многочлены — простые неразложимые в кольце алгебраических целых поля $K = \mathbb{Q}[\theta]$. Ограничиваются абсолютные значения норм полиномов из FB_1 константой B_1 ;

9) определяется рациональная факторная база FB_2 , состоящая из всех простых чисел, ограниченных сверху константой B_2 ;

10) определяется множество FB_3 , называемое факторной базой квадратичных характеров. Данное множество полиномов первого порядка $c - d\theta$, норма которых — простое число. Должно выполняться условие $FB_1 \cap FB_3 = \emptyset$;

11) выполняется просеивание многочленов $\{a - b\theta | (a, b) \in SR\}$ по факторной базе FB_1 и целых чисел $\{a - bm | (a, b) \in SR\}$ по факторной базе FB_2 . В результате получается множество M , состоящее из гладких пар (a, b) , то есть таких пар (a, b) , что $\text{НОД}(a, b) = 1$, полином и число $a - b\theta$ и $a - bm$ раскладываются полностью по FB_1 и FB_2 соответственно;

12) находится такое подмножество $S \subseteq M$, что

$$\prod_{(a,b) \in S} \text{Nr}(a - b\theta) = H^2, H \in \mathbb{Z}; \quad \prod_{(a,b) \in S} (a - bm) = B^2, B \in \mathbb{Z};$$

13) определяется многочлен

$$g(\theta) = f_1'^2(\theta) * \prod_{(a,b) \in S} (a - bm), \text{ где } f_1'^2(x) - \text{производная } f_1(x);$$

14) многочлен $g(\theta)$ является полным квадратом в кольце полиномов $\mathbb{Z}(\theta)$. Пусть тогда $\alpha(\theta)$ есть корень из $g(\theta)$ и B — корень из B^2 ;

15) строится отображение $\phi: \theta \rightarrow m$, заменяя полином $\alpha(\theta)$ числом $\alpha(m)$. Данное отображение является кольцевым гомоморфизмом кольца алгебраических целых чисел \mathbb{Z}_K в кольцо \mathbb{Z} , откуда получаем соотношение:

$$A^2 = g(m)^2 \equiv \phi(g(\alpha))^2 \equiv \phi(f_1'^2(\theta) * \prod_{(a,b) \in S} (a - b\theta)) \equiv f_1'^2(m) * \prod_{(a,b) \in S} (a - bm) \equiv f_1'^2(m) * C^2 \pmod{n};$$

16) пусть $B = f'(m) * C$. Находится пара чисел (A, B) таких, что $A \equiv B \pmod{n}$. Тогда находится делитель числа n , вычисляя НОД($n, A \pm B$) [10].

Была реализована модификация алгоритма, состоящая в том, что на 2 шаге алгоритма выбирается степень неприводимого многочлена, равное количеству байт входного числа n . Таким образом степень полинома будет выбираться не случайным образом и более эффективно, тем самым повышая скорость вычисления алгоритма. После вычислений 2 шага алгоритма, теоретическая оценка сложности 3 шага базового алгоритма $O(n^{1/d})$, а модифицированного $O(n^{8/n})$.

Был сгенерирован параметр и проведены тесты базового (таблица 31) и модифицированного (таблица 32) алгоритма решето числового поля, где N - 64 битное число.

Таблица 31 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
1198061138515093319	107	11196833070234517	55236	100243
2542692549626073869	47	54099841481405827	23526	822474
3353286029619116537	83	40401036501435139	16487	100027
1148692865944933531	709	1620159190331359	52364	822484
277140607703415601	19	14586347773863979	87457	104285
8882060243859981047	17	522474131991763591	28456	102134
3401883967797524099	209	16276956783720211	25474	109287
793738038913186267	1241	639595518866387	83467	102195
7074765594289533221	8543	828135970301947	28546	490486
3047154990597365849	401	7598890250866249	56586	822495

Таблица 32 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
1198061138515093319	107	11196833070234517	75236	822445
2542692549626073869	47	54099841481405827	33526	244165
3353286029619116537	83	40401036501435139	26487	100085
1148692865944933531	709	1620159190331359	82364	822449
277140607703415601	19	14586347773863979	37457	104246
8882060243859981047	127	69937482235117961	98456	102139
3401883967797524099	19	179046524620922321	45474	109257
793738038913186267	1241	639595518866387	53467	102184
7074765594289533221	8543	828135970301947	98546	712672
3047154990597365849	9619	316785007859171	66586	408648

В результате тестов, где N - 64 битное число, среднее время выполнения базового алгоритма решето числового поля равно 45759.9 мс, а модифицированного алгоритма решето числового поля равно 61759.9 мс. Средняя затраченная память базового алгоритма решето числового поля равна 357611 байт, а модифицированного алгоритма решето числового поля равна 352829 байт. Базовый алгоритм показал лучше результаты скорости, а модифицированный алгоритм лучше результаты в затраченной памяти.

Был сгенерирован параметр и проведены тесты базового (таблица 33) и модифицированного (таблица 34) алгоритма решето числового поля, где N - 128 битное число.

Таблица 33 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5304742162021764734016 5842779605199029	827	641444034101785336640 45759104722127	355236	4100243
6467502114478478304366 4114042046884567	3740311	172913485388741158271 76968450497	723526	8522474
1512755348614952377181 28655744312609603	6473	233702355726085644551 41148732320811	316487	1080027
6269857918239220212619 4165118939962221	47	133401232302962132183 3918406785956643	852364	8252484
7466439770311667276478 1934238110756297	11	678767251846515206952 5630385282796027	487457	1084285
1066983739667314805769 87119246859208349	545087	195745585506041201820 970082293027	928456	1092134
1079820523297751389209 2037158655880069	3889	277660201413667109593 5211406185621	245474	1059287
1404296617156021355463 97235345102518067	7096693	197880423622104176616 34402861319	863467	1042195
3177088377902487870571 1637938201735687	577	550621902582753530428 27795386831431	288546	4980486
1422468161301429155145 94836735221133703	107	132940949654339173378 1260156403935829	546586	8242495

Таблица 34 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5304742162021764734016 5842779605199029	827	641444034101785336640 45759104722127	755236	3822445
6467502114478478304366 4114042046884567	3740311	172913485388741158271 76968450497	336526	6244165

Таблица 34 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
1512755348614952377181 28655744312609603	6473	233702355726085644551 41148732320811	264787	7100085
6269857918239220212619 4165118939962221	47	133401232302962132183 3918406785956643	823364	8322449
7466439770311667276478 1934238110756297	11	678767251846515206952 5630385282796027	374757	7104246
1066983739667314805769 87119246859208349	545087	195745585506041201820 970082293027	983456	1802139
1079820523297751389209 2037158655880069	3889	277660201413667109593 5211406185621	475474	1409257
1404296617156021355463 97235345102518067	7096693	197880423622104176616 34402861319	353467	1802184
3177088377902487870571 1637938201735687	577	550621902582753530428 27795386831431	798546	2712672
1422468161301429155145 94836735221133703	107	132940949654339173378 1260156403935829	866586	6408648

В результате тестов, где N - 128 битное число, среднее время выполнения базового алгоритма решето числового поля равно 560759.9 мс, а модифицированного алгоритма решето числового поля равно 503219.9 мс. Средняя затраченная память базового алгоритма решето числового поля равна 3945611 байт, а модифицированного алгоритма решето числового поля равна 3672829 байт. Модифицированный алгоритм показал лучше результаты в скорости и затраченной памяти.

Был сгенерирован параметр и проведены тесты базового (таблица 35) и модифицированного (таблица 36) алгоритма решето числового поля, где N - 256 битное число.

Таблица 35 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5592095948104737821955 7658709962745606169578 0436487079925131839262 52335619753	5813	961998270790424535000 131751418591873493369 654974173541932103628 5266185381	5355236	41050243
7356173465255652220674 1831742753736456343477 0472620626167290655922 9173461517	467	157519774416609255260 689147200757465645275 111450239962776721767 86357973151	7723526	85272474
4878806315577460308885 4278430316470438712906 0612852749404217410556 56164838577	151	323099755998507305224 200519406069340653727 854710498509539216828 183153409527	3126487	10880027
1518487621654552869181 4408742071315766621482 0004343678738465172981 7773968283	19	799204011397133089042 863618003753461401130 631601808835465606173 58830208857	8524364	82542484
4906805766317448348063 9231400960337418957239 8514533752403617272230 62622312109	31	158284056977982204776 255585164388185222442 709198236694323747507 1711697493939	4874757	10884285
4848086647019923465940 3676777532059900639989 2478083252581413135630 16593623489	1031	470231488556733604843 876593380524344332104 648378354270205056387 61412796919	9238456	10924134
2925124859632571854906 1040098085324119415052 9222822593035355403459 64059110299	97330456 5503	300535409296151687744 982581115426087509505 523798238133779678329 33	2745474	10589287

Таблица 35 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
4203797406551405697656 5180303117273081734747 2073406239206073761463 35258775061	11	382163400595582336150 592548210157028015770 429157642035641885237 6939568979551	8683467	10424195
7611374973571945283956 6824952295618651058852 2743659734979500132399 9114734667	23	330929346677041099302 464456314328776743734 140323330319556304405 391265858029	2884546	49805486
2677482936572229301232 1182202736384890564781 1112540012939004916919 3770643059	23	116412301590096926140 526879142332108219846 874396756527364784746 486685680133	5486586	82428495

Таблица 36 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5592095948104737821955 7658709962745606169578 0436487079925131839262 52335619753	5813	961998270790424535000 131751418591873493369 654974173541932103628 5266185381	7455236	53822445
7356173465255652220674 1831742753736456343477 0472620626167290655922 9173461517	467	157519774416609255260 689147200757465645275 111450239962776721767 86357973151	6336526	67244165
4878806315577460308885 4278430316470438712906 0612852749404217410556 56164838577	151	323099755998507305224 200519406069340653727 854710498509539216828 183153409527	7264787	78100085

Таблица 36 - Результаты тестов модифицированного алгоритма решето
числового поля

N	P	Q	Время (мс)	Память (байт)
1518487621654552869181 4408742071315766621482 0004343678738465172981 7773968283	19	799204011397133089042 863618003753461401130 631601808835465606173 58830208857	9823364	83292449
4906805766317448348063 9231400960337418957239 8514533752403617272230 62622312109	31	158284056977982204776 255585164388185222442 709198236694323747507 1711697493939	4374757	75104246
4848086647019923465940 3676777532059900639989 2478083252581413135630 16593623489	1031	470231488556733604843 876593380524344332104 648378354270205056387 61412796919	9783456	71802139
2925124859632571854906 1040098085324119415052 9222822593035355403459 64059110299	97330456 5503	300535409296151687744 982581115426087509505 523798238133779678329 33	6475474	71409257
4203797406551405697656 5180303117273081734747 2073406239206073761463 35258775061	11	382163400595582336150 592548210157028015770 429157642035641885237 6939568979551	7353467	87902184
7611374973571945283956 6824952295618651058852 2743659734979500132399 9114734667	23	330929346677041099302 464456314328776743734 140323330319556304405 391265858029	7898546	29712672
2677482936572229301232 1182202736384890564781 1112540012939004916919 3770643059	23	116412301590096926140 526879142332108219846 874396756527364784746 486685680133	8666586	66408648

В результате тестов, где N - 256 битное число, среднее время выполнения базового алгоритма решето числового поля равно 5864289.9 мс, а модифицированного алгоритма решето числового поля равно 4543219.9 мс.

Средняя затраченная память базового алгоритма решето числового поля равна 39480111 байт, а модифицированного алгоритма решето числового поля равна 38479829 байт. Модифицированный алгоритм показал лучшие результаты в скорости и затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма решето числового поля можно сделать вывод, что модифицированный алгоритм показал лучшие результаты во всех тестах, кроме скорости на 64 битном входном параметре. (рисунок 14, 15).

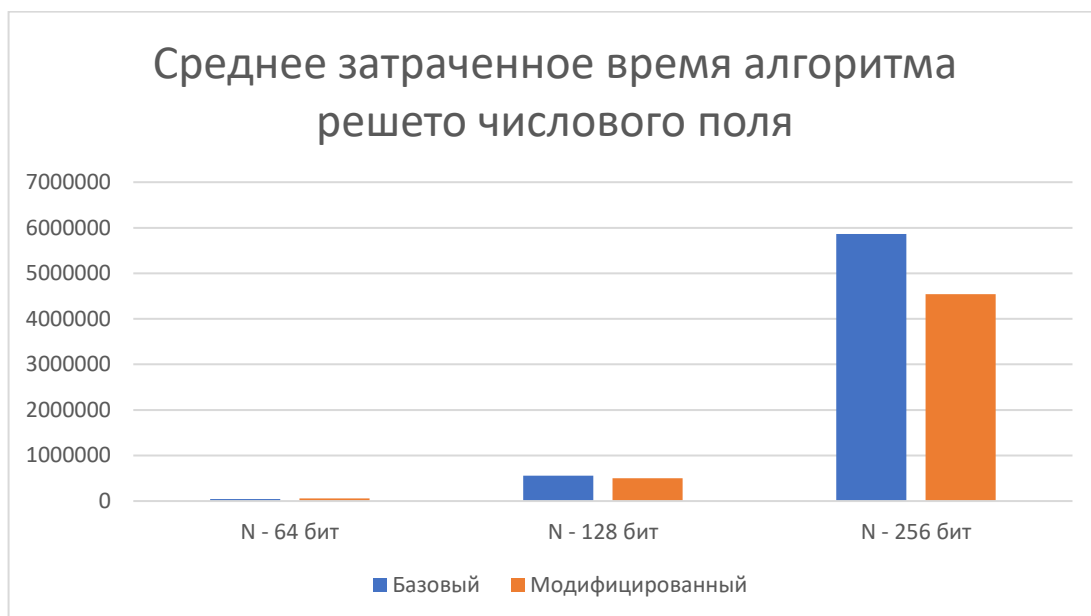


Рисунок 14 - Среднее затраченное время алгоритма решето числового поля

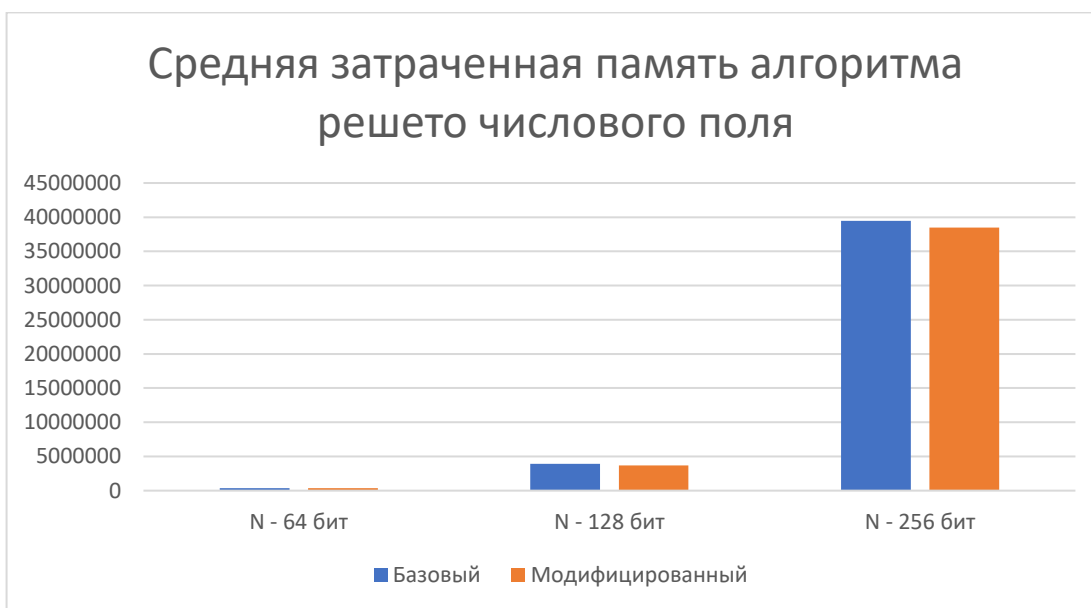


Рисунок 15 - Средняя затраченная память алгоритма решето числового поля

Заключение

В результате выпускной работы были реализованы и исследованы базовые и модифицированные алгоритмы дискретного логарифмирования.

Алгоритм Шенкса показал лучшие результаты в затраченном времени выполнения на маленьких параметрах, где g , p и A - 16 битные числа, а параметр a - 8 битное число. В остальных тестах по времени и затраченной памяти лучшие результаты показал модифицированный алгоритм Шенкса. Также базовый и модифицированный алгоритм Шенкса показал лучшие результаты, где g , p и A - 32 битные числа, а параметр a - 16 битное число, чем при параметрах, где g , p и A - 32 битные числа, а параметр a - 8 битное число.

Базовый алгоритм Полига-Хеллмана показал лучшие результаты в затраченном времени выполнения и затраченной памяти на маленьких параметрах, где g , p и A - 16 битные числа, а параметр a - 8 битное число. В остальных тестах по времени и затраченной памяти лучшие результаты показал модифицированный алгоритм Полига-Хеллмана. Также базовый и модифицированный алгоритм Полига-Хеллмана показал лучшие результаты в затраченном времени выполнения, где g , p и A - 32 битные числа, а параметр a - 16 битное число, чем при параметрах, где g , p и A - 32 битные числа, а параметр a - 8 битное число. Модифицированный алгоритм показал сильно лучшие результаты в затраченной памяти, где g , p и A - 32 битные числа, а параметр a - 8 битное число.

Модифицированный алгоритм ро-метод Полларда продемонстрировал лучшие результаты в затраченном времени выполнения, где N - 64 бит и N - 256 бит. В остальных показателях базовый алгоритм показал лучшие результаты.

Модифицированный алгоритм Адлемана показал лучшие результаты во всех тестах, кроме скорости выполнения, где g , p и A - 16 битные числа, а параметр a - 8 битное число, и где g , p и A - 32 битные числа, а параметр a - 16 битное число.

Модифицированный алгоритм COS показал лучше результаты во всех тестах, кроме скорости выполнения, где g , p и A - 32 битные числа, а параметр a - 8 битное число, и где g , p и A - 32 битные числа, а параметр a - 16 битное число.

На основе экспериментов базового и модифицированного алгоритма решето числового поля можно сделать вывод, что модифицированный алгоритм показал лучше результаты во всех тестах, кроме скорости на 64 битном входном параметре.

Таблица 37 - Компетенции

Компетенция	Расшифровка компетенции	Описание приобретенных знаний, умений и навыков
УК-1	Способен осуществлять критический анализ проблемных ситуаций на основе системного подхода, вырабатывать стратегию действий	Выработал навык критического анализа на основе системного подхода в области дискретного логарифмирования.
УК-2	Способен управлять проектом на всех этапах его жизненного цикла	Развил умение управлять проектом на всех этапах его жизненного цикла при работе с криптографией.
УК-3	Способен организовывать и руководить работой команды, вырабатывая командную стратегию для достижения поставленной цели	Развил способность организовывать работу команды и вырабатывать стратегию для достижения намеченной цели в области информационной безопасности.
УК-4	Способен применять современные коммуникативные технологии, в том числе на иностранном(ых) языке(ах), для академического и профессионального взаимодействия	Обрёл навык применения современных коммуникативных технологий для профессионального взаимодействия в области дискретного логарифмирования.
УК-5	Способен анализировать и учитывать разнообразие культур в процессе межкультурного взаимодействия	Получил навык анализа разнообразия культур в процессе межкультурного взаимодействия при работе с информационной безопасностью
УК-6	Способен определять и реализовывать приоритеты собственной деятельности и способы ее совершенствования на основе самооценки	Обрёл навык определения и реализации приоритетов собственной деятельности при работе в области информационной безопасности
ОПК-1	Способен находить, формулировать и решать актуальные проблемы	Получен навык нахождения, формулирования и решения

	прикладной математики, фундаментальной информатики и информационных технологий	актуальных проблем фундаментальной информатики и информационных технологий при работе с криптографией
ОПК-2	Способен применять компьютерные/суперкомпьютерные методы, современное программное обеспечение, в том числе отечественного происхождения, для решения задач профессиональной деятельности	Развил способность применения компьютерных методов для решения задач профессиональной деятельности при работе с дискретным логарифмированием
ОПК-3	Способен проводить анализ математических моделей, создавать инновационные методы решения прикладных задач профессиональной деятельности в области информатики и математического моделирования	Выработал навык проведения анализа математических моделей, создания инновационных методов решения прикладных задач профессиональной деятельности в области информационной безопасности
ОПК-4	Способен оптимальным образом комбинировать существующие информационно-коммуникационные технологии для решения задач в области профессиональной деятельности с учетом требований информационной безопасности	Получил навык оптимальным образом комбинировать существующие информационно-коммуникационные технологии для решения задач в области профессиональной деятельности с учётом информационной безопасности
ОПК-5	Способен устанавливать и сопровождать программное обеспечение информационных систем, осуществлять эффективное управление разработкой программных средств и проектов	Развил способность устанавливать и сопровождать программное обеспечение информационных систем при работе с дискретным логарифмом
ПК-1	Разработка требований и проектирование программного обеспечения	Обрёл навык разработки требований и проектирования программного обеспечения в области криптографии
ПК-2	Управление работами по сопровождению и проектами создания (модификации) информационных систем, автоматизирующих задачи организационного управления и бизнес-процессы	Выработал навык управления работами по сопровождению и проектами создания информационных систем, автоматизирующих задачи организационного управления в области информационной безопасности
ПК-3	Управление проектами в области информационных технологий малого и среднего уровня сложности в условиях неопределенностей, порождаемых запросами на изменения, с применением	Развил навык управления проектами в области информационных технологий малого и среднего уровня сложности в условиях

	формальных инструментов управления рисками и проблемами проекта	неопределённостей при работе с криптографией
ПК-4	Управление проектами в области информационных технологий любого масштаба в условиях высокой неопределенности, вызываемой запросами на изменения и рисками, и с учетом влияния организационного окружения проекта; разработка новых инструментов и методов управления проектами в области информационных технологий	Обрёл навык управления проектами в области информационных технологий любого масштаба в условиях высокой неопределённости, вызываемой запросами на изменения и рисками, при работе с дискретным логарифмированием
ПК-5	Создание и внедрение средств разработки технической документации	Получил навык создания и внедрения средств разработки технической документации при работе с дискретным логарифмированием
ПК-6	Управление аналитическими работами и подразделением	Развил навык управления аналитическими работами и подразделением при работе с дискретным логарифмированием

Список литературы

1. Applied Cryptography: Protocols, Algorithms, and Source Code in C. / Schneier, Bruce – Текст: непосредственный // Second Edition. — 2nd. — Wiley, 1996.
2. Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — С. 10.
3. 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.
4. Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — С. 52.
5. 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.
6. 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.
7. 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.
8. A subexponential algorithm for discrete logarithms over all finite fields / Adleman L. M., Demarrais J. - Текст: непосредственный // Mathematics of computation. — 1993.
9. Теоретико-числовые алгоритмы в криптографии. / Василенко О.Н. - Текст: непосредственный // N— М.: МЦНМО, 2003. — С. 328.
10. Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — С. 190.

Приложения

Приложение 1. Модифицированный алгоритм Шенкса

```
using DiscreteLogarithm.MathFunctionsForCalculation;
using System.Numerics;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ModifiedExponentialAlgorithms
{
    public class ModifiedShenks
    {
        MathFunctions mathFunctions;
        public ModifiedShenks()
        {
            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 g";
            };
            if (!BigInteger.TryParse(input_A, out A) || A <= 0)
            {
                theValuesAreCorrect = false;
                inputLabel.Text += "\nОшибка A";
            };
            if (!BigInteger.TryParse(input_p, out p) || p <= 0)
            {
                theValuesAreCorrect = false;
                inputLabel.Text += "\nОшибка p";
            };
        }

        async public Task CalculateModifiedShenksAsync(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>();

await Step2Async(g_km_degree, Ag_m_degree, g, A, p, m, k);

BigInteger i, j;
(i, j) = await Step3Async(g_km_degree, Ag_m_degree);
BigInteger result = i * m - j;

inputLabel.Text = "Результат: \na = " + result.ToString();
}

private void Step1(BigInteger p, out BigInteger m, out
BigInteger k)
{
    m = k = mathFunctions.Sqrt(p) + 1;
}

async private Task Step2Async(List<BigInteger> g_km_degree,
List<BigInteger> Ag_m_degree, BigInteger g, BigInteger A,
BigInteger p, BigInteger m, BigInteger k)
{
    Task task_Step2_g_km_degree =
Step2_g_km_degreeAsync(g_km_degree, g, A, p, m, k);
    Task task_Step2_Ag_m_degree =
Step2_Ag_m_degreeAsync(Ag_m_degree, g, A, p, m, k);
    await Task.WhenAll(task_Step2_g_km_degree,
task_Step2_Ag_m_degree);
}

async private Task Step2_g_km_degreeAsync(List<BigInteger>
g_km_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));
    }
}

async private Task Step2_Ag_m_degreeAsync(List<BigInteger>
Ag_m_degree, BigInteger g, BigInteger A, BigInteger p, BigInteger
m, BigInteger k)
{
    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));
    }
}

```

```

        async        private        Task<(BigInteger,        BigInteger)>
Step3Async(List<BigInteger>        g_km_degree,        List<BigInteger>
Ag_m_degree)
{
    var        task_Step3Async_0        =        Step3Async_0(g_km_degree,
Ag_m_degree);
    var        task_Step3Async_1        =        Step3Async_1(g_km_degree,
Ag_m_degree);
    (BigInteger,        BigInteger)        result        =        await
Task.WhenAny(task_Step3Async_0, task_Step3Async_1).Result;
    return result; // распараллелил алгоритм на 3 шаге, чтобы с
двух сторон был поиск результата
}

```

```

        async        private        Task<(BigInteger,        BigInteger)>
Step3Async_0(List<BigInteger>        g_km_degree,        List<BigInteger>
Ag_m_degree)
{
    for (int i = 0; i < g_km_degree.Count / 2; i += 1)
    {
        for (int j = 0; j < Ag_m_degree.Count; j++)
        {
            if (g_km_degree[i] == Ag_m_degree[j])
            {
                return (i + 1, j);
            }
        }
    }
    await Task.Delay(100000);
    return (0, 0);
}

```

```

        async        private        Task<(BigInteger,        BigInteger)>
Step3Async_1(List<BigInteger>        g_km_degree,        List<BigInteger>
Ag_m_degree)
{
    for (int i = g_km_degree.Count; i > g_km_degree.Count / 2; i-
-)
    {
        for (int j = 0; j < Ag_m_degree.Count; j++)
        {
            if (g_km_degree[i] == Ag_m_degree[j])
            {
                return (i + 1, j);
            }
        }
    }
    await Task.Delay(100000);
    return (0, 0);
}
}
}

```

Приложение 2. Модифицированный алгоритм Полига-Хеллмана

```
using DiscreteLogarithm.MathFunctionsForCalculation;
using System.Numerics;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ModifiedExponentialAlgorithms
{
    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 ModifiedPoligHellman
    {
        MathFunctions mathFunctions;
        public ModifiedPoligHellman()
        {
            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 g";
            };
            if (!BigInteger.TryParse(input_A, out A) || A <= 0)
            {
                theValuesAreCorrect = false;
                inputLabel.Text += "\nОшибка A";
            };
            if (!BigInteger.TryParse(input_p, out p) || p <= 0)
```

```

    {
        theValuesAreCorrect = false;
        inputLabel.Text += "\nОшибка 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();
        foreach (ListGroupedValues listGroupedValues in
fi_p_dividers_grouped)
        {
            listGroupedValues.Key = listGroupedValues.key_degree;
            listGroupedValues.degree_number = 1;
        }
        // сделал так, чтобы число p - 1 было разложено без степеней
        // таким образом на 1 шаге строится таблица с единичными
        значениями в каждой строке

        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: \na = {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. Модифицированный алгоритм ро-метод Полларда

```
using System.Numerics;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ModifiedExponentialAlgorithms
{
    public class ModifiedRoPollard
    {
        public void CheckingTheInputValues(
            string input_N,
            TextBox inputTextBox,
            ref bool theValuesAreCorrect,
            out BigInteger a)
        {
            inputTextBox.Text = "";
            if (!BigInteger.TryParse(input_N, out a) || a < 5)
            {
                theValuesAreCorrect = false;
                inputTextBox.Text = "Ошибка 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; // сделал вычисление x, не возводя в
квadrat x
                x = (x * x * x - 1) % n;
                i = i + 1;
            }
            return BigInteger.GreatestCommonDivisor(n, BigInteger.Abs(x
- y));
        }
    }
}
```

```

        public void CalculateRoPollard(BigInteger N, TextBox
inputTextBox)
        {
            BigInteger p = ro_Pollard(N);
            BigInteger q = N / p;

            inputTextBox.Text = string.Format("P = {0} \nQ = {1}", p, q);
        }
    }
}

```

Приложение 4. Модифицированный алгоритм Адлемана

```

using DiscreteLogarithm.ExponentialAlgorithms;
using DiscreteLogarithm.MathFunctionsForCalculation;
using ExtendedNumerics;
using System.Numerics;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ModifiedSubExponentialAlgorithms
{
    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 ModifiedAdleman
    {
        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 ModifiedAdleman()
        {
            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 g";
            };
            if (!BigInteger.TryParse(input_A, out A) || A <= 0)
            {
                theValuesAreCorrect = false;
                inputLabel.Text += "\nОшибка A";
            };
            if (!BigInteger.TryParse(input_p, out p) || p <= 0)
            {
                theValuesAreCorrect = false;
                inputLabel.Text += "\nОшибка p";
            };
        }

```

```

        public void CalculateAdleman(BigInteger input_g, BigInteger
input_A, BigInteger input_p, Label inputLabel)
        {
            g = input_g;
            p = input_p;
            A = input_A;
            Step1();
            Step2();
            Step3();
            BigInteger result = Step4();

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

        private void Step1()
        {
            BigInteger degree = BigRational.Sqrt(BigInteger.Log2(p) *
BigInteger.Log2(p)).WholePart;
            // поменял формулу вычисления B, чтобы повысить факторную
базу при вычислении логарифмов
            B = (BigInteger)BigRational.Pow(expNumber,
degree).FractionalPart;
        }

        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(listGroupedValue
sIndex);
}
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.Cou
nt; j++)
        {
            log_g_NUM.Add(exponentiationModuloDividersGroupedList[i].lis
tGroupedValues[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 isContainsList_0_0 = false;
    bool isContainsList_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])
                {
                    isContainsList_0_0 = true;
                }
                if (log_g_NUM_result[j].num == slauArrayResults[i][0, 1] &&
                    log_g_NUM_result[j].result == slauArrayResults[i][2, 2])
                {
                    isContainsList_0_1 = true;
                }
            }
            if (isContainsList_0_0 == false)
            {
                log_g_NUM_result.Add(new
                    Log_g_NUM_result(slauArrayResults[i][0, 0], 0,
                    slauArrayResults[i][1, 2]));
            }
            if (isContainsList_0_1 == false)
            {
                log_g_NUM_result.Add(new
                    Log_g_NUM_result(slauArrayResults[i][0, 1], 1,
                    slauArrayResults[i][2, 2]));
            }
            isContainsList_0_0 = false;
            isContainsList_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 = "")
{

```

```

return;
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()
{
return;
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();
}
}
}
}

```

Приложение 5. Модифицированный алгоритм COS

```

using DiscreteLogarithm.ExponentialAlgorithms;
using DiscreteLogarithm.MathFunctionsForCalculation;
using ExtendedNumerics;
using System.Numerics;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.ModifiedSubExponentialAlgorithms
{
public class ModifiedCOS
{
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;
BigInteger H;
BigInteger J;
Random rnd;
public ModifiedCOS()
{
    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;
    rnd = new Random();
}

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 g";
    };
    if (!BigInteger.TryParse(input_A, out A) || A <= 0)
    {
        theValuesAreCorrect = false;
        inputLabel.Text += "\nОшибка A";
    };
    if (!BigInteger.TryParse(input_p, out p) || p <= 0)
    {
        theValuesAreCorrect = false;
        inputLabel.Text += "\nОшибка p";
    };
}

```

```

        public void CalculateCOS(BigInteger input_g, BigInteger
input_A, BigInteger input_p, Label inputLabel)
        {
            g = input_g;
            p = input_p;
            A = input_A;
            _Step1();
            _Step2();
            Step3();
            BigInteger result = Step4();

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

        public void CalculateCOS1(BigInteger input_g, BigInteger
input_A, BigInteger input_p, Label inputLabel)
        {
            g = input_g;
            p = input_p;
            A = input_A;
            Step1();
            Step2();
            Step3();
            BigInteger result = Step4();

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

        private void Step1()
        {
            H = (BigInteger)BigRational.Sqrt(p).FractionalPart + 1;
            J = H * H - p;
        }

        private void Step2()
        {
            BigInteger exponentiationModuloResult = 0;
            List<BigInteger> exponentiationModuloList = new
List<BigInteger>();
            List<ListGroupedValues> exponentiationModuloDividersGrouped
= new List<ListGroupedValues>();
            bool isSmooth = true;
            for (int c = 1; c < 10; c++)
            {
                exponentiationModuloList = mathFunctions.Factorization((H +
rnd.Next(1, c)) * (H + rnd.Next(1, c)) * (H + rnd.Next(1, c)));
                // повысил значение логарифмов (H + c), добавив значение c3,
чтобы увеличить разложение чисел для формирования СЛАУ

```

```

        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(c, exponentiationModuloDividersGrouped);
            exponentiationModuloDividersGroupedList.Add(listGroupedValue
sIndex);
        }
        isSmooth = true;
    }

    private void _Step1()
    {
        BigInteger degree = BigRational.Sqrt(BigInteger.Log2(p) *
BigInteger.Log2(BigInteger.Log2(p))).WholePart;
        B = (BigInteger)BigRational.Pow(expNumber,
degree).FractionalPart;
    }

    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(listGroupedValue
sIndex);
            }
            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.Cou
nt; j++)
                {
                    log_g_NUM.Add(exponentiationModuloDividersGroupedList[i].lis
tGroupedValues[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.Cou
nt; j++)
            {
                rowSLAUindex =
log_g_NUM.IndexOf(exponentiationModuloDividersGroupedList[i].lis
tGroupedValues[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 isContainsList_0_0 = false;
    bool isContainsList_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])
            {
                isContainsList_0_0 = true;
            }
            if (log_g_NUM_result[j].num == slauArrayResults[i][0, 1] &&
log_g_NUM_result[j].result == slauArrayResults[i][2, 2])
            {
                isContainsList_0_1 = true;
            }
        }
        if (isContainsList_0_0 == false)
        {
            log_g_NUM_result.Add(new
Log_g_NUM_result(slauArrayResults[i][0, 0], 0],
slauArrayResults[i][1, 2]));
        }
        if (isContainsList_0_1 == false)
        {
            log_g_NUM_result.Add(new
Log_g_NUM_result(slauArrayResults[i][0, 1], 1],
slauArrayResults[i][2, 2]));
        }
        isContainsList_0_0 = false;
        isContainsList_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();
    }
}
}

```

Приложение 6. Модифицированный алгоритм решето числового поля

```

using DiscreteLogarithm.MathFunctionsForCalculation;
using ExtendedArithmetic;
using ExtendedNumerics;
using System.Management;
using System.Numerics;
using System.Runtime.Serialization;

```

```

using System.Text;
using Label = System.Windows.Forms.Label;

namespace DiscreteLogarithm.SubExponentialAlgorithms
{
    public class GNFS
    {
        public BigInteger N;
        public Solution Factorization { get; private set; }
        public int PolynomialDegree { get; internal set; }
        public BigInteger PolynomialBase { get; private set; }
        public Polynomial CurrentPolynomial { get; internal set; }

        public PolyRelationsSieveProgress CurrentRelationsProgress {
get; set; }
        public FactorBase PrimeFactorBase { get; set; }
        public List<Polynomial> PolynomialCollection { get; set; }
        public FactorPairCollection QuadraticFactorPairCollection {
get; set; }
        int relationQuantity { get; set; }
        int relationValueRange { get; set; }

        /// <summary>
        /// Array of (p, m % p)
        /// </summary>
        public FactorPairCollection RationalFactorPairCollection {
get; set; }

        /// <summary>
        /// Array of (p, r) where f(r) % p == 0
        /// </summary>
        public FactorPairCollection AlgebraicFactorPairCollection {
get; set; }

        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 N";
            };
        }

        public void CalculateGNFS(BigInteger N, Label inputLabel)
        {
            Step1(N);
            Step2();
            Step3();
        }
    }
}

```

```

Step4();
BigInteger p = this.Factorization.P;
BigInteger q = this.Factorization.Q;

inputLabel.Text = string.Format("P = {0} \nQ = {1}", p, q);
}

private void Step1(BigInteger N) // Create Polynomial, Factor
Bases && Roots
{
    PolynomialBase = 31;
    PolynomialDegree = 3;
    relationQuantity = 65;
    relationValueRange = 1000;
    PrimeFactorBase = new FactorBase();
    PolynomialCollection = new List<Polynomial>();
    RationalFactorPairCollection = new FactorPairCollection();
    AlgebraicFactorPairCollection = new FactorPairCollection();
    QuadraticFactorPairCollection = new FactorPairCollection();
    CurrentPolynomial = new Polynomial(N, PolynomialBase,
PolynomialDegree);
    CaclulatePrimeFactorBaseBounds(50);
    SetPrimeFactorBases();
    NewFactorPairCollections();
    CurrentRelationsProgress = new
PolyRelationsSieveProgress(this, relationQuantity,
relationValueRange);
    this.N = N;
}

private void Step2() // Sieve Relations
{
    this.CurrentRelationsProgress.GenerateRelations();
}

private void Step3() // Matrix
{
    MatrixSolve.GaussianSolve(this);
}

private void Step4() // Square Root Solve
{
    SquareFinder.Solve(this);
}

public void CaclulatePrimeFactorBaseBounds(BigInteger bound)
{
    PrimeFactorBase = new FactorBase();

    PrimeFactorBase.RationalFactorBaseMax = bound;
    PrimeFactorBase.AlgebraicFactorBaseMax =
    (PrimeFactorBase.RationalFactorBaseMax) * 3;
}

```

```

        PrimeFactorBase.QuadraticBaseCount =
CalculateQuadraticBaseSize(PolynomialDegree);

        PrimeFactorBase.QuadraticFactorBaseMin =
PrimeFactorBase.AlgebraicFactorBaseMax + 20;
        PrimeFactorBase.QuadraticFactorBaseMax =
PrimeFactory.GetApproximateValueFromIndex((UInt64)(PrimeFactorBa
se.QuadraticFactorBaseMin + PrimeFactorBase.QuadraticBaseCount));
    }
    private static int CalculateQuadraticBaseSize(int polyDegree)
    {
        int result = -1;

        if (polyDegree <= 3)
        {
            result = 10;
        }
        else if (polyDegree == 4)
        {
            result = 20;
        }
        else if (polyDegree == 5 || polyDegree == 6)
        {
            result = 40;
        }
        else if (polyDegree == 7)
        {
            result = 80;
        }
        else if (polyDegree >= 8)
        {
            result = 100;
        }

        return result;
    }

    public void SetPrimeFactorBases()
    {
        PrimeFactory.IncreaseMaxValue(PrimeFactorBase.QuadraticFacto
rBaseMax);

        PrimeFactorBase.RationalFactorBase =
PrimeFactory.GetPrimesTo(PrimeFactorBase.RationalFactorBaseMax);

        PrimeFactorBase.AlgebraicFactorBase =
PrimeFactory.GetPrimesTo(PrimeFactorBase.AlgebraicFactorBaseMax)
;

        PrimeFactorBase.QuadraticFactorBase =
PrimeFactory.GetPrimesFrom(PrimeFactorBase.QuadraticFactorBaseMi
n).Take(PrimeFactorBase.QuadraticBaseCount);

```

```

    }

    private void NewFactorPairCollections()
    {
        if (!RationalFactorPairCollection.Any())
        {
            RationalFactorPairCollection =
FactorPairCollection.Factory.BuildRationalFactorPairCollection(t
his);
        }

        if (!AlgebraicFactorPairCollection.Any())
        {
            AlgebraicFactorPairCollection =
FactorPairCollection.Factory.BuildAlgebraicFactorPairCollection(
this);
        }

        if (!QuadraticFactorPairCollection.Any())
        {
            QuadraticFactorPairCollection =
FactorPairCollection.Factory.BuildQuadraticFactorPairCollection(
this);
        }
    }

    public bool SetFactorizationSolution(BigInteger p, BigInteger
q)
    {
        BigInteger n = p * q;

        if (n == this.N)
        {
            Factorization = new Solution(p, q);
            return true;
        }
        return false;
    }

    }

    public class PolyRelationsSieveProgress
    {
        MathFunctions mathFunctions;
        public BigInteger A { get; private set; }
        public BigInteger B { get; private set; }
        public int SmoothRelations_TargetQuantity { get; private set;
}

        public BigInteger ValueRange { get; private set; }
    }

```

```

        public List<List<Relation>> FreeRelations { get { return
Relations.FreeRelations; } }
        public List<Relation> SmoothRelations { get { return
Relations.SmoothRelations; } }
        public List<Relation> RoughRelations { get { return
Relations.RoughRelations; } }

        public RelationContainer Relations { get; set; }

        public BigInteger MaxB { get; set; }
        public int SmoothRelationsCounter { get; set; }
        public int FreeRelationsCounter { get; set; }

        public int SmoothRelationsRequiredForMatrixStep
        {
            get
            {
                return
PrimeFactory.GetIndexFromValue(_gnfs.PrimeFactorBase.RationalFac
torBaseMax)
                +
PrimeFactory.GetIndexFromValue(_gnfs.PrimeFactorBase.AlgebraicFa
ctorBaseMax)
                + _gnfs.QuadraticFactorPairCollection.Count + 3;
            }
        }

        internal GNFS _gnfs;

        public PolyRelationsSieveProgress(GNFS gnfs, int
smoothRelationsTargetQuantity, BigInteger valueRange)
        {
            mathFunctions = new MathFunctions();
            _gnfs = gnfs;
            Relations = new RelationContainer();

            A = 0;
            B = 3;
            ValueRange = valueRange;

            if (smoothRelationsTargetQuantity == -1)
            {
                SmoothRelations_TargetQuantity =
SmoothRelationsRequiredForMatrixStep;
            }
            else
            {
                SmoothRelations_TargetQuantity =
Math.Max(smoothRelationsTargetQuantity,
SmoothRelationsRequiredForMatrixStep);
            }

            if (MaxB == 0)

```

```

{
    MaxB = (uint)gnfs.PrimeFactorBase.AlgebraicFactorBaseMax;
}
}

public void GenerateRelations()
{
    SmoothRelations_TargetQuantity =
    Math.Max(SmoothRelations_TargetQuantity,
    SmoothRelations_RequiredForMatrixStep); ;

    if (A >= ValueRange)
    {
        ValueRange += 200;
    }

    ValueRange = (ValueRange % 2 == 0) ? ValueRange + 1 :
    ValueRange;
    A = (A % 2 == 0) ? A + 1 : A;

    BigInteger startA = A;

    while (B >= MaxB)
    {
        MaxB += 100;
    }

    while (SmoothRelationsCounter <
    SmoothRelations_TargetQuantity)
    {
        if (B > MaxB)
        {
            break;
        }

        foreach (BigInteger a in
        SieveRange.GetSieveRangeContinuation(A, ValueRange))
        {
            A = a;
            if (BigInteger.GreatestCommonDivisor(A, B) == 1)
            {
                Relation rel = new Relation(_gnfs, A, B);

                rel.Sieve(_gnfs.CurrentRelationsProgress);

                bool smooth = rel.IsSmooth;
                if (smooth)
                {
                    Serialization.Save.Relations.Smooth.Append(_gnfs, rel);

```

```

        _gnfs.CurrentRelationsProgress.Relations.SmoothRelations.Add
(rel);

    }
    else
    {

    }
    }
    }

    B += 1;
    A = startA;

    }
    }

    public void IncreaseTargetQuantity()
    {
        IncreaseTargetQuantity(SmoothRelations_TargetQuantity -
SmoothRelationsRequiredForMatrixStep);
    }

    public void IncreaseTargetQuantity(int ammount)
    {
        SmoothRelations_TargetQuantity += ammount;
    }

    public void PurgePrimeRoughRelations()
    {
        List<Relation>          roughRelations          =
Relations.RoughRelations.ToList();

        IEnumerable<Relation> toRemoveAlg = roughRelations
        .Where(r => r.AlgebraicQuotient != 1 &&
mathFunctions.TestMillerRabin(r.AlgebraicQuotient) == "Вероятно
простое");

        roughRelations          =
roughRelations.Except(toRemoveAlg).ToList();

        Relations.RoughRelations = roughRelations;

        IEnumerable<Relation> toRemoveRational = roughRelations
        .Where(r => r.RationalQuotient != 1 &&
mathFunctions.TestMillerRabin(r.AlgebraicQuotient) == "Вероятно
простое");

        roughRelations          =
roughRelations.Except(toRemoveRational).ToList();

        Relations.RoughRelations = roughRelations;

```



```

    }

    public void AddFreeRelationSolution(List<Relation>
freeRelationSolution)
    {
        Relations.FreeRelations.Add(freeRelationSolution);
    }
}

public static class SieveRange
{
    public static IEnumerable<BigInteger>
GetSieveRangeContinuation(BigInteger currentValue, BigInteger
maximumRange)
    {
        BigInteger max = maximumRange;
        BigInteger counter = BigInteger.Abs(currentValue);
        bool flipFlop = !(currentValue.Sign == -1);

        while (counter <= max)
        {
            if (flipFlop)
            {
                yield return counter;
                flipFlop = false;
            }
            else if (!flipFlop)
            {
                yield return -counter;
                counter++;
                flipFlop = true;
            }
        }
    }

    public class Relation
    {
        public BigInteger A { get; protected set; }

        /// <summary>
        /// Root of f(x) in algebraic field
        /// </summary>
        public BigInteger B { get; protected set; }

        /// <summary> f(b) ≡ 0 (mod a); Calculated as: f(-a/b) * -
b^deg </summary>
        public BigInteger AlgebraicNorm { get; protected set; }
        /// <summary> a + bm </summary>
        public BigInteger RationalNorm { get; protected set; }

        internal BigInteger AlgebraicQuotient;
        internal BigInteger RationalQuotient;
    }
}

```

```

        public CountDictionary AlgebraicFactorization { get; private
set; }
        public CountDictionary RationalFactorization { get; private
set; }

        public bool IsSmooth { get { return (IsRationalQuotientSmooth
&& IsAlgebraicQuotientSmooth); } }

        public bool IsRationalQuotientSmooth { get { return
(RationalQuotient == 1 || RationalQuotient == 0); } }

        public bool IsAlgebraicQuotientSmooth { get { return
(AlgebraicQuotient == 1 || AlgebraicQuotient == 0); } }

        public bool IsPersisted { get; set; }

        public Relation()
        {
            IsPersisted = false;
            RationalFactorization = new CountDictionary();
            AlgebraicFactorization = new CountDictionary();
        }

        public Relation(GNFS gnfs, BigInteger a, BigInteger b)
        : this()
        {
            A = a;
            B = b;

            AlgebraicNorm = Normal.Algebraic(A, B,
gnfs.CurrentPolynomial); //  $b^{\deg} * f(a/b)$ 
            RationalNorm = Normal.Rational(A, B, gnfs.PolynomialBase); //
a + bm

            AlgebraicQuotient = BigInteger.Abs(AlgebraicNorm);
            RationalQuotient = BigInteger.Abs(RationalNorm);

            if (AlgebraicNorm.Sign == -1)
            {
                AlgebraicFactorization.Add(BigInteger.MinusOne);
            }

            if (RationalNorm.Sign == -1)
            {
                RationalFactorization.Add(BigInteger.MinusOne);
            }

            public void Sieve(PolyRelationsSieveProgress relationsSieve)
            {
                Sieve(relationsSieve._gnfs.PrimeFactorBase.RationalFactorBas
e, ref RationalQuotient, RationalFactorization);
            }
        }
    }

```

```

        if (IsRationalQuotientSmooth) // No sense wasting time on
factoring the AlgebraicQuotient if the relation is ultimately
going to be rejected anyways.
    {
        Sieve(relationsSieve._gnfs.PrimeFactorBase.AlgebraicFactorBa
se, ref AlgebraicQuotient, AlgebraicFactorization);
    }
}

private static void Sieve(IEnumerable<BigInteger>
primeFactors, ref BigInteger quotientValue, CountDictionary
dictionary)
{
    if (quotientValue.Sign == -1 || primeFactors.Any(f => f.Sign
== -1))
    {
        throw new Exception("There shouldn't be any negative values
either in the quotient or the factors");
    }

    foreach (BigInteger factor in primeFactors)
    {
        if (quotientValue == 1)
        {
            return;
        }

        if ((factor * factor) > quotientValue)
        {
            if (primeFactors.Contains(quotientValue))
            {
                dictionary.Add(quotientValue);
                quotientValue = 1;
            }
            return;
        }

        while (quotientValue != 1 && quotientValue % factor == 0)
        {
            dictionary.Add(factor);
            quotientValue = BigInteger.Divide(quotientValue, factor);
        }
    }

}

public class FactorBase
{
    public FactorBase()
    {
        RationalFactorBase = new List<BigInteger>();
    }
}

```

```

        AlgebraicFactorBase = new List<BigInteger>();
        QuadraticFactorBase = new List<BigInteger>();
    }

    public BigInteger RationalFactorBaseMax { get; internal set;
}
    public BigInteger AlgebraicFactorBaseMax { get; internal set;
}
    public BigInteger QuadraticFactorBaseMin { get; internal set;
}
    public BigInteger QuadraticFactorBaseMax { get; internal set;
}

    public int QuadraticBaseCount { get; internal set; }
    public IEnumerable<BigInteger> RationalFactorBase { get;
internal set; }
    public IEnumerable<BigInteger> AlgebraicFactorBase { get;
internal set; }
    public IEnumerable<BigInteger> QuadraticFactorBase { get;
internal set; }
    }

    public class RelationContainer
    {
        public List<Relation> SmoothRelations { get; internal set; }
        public List<Relation> RoughRelations { get; internal set; }
        public List<List<Relation>> FreeRelations { get; internal
set; }

        public RelationContainer()
        {
            SmoothRelations = new List<Relation>();
            RoughRelations = new List<Relation>();
            FreeRelations = new List<List<Relation>>();
        }
    }

    public class CountDictionary : SortedDictionary<BigInteger,
BigInteger>
    {
        public CountDictionary()
        : base(Comparer<BigInteger>.Create(BigInteger.Compare))
        {
        }

        public void Add(BigInteger key)
        {
            this.AddSafe(key, 1);
        }
        private void AddSafe(BigInteger key, BigInteger value)
        {
            if (!ContainsKey(key)) { this.Add(key, value); }
            else { this[key] += value; }
        }
    }

```

```

public void Combine(CountDictionary dictionary)
{
    foreach (var kvp in dictionary)
    {
        AddSafe(kvp.Key, kvp.Value);
    }
}

#region String Formatting

public string FormatStringAsFactorization()
{
    //Order();
    StringBuilder result = new StringBuilder();
    result.Append(
        " -> {\t" +
        string.Join(" * ", this.Select(kvp =>
        $"{kvp.Key}^{kvp.Value}")) +
        "\t};"
    );
    return result.ToString();
}

#endregion

public static class Normal
{
    /// <summary>
    /// a + bm
    /// </summary>
    /// <param name="polynomialBase">Base m of f(m) = N</param>
    /// <returns></returns>
    public static BigInteger Rational(BigInteger a, BigInteger b,
    BigInteger polynomialBase)
    {
        return BigInteger.Add(a, BigInteger.Multiply(b,
    polynomialBase));
    }

    /// <summary>
    /// f(b)  $\equiv 0 \pmod{a}$ 
    ///
    /// Calculated as:
    /// f(-a/b) * -b^deg
    /// </summary>
    /// <param name="a">Divisor in the equation f(b)  $\equiv 0 \pmod{a}$ </param>
    /// <param name="b">A root of f(x)</param>
    /// <param name="poly">Base m of f(m) = N</param>
    /// <returns></returns>

```

```

    public static BigInteger Algebraic(BigInteger a, BigInteger
b, Polynomial poly)
    {
        BigRational aD = (BigRational)a;
        BigRational bD = (BigRational)b;
        BigRational ab = BigRational.Negate(aD) / bD;

        BigRational left = PolynomialEvaluate_BigRational(poly, ab);
        BigInteger right = BigInteger.Pow(BigInteger.Negate(b),
poly.Degree);

        BigRational product = right * left;

        Fraction fractionalPart = product.FractionalPart;

        BigInteger result = product.WholePart;
        return result;
    }

    private static BigRational
PolynomialEvaluate_BigRational(Polynomial polynomial, BigRational
indeterminateValue)
    {
        int num = polynomial.Degree;

        BigRational result = (BigRational)polynomial[num];
        while (--num >= 0)
        {
            result *= indeterminateValue;
            result += (BigRational)polynomial[num];
        }

        return result;
    }

    public static class PrimeFactory
    {
        private static MathFunctions mathFunctions;
        private static BigInteger MaxValue = 10;

        private static int primesCount;
        private static BigInteger primesLast;
        private static List<BigInteger> primes = new
List<BigInteger>() { 2, 3, 5, 7, 11, 13 };

        static PrimeFactory()
        {
            SetPrimes();
            mathFunctions = new MathFunctions();
        }

        private static void SetPrimes()

```

```

    {
        primes = FastPrimeSieve.GetRange(2,
(Int32.MaxValue).ToList());
        primesCount = primes.Count;
        primesLast = primes.Last();
    }

    public static IEnumerable<BigInteger> GetPrimeEnumerator(int
startIndex = 0, int stopIndex = -1)
    {
        int index = startIndex;
        int maxIndex = stopIndex > 0 ? stopIndex : primesCount - 1;
        while (index < maxIndex)
        {
            yield return primes[index];
            index++;
        }
        yield break;
    }

    public static void IncreaseMaxValue(BigInteger newMaxValue)
    {
        // Increase bound
        BigInteger temp = BigInteger.Max(newMaxValue + 1000, MaxValue
+ 100000 /*MaxValue*/);
        MaxValue = BigInteger.Min(temp, (Int32.MaxValue - 1));
        SetPrimes();
    }

    public static int GetIndexFromValue(BigInteger value)
    {
        if (value == -1)
        {
            return -1;
        }
        if (primesLast < value)
        {
            IncreaseMaxValue(value);
        }

        BigInteger primeValue = primes.First(p => p >= value);

        int index = primes.IndexOf(primeValue) + 1;
        return index;
    }

    public static BigInteger GetApproximateValueFromIndex(UInt64
n)
    {
        if (n < 6)
        {
            return primes[(int)n];
        }
    }

```

```

double fn = (double)n;
double flogn = Math.Log(n);
double flog2n = Math.Log(flogn);

double upper;

if (n >= 688383)      /* Dusart 2010 page 2 */
{
    upper = fn * (flogn + flog2n - 1.0 + ((flog2n - 2.00) /
flogn));
}
else if (n >= 178974)    /* Dusart 2010 page 7 */
{
    upper = fn * (flogn + flog2n - 1.0 + ((flog2n - 1.95) /
flogn));
}
else if (n >= 39017)     /* Dusart 1999 page 14 */
{
    upper = fn * (flogn + flog2n - 0.9484);
}
else                    /* Modified from Robin 1983 for 6-
39016 _only_ */
{
    upper = fn * (flogn + 0.6000 * flog2n);
}

if (upper >= (double)UInt64.MaxValue)
{
    throw new OverflowException($"{upper} > {UInt64.MaxValue}");
}

return new BigInteger((UInt64)Math.Ceiling(upper));
}

public          static          IEnumerable<BigInteger>
GetPrimesFrom(BigInteger minValue)
{
    return GetPrimeEnumerator(GetIndexFromValue(minValue));
}

public static IEnumerable<BigInteger> GetPrimesTo(BigInteger
maxValue)
{
    if (primesLast < maxValue)
    {
        IncreaseMaxValue(maxValue);
    }
    return GetPrimeEnumerator(0).TakeWhile(p => p < maxValue);
}

public static BigInteger GetNextPrime(BigInteger fromValue)
{

```



```

        BigInteger result = fromValue + 1;

        if (result.IsEven)
        {
            result += 1;
        }

        while (mathFunctions.TestMillerRabin(result) != "Вероятно
простое")
        {
            result += 2;
        }

        return result;
    }
}

public class FactorPairCollection : List<FactorPair>
{
    public FactorPairCollection()
    : base()
    {
    }

    public FactorPairCollection(IEnumerable<FactorPair>
collection)
    : base(collection)
    {
    }

    public static class Factory
    {
        // array of (p, m % p) up to bound
        // quantity = phi(bound)
        public static FactorPairCollection
BuildRationalFactorPairCollection(GNFS gnfs)
        {
            IEnumerable<FactorPair> result =
gnfs.PrimeFactorBase.RationalFactorBase.Select(p => new
FactorPair(p, (gnfs.PolynomialBase % p))).Distinct();
            return new FactorPairCollection(result);
        }

        // array of (p, r) where f(r) % p == 0
        // quantity = 2-3 times RFB.quantity
        public static FactorPairCollection
BuildAlgebraicFactorPairCollection(GNFS gnfs)
        {
            return new
FactorPairCollection(FindPolynomialRootsInRange(gnfs.CurrentPoly
nomial, gnfs.PrimeFactorBase.AlgebraicFactorBase, 0,
gnfs.PrimeFactorBase.AlgebraicFactorBaseMax, 2000));
        }
    }
}

```

```

    // array of (p, r) where  $f(r) \equiv 0 \pmod p$ 
    // quantity  $\leq 100$ 
    // magnitude  $p > \text{AFB.Last().p}$ 
    public static FactorPairCollection
BuildQuadraticFactorPairCollection(GNFS gnfs)
    {
        return new
FactorPairCollection(FindPolynomialRootsInRange(gnfs.CurrentPoly
nomial, gnfs.PrimeFactorBase.QuadraticFactorBase, 2,
gnfs.PrimeFactorBase.QuadraticFactorBaseMax,
gnfs.PrimeFactorBase.QuadraticBaseCount));
    }
}

    public static List<FactorPair>
FindPolynomialRootsInRange(Polynomial polynomial,
IEnumerable<BigInteger> primes, BigInteger rangeFrom, BigInteger
rangeTo, int totalFactorPairs)
    {
        List<FactorPair> result = new List<FactorPair>();

        BigInteger r = rangeFrom;
        IEnumerable<BigInteger> modList = primes.AsEnumerable();
        while (r < rangeTo && result.Count < totalFactorPairs)
        {
            // Finds p such that  $f(r) \equiv 0 \pmod p$ 
            List<BigInteger> roots = GetRootsMod(polynomial, r, modList);
            if (roots.Any())
            {
                result.AddRange(roots.Select(p => new FactorPair(p, r)));
            }
            r++;
        }

        return result.OrderBy(tup => tup.P).ToList();
    }

    /// <summary>
    /// Given a list of primes, returns primes p such that  $f(r)$ 
 $\equiv 0 \pmod p$ 
    /// </summary>
    public static List<BigInteger> GetRootsMod(Polynomial
polynomial, BigInteger baseM, IEnumerable<BigInteger> modList)
    {
        BigInteger polyResult = polynomial.Evaluate(baseM);
        IEnumerable<BigInteger> result = modList.Where(mod =>
(polyResult % mod) == 0);
        return result.ToList();
    }
}

    public struct FactorPair

```

```

    {
    public int P { get; private set; }
    public int R { get; private set; }

    public FactorPair(BigInteger p, BigInteger r)
    {
        P = (int)p;
        R = (int)r;
    }

    }

    public class FastPrimeSieve
    {
        private static readonly uint PageSize; // L1 CPU cache size
in bytes
        private static readonly uint BufferBits;
        private static readonly uint BufferBitsNext;

        static FastPrimeSieve()
        {
            uint cacheSize = 393216;
            List<uint> cacheSizes =
CPUInfo.GetCacheSizes(CPUInfo.CacheLevel.Level1);
            if (cacheSizes.Any())
            {
                cacheSize = cacheSizes.First() * 1024;
            }

            PageSize = cacheSize; // L1 CPU cache size in bytes
            BufferBits = PageSize * 8; // in bits
            BufferBitsNext = BufferBits * 2;
        }

        public static IEnumerable<BigInteger> GetRange(BigInteger
floor, BigInteger ceiling)
        {
            FastPrimeSieve primesPaged = new FastPrimeSieve();
            IEnumerator<BigInteger> enumerator =
primesPaged.GetEnumerator();

            while (enumerator.MoveNext())
            {
                if (enumerator.Current >= floor)
                {
                    break;
                }
            }

            do
            {
                if (enumerator.Current > ceiling)
                {

```

```

break;
}
yield return enumerator.Current;
}
while (enumerator.MoveNext());

yield break;
}

public IEnumerator<BigInteger> GetEnumerator()
{
return Iterator();
}

private static IEnumerator<BigInteger> Iterator()
{
IEnumerator<BigInteger> basePrimes = null;
List<uint> basePrimesArray = new List<uint>();
uint[] cullBuffer = new uint[PageSize / 4]; // 4 byte words

yield return 2;

for (var low = (BigInteger)0; ; low += BufferBits)
{
for (var bottomItem = 0; ; ++bottomItem)
{
if (bottomItem < 1)
{
if (bottomItem < 0)
{
bottomItem = 0;
yield return 2;
}

BigInteger next = 3 + low + low + BufferBitsNext;
if (low <= 0)
{
// cull very first page
for (int i = 0, sqr = 9, p = 3; sqr < next; i++, p += 2, sqr
= p * p)
{
if ((cullBuffer[i >> 5] & (1 << (i & 31))) == 0)
{
for (int j = (sqr - 3) >> 1; j < BufferBits; j += p)
{
cullBuffer[j >> 5] |= 1u << j;
}
}
}
}
else
{
// Cull for the rest of the pages

```

```

Array.Clear(cullBuffer, 0, cullBuffer.Length);

if (basePrimesArray.Count == 0)
{
    // Init second base primes stream
    basePrimes = Iterator();
    basePrimes.MoveNext();
    basePrimes.MoveNext();
    basePrimesArray.Add((uint)basePrimes.Current); // Add 3 to
base primes array
    basePrimes.MoveNext();
}

// Make sure basePrimesArray contains enough base primes...
for (BigInteger p = basePrimesArray[basePrimesArray.Count -
1], square = p * p; square < next;)
{
    p = basePrimes.Current;
    basePrimes.MoveNext();
    square = p * p;
    basePrimesArray.Add((uint)p);
}

for (int i = 0, limit = basePrimesArray.Count - 1; i < limit;
i++)
{
    var p = (BigInteger)basePrimesArray[i];
    var start = (p * p - 3) >> 1;

    // adjust start index based on page lower limit...
    if (start >= low)
    {
        start -= low;
    }
    else
    {
        var r = (low - start) % p;
        start = (r != 0) ? p - r : 0;
    }
    for (var j = (uint)start; j < BufferBits; j += (uint)p)
    {
        cullBuffer[j >> 5] |= 1u << ((int)j);
    }
}

while (bottomItem < BufferBits && (cullBuffer[bottomItem >>
5] & (1 << (bottomItem & 31))) != 0)
{
    ++bottomItem;
}

```

```

        if (bottomItem < BufferBits)
        {
            var result = 3 + (((BigInteger)bottomItem + low) << 1);
            yield return result;
        }
        else break; // outer loop for next page segment...
    }
}

public static class CPUInfo
{
    public static List<uint> GetCacheSizes(CacheLevel level)
    {
        ManagementClass mc = new
ManagementClass("Win32_CacheMemory");
        ManagementObjectCollection moc = mc.GetInstances();
        List<uint> cacheSizes = new List<uint>(moc.Count);

        cacheSizes.AddRange(moc
            .Cast<ManagementObject>()
            .Where(p => (ushort)(p.Properties["Level"].Value) ==
(ushort)level)
            .Select(p => (uint)(p.Properties["MaxCacheSize"].Value)));

        return cacheSizes;
    }

    public enum CacheLevel : ushort
    {
        Level1 = 3,
        Level2 = 4,
        Level3 = 5,
    }
}

public static partial class Serialization
{
    public static class Save
    {
        public static class Relations
        {
            public static class Smooth
            {
                public static void Append(GNFS gnfs)
                {
                    if
(gnfs.CurrentRelationsProgress.Relations.SmoothRelations.Any())
                    {
                        List<Relation> toSave =
gnfs.CurrentRelationsProgress.Relations.SmoothRelations.Where(re
l => !rel.IsPersisted).ToList();

```

```

foreach (Relation rel in toSave)
{
Append(gnfs, rel);
}
}

public static void Append(GNFS gnfs, Relation relation)
{
if (relation != null && relation.IsSmooth &&
!relation.IsPersisted)
{
gnfs.CurrentRelationsProgress.SmoothRelationsCounter += 1;

relation.IsPersisted = true;
}
}

}

}

}

public static class MatrixSolve
{
public static void GaussianSolve(GNFS gnfs)
{
Serialization.Save.Relations.Smooth.Append(gnfs); // Persist
any relations not already persisted to disk

List<Relation> smoothRelations =
gnfs.CurrentRelationsProgress.SmoothRelations.ToList();

int smoothCount = smoothRelations.Count;

BigInteger requiredRelationsCount =
gnfs.CurrentRelationsProgress.SmoothRelationsRequiredForMatrixSt
ep;

while (smoothRelations.Count >= requiredRelationsCount)
{
// Randomly select n relations from smoothRelations
List<Relation> selectedRelations = new List<Relation>();
while (
selectedRelations.Count < requiredRelationsCount
||
selectedRelations.Count % 2 != 0 // Force number of relations
to be even
)
{
int randomIndex = StaticRandom.Next(0,
smoothRelations.Count);
selectedRelations.Add(smoothRelations[randomIndex]);

```

```

smoothRelations.RemoveAt(randomIndex);
}

GaussianMatrix gaussianReduction = new GaussianMatrix(gnfs,
selectedRelations);
gaussianReduction.TransposeAppend();
gaussianReduction.Elimination();

int number = 1;
int solutionCount = gaussianReduction.FreeVariables.Count(b
=> b) - 1;
List<List<Relation>> solution = new List<List<Relation>>();
while (number <= solutionCount)
{
    List<Relation> relations =
gaussianReduction.GetSolutionSet(number);
    number++;

    BigInteger algebraic = relations.Select(rel =>
rel.AlgebraicNorm).Product();
    BigInteger rational = relations.Select(rel =>
rel.RationalNorm).Product();

    CountDictionary algCountDict = new CountDictionary();
    foreach (var rel in relations)
    {
        algCountDict.Combine(rel.AlgebraicFactorization);
    }

    bool isAlgebraicSquare = algebraic.IsSquare();
    bool isRationalSquare = rational.IsSquare();

    if (isAlgebraicSquare && isRationalSquare)
    {
        solution.Add(relations);
        gnfs.CurrentRelationsProgress.AddFreeRelationSolution(relati
ons);
    }
}

}
}

public static class StaticRandom
{
    private static readonly Random rand = new Random();
    static StaticRandom()
    {
        int counter = rand.Next(100, 200);
        while (counter-- > 0)
        {
            rand.Next();
        }
    }
}

```



```

    }
    }

    public static int Next(int minValue, int maxValue)
    {
        return rand.Next(minValue, maxValue);
    }
}

public class GaussianMatrix
{
    public List<bool[]> Matrix { get { return M; } }
    public bool[] FreeVariables { get { return freeCols; } }

    public int RowCount { get { return M.Count; } }
    public int ColumnCount { get { return M.Any() ?
M.First().Length : 0; } }

    private List<bool[]> M;
    private bool[] freeCols;
    private bool eliminationStep;

    private GNFS _gnfs;
    private List<Relation> relations;
    public Dictionary<int, Relation>
ColumnIndexRelationDictionary;
    private List<Tuple<Relation, bool[]>> relationMatrixTuple;

    public GaussianMatrix(GNFS gnfs, List<Relation> rels)
    {
        _gnfs = gnfs;
        relationMatrixTuple = new List<Tuple<Relation, bool[]>>();
        eliminationStep = false;
        freeCols = new bool[0];
        M = new List<bool[]>();

        relations = rels;

        List<GaussianRow> relationsAsRows = new List<GaussianRow>();

        foreach (Relation rel in relations)
        {
            GaussianRow row = new GaussianRow(_gnfs, rel);

            relationsAsRows.Add(row);
        }

        //List<GaussianRow> orderedRows =
relationsAsRows.OrderBy(row1 =>
row1.LastIndexOfAlgebraic).ThenBy(row2 =>
row2.LastIndexOfQuadratic).ToList();

```

```

        List<GaussianRow> selectedRows =
relationsAsRows.Take(_gnfs.CurrentRelationsProgress.SmoothRelationsRequiredForMatrixStep).ToList();

        int maxIndexRat = selectedRows.Select(row =>
row.LastIndexOfRational()).Max();
        int maxIndexAlg = selectedRows.Select(row =>
row.LastIndexOfAlgebraic()).Max();
        int maxIndexQua = selectedRows.Select(row =>
row.LastIndexOfQuadratic()).Max();

        foreach (GaussianRow row in selectedRows)
        {
            row.ResizeRationalPart(maxIndexRat);
            row.ResizeAlgebraicPart(maxIndexAlg);
            row.ResizeQuadraticPart(maxIndexQua);
        }

        GaussianRow exampleRow = selectedRows.First();
        int newLength = exampleRow.GetBoolArray().Length;

        newLength++;

        selectedRows = selectedRows.Take(newLength).ToList();

        foreach (GaussianRow row in selectedRows)
        {
            relationMatrixTuple.Add(new Tuple<Relation,
bool[]>(row.SourceRelation, row.GetBoolArray()));
        }

        public void TransposeAppend()
        {
            List<bool[]> result = new List<bool[]>();
            ColumnIndexRelationDictionary = new Dictionary<int,
Relation>();

            int index = 0;
            int numRows = relationMatrixTuple[0].Item2.Length;
            while (index < numRows)
            {
                ColumnIndexRelationDictionary.Add(index,
relationMatrixTuple[index].Item1);

                List<bool> newRow = relationMatrixTuple.Select(bv =>
bv.Item2[index]).ToList();
                newRow.Add(false);
                result.Add(newRow.ToArray());

                index++;
            }

```

```

M = result;
freeCols = new bool[M.Count];
}

public void Elimination()
{
    if (eliminationStep)
    {
        return;
    }

    int numRows = RowCount;
    int numCols = ColumnCount;

    freeCols = Enumerable.Repeat(false, numCols).ToArray();

    int h = 0;

    for (int i = 0; i < numRows && h < numCols; i++)
    {
        bool next = false;

        if (M[i][h] == false)
        {
            int t = i + 1;

            while (t < numRows && M[t][h] == false)
            {
                t++;
            }

            if (t < numRows)
            {
                //swap rows M[i] and M[t]

                bool[] temp = M[i];
                M[i] = M[t];
                M[t] = temp;
                temp = null;
            }
            else
            {
                {
                    freeCols[h] = true;
                    i--;
                    next = true;
                }
            }
            if (next == false)
            {
                for (int j = i + 1; j < numRows; j++)
                {
                    if (M[j][h] == true)

```

```

{
// Add rows
//  $M[j] \leftarrow M[j] + M[i]$ 

M[j] = Add(M[j], M[i]);
}
}
for (int j = 0; j < i; j++)
{
if (M[j][h] == true)
{
// Add rows
//  $M[j] \leftarrow M[j] + M[i]$ 

M[j] = Add(M[j], M[i]);
}
}
}
h++;
}

eliminationStep = true;
}

public List<Relation> GetSolutionSet(int numberOfSolutions)
{
bool[] solutionSet = GetSolutionFlags(numberOfSolutions);

int index = 0;
int max = ColumnIndexRelationDictionary.Count;

List<Relation> result = new List<Relation>();
while (index < max)
{
if (solutionSet[index] == true)
{
result.Add(ColumnIndexRelationDictionary[index]);
}

index++;
}

return result;
}

private bool[] GetSolutionFlags(int numSolutions)
{
if (!eliminationStep)
{
throw new Exception("Must call Elimination() method first!");
}

if (numSolutions < 1)

```

```

    {
        throw new ArgumentException($"{nameof(numSolutions)} must be
greater than 1.");
    }

    int numRows = RowCount;
    int numCols = ColumnCount;

    if (numSolutions >= numCols)
    {
        throw new ArgumentException($"{nameof(numSolutions)} must be
less than the column count.");
    }

    bool[] result = new bool[numCols];

    int j = -1;
    int i = numSolutions;

    while (i > 0)
    {
        j++;

        while (freeCols[j] == false)
        {
            j++;
        }

        i--;
    }

    result[j] = true;

    for (i = 0; i < numRows - 1; i++)
    {
        if (M[i][j] == true)
        {
            int h = i;
            while (h < j)
            {
                if (M[i][h] == true)
                {
                    result[h] = true;
                    break;
                }
                h++;
            }
        }
    }

    return result;
}

```

```

    public static bool[] Add(bool[] left, bool[] right)
    {
        if (left.Length != right.Length) throw new
ArgumentException($"Both vectors must have the same length.");

        int length = left.Length;
        bool[] result = new bool[length];

        int index = 0;
        while (index < length)
        {
            result[index] = left[index] ^ right[index];
            index++;
        }

        return result;
    }

    public class GaussianRow
    {
        public bool Sign { get; set; }

        public List<bool> RationalPart { get; set; }
        public List<bool> AlgebraicPart { get; set; }
        public List<bool> QuadraticPart { get; set; }

        public int LastIndexOfRational { get { return
RationalPart.LastIndexOf(true); } }
        public int LastIndexOfAlgebraic { get { return
AlgebraicPart.LastIndexOf(true); } }
        public int LastIndexOfQuadratic { get { return
QuadraticPart.LastIndexOf(true); } }

        public Relation SourceRelation { get; private set; }

        public GaussianRow(GNFS gnfs, Relation relation)
        {
            SourceRelation = relation;

            if (relation.RationalNorm.Sign == -1)
            {
                Sign = true;
            }
            else
            {
                Sign = false;
            }

            FactorPairCollection qfb =
gnfs.QuadraticFactorPairCollection;

```

```

        BigInteger          rationalMaxValue          =
gnfs.PrimeFactorBase.RationalFactorBaseMax;
        BigInteger          algebraicMaxValue         =
gnfs.PrimeFactorBase.AlgebraicFactorBaseMax;

        RationalPart    =    GetVector(relation.RationalFactorization,
rationalMaxValue).ToList();
        AlgebraicPart    =    GetVector(relation.AlgebraicFactorization,
algebraicMaxValue).ToList();
        QuadraticPart    =          qfb.Select(qf          =>
QuadraticResidue.GetQuadraticCharacter(relation, qf)).ToList();
    }

    protected    static    bool[]    GetVector(CountDictionary
primeFactorizationDict, BigInteger maxValue)
    {
        int primeIndex = PrimeFactory.GetIndexFromValue(maxValue);

        bool[] result = new bool[primeIndex];

        if (primeFactorizationDict.Any())
        {
            foreach    (KeyValuePair<BigInteger,    BigInteger>    kvp    in
primeFactorizationDict)
            {
                if (kvp.Key > maxValue)
                {
                    continue;
                }
                if (kvp.Key == -1)
                {
                    continue;
                }
                if (kvp.Value % 2 == 0)
                {
                    continue;
                }

                int index = PrimeFactory.GetIndexFromValue(kvp.Key);
                result[index] = true;
            }
        }

        return result;
    }

    public bool[] GetBoolArray()
    {
        List<bool> result = new List<bool>() { Sign };
        result.AddRange(RationalPart);
        result.AddRange(AlgebraicPart);
        result.AddRange(QuadraticPart);
        //result.Add(false);
    }

```

```

return result.ToArray();
}

public void ResizeRationalPart(int size)
{
RationalPart = RationalPart.Take(size + 1).ToList();
}

public void ResizeAlgebraicPart(int size)
{
AlgebraicPart = AlgebraicPart.Take(size + 1).ToList();
}

public void ResizeQuadraticPart(int size)
{
QuadraticPart = QuadraticPart.Take(size + 1).ToList();
}

public class QuadraticResidue
{
    public static bool GetQuadraticCharacter(Relation rel,
FactorPair quadraticFactor)
    {
        BigInteger ab = rel.A + rel.B;
        BigInteger abp = BigInteger.Abs(BigInteger.Multiply(ab,
quadraticFactor.P));

        int legendreSymbol = Legendre.Symbol(abp, quadraticFactor.R);
        return (legendreSymbol != 1);
    }
}

public static class Legendre
{
    /// <summary>
    /// Legendre Symbol returns 1 for a (nonzero) quadratic
    residue mod p, -1 for a non-quadratic residue (non-residue), or 0
    on zero.
    /// </summary>
    public static int Symbol(BigInteger a, BigInteger p)
    {
        if (p < 2) { throw new ArgumentOutOfRangeException(nameof(p),
$"Parameter '{nameof(p)}' must not be < 2, but you have supplied:
{p}"); }
        if (a == 0) { return 0; }
        if (a == 1) { return 1; }

        int result;
        if (a.Mod(2) == 0)
        {
            result = Symbol(a >> 2, p); // >> right shift == /2

```



```

        if ((p * p - 1) & 8) != 0) // instead of dividing by 8, shift
the mask bit
    {
        result = -result;
    }
    else
    {
        result = Symbol(p.Mod(a), a);
        if ((a - 1) * (p - 1) & 4) != 0) // instead of dividing by
4, shift the mask bit
    {
        result = -result;
    }
    }
    return result;
}

    /// <summary>
    /// Find r such that  $(r \mid m) = \text{goal}$ , where  $(r \mid m)$  is the
Legendre symbol, and m = modulus
    /// </summary>
    public static BigInteger SymbolSearch(BigInteger start,
BigInteger modulus, BigInteger goal)
    {
        if (goal != -1 && goal != 0 && goal != 1)
        {
            throw new Exception($"Parameter '{nameof(goal)}' may only be
-1, 0 or 1. It was {goal}.");
        }

        BigInteger counter = start;
        BigInteger max = counter + modulus + 1;
        do
        {
            if (Symbol(counter, modulus) == goal)
            {
                return counter;
            }
            counter++;
        }
        while (counter <= max);

        //return counter;
        throw new Exception("Legendre symbol matching criteria not
found.");
    }
}

    public static class BigIntegerCollectionExtensionMethods
    {
        public static BigInteger Product(this IEnumerable<BigInteger>
input)

```

```

{
    BigInteger result = 1;
    foreach (BigInteger bi in input)
    {
        result = BigInteger.Multiply(result, bi);
    }
    return result;
}

}

public partial class SquareFinder
{
    public BigInteger RationalProduct { get; set; }
    public BigInteger RationalSquare { get; set; }
    public BigInteger RationalSquareRootResidue { get; set; }
    public bool IsRationalSquare { get; set; }
    public bool IsRationalIrreducible { get; set; }

    public BigInteger AlgebraicProduct { get; set; }
    public BigInteger AlgebraicSquare { get; set; }
    public BigInteger AlgebraicProductModF { get; set; }
    public BigInteger AlgebraicSquareResidue { get; set; }
    public BigInteger AlgebraicSquareRootResidue { get; set; }
    public List<BigInteger> AlgebraicPrimes { get; set; }
    public List<BigInteger> AlgebraicResults { get; set; }
    public bool IsAlgebraicSquare { get; set; }
    public bool IsAlgebraicIrreducible { get; set; }

    public BigInteger N { get; set; }
    public Polynomial S { get; set; }
    public Polynomial TotalS { get; set; }
    public List<Tuple<BigInteger, BigInteger>> RootsOfS { get;
set; }
    public Polynomial PolynomialRing { get; set; }
    public List<Polynomial> PolynomialRingElements { get; set; }

    public BigInteger PolynomialBase { get; set; }
    public Polynomial MonicPolynomial { get; set; }
    public Polynomial PolynomialDerivative { get; set; }
    public Polynomial MonicPolynomialDerivative { get; set; }

    public Polynomial PolynomialDerivativeSquared { get; set; }
    public Polynomial PolynomialDerivativeSquaredInField { get;
set; }

    public BigInteger PolynomialDerivativeValue { get; set; }
    public BigInteger PolynomialDerivativeValueSquared { get;
set; }

    public Polynomial MonicPolynomialDerivativeSquared { get;
set; }

```

```

    public Polynomial MonicPolynomialDerivativeSquaredInField {
get; set; }

    public BigInteger MonicPolynomialDerivativeValue { get; set;
}
    public BigInteger MonicPolynomialDerivativeValueSquared {
get; set; }

    private GNFS gnfs { get; set; }
    private List<BigInteger> rationalNorms { get; set; }
    private List<BigInteger> algebraicNormCollection { get; set;
}
    private List<Relation> relationsSet { get; set; }

    public SquareFinder(GNFS sieve)
    {
        RationalSquareRootResidue = -1;
        RootsOfS = new List<Tuple<BigInteger, BigInteger>>();

        gnfs = sieve;
        N = gnfs.N;
        PolynomialBase = gnfs.PolynomialBase;

        PolynomialDerivative =
Polynomial.GetDerivativePolynomial(gnfs.CurrentPolynomial);
        PolynomialDerivativeSquared =
Polynomial.Square(PolynomialDerivative);
        PolynomialDerivativeSquaredInField =
Polynomial.Field.Modulus(PolynomialDerivativeSquared,
gnfs.CurrentPolynomial);

        PolynomialDerivativeValue =
PolynomialDerivative.Evaluate(gnfs.PolynomialBase);
        PolynomialDerivativeValueSquared =
BigInteger.Pow(PolynomialDerivativeValue, 2);

        MonicPolynomial =
Polynomial.MakeMonic(gnfs.CurrentPolynomial, PolynomialBase);
        MonicPolynomialDerivative =
Polynomial.GetDerivativePolynomial(MonicPolynomial);
        MonicPolynomialDerivativeSquared =
Polynomial.Square(MonicPolynomialDerivative);
        MonicPolynomialDerivativeSquaredInField =
Polynomial.Field.Modulus(MonicPolynomialDerivativeSquared,
MonicPolynomial);

        MonicPolynomialDerivativeValue =
MonicPolynomialDerivative.Evaluate(gnfs.PolynomialBase);
        MonicPolynomialDerivativeValueSquared =
MonicPolynomialDerivativeSquared.Evaluate(gnfs.PolynomialBase);
    }

    public static bool Solve(GNFS gnfs)

```

```

{
    List<int> triedFreeRelationIndices = new List<int>();

    BigInteger polyBase = gnfs.PolynomialBase;
    List<List<Relation>> freeRelations =
gnfs.CurrentRelationsProgress.FreeRelations;
    SquareFinder squareRootFinder = new SquareFinder(gnfs);

    int freeRelationIndex = 0;
    bool solutionFound = false;

    // Below randomly selects a solution set to try and find a
    square root of the polynomial in.
    while (!solutionFound)
    {
        // Each time this step is stopped and restarted, it will try
        a different solution set.
        // Previous used sets are tracked with the List<int>
        triedFreeRelationIndices
        if (triedFreeRelationIndices.Count == freeRelations.Count) //
        If we have exhausted our solution sets, alert the user. Number
        wont factor for some reason.
        {
            break;
        }

        do
        {
            // Below randomly selects a solution set to try and find a
            square root of the polynomial in.
            freeRelationIndex = StaticRandom.Next(0,
freeRelations.Count);
        }
        while
        (triedFreeRelationIndices.Contains(freeRelationIndex));

        triedFreeRelationIndices.Add(freeRelationIndex); // Add
        current selection to our list

        List<Relation> selectedRelationSet =
freeRelations[freeRelationIndex]; // Get the solution set

        squareRootFinder.CalculateRationalSide(selectedRelationSet);

        Tuple<BigInteger, BigInteger> foundFactors =
squareRootFinder.CalculateAlgebraicSide();

        BigInteger P = foundFactors.Item1;
        BigInteger Q = foundFactors.Item2;

        bool nonTrivialFactorsFound = (P != 1 || Q != 1);
        if (nonTrivialFactorsFound)
        {

```

```

        solutionFound = gnfs.SetFactorizationSolution(P, Q);

        break;
    }
    else
    {

    }
}

return solutionFound;
}

public void CalculateRationalSide(List<Relation> relations)
{
    relationsSet = relations;
    rationalNorms = relationsSet.Select(rel =>
rel.RationalNorm).ToList();

    CountDictionary rationalSquareFactorization = new
CountDictionary();
    foreach (var rel in relationsSet)
    {
        rationalSquareFactorization.Combine(rel.RationalFactorizatio
n);
    }

    string rationalSquareFactorizationString =
rationalSquareFactorization.FormatStringAsFactorization();

    RationalProduct = rationalNorms.Product();

    BigInteger RationalProductSquareRoot =
RationalProduct.SquareRoot();

    var product = PolynomialDerivativeValue *
RationalProductSquareRoot;

    RationalSquareRootResidue = product.Mod(N);

    IsRationalSquare = RationalProduct.IsSquare();
    if (!IsRationalSquare) // This is an error in implementation.
This should never happen, and so must be a bug
    {
        throw new Exception($"{nameof(IsRationalSquare)} evaluated to
false. This is a sign that there is a bug in the implementation,
as this should never be the case if the algorithm has been
correctly implemented.");
    }
}

public Tuple<BigInteger, BigInteger>
CalculateAlgebraicSide()

```

```

        {
            RootsOfS.AddRange(relationsSet.Select(rel => new
Tuple<BigInteger, BigInteger>(rel.A, rel.B)));

            PolynomialRingElements = new List<Polynomial>();
            foreach (Relation rel in relationsSet)
            {
                // poly(x) = A + (B * x)
                Polynomial newPoly =
                new Polynomial(
                new Term[]
                {
                    new Term( rel.B, 1),
                    new Term( rel.A, 0)
                }
                );

                PolynomialRingElements.Add(newPoly);
            }

            PolynomialRing = Polynomial.Product(PolynomialRingElements);
            Polynomial PolynomialRingInField =
            Polynomial.Field.Modulus(PolynomialRing, MonicPolynomial);

            // Multiply the product of the polynomial elements by f'(x)^2
            // This will guarantee that the square root of product of
            polynomials
            // is an element of the number field defined by the algebraic
            polynomial.
            TotalS = Polynomial.Multiply(PolynomialRing,
            MonicPolynomialDerivativeSquared);
            S = Polynomial.Field.Modulus(TotalS, MonicPolynomial);

            bool solutionFound = false;

            int degree = MonicPolynomial.Degree;
            Polynomial f = MonicPolynomial; // gnfs.CurrentPolynomial;

            BigInteger lastP =
            gnfs.QuadraticFactorPairCollection.Last().P;
            //quadraticPrimes.First(); //BigInteger.Max(fromRoot,
            fromQuadraticFactorPairs); //N / N.ToString().Length; //((N * 3)
            + 1).NthRoot(3); //gnfs.QFB.Select(fp => fp.P).Max();
            lastP = PrimeFactory.GetNextPrime(lastP + 1);

            List<BigInteger> primes = new List<BigInteger>();
            List<BigInteger> values = new List<BigInteger>();

            int attempts = 7;
            while (!solutionFound && attempts > 0)
            {
                if (primes.Count > 0 && values.Count > 0)
                {

```

```

primes.Clear();
values.Clear();
}

do
{
lastP = PrimeFactory.GetNextPrime(lastP + 1);

Polynomial g = Polynomial.Parse($"X^{lastP} - X");
Polynomial h = FiniteFieldArithmetic.ModMod(g, f, lastP);

Polynomial gcd = Polynomial.Field.GCD(h, f, lastP);

bool isIrreducible = gcd.CompareTo(Polynomial.One) == 0;
if (!isIrreducible)
{
continue;
}

primes.Add(lastP);
}
while (primes.Count < degree);

if (primes.Count > degree)
{
primes.Remove(primes.First());
values.Remove(values.First());
}

BigInteger primeProduct = primes.Product();

if (primeProduct < N)
{
continue;
}

bool takeInverse = false;
foreach (BigInteger p in primes)
{
Polynomial choosenPoly = FiniteFieldArithmetic.SquareRoot(S,
f, p, degree, gnfs.PolynomialBase);
BigInteger choosenX;

//if (takeInverse)
//{
//    Polynomial inverse = ModularInverse(choosenPoly, p);
//    BigInteger inverseEval =
inverse.Evaluate(gnfs.PolynomialBase);
//    BigInteger inverseX = inverseEval.Mod(p);
//
//    choosenPoly = inverse;
//    choosenX = inverseX;
//}

```

```

//else
//{
BigInteger eval = choosenPoly.Evaluate(gnfs.PolynomialBase);
BigInteger x = eval.Mod(p);

choosenX = x;
//}

values.Add(choosenX);

takeInverse = !takeInverse;
}

BigInteger commonModulus =
Polynomial.Algorithms.ChineseRemainderTheorem(primes.ToArray(),
values.ToArray());
//FiniteFieldArithmetic.ChineseRemainder(primes, values);
AlgebraicSquareRootResidue = commonModulus.Mod(N);

int index = -1;
while ((++index) < primes.Count)
{
var tp = primes[index];
var tv = values[index];
}

BigInteger algebraicSquareRoot = 1;

BigInteger min;
BigInteger max;
BigInteger A;
BigInteger B;
BigInteger U;
BigInteger V;
BigInteger P = 0;
BigInteger Q;

min = BigInteger.Min(RationalSquareRootResidue,
AlgebraicSquareRootResidue);
max = BigInteger.Max(RationalSquareRootResidue,
AlgebraicSquareRootResidue);

A = max + min;
B = max - min;

U = GCD.FindGCD(N, A);
V = GCD.FindGCD(N, B);

if (U > 1 && U != N)
{
P = U;
solutionFound = true;
}

```



```

else if (V > 1 && V != N)
{
P = V;
solutionFound = true;
}

if (solutionFound)
{
BigInteger rem;
BigInteger other = BigInteger.DivRem(N, P, out rem);

if (rem != 0)
{
solutionFound = false;
}
else
{
Q = other;
AlgebraicResults = values;
//AlgebraicSquareRootResidue = AlgebraicSquareRootResidue;
AlgebraicPrimes = primes;

return new Tuple<BigInteger, BigInteger>(P, Q);
}

if (!solutionFound)
{
attempts--;
}
}

return new Tuple<BigInteger, BigInteger>(1, 1);
}

private static Tuple<BigInteger, BigInteger>
AlgebraicSquareRoot(Polynomial f, BigInteger m, int degree,
Polynomial dd, BigInteger p)
{
Polynomial startPolynomial = Polynomial.Field.Modulus(dd, p);
Polynomial startInversePolynomial =
ModularInverse(startPolynomial, p);

Polynomial startSquared1 =
FiniteFieldArithmetic.ModMod(Polynomial.Square(startPolynomial),
f, p);
Polynomial startSquared2 =
FiniteFieldArithmetic.ModMod(Polynomial.Square(startInversePolyn
omial), f, p);

Polynomial resultPoly1 =
FiniteFieldArithmetic.SquareRoot(startPolynomial, f, p, degree,
m);

```

```

        Polynomial resultPoly2 = ModularInverse(resultPoly1, p);

        Polynomial resultSquared1 =
FiniteFieldArithmetic.ModMod(Polynomial.Square(resultPoly1), f,
p);
        Polynomial resultSquared2 =
FiniteFieldArithmetic.ModMod(Polynomial.Square(resultPoly2), f,
p);

        bool bothResultsAgree =
(resultSquared1.CompareTo(resultSquared2) == 0);

        BigInteger result1 = resultPoly1.Evaluate(m).Mod(p);
        BigInteger result2 = resultPoly2.Evaluate(m).Mod(p);

        BigInteger inversePrime = p - result1;
        bool testEvaluationsAreModularInverses = inversePrime ==
result2;

        if (bothResultsAgree && testEvaluationsAreModularInverses)
        {
            return new Tuple<BigInteger,
BigInteger>(BigInteger.Min(result1, result2),
BigInteger.Max(result1, result2));
        }

        return new Tuple<BigInteger, BigInteger>(BigInteger.Zero,
BigInteger.Zero);
    }

    private static Polynomial ModularInverse(Polynomial poly,
BigInteger mod)
    {
        return new Polynomial(Term.GetTerms(poly.Terms.Select(trm =>
(mod - trm.Coefficient).Mod(mod)).ToArray()));
    }

}

public static class GCD
{
    public static BigInteger FindGCD(BigInteger left, BigInteger
right)
    {
        return BigInteger.GreatestCommonDivisor(left, right);
    }

}

public class Solution
{
    [DataMember]
    public BigInteger P { get; private set; }
}

```

```

[DataMember]
public BigInteger Q { get; private set; }

public Solution(BigInteger p, BigInteger q)
{
    P = p;
    Q = q;
}
}

public static class FiniteFieldArithmetic
{
    /// <summary>
    /// Tonelli-Shanks algorithm for finding polynomial modular
square roots
    /// </summary>
    /// <returns></returns>
    public static Polynomial SquareRoot(Polynomial
startPolynomial, Polynomial f, BigInteger p, int degree,
BigInteger m)
    {
        BigInteger q = BigInteger.Pow(p, degree);
        BigInteger s = q - 1;

        int r = 0;
        while (s.Mod(2) == 0)
        {
            s /= 2;
            r++;
        }

        BigInteger halfS = ((s + 1) / 2);
        if (r == 1 && q.Mod(4) == 3)
        {
            halfS = ((q + 1) / 4);
        }

        BigInteger quadraticNonResidue = Legendre.SymbolSearch(m + 1,
q, -1);
        BigInteger theta = quadraticNonResidue;
        BigInteger minusOne = BigInteger.ModPow(theta, ((q - 1) / 2),
p);

        Polynomial omegaPoly =
Polynomial.Field.ExponentiateMod(startPolynomial, halfS, f, p);

        BigInteger lambda = minusOne;
        BigInteger zeta = 0;

        int i = 0;
        do
        {
            i++;

```

```

        zeta = BigInteger.ModPow(theta, (i * s), p);

        lambda = (lambda * BigInteger.Pow(zeta, (int)Math.Pow(2, (r
- i))))).Mod(p);

        omegaPoly      =      Polynomial.Field.Multiply(omegaPoly,
BigInteger.Pow(zeta, (int)Math.Pow(2, ((r - i) - 1))), p);
    }
    while (!((lambda == 1) || (i > (r))));

    return omegaPoly;
}

/// <summary>
/// Finds X such that a*X = 1 (mod p)
/// </summary>
/// <param name="a">a.</param>
/// <param name="p">The modulus</param>
/// <returns></returns>
public static BigInteger
ModularMultiplicativeInverse(BigInteger a, BigInteger p)
{
    if (p == 1)
    {
        return 0;
    }

    BigInteger divisor;
    BigInteger dividend = a;
    BigInteger diff = 0;
    BigInteger result = 1;
    BigInteger quotient = 0;
    BigInteger lastDivisor = 0;
    BigInteger remainder = p;

    while (dividend > 1)
    {
        divisor = remainder;
        quotient = BigInteger.DivRem(dividend, divisor, out
remainder); // Divide
        dividend = divisor;
        lastDivisor = diff; // The thing to divide will be the last
divisor

        // Update diff and result
        diff = result - quotient * diff;
        result = lastDivisor;
    }

    if (result < 0)
    {
        result += p; // Make result positive
    }
}

```

```

    return result;
}

    /// <summary>
    /// Finds N such that primes[i]  $\equiv$  values[i] (mod N) for all
    values[i] with 0 < i < a.Length
    /// </summary>
    public static BigInteger ChineseRemainder(List<BigInteger>
primes, List<BigInteger> values)
    {
        BigInteger primeProduct = primes.Product();

        int indx = 0;
        BigInteger Z = 0;
        foreach (BigInteger pi in primes)
        {
            BigInteger Pj = primeProduct / pi;
            BigInteger Aj = ModularMultiplicativeInverse(Pj, pi);
            BigInteger AXPj = values[indx] * Aj * Pj;

            Z += AXPj;
            indx++;
        }

        BigInteger r = Z / primeProduct;
        BigInteger rP = r * primeProduct;
        BigInteger finalResult_sqrt = (Z - rP);
        return finalResult_sqrt;
    }

    /// <summary>
    /// Reduce a polynomial by a modulus polynomial and modulus
    integer.
    /// </summary>
    public static Polynomial ModMod(Polynomial toReduce,
Polynomial modPoly, BigInteger primeModulus)
    {
        int compare = modPoly.CompareTo(toReduce);
        if (compare > 0)
        {
            return toReduce;
        }
        if (compare == 0)
        {
            return Polynomial.Zero;
        }

        return Remainder(toReduce, modPoly, primeModulus);
    }

    public static Polynomial Remainder(Polynomial left,
Polynomial right, BigInteger mod)
    {

```

```

    if (left == null)
    {
        throw new ArgumentNullException("left");
    }
    if (right == null)
    {
        throw new ArgumentNullException("right");
    }
    if (right.Degree > left.Degree || right.CompareTo(left) == 1)
    {
        return Polynomial.Zero.Clone();
    }

    int rightDegree = right.Degree;
    int quotientDegree = left.Degree - rightDegree + 1;

    BigInteger leadingCoefficient = right[rightDegree].Mod(mod);
    if (leadingCoefficient != 1) { throw new
ArgumentNullException("right", "This method was expecting only
monomials (leading coefficient is 1) for the right-hand-side
polynomial."); }

    Polynomial rem = left.Clone();
    BigInteger quot = 0;

    for (int i = quotientDegree - 1; i >= 0; i--)
    {
        quot = BigInteger.Remainder(rem[rightDegree + i],
mod); //.Mod(mod);

        rem[rightDegree + i] = 0;

        for (int j = rightDegree + i - 1; j >= i; j--)
        {
            rem[j] = BigInteger.Subtract(
rem[j],
BigInteger.Multiply(quot, right[j - i]).Mod(mod)
).Mod(mod);
        }
    }

    return new Polynomial(rem.Terms);
}
}
}

```