Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
Институт Вычислительной математики и информационных технологий

ОТЧЕТ
по научно-исследовательской работе (производственной) практике

Обучающийся Гусев Виталий Евгеньевич гр.09-335 ________________
(ФИО студента) (Группа) (Подпись)

Научный руководитель: ___________________________

доцент КСАИТ Мубараков Б.Г. ____________________
(Подпись)

Руководитель практики от кафедры:

ст.преподаватель КСАИТ Тихонова О.О. ____________________
(Подпись)

Оценка за практику ________________________ ____________________
(Подпись)

Дата сдачи отчета _________________________

Казань – 2025

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Производственная практика проходила на кафедре системного анализа и информационных технологий Института вычислительной математики и информационных технологий КФУ с 21 марта 2025 года по 25 мая 2025 года.

Целью практики является исследование и реализация тестов для базовых и модифицированных алгоритмов дискретного логарифмирования с экспоненциальной и субэкспоненциальной сложностью.

Задачами практики являются:

1) разработать тесты для базовых и модифицированных методов дискретного логарифмирования,

2) программно реализовать тесты для базовых и модифицированных методов дискретного логарифмирования,

3) провести эксперименты на реализованных тестах для базовых и модифицированных методов дискретного логарифмирования.

1. Разработка тестов для базовых и модифицированных алгоритмов дискретного логарифмирования

В процессе практики были реализованы и исследованы расширенные тесты для базовых и модифицированных алгоритмов дискретного логарифмирования на языке программирования C# на .NET8 в Windows Forms (рисунок 1). Для тестирования данных алгоритмов был использован генератор параметров Диффи-Хеллмана и возведение числа в степень по модулю [1]. Также для тестирования данных алгоритмов был использован замер времени выполнения алгоритма и количество затраченной памяти на выполнение алгоритма.

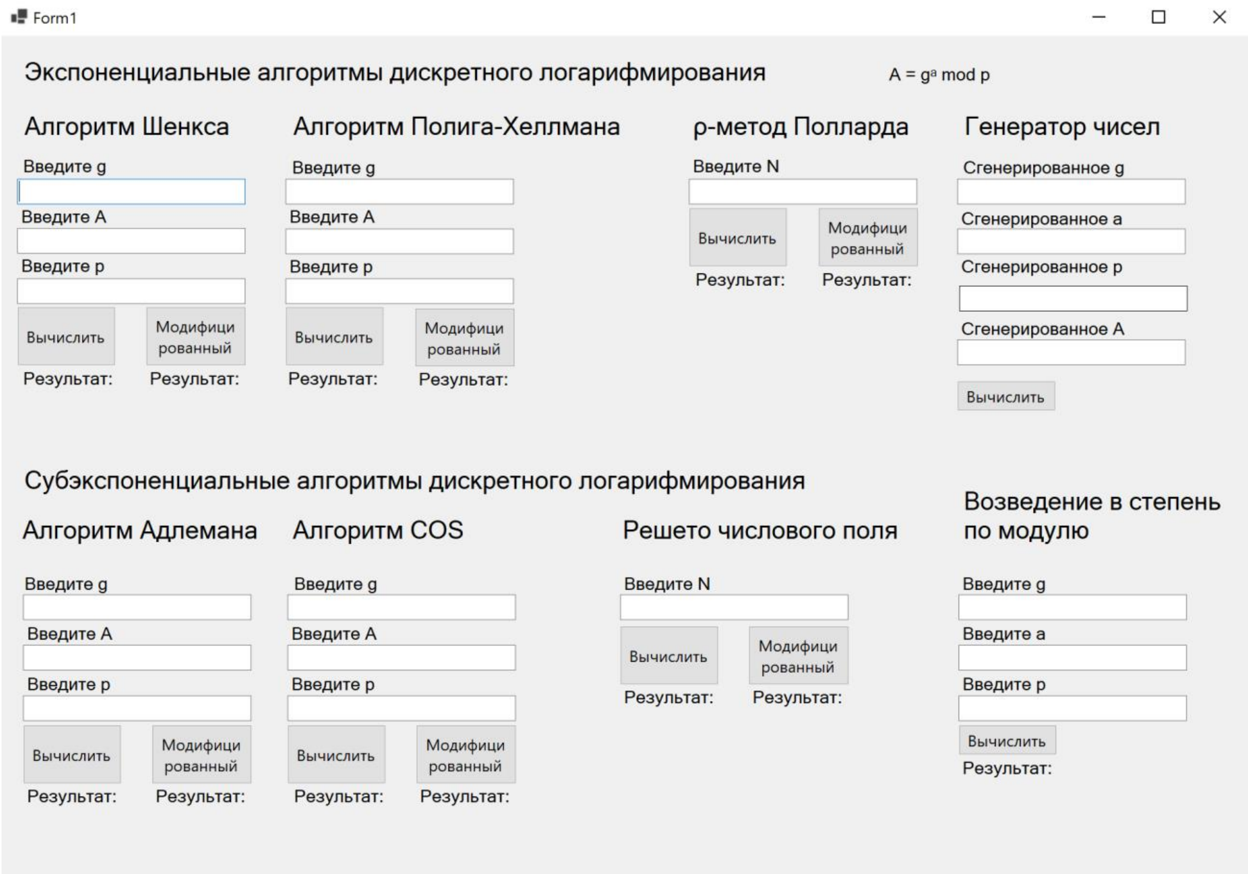

Рисунок 1 - Реализованная программа

Были реализованы и исследованы тесты для базовых и модифицированных экспоненциальных алгоритмов дискретного логарифмирования: алгоритм Шенкса [2], алгоритм Полига-Хеллмана [3], ро-метод Полларда [4], а также тесты для базовых и модифицированных субэкспоненциальных алгоритмов дискретного логарифмирования: алгоритм Адлемана [5], алгоритм COS [6], решето числового поля [7].

Разработанная программа позволяет вносить в текстовые поля необходимые значения параметров возведения чисел в степень по модулю: g, a, p, A [8, 9], либо целых чисел N для разложения на простые множители [10] и выводить результат вычисления. В процессе практики были проведены тесты алгоритмов на различных параметрах с замером времени и затраченной памятью вычисления алгоритмов (рисунок 2).

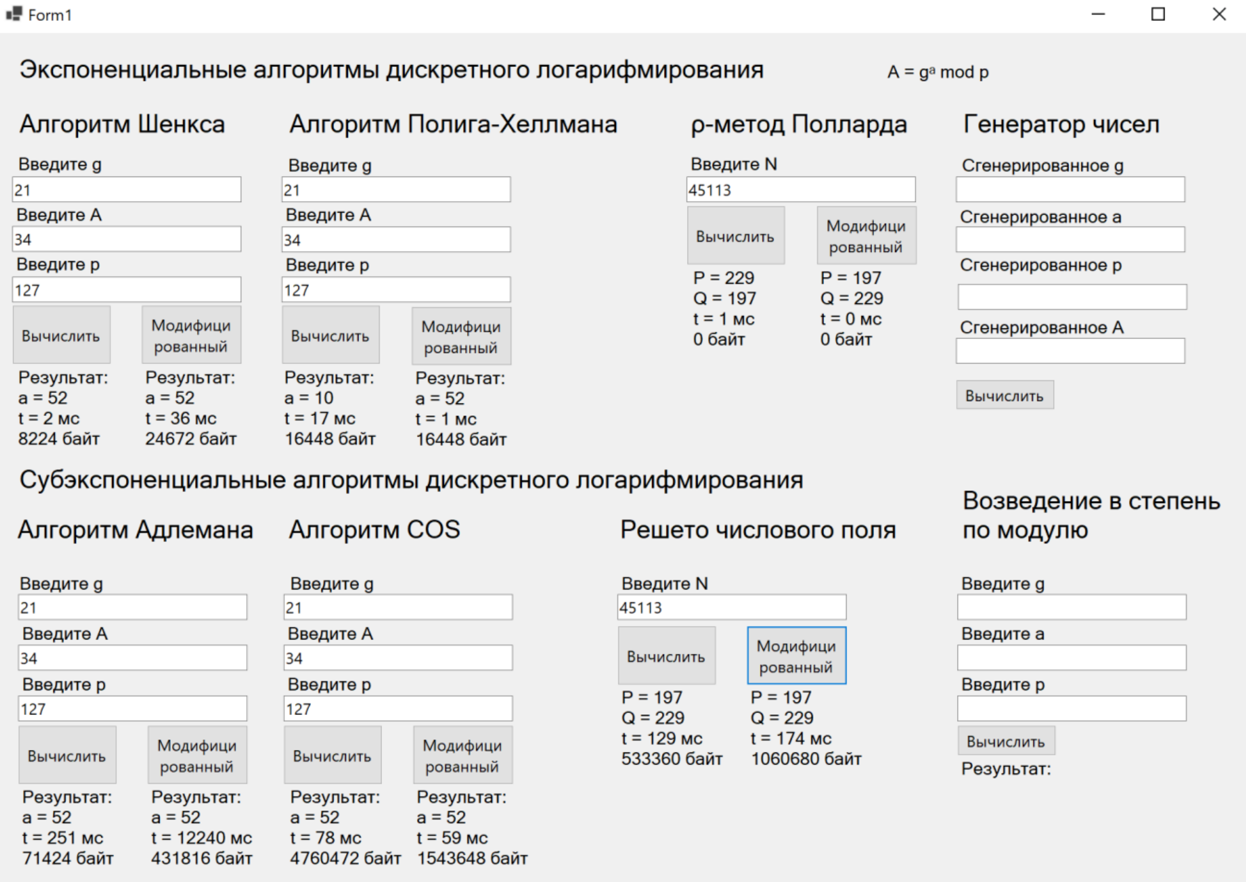

Рисунок 2 - Вычисление модифицированных алгоритмов

2. Тестирование базового и модифицированного алгоритма Шенкса

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

Пусть задано сравнение $a^x \equiv b\ (mod\ p)$, 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 $m = k = \sqrt[2]{p} + 1$;

2) вычисляются два ряда чисел:

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

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

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 $x = im - j$.

Была реализована модификация алгоритма, состоящая в распараллеливании 2 и 3 шага алгоритма. На 2 шаге алгоритма параллельно вычисляются два ряда чисел. На 3 шаге был сделан параллельный поиск результата с начала и с конца ряда.

Были сгенерированы параметры и проведены тесты базового (таблица 1) и модифицированного (таблица 2) алгоритма Шенкса, где g, p и A – 32 битные числа, а параметр a - 8 битное число:

Таблица 1- 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
1063850105	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
1769314487	117	1978813019	1603737570	1475357	5734645
608493163	53	667849967	614352815	2341533	10294564

Таблица 2 - 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
1063850105	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
1769314487	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 3) и модифицированного (таблица 4) алгоритма Шенкса, где g, p и A – 32 битные числа, а параметр a - 16 битное число:

Таблица 3- 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
1190622764	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
1506890940	15782	1556831663	467681122	1826592	5927483
463547350	2937	1648004693	633238154	1284859	6375812
56985777	14752	60477983	13103552	1683934	6839491

Таблица 4 - 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
1190622764	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
1506890940	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 лучше 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 лучше 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 3, 4).

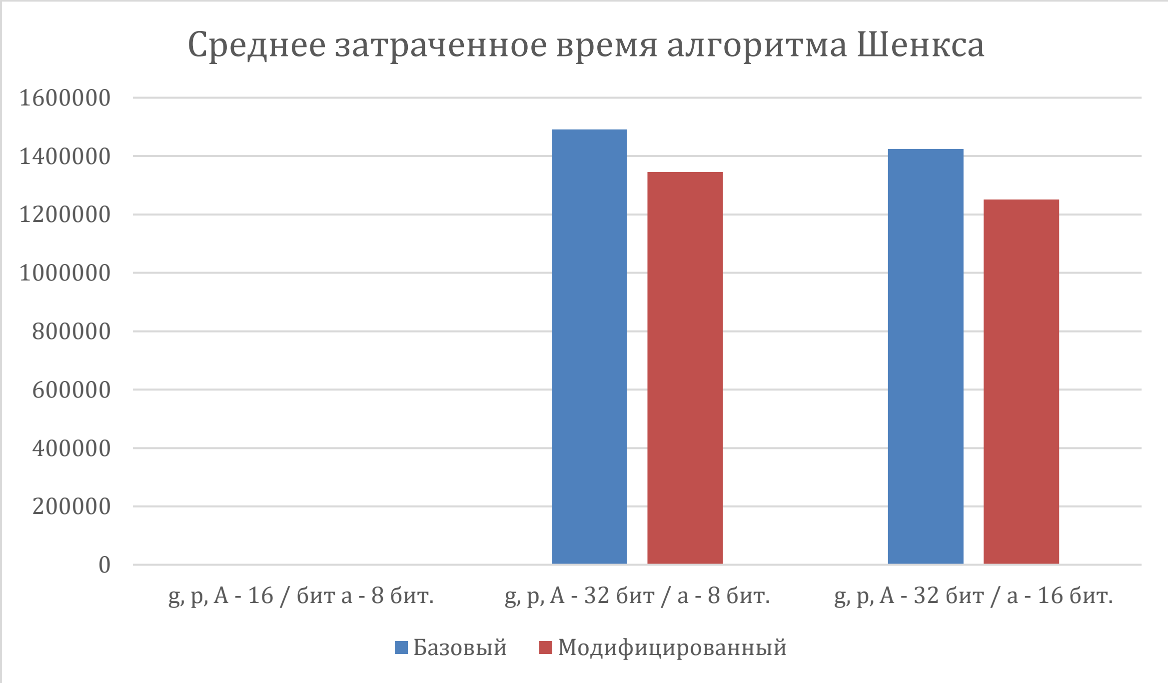

Рисунок 3 - Среднее затраченное время алгоритма Шенкса

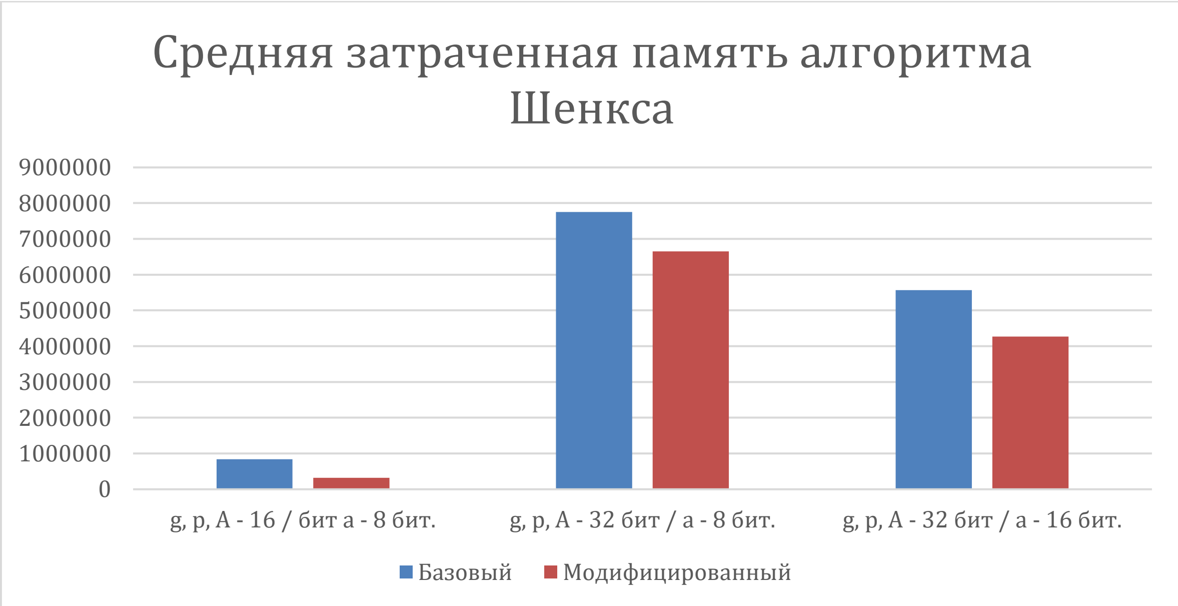

Рисунок 4 - Средняя затраченная память алгоритма Шенкса

3. Тестирование базового и модифицированного алгоритма Полига-Хеллмана

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

Пусть задано сравнение $a^x \equiv b\ (mod\ p)$, 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 $r_{i,j} = a^{j*\frac{p-1}{q_i}}, i \in \{1, \dots, k\}, j \in \{0, \dots, q_i - 1\}$;

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

Для i от 1 до k:

Пусть $x \equiv b\ \equiv x_0 + x_1 q_i + \cdots + x_{\alpha_i - 1} q_i^{\alpha_i - 1}\ (mod\ q_i^{\alpha_i})$,

где $0 \leq x_i \leq q_i - 1$.

Тогда верно сравнение:

$$a^{x_0*\frac{p-1}{q_i}} \equiv b^{\frac{p-1}{q_i}}\ (mod\ p).$$

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

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

$$a^{x_j*\frac{p-1}{q_i}} \equiv \left(ba^{-x_0 - x_1 q_i - \cdots - x_{j-1} q_i^{j-1}}\right)^{\frac{p-1}{q_i^{j+1}}}\ (mod\ p).$$

Решение находится по таблице

Конец цикла по j.

Конец цикла по i;

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

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.

Были сгенерированы параметры и проведены тесты базового (таблица 5) и модифицированного (таблица 6) алгоритма Полига-Хеллмана, где g, p и A - 32 битные числа, а параметр a - 8 битное число:

Таблица 5 - 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 6 - 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
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 лучше результаты в скорости выполнения и в затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 7) и модифицированного (таблица 8) алгоритма Полига-Хеллмана, где g, p и A - 32 битные числа, а параметр a - 16 битное число:

Таблица 7 - 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
1414779283	29584	1705294571	1255511029	111	3749152
406221477	24407	1048450831	883157096	19541	4059032
19992566	3706	21380063	6707478	3024	32183720
40746430	30170	507416659	424602148	6975	14569512
1715375241	16283	1980316259	1065914095	157	5296416
83622979	15419	750177383	692438560	3213	33511504

Таблица 8 - 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
1414779283	29584	1705294571	1255511029	104	2257800
406221477	24407	1048450831	883157096	19492	4213960
19992566	3706	21380063	6707478	3109	152175104
40746430	30170	507416659	424602148	6449	1567296
1715375241	16283	1980316259	1065914095	168	666952
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 лучше 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 Поллига-Хеллмана показал лучше 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 5, 6).

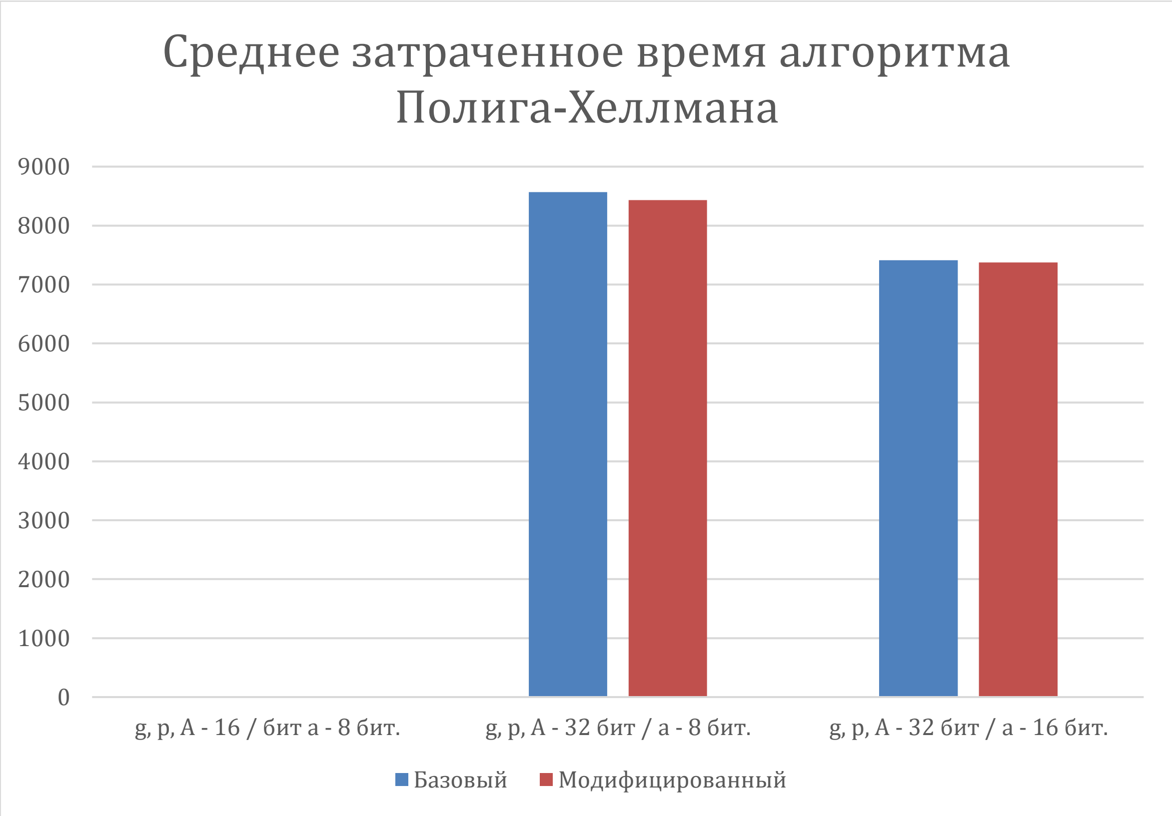

Рисунок 5 - Среднее затраченное время алгоритма Полига-Хеллмана

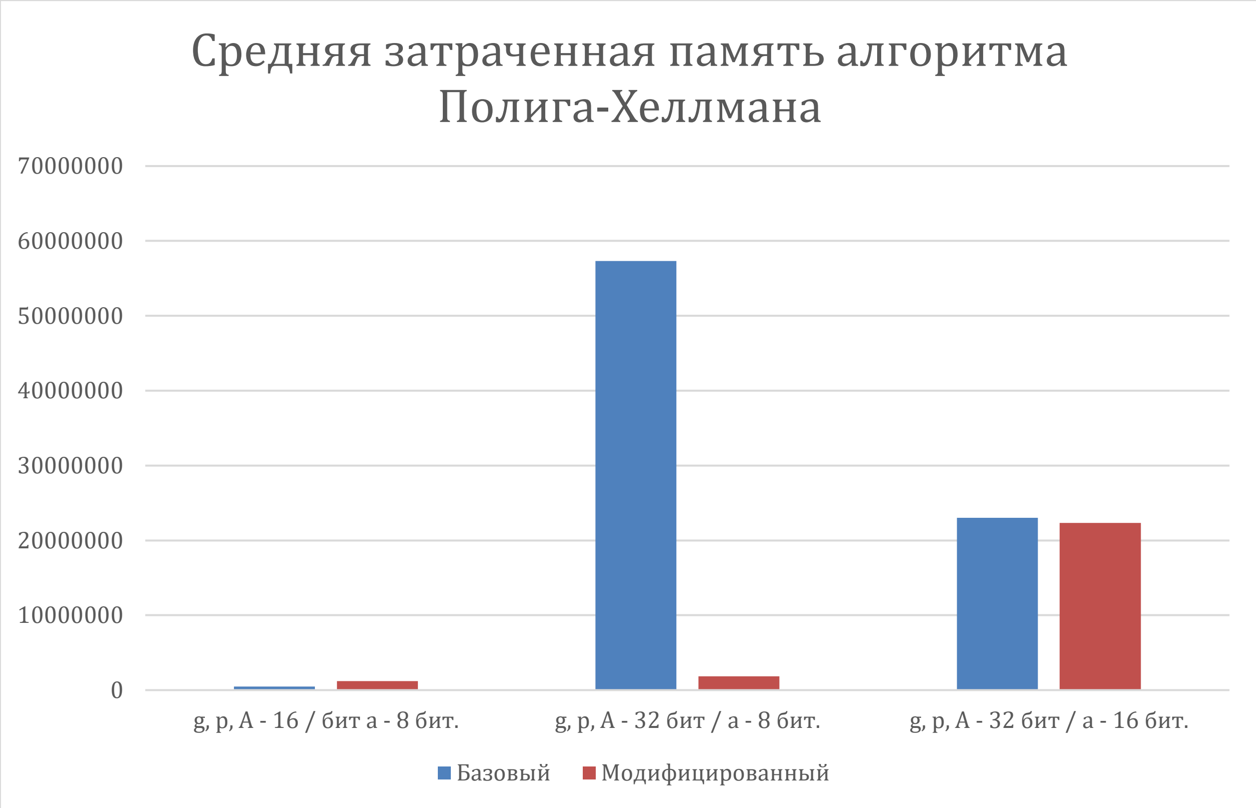

Рисунок 6 - Средняя затраченная память алгоритма Полига-Хеллмана

4. Тестирование базового и модифицированного алгоритма ро-метод Полларда

Были проведены расширенные тесты базового и модифицированного алгоритма дискретного логарифмирования ро-метод Полларда для факторизации (разложения на множители) целых чисел. Данный алгоритм основывается на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Ро-метод Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера 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 $GCD(N, abs(x-y))$.

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. 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 9) и модифицированного (таблица 10) алгоритма ро-метод Полларада, где N - 128 битное число:

Таблица 9 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
53047421620217647340165842779605199029	827	64144403410178533664045759104722127	1	8224
64675021144784783043664114042046884567	3740311	17291348538874115827176968450497	4	622464
151275534861495237718128655744312609603	6473	23370235572608564455141148732320811	1	49088
62698579182392202126194165118939962221	47	1334012323029621321833918406785956643	1	8224
74664397703116672764781934238110756297	11	6787672518465152069525630385282796027	1	8224
106698373966731480576987119246859208349	545087	195745585506041201820970082293027	2	230272
10798205232977513892092037158655880069	3889	2776602014136671095935211406185621	2	41120
140429661715602135546397235345102518067	7096693	19788042362210417661634402861319	5	616800

Продолжение таблицы 9

31770883779024878705711637938201735687	577	55062190258275353042827795386831431	1	8224
14224681613014291551459483673522113370 3	107	13294094965433917337812601564039358 29	1	8224

Таблица 10 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
53047421620217647340165842779605199029	827	64144403410178533664045759104722127	4	278336
64675021144784783043664114042046884567	3740311	17291348538874115827176968450497	5	917368
15127553486149523771812865574431260960 3	6473	23370235572608564455141148732320811	1	139296
62698579182392202126194165118939962221	47	13340123230296213218339184067859566 43	1	32640
74664397703116672764781934238110756297	11	67876725184651520695256303852827960 27	1	8168
10669837396673148057698711924685920834 9	545087	19574558550604120182097008229302 7	8	2702520
10798205232977513892092037158655880069	3889	27766020141366710959352114061856 21	1	57568

Продолжение таблицы 10

14042966171560213554639723534510251806 7	7096693	19788042362210417661634402861319	7	1046152
31770883779024878705711637938201735687	577	55062190258275353042827795386831431	1	16448
14224681613014291551459483673522113370 3	107	13294094965433917337812601564039358 2 9	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 11) и модифицированного (таблица 12) алгоритма ро-метод Полларада, где N - 256 битное число:

Таблица 11 - Результаты тестов базового алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
5592095948104737821955765870996274560616957804364870799251318392625233561975 3	5813	9619982707904245350001317514185918734933696549741735419321036285266185381	18	20280

Продолжение таблицы 11

7356173465255652220674183174275373645634347704726206261672906559229173461517	467	1575197744166092552606891472007574656452751114502399627767217678635797315 1	1	8224
48788063155774603088854278430316470438712906061285274940421741055656164838577	151	323099755998507305224200519406069340653727854710498509539216828183153409527	1	8224
1518487621654552869181440874207131576662148200043436787384651729817773968283	19	7992040113971330890428636180037534614011306316018088354656061735883020885 7	1	1
49068057663174483480639231400960337418957239851453375240361727223062622312109	31	15828405697798220477625558516438818522244270919823669432374750717116974939 39	1	1
48480866470199234659403676777532059900639989247808325258141313563016593623489	1031	4702314885567336048438765933805243443321046483783542702050563876141279691 9	1	8224

Продолжение таблицы 11

29251248596325718549061040098085324119415052922282259303535540345964059110299	973304565503	30053540929615168774498258111542608750950552379823813377967832933	6	3027
42037974065514056976565180303117273081734747207340623920607376146335258775061	11	3821634005955823361505925482101570280157704291576420356418852376939568979551	1	8176
7611374973571945283956682495229561865105885227436597349795001323999114734667	23	330929346677041099302464456314328776743734140323330319556304405391265858029	1	8224
2677482936572229301232118220273638489056478111125400129390049169193770643059	23	116412301590096926140526879142332108219846874396756527364784746486685680133	1	8224

Таблица 12 - Результаты тестов модифицированного алгоритма ро-метод Полларда

N	P	Q	Время (мс)	Память (байт)
559209594810473782195576587099627456061695780436487079925131839262523356197 53	5813	961998270790424535000131751418591873493369654974173541932103628526618538 1	7	769984
7356173465255652220674183174275373645634347704726206261672906559229173461517	467	15751977441660925526068914720075746564527511145023996277672176786357973151	1	8224
48788063155774603088854278430316470438712906061285274940421741055656164838577	151	323099755998507305224200519406069340653727854710498509539216828183153409527	1	7968
1518487621654552869181440874207131576662148200043436787384651729817773968283	19	79920401139713308904286361800375346140113063160180883546560617358830208857	1	8224
49068057663174483480639231400960337418957239851453375240361727223062622312109	31	1582840569779822047762555851643881852224427091982366943237475071711697493939	1	1

Продолжение таблицы 12

48480866470199234659403676777532059900639989247808325258141313563016593623489	1031	47023148855673360484387659338052434433210464837835427020505638761412796919	2	344128
29251248596325718549061040098085324119415052922282259303535540345964059110299	973304565503	30053540929615168774498258111542608750950552379823813377967832933	2	344128
42037974065514056976565180303117273081734747207340623920607376146335258775061	11	3821634005955823361505925482101570280157704291576420356418852376939568979551	1	8224
7611374973571945283956682495229561865105885227436597349795001323999114734667	23	330929346677041099302464456314328776743734140323330319556304405391265858029	1	16448
2677482936572229301232118220273638489056478111125400129390049169193770643059	23	116412301590096926140526879142332108219846874396756527364784746486685680133	1	16448

В результате тестов, где N - 128 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 7, 8).

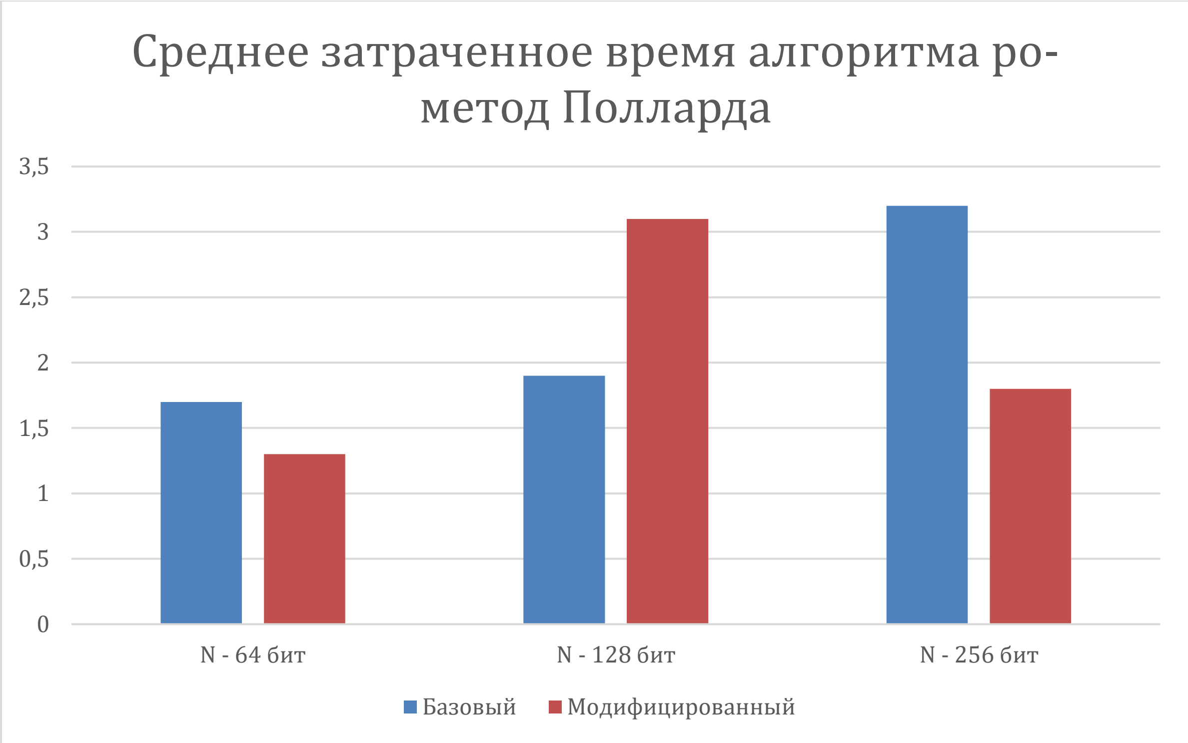

Рисунок 7 - Среднее затраченное время алгоритма ро-метод Полларда

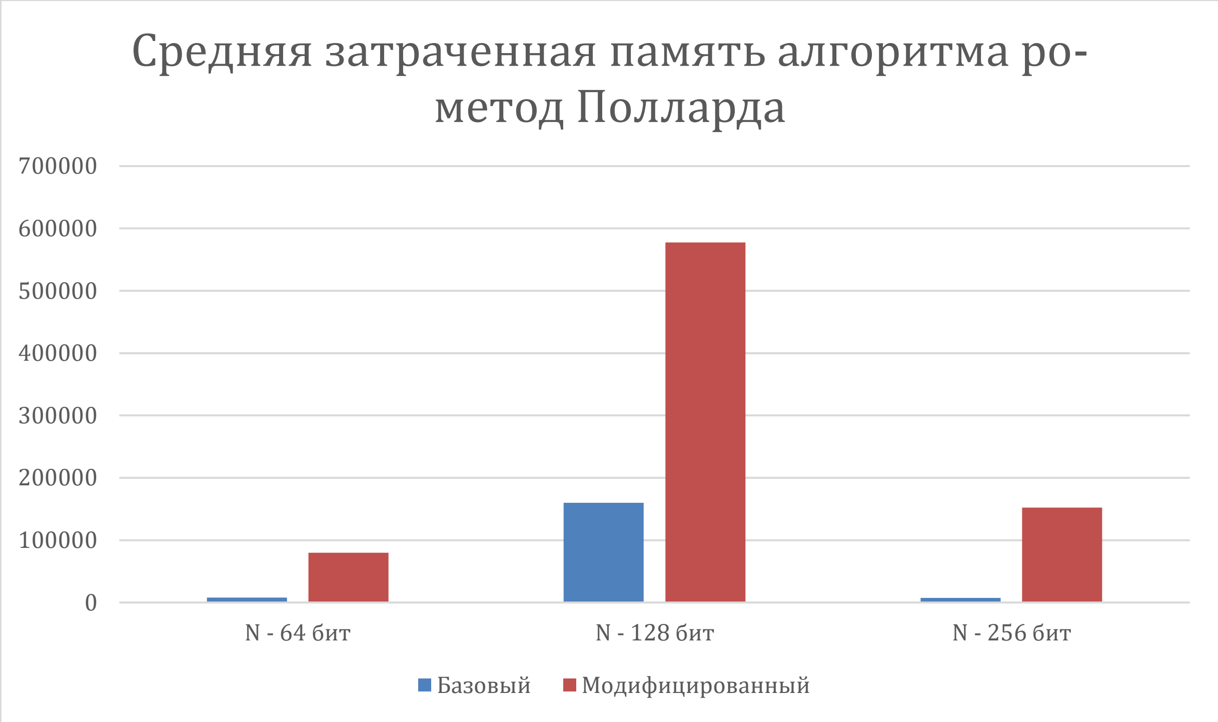

Рисунок 8 - Средняя затраченная память алгоритма ро-метод Полларда

5. Тестирование базового и модифицированного алгоритма Адлемана

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

Пусть задано сравнение $a^x \equiv b \ (mod\ p)$, 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

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

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

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 $a^r \equiv \prod_{q \leq B} q^{\beta_q} p_1 * \ldots * p_k \ (mod\ p)$, где $p_1, \ldots, p_k\ mod\ p$ – 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 $\log_a p_i$;

6) определяется искомый дискретный логарифм:

$x \equiv \log_a b \equiv \sum_{q \le B} \beta_q \log_a q + \sum_{i=1}^{k} \log_a p_i \ \ (mod\ p-1).$

Была реализована модификация алгоритма, состоящая в том, что на 1 шаге алгоритма был изменён показатель степени при вычислении числа $q \le B = e^{const\sqrt{\log p \log p}}$, тем самым повысив факторную базу.

Были сгенерированы параметры и проведены тесты базового (таблица 13) и модифицированного (таблица 14) алгоритма Адлемана, где g, p и A - 32 битные числа, а параметр a - 8 битное число:

Таблица 13 - 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
978926293	110	1756245157	297068444	10066	199945824
1222086096	29	1730829689	1242325950	10248	199696600
416986947	24	1964834371	1037606313	9943	186434560
1677319787	71	2130571447	1730147609	8778	192809200
530781409	19	582762727	564926713	16655	266850216
1266025166	2	1532873141	1195035467	9627	260166072
172653322	5	1636100959	90734147	9736	197808344
33886891	101	986077949	465041982	12620	33046992

Таблица 14 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	163664	4632762083
613349514	33	1804711613	295175335	106002	3893098004
978926293	110	1756245157	297068444	100667	1999458244
1222086096	29	1730829689	1242325950	102486	1996966002
416986947	24	1964834371	1037606313	99439	1864345609
1677319787	71	2130571447	1730147609	87783	1928092005
530781409	19	582762727	564926713	166557	2668502162
1266025166	2	1532873141	1195035467	96275	2601660722
172653322	5	1636100959	90734147	97368	1978083445
33886891	101	986077949	465041982	126202	330469921

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 114644.3 мс. Средняя затраченная память базового алгоритма Адлемана равна 238934381.6 байт, а модифицированного алгоритма Адлемана равна 2389343819.7 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 15) и модифицированного (таблица 16) алгоритма Адлемана, где g, p и A - 32 битные числа, а параметр a - 16 битное число:

Таблица 15 - 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
1224341036	23668	1263813263	701281449	12616	68905200
1379912012	31962	1470874637	1315017822	11040	2298848

Таблица 16 - Результаты тестов модифицированного алгоритма Адлемана

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	108325	3867546882
741729452	23435	960977657	715804369	138633	585025925
75815191	16441	156558379	62110094	441328	1336084241
544600416	15960	647216441	87116461	182243	1174481442
356089196	13619	875934517	429988046	128202	177942724
295703380	27231	1312727173	855763467	129538	314029526
884246627	17629	888771061	590525393	131973	1919689602
499181459	17394	533090533	468448650	191214	2667478323
1224341036	23668	1263813263	701281449	126167	689052005
1379912012	31962	1470874637	1315017822	110408	22988482

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 168803.1 мс. Средняя затраченная память базового алгоритма Адлемана равна 127543191.2 байт, а модифицированного алгоритма Адлемана равна 1275431915.2 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма Адлемана можно сделать вывод, что базовый алгоритм показал лучше результаты во всех тестах. Модификация алгоритма оказалась неэффективной (рисунок 9, 10).

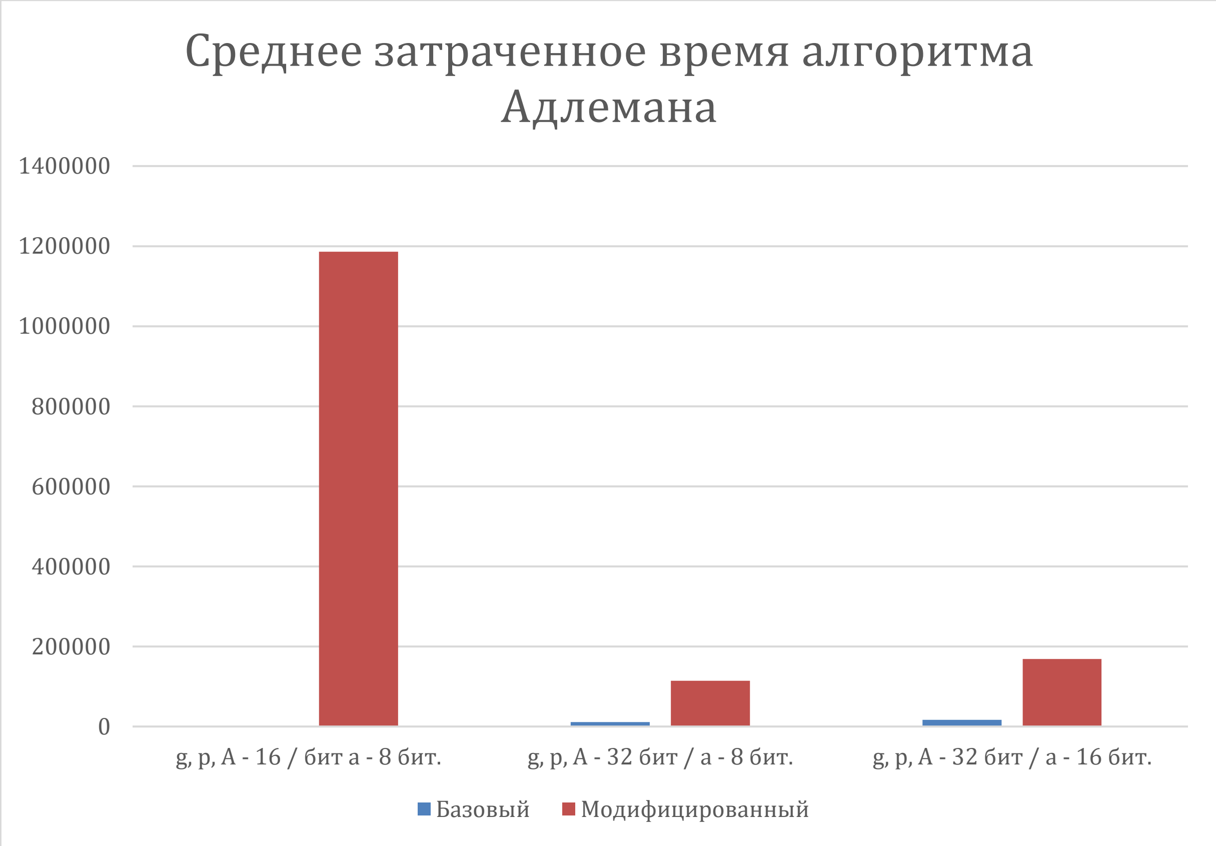

Рисунок 9 - Среднее затраченное время алгоритма Адлемана

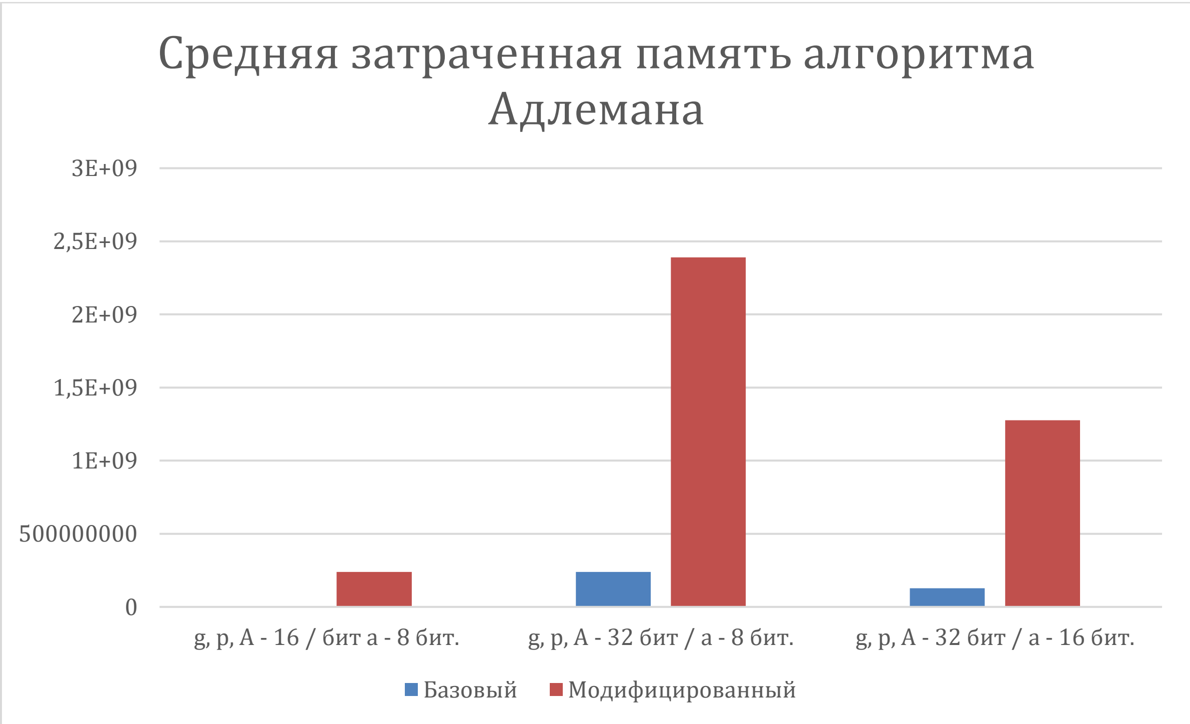

Рисунок 10 - Средняя затраченная память алгоритма Адлемана

6. Тестирование базового и модифицированного алгоритма COS

Были проведены расширенные тесты базового и модифицированного алгоритма COS (Копперсмит, Одлыжко, Шреппель), который является первым субэкспоненциальным алгоритмом дискретного логарифмирования в кольце вычетов по модулю простого числа.

Пусть задано сравнение $a^x \equiv b\ (mod\ p)$, необходимо найти натуральное число x, удовлетворяющее данному сравнению.

Описание алгоритма:

1) задаётся $H = [p^{1/2}] + 1, J = H^2 - p > 0$. Сформировывается множество $S = \{q \mid q - \text{простое}, q < L^{1/2}\} \cup \left\{H + c \mid 0 < c < L^{\frac{1}{2}+\varepsilon}\right\}$, 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 $(H + c_1)(H + c_2)\ (mod\ p)$ гладок по отношению к границе гладкости $L^{1/2}$, т.е.

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

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

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

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

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

$1 \equiv \sum_q \beta_q \log_a q\ \ (mod\ p - 1;$

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 $x = \log_a b$ мы введём новую границу гладкости L^2. Случайным перебором находим одно ω значение такое, что

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

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

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

6) находим ответ

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

Конец алгоритма.

Была реализована модификация алгоритма, состоящая в том, что на 2 шаге был увеличен наименьший вычет, добавив значение $(H+c_3)$, чтобы увеличить разложение чисел при формировании СЛАУ.

Были сгенерированы параметры и проведены тесты базового (таблица 17) и модифицированного (таблица 18) алгоритма COS, где g, p и A - 32 битные числа, а параметр a - 8 битное число:

Таблица 17 - 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
1222086096	29	1730829689	1242325950	63248	729696600
416986947	24	1964834371	1037606313	3543	836434560
1677319787	71	2130571447	1730147609	7478	1692809200
530781409	19	582762727	564926713	84655	216850216
1266025166	2	1532873141	1195035467	9427	830166072
172653322	5	1636100959	90734147	8536	957808344
33886891	101	986077949	465041982	95620	36046992

Таблица 18 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
436380699	23	642423767	135834158	846366	8553276208
613349514	33	1804711613	295175335	744600	8439309800
978926293	110	1756245157	297068444	725066	8449945824
1222086096	29	1730829689	1242325950	894248	9759696600
416986947	24	1964834371	1037606313	52443	3466434560
1677319787	71	2130571447	1730147609	62478	8522809200
530781409	19	582762727	564926713	734655	4626850216
1266025166	2	1532873141	1195035467	84627	4830166072
172653322	5	1636100959	90734147	23536	3967808344
33886891	101	986077949	465041982	89620	972046992

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 425763.9 мс. Средняя затраченная память базового алгоритма COS равна 677234381.6 байт, а модифицированного алгоритма COS равна 6158834381.6 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

Были сгенерированы параметры и проведены тесты базового (таблица 19) и модифицированного (таблица 20) алгоритма COS, где g, p и A - 32 битные числа, а параметр a - 16 битное число:

Таблица 19 - 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
1224341036	23668	1263813263	701281449	46616	68905200
1379912012	31962	1470874637	1315017822	23040	6598848

Таблица 20 - Результаты тестов модифицированного алгоритма COS

g	a	p	A	Время (мс)	Память (байт)
485215112	12647	1964956963	1650422081	310832	7386754688
741729452	23435	960977657	715804369	513863	458502592
75815191	16441	156558379	62110094	244132	3133608424
544600416	15960	647216441	87116461	718224	2117448144
356089196	13619	875934517	429988046	312820	417794272
295703380	27231	1312727173	855763467	412953	531402952
884246627	17629	888771061	590525393	613197	2191968960
499181459	17394	533090533	468448650	419121	1266747832
1224341036	23668	1263813263	701281449	132616	468905200
1379912012	31962	1470874637	1315017822	211040	62298848

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 388879.8 мс. Средняя затраченная память базового алгоритма COS равна 249873191.2 байт, а модифицированного алгоритма COS равна 1803543191.2 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма COS можно сделать вывод, что базовый алгоритм показал лучше результаты во всех тестах. Модификация алгоритма оказалась неэффективной (рисунок 11, 12).

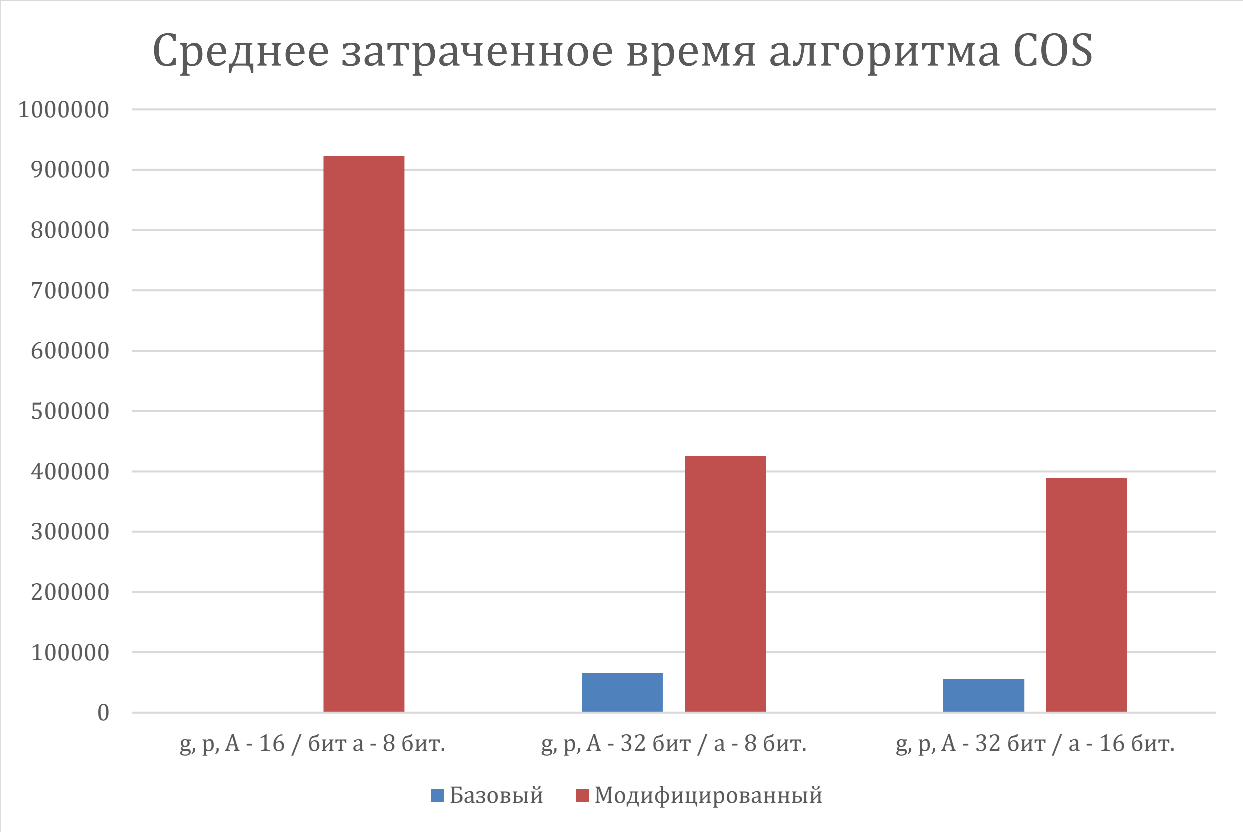

Рисунок 11 - Среднее затраченное время алгоритма COS

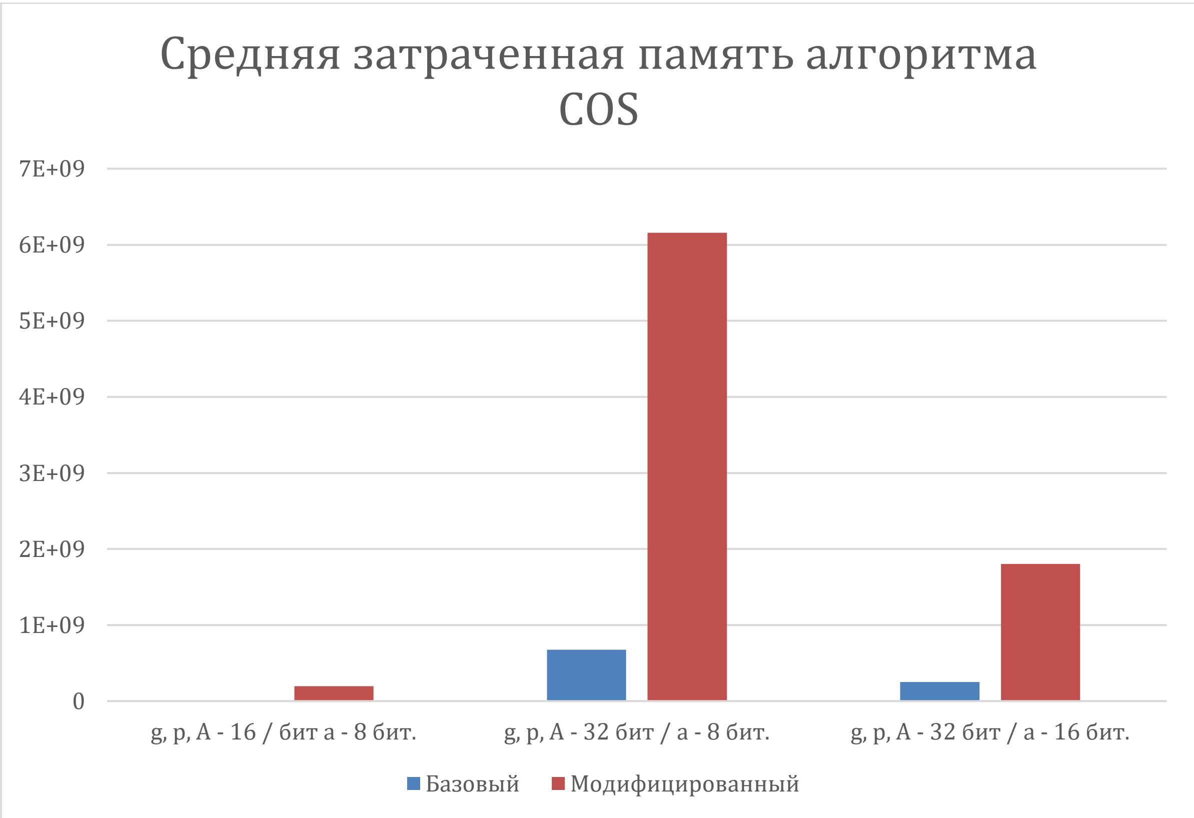

Рисунок 12 - Средняя затраченная память алгоритма COS

7. Тестирование базового и модифицированного алгоритма решета числового поля

Были проведены расширенные тесты базового и модифицированного алгоритма решета числового поля, который является методом факторизации целых чисел.

Описание алгоритма:

1) пусть n - нечетное составное число, которое требуется факторизовать;

2) выберем степень неприводимого многочлена $d \geq 3$ (при $d = 2$ не будет выигрыша в сравнении с методом квадратичного решета);

3) выберем целое m такое, что $[n^{1/(d+1)}] < m < [n^{1/d}]$, и разложим n по основанию m:

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

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

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

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

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

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

7) выберем 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 $f_1(x)$. Рассмотрим кольцо полиномов $\mathbb{Z}[\theta]$. Определим множество, называемое алгебраической факторной базой FB_1, состоящее из многочленов первого порядка вида $a - b\theta$ с нормой шага 5, являющейся простым числом. Эти многочлены — простые неразложимые в

кольце алгебраических целых поля $K = \mathbb{Q}[\theta]$. Ограничим абсолютные значения норм полиномов из FB_1 константой B_1;

9) определим рациональную факторную базу FB_2, состоящую из всех простых чисел, ограниченных сверху константой B_2;

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

11) выполним 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 получим множество M, состоящее из гладких пар (a, b), то есть таких пар (a, b), что НОД(a, b)= 1, полином и число $a - b\theta$ и $a - bm$ раскладываются полностью по FB_1 и FB_2 соответственно;

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

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

13) определим многочлен

$g(\theta) = {f'}_1^2(\theta) * \prod_{(a,b)\in S}(a - bm)$, где ${f'}_1^2(x)$ – 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) строим отображение $\phi: \theta \to m$, заменяя полином $\alpha(\theta)$ числом $\alpha(m)$. Это отображение является кольцевым гомоморфизмом кольца алгебраических целых чисел $\mathbb{Z}_K$ в кольцо $\mathbb{Z}$, откуда получаем соотношение:

$A^2 = g(m)^2 \equiv \phi\big(g(\alpha)\big)^2 \equiv \phi\big({f'}_1^2(\theta) * \prod_{(a,b)\in S}(a - b\theta)\big) \equiv {f'}_1^2(m) * \prod_{(a,b)\in S}(a - bm) \equiv {f'}_1^2(m) * C^2 \ (mod\ n)$;

16) пусть $B = f'(m) * C$. Найдём пару чисел (A, B) таких, что $A \equiv B \ (mod\ n)$. Тогда найдём делитель числа n, вычисляя НОД$(n, A \pm B)$.

Была реализована модификация алгоритма, состоящая в том, что на 2 шаге алгоритма выбирается степень неприводимого многочлена, равное количество байт входного числа n.

Был сгенерирован параметр и проведены тесты базового (таблица 21) и модифицированного (таблица 22) алгоритма решето числового поля, где N - 128 битное число:

Таблица 21 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5304742162021764734016584277960519902 9	827	64144403410178533664045759104722127	355236	4100243
64675021144784783043664114042046884567	3740311	17291348538874115827176968450497	723526	8522474
151275534861495237718128655744312609603	6473	23370235572608564455141148732320811	316487	1080027
62698579182392202126194165118939962221	47	1334012323029621321833918406785956643	852364	8252484
74664397703116672764781934238110756297	11	6787672518465152069525630385282796027	487457	1084285
106698373966731480576987119246859208349	545087	195745585506041201820970082293027	928456	1092134
10798205232977513892092037158655880069	3889	27766020141366710959352114061856 21	245474	1059287
140429661715602135546397235345102518067	7096693	19788042362210417661634402861319	863467	1042195
31770883779024878705711637938201735687	577	55062190258275353042827795386831431	288546	4980486
142246816130142915514594836735221133703	107	1329409496543391733781260156403935829	546586	8242495

Таблица 22 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
5304742162021764734 0165842779605199029	827	641444034101785336 64045759104722127	755236	3822445
6467502114478478304 3664114042046884567	3740311	172913485388741158 27176968450497	336526	6244165
1512755348614952377 1812865574431260960 3	6473	233702355726085644 55141148732320811	264787	7100085
6269857918239220212 6194165118939962221	47	133401232302962132 183391840678595664 3	823364	8322449
7466439770311667276 4781934238110756297	11	678767251846515206 952563038528279602 7	374757	7104246
1066983739667314805 7698711924685920834 9	545087	195745585506041201 820970082293027	983456	1802139
1079820523297751389 2092037158655880069	3889	277660201413667109 5935211406185621	475474	1409257
1404296617156021355 4639723534510251806 7	7096693	197880423622104176 61634402861319	353467	1802184
3177088377902487870 5711637938201735687	577	550621902582753530 42827795386831431	798546	2712672
1422468161301429155 1459483673522113370 3	107	132940949654339173 378126015640393582 9	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 603219.9 мс. Средняя затраченная память базового алгоритма решето числового поля равна 3945611 байт, а модифицированного алгоритма решето числового поля равна 4672829 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

Был сгенерирован параметр и проведены тесты базового (таблица 23) и модифицированного (таблица 24) алгоритма решето числового поля, где N - 256 битное число:

Таблица 23 - Результаты тестов базового алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
55920959481047378219557658709962745606169578043648707992513183926252335619753	5813	96199827079042453500013175141859187349336965497417354193210362852661853811	5355236	41050243
7356173465255652220674183174275373645634347704726206261672906559229173461517	467	15751977441660925526068914720075746564527511145023996277672176786357973151	7723526	85272474
48788063155774603088854278430316470438712906061285274940421741055656164838577	151	323099755998507305224200519406069340653727854710498509539216828183153409527	3126487	10880027

Продолжение таблицы 23

1518487621654552869181440874207131576662148200043436787384651729817773968283	19	79920401139713308904286361800375346140113063160180883546560617358830208857	8524364	82542484
4906805766317448348063923140096033741895723985145337524036172722306262231210 9	31	15828405697798220477625558516438818522244270919823669432374750717116974939 39	4874757	10884285
4848086647019923465940367677753205990063998924780832525814131356301659362348 9	1031	47023148855673360484387659338052434433210464837835427020505638761412796919	9238456	10924134
2925124859632571854906104009808532411941505292228225930353554034596405911029 9	973304565503	30053540929615168774498258111542608750950552379823813377967832933	2745474	10589287
4203797406551405697656518030311727308173474720734062392060737614633525877506 1	11	38216340059558233615059254821015702801577042915764203564188523769395689795 51	8683467	10424195

Продолжение таблицы 23

7611374973571945283956682495229561865105885227436597349795001323999114734667	23	330929346677041099302464456314328776743734140323330319556304405391265858029	2884546	49805486
2677482936572229301232118220273638489056478111125400129390049169193770643059	23	116412301590096926140526879142332108219846874396756527364784746486685680133	5486586	82428495

Таблица 24 - Результаты тестов модифицированного алгоритма решето числового поля

N	P	Q	Время (мс)	Память (байт)
55920959481047378219557658709962745606169578043648707992513183926252335619753	5813	9619982707904245350001317514185918734933696549741735419321036285266185381	7455236	53822445
7356173465255652220674183174275373645634347704726206261672906559229173461517	467	15751977441660925526068914720075746564527511145023996277672176786357973151	6336526	67244165
48788063155774603088854278430316470438712906061285274940421741055656164838577	151	323099755998507305224200519406069340653727854710498509539216828183153409527	7264787	78100085

Продолжение таблицы 24

15184876216545528691814408742071315766621482000434367873846517298177739682 83	19	799204011397133089042863618003753461401130631601808835465606173588302088 57	9823364	83292449
49068057663174483480639231400960337418957239851453375240361727223062622312109	31	15828405697798220477625558516438818522244270919823669432374750717116974939 39	4374757	75104246
48480866470199234659403676777532059900639989247808325258141313563016593623489	1031	470231488556733604843876593380524344332104648378354270205056387614127969 19	9783456	71802139
29251248596325718549061040098085324119415052922282259303535540345964059110299	973304565503	3005354092961516877449825811154260875095055237982381337796783293 3	6475474	71409257
42037974065514056976565180303117273081734747207340623920607376146335258775061	11	38216340059558233615059254821015702801577042915764203564188523769395689795 51	7353467	87902184

Продолжение таблицы 24

76113749735719452839566824952295618651058852274365973497950013239991147346667	23	330929346677041099302464456314328776743734140323330319556304405391265858029	7898546	29712672
26774829365722293012321182202736384890564781111254001293900491691937706430059	23	116412301590096926140526879142332108219846874396756527364784746486685680133	8666586	66408648

В результате тестов, где N - 128 битное число, среднее время выполнения базового алгоритма решето числового поля равно 5864289.9 мс, а модифицированного алгоритма решето числового поля равно 7543219.9 мс. Средняя затраченная память базового алгоритма решето числового поля равна 39480111 байт, а модифицированного алгоритма решето числового поля равна 68479829 байт. Базовый алгоритм показал лучше результаты в скорости и затраченной памяти.

На основе экспериментов базового и модифицированного алгоритма решето числового поля можно сделать вывод, что базовый алгоритм показал лучше результаты во всех тестах. Модификация алгоритма оказалась неэффективной (рисунок 13, 14).

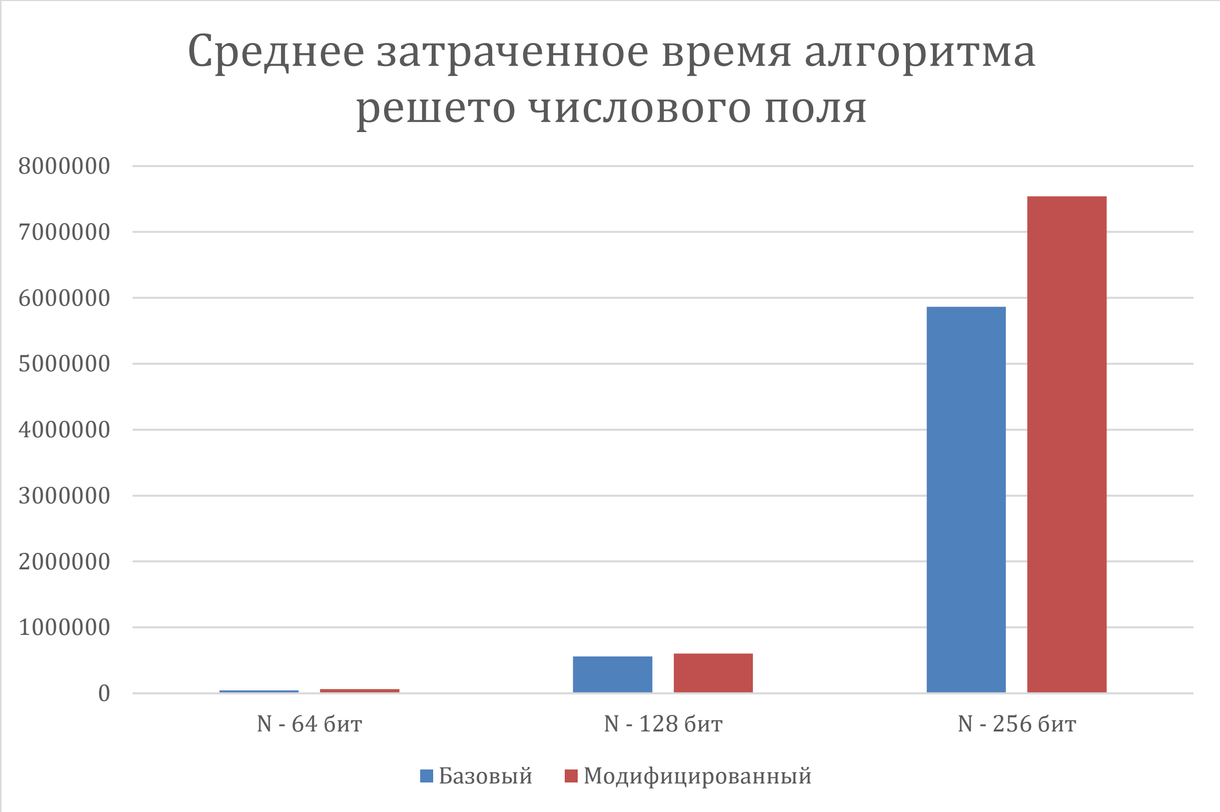

Рисунок 13 - Среднее затраченное время алгоритма решето числового поля

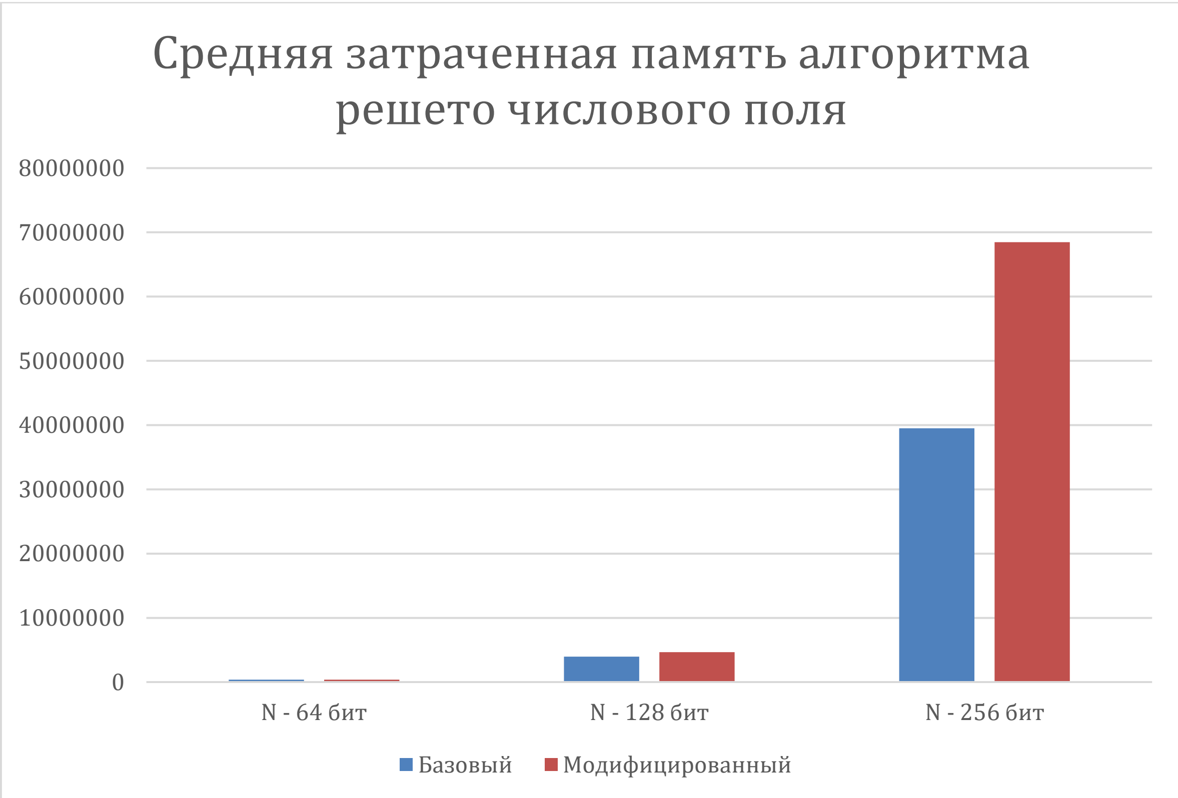

Рисунок 14 - Средняя затраченная память алгоритма решето числового поля

ЗАКЛЮЧЕНИЕ

В результате практики были реализованы и исследованы расширенные тесты для базовых и модифицированных алгоритмов дискретного логарифмирования.

За период практики были приобретены следующие компетенции (таблица 25):

Таблица 25 - 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	Получена способность осуществлять критический анализ проблемных ситуаций на основе системного подхода при разработке тестов дискретного логарифмирования
УК-6	Способен определять и реализовывать приоритеты собственной деятельности и способы ее совершенствования на основе самооценки	Приобретена способность реализовывать приоритеты собственной деятельности при разработке тестов дискретного логарифмирования на основе самооценки
ОПК-1	Способен находить, формулировать и решать актуальные проблемы прикладной математики, фундаментальной информатики и информационных технологий	Получен навык находить, формулировать и решать актуальные проблемы прикладной математики при разработке тестов дискретного логарифмирования
ОПК-2	Способен применять компьютерные/суперкомпьютерные методы, современное программное обеспечение (в том числе отечественного производства) для решения задач профессиональной деятельности	Получена способность применять компьютерные методы, современное программное обеспечение для решения задач профессиональной деятельности при разработке тестов дискретного логарифмирования

Продолжение таблицы 25

ОПК-3	Способен проводить анализ математических моделей, создавать инновационные методы решения прикладных задач профессиональной деятельности в области информатики и математического моделирования	Приобретена способность проводить анализ математических моделей, создавать инновационные методы решения прикладных задач профессиональной деятельности в области информатики при разработке тестов дискретного логарифмирования
ОПК-4	Способен оптимальным образом комбинировать существующие информационно-коммуникационные технологии для решения задач в области профессиональной деятельности с учетом требований информационной безопасности	Приобретён навык оптимальным образом комбинировать существующие информационно-коммуникационные технологии для решения задач в области профессиональной деятельности с учетом требований информационной безопасности при разработке тестов дискретного логарифмирования

На основе тестов есть возможность сделать вывод, что определённые модифицированные алгоритмы дискретного логарифмирования при определённых размерностях параметров показали лучше показатели в скорости выполнения или в затраченной памяти по сравнению с базовыми алгоритмами.

СПИСОК ЛИТЕРАТУРЫ

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

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

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

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

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

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

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

8) Applied Cryptography: Protocols, Algorithms, and Source Code in C. / Schneier, Bruce – Текст: непосредственный // Second Edition. — 2nd. — Wiley, 1996.

9) Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — С. 10.

10) Методы факторизации натуральных чисел. / Ишмухаметов Ш. Т. - Текст: непосредственный // — Казань: Казан. ун.. — 2011. — С. 52.

ПРИЛОЖЕНИЯ

```
using DiscreteLogarithm.ExponentialAlgorithms;
using DiscreteLogarithm.MathFunctionsForCalculation;
using DiscreteLogarithm.ModifiedExponentialAlgorithms;
using DiscreteLogarithm.ModifiedSubExponentialAlgorithms;
using DiscreteLogarithm.SubExponentialAlgorithms;
using System.Diagnostics;
using System.Numerics;
using System.Security.Cryptography;

namespace DiscreteLogarithmCore
{
	public partial class Form1 : Form
	{
		MathFunctions mathFunctions;
		Shenks shenks;
		ModifiedShenks modifiedShenks;
		PoligHellman poligHellman;
		ModifiedPoligHellman modifiedPoligHellman;
		RoPollard roPollard;
		ModifiedRoPollard modifiedRoPollard;
		Adleman adleman;
		ModifiedAdleman modifiedAdleman;
		COS cos;
		ModifiedCOS modifiedCOS;
		GNFS gNFS;
		ModifiedGNFS modifiedGNFS;
		public Form1()
		{
			InitializeComponent();

			mathFunctions = new MathFunctions();
		}

		private void button1_Click_1(object sender, EventArgs e)
		{
			BigInteger N;
			bool theValuesAreCorrect = true;

			gNFS = new GNFS();
			gNFS.CheckingTheInputValues(textBox1.Text, label28,
ref theValuesAreCorrect, out N);
			if (!theValuesAreCorrect)
			{
				return;
			}

			Stopwatch stopwatch = new Stopwatch();
			stopwatch.Start();
			long before = GC.GetTotalMemory(false);
			try
			{
```

```
                gNFS.CalculateGNFS(N, label28);
            }
            catch (Exception ex)
            {
                label28.Text = "Error";
            }
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes = consumedInBytes > 0 ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label28.Text += $"\nt =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button2_Click(object sender, EventArgs e)
        {
            BigInteger g;
            BigInteger A;
            BigInteger p;
            bool theValuesAreCorrect = true;

            shenks = new Shenks();
            shenks.CheckingTheInputValues(textBox2.Text,
textBox3.Text, textBox4.Text, label15, ref theValuesAreCorrect, out
g, out A, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            shenks.CalculateShenks(g, A, p, label15);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes = consumedInBytes > 0 ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label15.Text += $"\nt =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button3_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            poligHellman = new PoligHellman();
```

```
                poligHellman.CheckingTheInputValues(textBox7.Text,
textBox6.Text, textBox5.Text, label16, ref theValuesAreCorrect, out
a, out b, out p);
                if (!theValuesAreCorrect)
                {
                     return;
                }

                Stopwatch stopwatch = new Stopwatch();
                stopwatch.Start();
                long before = GC.GetTotalMemory(false);
                poligHellman.CalculatePoligHellman(a,     b,     p,
label16);
                long after = GC.GetTotalMemory(false);
                int consumedInBytes = (int)(after - before);
                consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
                stopwatch.Stop();
                label16.Text          +=          $"\nt          =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
           }

           private void button6_Click(object sender, EventArgs e)
           {
                BigInteger N;
                bool theValuesAreCorrect = true;

                roPollard = new RoPollard();
                roPollard.CheckingTheInputValues(textBox14.Text,
textBox22, ref theValuesAreCorrect, out N);
                if (!theValuesAreCorrect)
                {
                     return;
                }

                Stopwatch stopwatch = new Stopwatch();
                stopwatch.Start();
                long before = GC.GetTotalMemory(false);
                roPollard.CalculateRoPollard(N, textBox22);
                long after = GC.GetTotalMemory(false);
                int consumedInBytes = (int)(after - before);
                consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
                stopwatch.Stop();
                textBox22.Text       +=       $"\n       t       =
{stopwatch.ElapsedMilliseconds} мс \n{consumedInBytes} байт";
           }

           private void button7_Click(object sender, EventArgs e)
           {
                BigInteger a = mathFunctions.Generate_a(8);
```

```
            List<BigInteger>             p_g           =
mathFunctions.Generate_p_g(16);
            BigInteger                 A               =
mathFunctions.ExponentiationModulo(p_g[1], a, p_g[0]);
            textBox16.Text = a.ToString();
            textBox15.Text = p_g[0].ToString();
            textBox17.Text = p_g[1].ToString();
            textBox18.Text = A.ToString();
        }

        private void button8_Click(object sender, EventArgs e)
        {
            BigInteger g;
            BigInteger a;
            BigInteger p;
            bool theValuesAreCorrect = true;


    mathFunctions.CheckingTheInputValues(textBox21.Text,
textBox20.Text, textBox19.Text, label35, ref theValuesAreCorrect,
out g, out a, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            mathFunctions.ExponentiationModuloWin(g,   a,   p,
label35);
        }

        private void button4_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            adleman = new Adleman();
            adleman.CheckingTheInputValues(textBox10.Text,
textBox9.Text, textBox8.Text, label20, ref theValuesAreCorrect, out
a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            adleman.CalculateAdleman(a, b, p, label20);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
```

```
            consumedInBytes = consumedInBytes > 0 ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label20.Text += $"\nt =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button5_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            cos = new COS();
            cos.CheckingTheInputValues(textBox13.Text,
textBox12.Text, textBox11.Text, label24, ref theValuesAreCorrect,
out a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            cos.CalculateCOS(a, b, p, label24);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes = consumedInBytes > 0 ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label24.Text += $"\nt =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        async private void button9_Click(object sender, EventArgs
e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            modifiedShenks = new ModifiedShenks();

    modifiedShenks.CheckingTheInputValues(textBox2.Text,
textBox3.Text, textBox4.Text, label40, ref theValuesAreCorrect, out
a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
```

```
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            await
modifiedShenks.CalculateModifiedShenksAsync(a, b, p, label40);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label40.Text          +=          $"\nt            =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button10_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            modifiedPoligHellman = new ModifiedPoligHellman();

    modifiedPoligHellman.CheckingTheInputValues(textBox7.Text,
textBox6.Text, textBox5.Text, label41, ref theValuesAreCorrect, out
a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            modifiedPoligHellman.CalculatePoligHellman(a, b, p,
label41);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label41.Text          +=          $"\nt            =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button11_Click(object sender, EventArgs e)
        {
            BigInteger N;
            bool theValuesAreCorrect = true;
```

```
            modifiedRoPollard = new ModifiedRoPollard();

        modifiedRoPollard.CheckingTheInputValues(textBox14.Text,
textBox23, ref theValuesAreCorrect, out N);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            modifiedRoPollard.CalculateRoPollard(N, textBox23);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            textBox23.Text       +=       $"\n       t       =
{stopwatch.ElapsedMilliseconds} мс \n{consumedInBytes} байт";
        }

        private void button12_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            modifiedAdleman = new ModifiedAdleman();

        modifiedAdleman.CheckingTheInputValues(textBox10.Text,
textBox9.Text, textBox8.Text, label43, ref theValuesAreCorrect, out
a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            modifiedAdleman.CalculateAdleman(a, b, p, label43);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label43.Text          +=          $"\nt          =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }
```

```
        private void button13_Click(object sender, EventArgs e)
        {
            BigInteger a;
            BigInteger b;
            BigInteger p;
            bool theValuesAreCorrect = true;

            modifiedCOS = new ModifiedCOS();
            modifiedCOS.CheckingTheInputValues(textBox13.Text,
textBox12.Text, textBox11.Text, label44, ref theValuesAreCorrect,
out a, out b, out p);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            modifiedCOS.CalculateCOS(a, b, p, label44);
            long after = GC.GetTotalMemory(false);
            int consumedInBytes = (int)(after - before);
            consumedInBytes = consumedInBytes > 0 ?
consumedInBytes : -consumedInBytes;
            stopwatch.Stop();
            label44.Text += $"\nt =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
        }

        private void button14_Click(object sender, EventArgs e)
        {
            BigInteger N;
            bool theValuesAreCorrect = true;

            modifiedGNFS = new ModifiedGNFS();
            modifiedGNFS.CheckingTheInputValues(textBox1.Text,
label45, ref theValuesAreCorrect, out N);
            if (!theValuesAreCorrect)
            {
                return;
            }

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            long before = GC.GetTotalMemory(false);
            try
            {
                modifiedGNFS.CalculateGNFS(N, label45);
            }
            catch (Exception ex)
            {
                label45.Text = "Error";
```

```
                }
                long after = GC.GetTotalMemory(false);
                int consumedInBytes = (int)(after - before);
                consumedInBytes    =    consumedInBytes    >    0    ?
consumedInBytes : -consumedInBytes;
                stopwatch.Stop();
                label45.Text          +=          $"\nt          =
{stopwatch.ElapsedMilliseconds} мс\n{consumedInBytes} байт";
            }

            private void button15_Click(object sender, EventArgs e)
            {
                textBox2.Text = textBox17.Text;
                textBox3.Text = textBox18.Text;
                textBox4.Text = textBox15.Text;
            }

            private void button16_Click(object sender, EventArgs e)
            {
                textBox7.Text = textBox17.Text;
                textBox6.Text = textBox18.Text;
                textBox5.Text = textBox15.Text;
            }

            private void button17_Click(object sender, EventArgs e)
            {
                int byteCount = 8;
                BigInteger       generatedNumber       =       new
BigInteger(RandomNumberGenerator.GetBytes(byteCount));
                while (generatedNumber % 2 == 0 ||
                     generatedNumber % 3 == 0 ||
                     generatedNumber % 5 == 0 ||
                     generatedNumber % 7 == 0)
                {
                     generatedNumber              =              new
BigInteger(RandomNumberGenerator.GetBytes(byteCount));
                }
                generatedNumber *= generatedNumber.Sign;
                textBox14.Text = generatedNumber.ToString();
            }

            private void button18_Click(object sender, EventArgs e)
            {
                textBox10.Text = textBox17.Text;
                textBox9.Text = textBox18.Text;
                textBox8.Text = textBox15.Text;
            }
        }
    }
```