МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Факторизация целых чисел

ОТЧЁТ
ПО ДИСЦИПЛИНЕ
«ТЕОРЕТИКО-ЧИСЛОВЫЕ МЕТОДЫ В КРИПТОГРАФИИ»

студента 5 курса 531 группы
специальности 10.05.01 Компьютерная безопасность
факультета компьютерных наук и информационных технологий
Сенокосова Владислава Владимировича

Преподаватель, профессор _____ В.А. Молчанов
подпись, дата

Саратов 2024

Содержание

1 Цель работы и порядок выполнения

Цель работы — изучение основных методов факторизации целых чисел и их программная реализация.

Порядок выполнения работы

1. Рассмотреть ρ-метод Полларда разложения целых чисел на множители и привести его программную реализацию.

2. Рассмотреть (p-1)-метод Полларда разложения целых чисел на множители и привести его программную реализацию.

3. Рассмотреть метод цепных дробей разложения целых чисел на множители и привести его программную реализацию.

2 ρ-метод Полларда разложения целых чисел на множители

Пусть n – нечетное составное число, $S = \{0, 1, \ldots, n - 1\}$ и $f: S \to S$ – случайное отображение, обладающее сжимающими свойствами, например $f(x) = x^2 + 1 \, (mod\, n)$. Основная идея метода состоит в следующем. Выбираем случайный элемент $x_0 \in S$ и строим последовательность $x_0, x_1, x_2, \ldots$, определяемую рекуррентными соотношением

$$x_{i+1} = f(x_i)$$

Где $i \geq 0$, до тех пор, пока не найдем такие числа i, j, что $i < j$ и $x_i = x_j$. Поскольку множество S конечно, такие индексы i, j существуют (последовательность «зацикливается»). Последовательность $\{x_i\}$ будет состоять из «хвоста» $x_0, x_1, \ldots, x_{i-1}$ длины $O(\sqrt{\frac{\pi n}{8}})$ и цикла $x_i = x_j, x_{i+1}, \ldots, x_{j-1}$ той же длины.

Если p – простой делитель числа n и $x_i = x_j \, (mod\, p)$, то разность $x_i - x_j$ делится на p и НОД$(x_i - x_j, n) > 1$. Число p нам нужно найти, поэтому все вычисления в алгоритме будем проводить по модулю n и на каждом шаге вычислять $d =$ НОД$(x_i - x_j, n)$. Нетривиальный наибольший общий делитель $1 < d < n$ (когда числа x_i и x_j принадлежит одному классу вычетов по модулю p, но разным классов вычетов по модулю n) как раз и будет искомым делителем p числа n. Случай $d = n$ имеет место с пренебрежимо малой вероятностью.

Если последовательно вычислять значения $x_{i+1} = f(x_i)$ запоминать их и искать равные, то число шагов алгоритма в среднем равно $\sqrt{\frac{\pi n}{2}}$. Кроме того, временная и емкостная сложность равны $O(\sqrt{n})$. Но можно этого избежать если последовательно вычислять значения (x_i, x_{2i}) из (x_{i-1}, x_{2i-2}), пока не найдется такое число k, что $x_m = x_{2m}$.

Алгоритм ρ-метода Полларда:

Вход: Число n, начальное значение c, функция f, обладающая сжимающими свойствами.

Выход: Вероятно нетривиальный делитель p числа n.

1. Положить $a = c, b = c$.
2. Вычислить $a = f(a)(mod\ n), b = f(b)(mod\ n), b = f(b)(mod\ n)$.
3. Найти $d = \text{НОД}(a - b, n)$.
4. Если $1 < d < n$, то положить $p = d$ и результат: p. При $d = n$ результат: «Делитель не найден»; при $d = 1$ вернуться на шаг 2.

В предположении, что функция f ведет себя как случайная, алгоритм требует для нахождения делителя p числа n порядка $O(\sqrt{p})$ модульных умножений. Следовательно, для нахождения нетривиального делителя числа n требуется $O(\sqrt[4]{n})$ модульных умножений. С учетом использования алгоритма Евклида это добавляет еще n^2.

Сложность алгоритма составляет: $O(n^2\sqrt[4]{n})$.

Стоит отметить, что алгоритм ρ-метод Полларда может зациклиться и не иметь оценки вероятности успеха. Для оценки трудоемкости алгоритма сначала рассмотрим более простой для анализа, но менее эффективный его вариант.

ДАНО: составное число $N, 0 < \varepsilon < 1$.

ВЫХОД: вероятно нетривиальный делитель d числа $N, 1 < d < N$, с вероятностью $1 - \varepsilon$

1. Вычислить $T = \left[\sqrt{2\sqrt{n}\, ln\frac{1}{\varepsilon}}\right] + 1$ и выбрать случайный многочлен $f \in Z_n$.
2. Случайно выбрать $x_0 \in Z_n$ и, последовательно вычисляя значения $x_{i+1} = f(x_i)\ mod\ n, 0 \le i \le T$, проверять тест на шаге 3.
3. Для каждого $0 \le k \le i$ вычислить $d_k = \text{НОД}(x - x_k, n)$ и проверить условие $1 < d_k < n$. Если это выполняется, то найден нетривиальный делитель d_k числа n. Если же $d_k = 1$ для всех $0 \le k \le$

i , то перейти к выбору следующего значения последовательности на шаге 2. Если найдется $d_k = n$ для некоторого $0 \le k \le i$, то перейти к выбору нового значения $x_0 \in Z_n$ на шаге 2.

4. Если вычислено T членов последовательности $\{x_i\}$, а делитель числа не найден, то остановить алгоритм.

Замечание: Условие $d_k = N$ означает, что $x_i = x_k$. Следовательно, $x_k, \ldots, x_{i-1}$ — период последовательности $\{x_i\}$. В этом случае дальнейшие вычисления членов последовательности $\{x_i\}$ бесполезны, поскольку все возможные $(x_i - x_k, N)$ уже вычислены. В этом случае действительно требуется выбрать новое значение $x_0 \in Z_n$ новый многочлен $f(x)$.

Для более точных выводов требуется использовать теорему, известную под названием «парадокс дней рождения».

Теорема (Парадокс дней рождения): Пусть $\lambda > 0, S$ — конечное множество из n элементов, $k = [\sqrt{2\lambda n}]$. Для случайной равновероятной выборки объема $k + 1$ из множества S вероятность $P_{n,k}$ того, что все элементы в выборке попарно различны, оценивается неравенством $P_{n,k} \le e^{-\lambda}$.

Положим в теореме $\lambda = \ln\left(\frac{1}{\varepsilon}\right), S = Z_p, R = \left[\sqrt{2pln\left(\frac{1}{\varepsilon}\right)}\right]$,

где $p \le \sqrt{N}$ − простой делитель числа N. Тогда по теореме среди членов последовательности $\{x_i \ (mod\ p)\}, 0 \le i \le R$ с вероятностью не менее

$$1 - P_{p,R} \ge 1 - e^{-\lambda} = 1 - \varepsilon$$

найдутся совпадающие члены, т. е.

$$x_i = x_k \ (mod\ p), 0 \le k < i \le R.$$

Оценим вероятность одновременного выполнения сравнения $x_i = x_k \ (mod\ N)$. Для этого потребуется оценить снизу вероятность $P_{n,k}$ из теоремы.

Лемма: Пусть выполнены условия теоремы и верно неравенство $n > \frac{8\lambda}{\ln^2 2}$. Тогда $P_{n,k} > e^{-4\lambda}$.

На основании этой леммы положим $\lambda' = \frac{\lambda p}{N}, \lambda = \ln\left(\frac{1}{\varepsilon}\right), S = Z_N, R = \left[\sqrt{2\lambda' N}\right] = \left[\sqrt{2pln\left(\frac{1}{\varepsilon}\right)}\right]$, где $p \leq \sqrt{N}$ - - простой делитель числа N. Для достаточно больших N условие $N > \frac{8\lambda}{\ln^2 2}$ заведомо выполнено. Тогда по лемме среди членов последовательности $\{x_i(mod\ N)\}, 0 \leq i < R$ с вероятностью не более

$$1 - P_{N,R} \leq 1 - e^{-4\lambda} = 1 - e^{-\frac{4\lambda p}{N}} \leq 1 - e^{-\frac{4\lambda p}{\sqrt{N}}}.$$

найдутся совпадающие члены. Поскольку $1 - e^{-\frac{4\lambda p}{\sqrt{N}}} \to 0$ при $N \to \infty$, то при достаточно больших N в выборке $\{x_i(mod\ N)\}$ объема $\left[\sqrt{2pln\left(\frac{1}{\varepsilon}\right)}\right] + 1$ с вероятностью близкой к единице все члены будут различны.

Итак, с вероятностью не менее $1 - \varepsilon$ по

$$R + 1 = \left[\sqrt{2pln\left(\frac{1}{\varepsilon}\right)}\right] + 1$$

членам последовательности $\{x_i\ (mod\ N)\}$ алгоритм найдет нетривиальный делитель числа N. Поскольку вычисление одного наибольшего общего делителя $(x_i - x_k, N)$ требует совершения $O(\log^2 N)$ арифметических операций, а всего в алгоритме потребуется вычислить $O(R^2)$ таких наибольших общих делителей, то трудоемкость алгоритма оценивается величиной $O(pln\left(\frac{1}{\varepsilon}\right)\log^2 N)$. Поскольку $p \leq \sqrt{N}$, то можно привести более грубую оценку: $O\left(\ln\left(\frac{1}{\varepsilon}\right)\sqrt{N}\log^2 N\right)$.

Так как рассмотренный алгоритм для каждого рассматриваемого значения i вычисляет только одно значение наибольшего общего делителя вида $(x_i - x_k, N)$, то трудоемкость алгоритма можно оценить как:

$$O(4Rlog^2 N) = O(\sqrt{pln\left(\frac{1}{\varepsilon}\right)}\log^2 N)$$

При этом нетривиальный делитель числа N будет найден с вероятностью не менее $1 - \varepsilon$.

Замечание: Учитывая полученную оценку трудоемкости метода Полларда, можно заметить, что этот метод может быть эффективно применен для поиска относительно небольших делителей составных чисел (сами факторизуемые числа при этом могут быть весьма большими)

Псевдокод реализованного алгоритма:

Входные данные: n — целое число, для которого нужно найти нетривиальный делитель, $0 < \varepsilon < 1$

k — количество итераций (находится по формуле $\left[\sqrt{2\sqrt{n}\, ln\frac{1}{\varepsilon}}\right]$, где $0 < \varepsilon < 1$)

Выходные данные: Список вероятно найденных делителей числа n, каждый из которых находится с вероятностью $1 - \varepsilon$

Начало алгоритма

$Pollard\_Rho(n, k = 10)$:

Если $n == 1$: вернуть 1

Инициализировать множество $divisors$ = пустое множество

Для i от 1 до k:

Случайным образом выбрать x в диапазоне [1, n - 1]

Установить $y = 1, i = 0, stage = 2$

Пока $gcd(n, |x - y|) == 1$:

Если $i == stage$:

$y = x$

$stage = stage * 2$

Вычислить $x = (x * x + 1)\, mod\, n$

Увеличить i на 1

Добавить $gcd(n, |x - y|)$ в $divisors$

Вернуть отсортированный список $divisors$

Конец алгоритма

Сложность алгоритма: $O(kn^2\sqrt[4]{n})$ где k – это количество раз запуска программы для поиска разложения

3 (p-1)-метод Полларда разложения целых чисел на множители

Определение: Число n называется сильно составным, если для любого числа $m < n$ выполняется неравенство:

$$d(m) < d(n)$$

где $d(n)$ — это функция количества делителей числа n.

Пусть n – нечетное составное число и p – его нетривиальный делитель. $(p-1)$ – Метод особенно эффективен при разложении таких чисел n, для которых число $p-1$ сильно составное.

Определение: Пусть $B = \{p_1, p_2, \dots., p_s\}$ – множество различных простых чисел. Назовем множество B базой разложения. Целое число назовем B-гладким, если все его простые делители являются элементами множества B.

Идея метода такова. Пусть $n = pq$ и пусть каноническое разложение числа $p-1$ имеет вид $p-1= p_1^{\alpha_1} p_2^{\alpha_2} \dots . p_s^{\alpha_s}$. Найдем максимальные показатели $l_1, l_2 \dots l_s$, для которых $p_i^{l_i} \leq n$. Прологарифмируем обе части этого неравенства: $l_i ln(p_i) \leq \ln(n)$, откуда $l_i \leq \left[\frac{\ln n}{\ln p_i}\right] = w_i$. Вычислим

$$M = p_1^{w_1} p_2^{w_2} \dots . p_s^{w_s}$$

Тогда $M = (p-1) * z$ для некоторого целого числа z.

Согласно малой теореме Ферма, выполняется сравнение $a^{p-1} = 1\ (mod\ p)$ для любого целого a, взаимно простого с p. Возводя обе части этого сравнения в степень z, получаем $a^M = 1\ (mod\ p)$.

Обозначим $d = \text{НОД}(a^M - 1, n)$. Если $a^M = 1\ (mod\ n)$, то число d должно делиться на p, поскольку разность $a^M - 1$ делится на p и число n делится на p.

Алгоритм (p-1)-метода Полларда:

Вход: Составное число n, число $B > 0$ (для построения базы)

Выход: Вероятно нетривиальный делитель p числа n.

1. Выбрать базу разложения $B = \{p_1, p_2, \dots . p_s\}$

2. Выбрать случайное целое a, $2 \leq a \leq n-2$, и вычислить $d = НОД(a, n)$. При $d \geq 2$ положить $p = d$ и результат: p

3. Для $i = 1, 2, \ldots, s$ выполнить следующие действия

   a. Вычислить $l = \left[\frac{\ln n}{\ln p_i}\right]$

   b. Положить $a = a^{p_i^l} \ (mod\ n)$

4. Вычислить $d = НОД(a-1, n)$

5. При $d = 1$ или $d = n$ результат: «Делитель не найден». В противном случае положить $p = d$ и результат: p.

Равенство $d = n$ на шаге 5 означает, что каноническое разложение числа $q - 1$ включает в себя те же простые числа, что и разложение числа $p - 1$. В силу малой теоремы Ферма $a^{p-1} = 1 \ (mod\ p)$ и $a^{q-1} = 1 \ (mod\ q)$, а поскольку в этом случае число M делится не только на $p - 1$, но и на $q - 1$, получаем $a^M = 1 \ (mod\ p)$ и $a^M = 1 \ (mod\ q)$, тогда $a^M = 1 \ (mod\ pq)$.

Если $p_i \leq P$ для $i = 1,2,3, \ldots, s$ и модульное умножение выполняется в столбик, то сложность алгоритма $O(PlogP\ (logn)^2)$. С учетом алгоритма Евклида можно записать сложность как: $O(n^2 PlogP\ (logn)^2)$.

Псевдокод реализованного алгоритма:

Входные данные:

n — целое число, которое нужно разложить на множители

B — ограничение для поиска показателей (База)

Выходные данные:

Список найденных вероятных делителей числа n

Начало алгоритма

$Pollard\_p(n, B = 1000)$:

Инициализировать множество $divisors$ = пустое множество

Для каждого из k:

Случайным образом выбрать a в диапазоне $[2, n - 2]$

Вычислить $d = gcd(a, n)$

Если $d > 1$:

Добавить d в $divisors$

Для i от 2 до B:

Вычислить $a = a^i \, mod \, n$

Вычислить $d = gcd(a - 1, n)$

Если $1 < d < n$:

Добавить d в $divisors$

Вернуть отсортированный список $divisors$

Конец алгоритма

Сложность реализованного алгоритма можно расценивать как: $O(kn^2 PlogP \, (logn)^2)$ где k – это количество раз запуска программы для поиска разложения.

4 Метод цепных дробей разложения целых чисел на множители

На данный момент под методом факторизации с помощью непрерывных дробей понимают метод Моррисона и Бриллхарта. Основная идея строится на введении процедуры алгоритмического построения сравнения $x^2 = y^2\ (mod\ m)$ по заданному множеству сравнений вида $u^2 = v\ (mod\ m)$. Для реализации этой процедуры использовались понятие факторной базы.

Будем искать x как произведение таких чисел u_i, что наименьший по абсолютной величине вычет u_i^2 по модулю m является произведением простых чисел. Тогда из тех же простых чисел можно построить и y.

Определим необходимые понятия.

Определение. Выражение такого вида

$$q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ldots + \cfrac{1}{q_{k-1} + \cfrac{1}{q_k}}}}$$

принято называть цепной (или непрерывной) дробью с неполными частными $q_1, q_2, \ldots, q_k$ и обозначать символом $(q_1; q_2, \ldots, q_k)$.

Таким образом, любая несократимая рациональная дробь $\frac{a_0}{a_1}$ может быть представлена в виде цепной дроби $\frac{a_0}{a_1} = (q_1; q_2, \ldots, q_k)$ с неполными частными $q_1; q_2, \ldots, q_k$, полученными в результате вычисления по алгоритму Евклида наибольшего общего делителя взаимно простых чисел a_0, a_1.

Определение. Для цепной дроби $\frac{a_0}{a_1} = (q_1; q_2, \ldots, q_k)$ выражения

$$\delta_1 = q_1, \delta_2 = q_1 + \frac{1}{q_2}, \delta_3 = q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3}}, \ldots, \delta_k = q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ldots + \cfrac{1}{q_{k-1} + \cfrac{1}{q_k}}}}$$

называются подходящими дробями конечной цепной дроби $(q_1; q_2, \ldots, q_k)$ и обозначаются символами $\delta_i = (q_1; q_2, \ldots, q_i)$ где $1 \le i \le k$.

Аналогично определяются подходящие дроби $\delta_i = (q_1; q_2, \ldots, q_i)$ для бесконечной цепной дроби $(q_1; q_2, \ldots, q_k, \ldots)$.

Каждая подходящая дробь $\delta_i = (q_1; q_2, \ldots, q_i)$ является несократимой рациональной дробью $\delta_i = \frac{P_i}{Q_i}$ с числителем P_i и знаменателем Q_i, которые вычисляются по следующим рекуррентным формулам:

$$P_i = q_i P_{i-1} + P_{i-2}, Q_i = q_i Q_{i-1} + Q_{i-2}$$

с начальными условиями $P_{-1} = 0, P_0 = 1, Q_{-1} = 1, Q_0 = 0$

Теорема 1: Пусть $n \in N, n > 16, \sqrt{n} \notin N$ и $\frac{P_i}{Q_i}$ и — подходящая дробь для представления числа $\sqrt{n}$ цепной дробью. Тогда абсолютно наименьший вычет $P_i^2 \pmod n$ равен значению $P_i^2 - nQ_i^2$ и выполняется $|P_i^2 - nQ_i^2| < 2\sqrt{n}$.

Для разложения числа $\sqrt{n}$ в цепную дробь с помощью целых чисел и нахождения целой части чисел вида $\frac{\sqrt{d}-u}{v}$ нужно воспользоваться теоремой 2.

Теорема 2: Пусть — квадратичная иррациональность вида $alpha = \frac{\sqrt{d}-u}{v}$ где $D \in N, \sqrt{D} \notin N, u \in N, v|D^2 - u$. Тогда для любого $k \geq 0$ справедливо разложение в бесконечную цепную дробь $alpha = [a_0, a_1, \ldots, a_k, a_{k+1}]$ где $a_0 \in Z, a_1, \ldots, a_k \in N, a_{k+1} - (k+1)$-й остаток. При этом справедливы соотношения $a_0 = [\alpha], v_0 = v, u_0 = u + a_0 v,$ и при $k \geq 0$ $a_{k+1} = [\alpha_{k+1}]$, где $v_{k+1} = \frac{D-u_k^2}{v_k} \in Z, v_{k+1} \neq 0, a_{k+1} = \frac{(\sqrt{D}+u_k)}{v_{k+1}} > 1$ и числа u_k получаются с помощью рекуррентной формулы $u_{k+1} = a_{k+1} v_{k+1} - u_k$.

Базой факторизации (или факторной базой) натурального числа m называется множество $B = \{p_0, p_1 \ldots., p_k\}$ различных простых чисел p_i, за возможным исключением p_0, которое может быть равным – 1. При этом число b, для которого $b^2 \bmod m$, является произведением степеней чисел из множества B, называется B-гладким числом. Пусть теперь $u_i - B$ – гладкие

числа, $v_i = u_i^2 \ mod \ m = \prod_{j=0}^{h} p_j^{\alpha_{ij}} -$ разложения их наименьших по абсолютной величине вычетов по модулю m. Положим

$$e_i = (e_{i0}, e_{i1}, .., e_{ih}) \in F_2^h$$

Где $e_{ij} = \alpha_{ij} \bmod 2$. F_2^h - векторное пространство над полем $GF(2)$, которое состоит из наборов нулей и единиц размерности h.

Подберем числа u_i так чтобы сумма векторов e_i была равна нулю. Определим

$$x = \left(\prod\nolimits_i u_i\right) mod \ m, \qquad y = \prod_{j=0}^{h} p_j^{\gamma_j}$$

Где $\gamma_j = \frac{1}{2}\sum_i \alpha_{ij}$ Тогда $x^2 = y^2 \ (mod \ m)$. Факторная база в алгоритме Моррисона и Бриллхарта состоит из тех простых чисел p_i, по модулю которых m является квадратичным вычетом.

Алгоритм Моррисона и Бриллхарта:

Вход: Составное число m (не является полным квадратом)

Выход: Нетривиальный делитель p числа m.

1. Построить базу разложения $B = \{p_0, p_1 \ldots., p_h\}$, где $p_0 = -1$ и $p_1, \ldots, p_h -$ попарно различные простые числа, по модулю которых m является квадратичным вычетом.
2. Берутся целые числа u_i являющиеся числителями подходящих дробей к обыкновенной непрерывной дроби, выражающей число $\sqrt{m}$. Из этих числителей, выбираются $h + 2$ чисел, для которых абсолютно наименьшие вычеты u_i^2 по модулю m являются $B -$ гладкими:

$$u_i^2 \ mod \ m = \prod_{j=0}^{h} p_j^{\alpha_{ij}} = v_i$$

Где $\alpha_{ij} \geq 0$. Также каждому числу u_i сопоставляется вектор показателей $(\alpha_{i0}, \alpha_{i1}, .., \alpha_{ih})$.

3. Найти (например, методом Гаусса) такое непустое множество $K \in \{1,2,\dots,h+1\}$, что путем применения к векторам e_i поэлементно операции исключающего ИЛИ мы должны получить нуль вектор

$$e_i = (e_{i0}, e_{i1}, .., e_{ih}), e_{ij} = \alpha_{ij} \ (mod\ 2), 0 \le j \le h.$$

4. Положить $x = \prod_{i \in K} u_i \ mod\ m,\ y = \prod_{j=1}^{h} p_j^{\gamma_j} mod\ m$, где $\gamma_j = \frac{1}{2}\sum_i \alpha_{ij}$. Тогда $x^2 = y^2\ mod\ m$
5. Если $x \neq y\ (mod\ m)$, то положить $p = \text{НОД}(x-y, m)$ и выдать результат: p

В противном случае вернуться на шаг 3 и поменять множество K Обычно есть несколько вариантов выбора множества K для одной и той же факторной базы B. Если все возможности исчерпаны, то следует увеличить размер факторной базы).

Сложность реализованного алгоритма основывается на следующем утверждении:

Если $a = \frac{1}{\sqrt{2}}, L = L(m) = \exp((\log m \log\log m)^{\frac{1}{2}})$ и факторная база состоит из $p = -1$ и всех простых чисел p таких, что:

$$2 \le p \le L^a, \left(\frac{m}{p}\right) = +1$$

То при вычислении L^b подходящих дробей, где $b = a + \frac{1}{4a}$, можно ожидать, что алгоритм разложит m на два множителя с эвристической оценкой $L_m\ [\frac{1}{2}; 2]$ арифметических операций.

Псевдокод реализованного алгоритма:

Входные данные:

n — целое число, которое нужно разложить на множители

len_base — длина базы факторов (по умолчанию 20)

Выходные данные:

Делитель числа n

$CFRAC(n, len\_base = 20)$:

Шаг 1: Разложение квадратного корня числа n в цепную дробь

$chain$ = Разложение квадратного корня n в цепную дробь с длиной len_base

Шаг 2: Найти подходящие дроби P и Q на основе цепной дроби

P, Q = Подходящие дроби, вычисленные на основе цепной дроби

Шаг 3: Генерация факторной базы для числа n

$base$ = Генерация факторной базы для числа n с длиной len_base

h = Длина факторной базы $base$

Инициализировать пустые списки:

$system$ = [] # Система уравнений для решения

P_k = [] # Подходящие числа P

$pows\_$ = [] # Показатели для факторизации чисел

Шаг 4: Построение системы уравнений на основе факторизации

Для каждого p из P:

Если длина $system$ не равна $h + 2$:

Найти разность квадратов и факторизовать её с помощью факторной базы

$min\_sub = abs(pow(p, 2, n), n)$

$factor, pows$ = Разложить min_sub с использованием факторной базы

Если $factor$ не равен $None$:

Добавить $factor$ в $system$

Добавить p в P_k # Сохранить число, соответствующее этому уравнению

Добавить $pows$ в $pows\_$ # Сохранить показатели простых чисел для этого числа

Иначе прервать цикл (если собрано достаточно уравнений)

Шаг 5: Найти комбинации векторов, которые дают нулевой вектор

$combinations$ = Найти комбинации векторов, которые дают нулевой вектор в системе $system$

Шаг 6: Проверить каждую комбинацию для нахождения делителя

Для каждой $comb$ из $combinations$:

$x = 1$ # Инициализация переменной для произведения подходящих P_k

Для каждого i из $comb$:

$x = x * P\_k[i]$ # Умножаем подходящие числа

$x = x \bmod n$ # Приводим x по модулю n

$y = 1$ # Инициализация переменной для произведения простых чисел из факторной базы

$i = 1$

Для каждого p из $base$ начиная с 1:

$sum\_$ = Найти сумму показателей для комбинации $comb$ и $pows\_$

$y = y * pow(base[i], sum\_)$ # Подсчитываем произведение простых чисел

$i \mathrel{+}= 1$

$y = y \bmod n$ # Приводим y по модулю n

Шаг 7: Если x и y не равны по модулю n, находим делитель

Если $x \bmod n$ не равно $y \bmod n$:

Вернуть $gcd(x - y, n)$ # Возвращаем НОД x и y, который является делителем

Шаг 8: Если делитель не найден, увеличиваем длину факторной базы и повторяем

Вернуть $CFRAC(n, len\_base + 10)$

Сложность реализованного алгоритма: $O(n^3)$

5 Тестирование алгоритмов факторизации

Программная реализация алгоритмов факторизации была выполнена на языке Python. Протестируем числа: *21299881, 1557697, 1728239.* Все данные о выводе будут представлены на соответствующих рисунках.

Тестирование ρ-метода Полларда

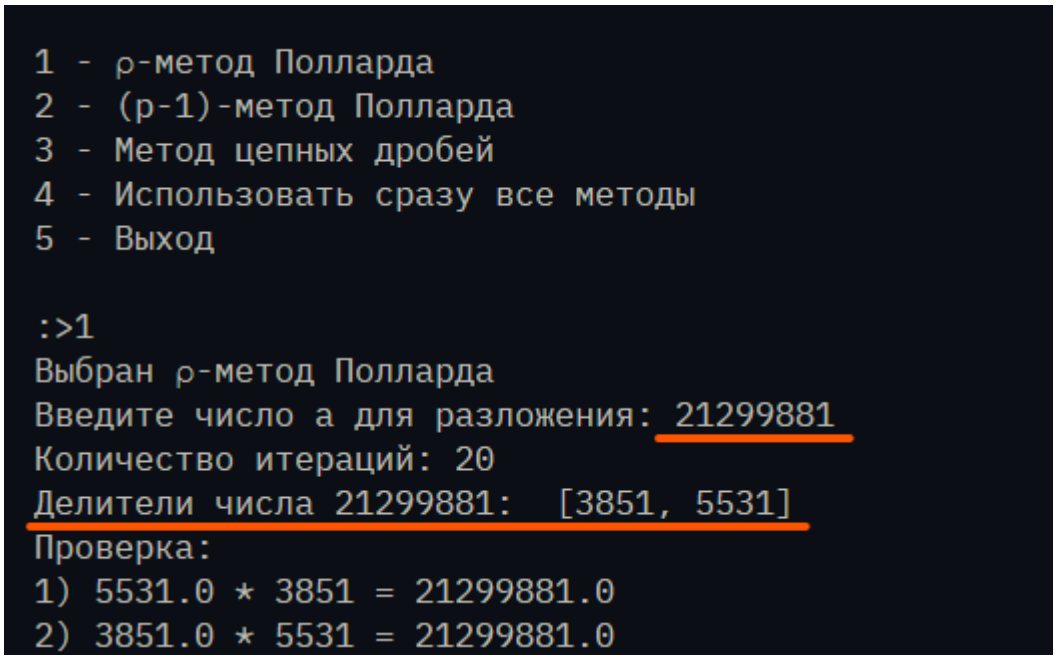

Рисунок 1 - Факторизация числа *21299881* ρ-методом Полларда

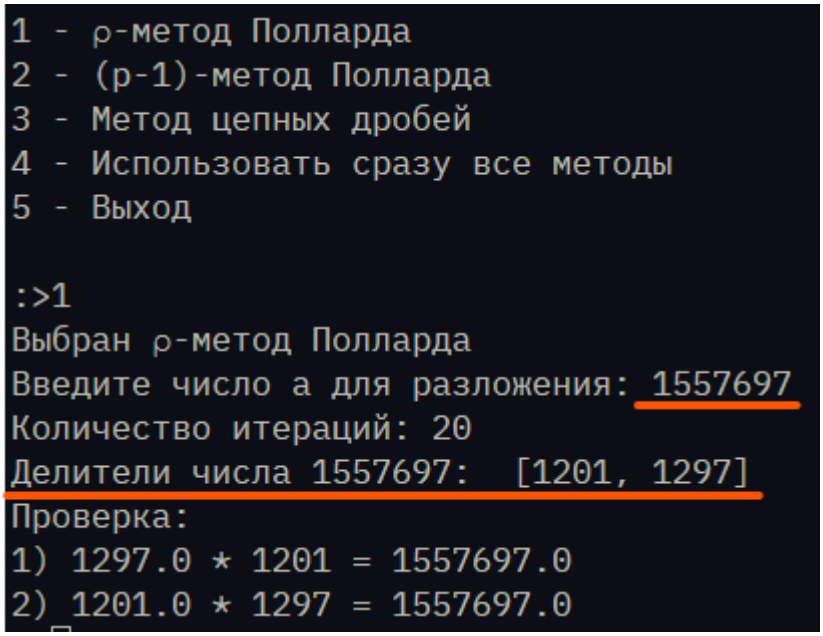

Рисунок 2 - Факторизация числа *1557697* ρ-методом Полларда

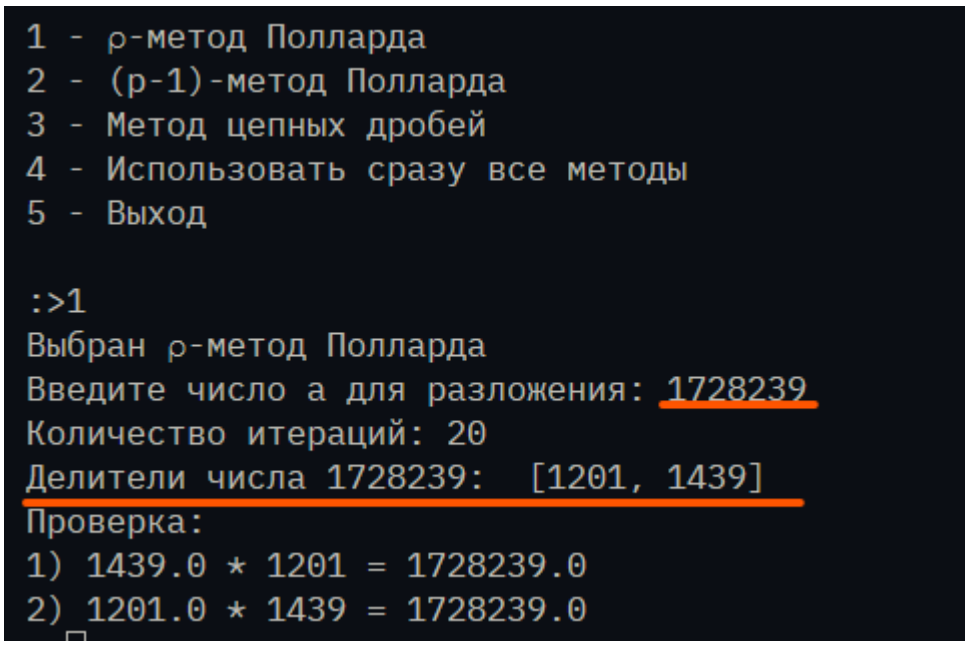

Рисунок 3 - Факторизация числа *1728239* ρ-методом Полларда

Тестирование (p-1)-метода Полларда

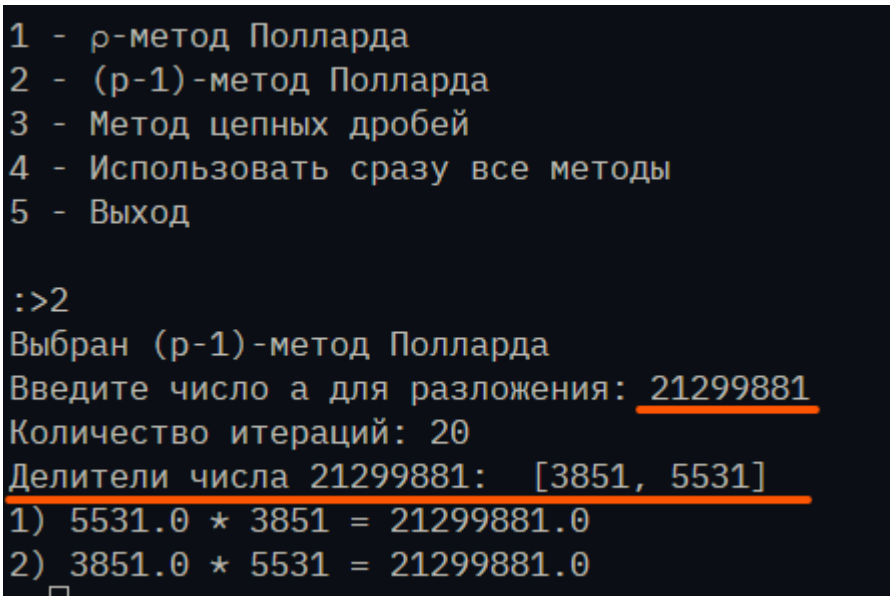

Рисунок 4 - Факторизация числа *21299881* (p-1)-методом Полларда

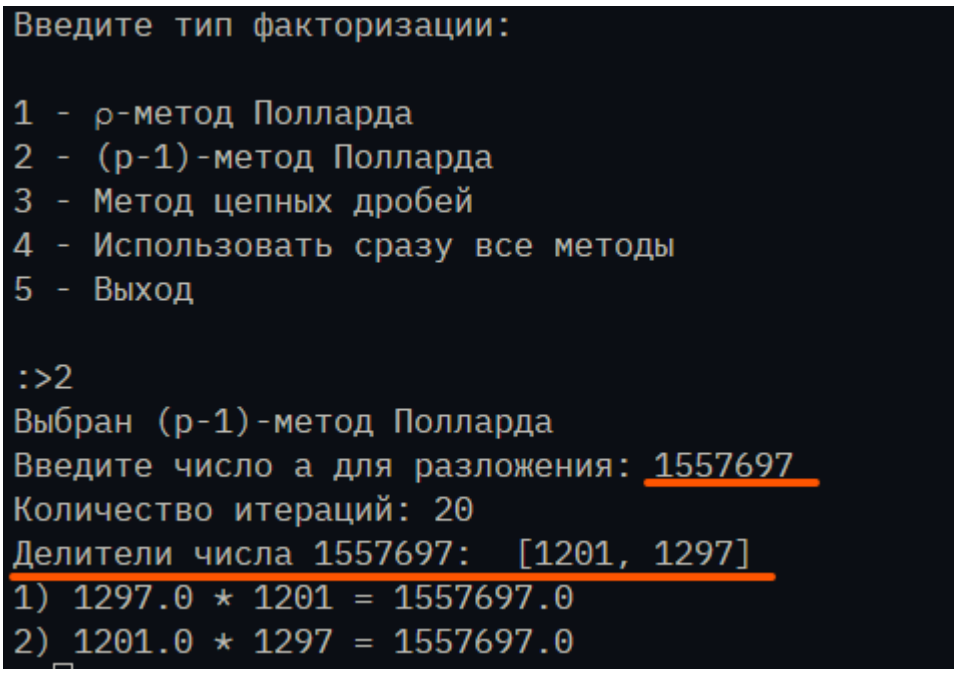

Рисунок 5 - Факторизация числа *1557697* (p-1)-методом Полларда

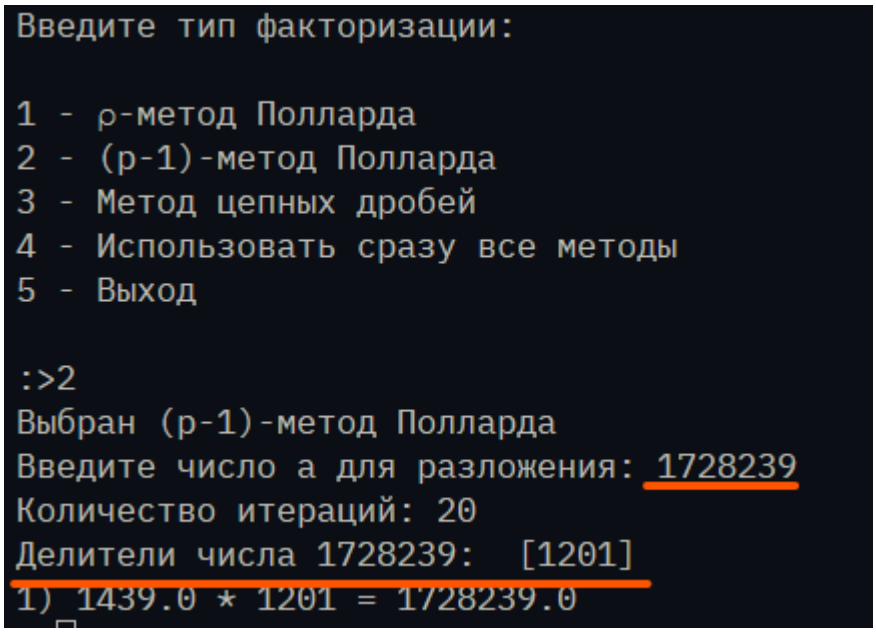

Рисунок 6 - Факторизация числа *1728239* (p-1)-методом Полларда

Тестирование метода Моррисона и Бриллхарта

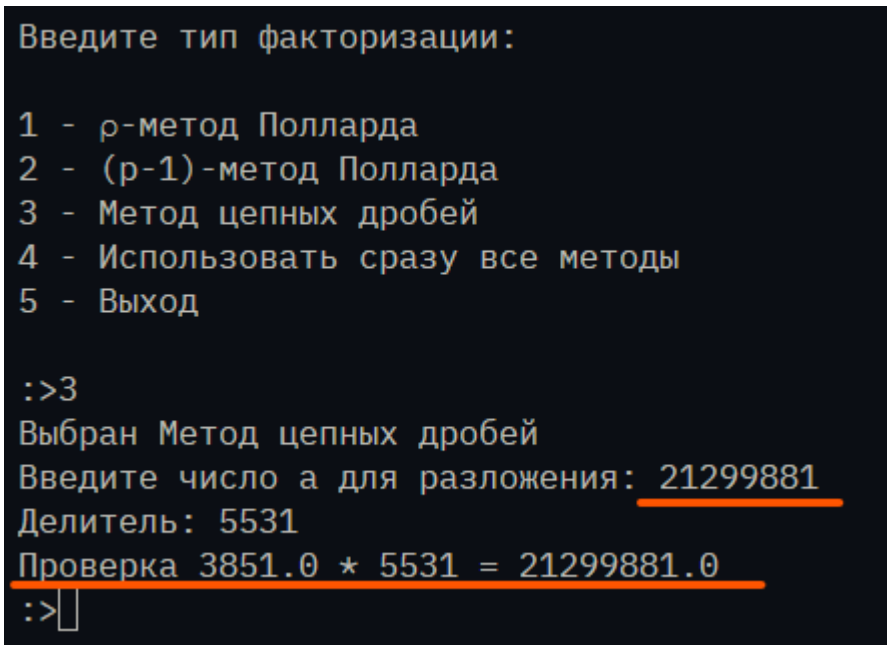

Рисунок 7 - Факторизация числа *21299881* методом Моррисона и Бриллхарта

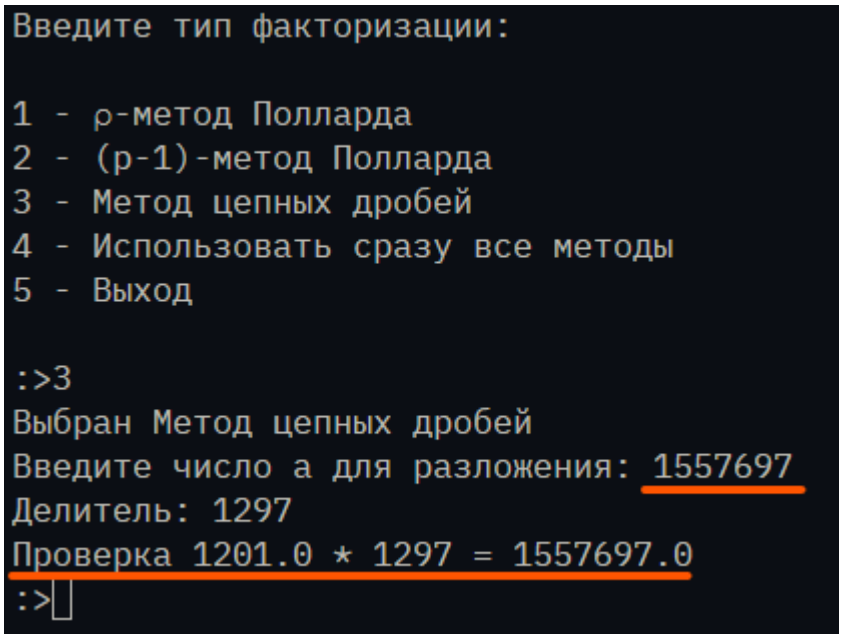

Рисунок 8 - Факторизация числа *1557697* методом Моррисона и Бриллхарта

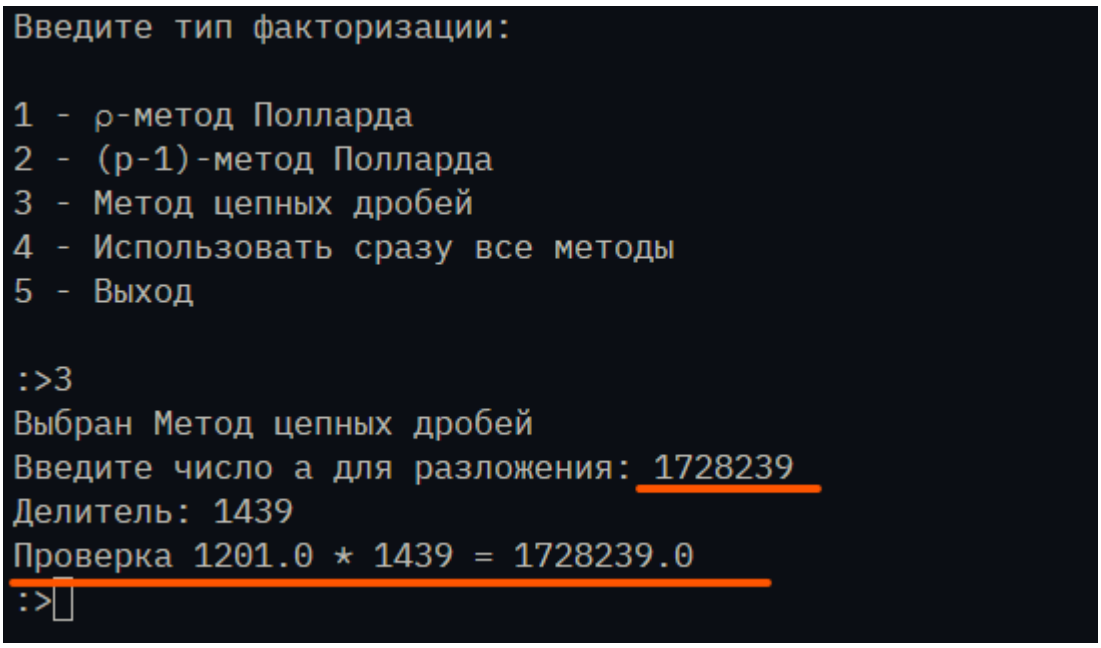

Рисунок 9 - Факторизация числа *1728239* методом Моррисона и Бриллхарта

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были изучены и реализованы три основных метода факторизации целых чисел: ρ-метод Полларда, $(p-1)$-метод Полларда и метод цепных дробей.

1. Реализация ρ-метода Полларда показала свою эффективность для разложения чисел среднего размера на простые множители. Этот метод основан на использовании случайных функций для нахождения делителей. В ходе реализации особое внимание уделялось выбору полиномиальной функции и использованию алгоритма Евклида для нахождения НОД (наибольшего общего делителя). Программа успешно справлялась с задачей разложения чисел с относительно малыми простыми делителями, демонстрируя быстроту при разложении на простые множители.

2. $(p-1)$-метод Полларда оказался особенно полезен для чисел, имеющих простые делители, такие что $p-1$ делится на небольшие простые множители. В отличие от ρ-метода, $(p-1)$-метод использует свойства теории групп, что делает его более эффективным для определённых классов чисел. Реализация метода включала использование быстрого алгоритма возведения в степень по модулю, что значительно улучшило производительность программы, особенно при работе с большими числами.

3. Метод цепных дробей был изучен как более сложный, но мощный метод факторизации. Он основан на разложении квадратного корня числа в цепную дробь и нахождении подходящих дробей. В ходе реализации использовались алгоритмы разложения квадратных корней в цепные дроби и вычисления подходящих дробей, что позволило эффективно находить возможные делители числа. Этот метод оказался более универсальным, но требовал значительных вычислительных ресурсов для больших чисел.

Работа над реализацией алгоритмов способствовала глубокому пониманию особенностей и возможностей различных методов факторизации. В процессе программирования были исследованы числовые примеры, что

позволило не только освоить каждый из методов, но и сравнить их между собой с точки зрения эффективности и применимости к различным типам чисел. Также был изучен и реализован алгоритм для разложения квадратных корней в цепные дроби, что добавило практический опыт работы с числовыми последовательностями и приближенными вычислениями.

В результате проделанной работы стало очевидно, что каждый метод имеет свою область применения, и в зависимости от структуры числа и его размера можно выбрать наиболее подходящий способ факторизации.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Глухов М. М. и др. Введение в теоретико-числовые методы криптографии: учеб. пособие - Москва : Лань, 2011.

2. Маховенко Е.Б. Теоретико-числовые методы в криптографии. М.: Гелиос АРВ, 2006.

3. Черемушкин, А. В. Лекции по арифметическим алгоритмам в криптографии. - Москва : МЦНМО, 2002.

4. Панкратова И.А. Теоретико-числовые методы в криптографии. Томск, 2009.

5. Василенко О.Н. Теоретико-числовые алгоритмы в криптографии. М.:МЦНМО, 2003.

6. Венбо Мао. Современная криптография: теория и практика. М.:Вильямс, 2005.

ПРИЛОЖЕНИЕ А

Реализованные программы для лабораторной работы

```
import random
from sympy import isprime
from decimal import Decimal
from fractions import Fraction
from itertools import combinations
import math

# Вывод матрицы в терминал
def print_matrix(matrix):
    for row in matrix:
        print(*row)
    print()

# Нод двух чисел
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

#Цепная дробь
def sqrt_to_chain_(a, k=20):
    sq = Decimal(a).sqrt()
    res = [int(sq)]
    while k != 0:
        fractional_part = sq - int(sq)
        if fractional_part == 0:
            break
        sq = 1 / Fraction(fractional_part)
        res.append(int(sq))
        sq -= int(sq)
        k -= 1
    return res

def suitable_fractions_chain(chain, k=20):
    P = [0, 1]
    Q = [1, 0]
    for i in range(len(chain)):
        q = chain[i]
        P.append(q * P[-1] + P[-2])
        Q.append(q * Q[-1] + Q[-2])
    return P[2:], Q[2:]

def pollard_ro(n, eps):
    if n == 1:
        return 1
    k = int(math.sqrt(2*math.sqrt(n)*math.log(1 / eps))) + 1
```

```
    divs = set()
    for i in range(k):
        x = random.randint(1, n - 1)
        y, i, stage = 1, 0, 2
        while gcd(n, abs(x - y)) == 1:
            if i == stage:
                y = x
                stage = stage * 2
            x = (x * x + 1) % n
            i += 1
        divs.add(gcd(n, abs(x - y)))
    return sorted(list(divs))

def pollard_p(n, B=1000, k=100):
    res = set()
    for _ in range(k):
        a = random.randint(2, n - 2)
        d = gcd(a, n)
        if d > 1:
            res.add(d)
        for i in range(2, B):
            a = pow(a, i, n)
            d = gcd(a - 1, n)
            if 1 < d < n:
                res.add(d)
    return sorted(list(res))

def jacobi(a, n):
    a = a % n
    t = 1
    while a != 0:
        while a % 2 == 0:
            a //= 2
            r = n % 8
            if r == 3 or r == 5:
                t = -t
        a, n = n, a
        if a % 4 == 3 and n % 4 == 3:
            t = -t
        a = a % n
    return t if n == 1 else 0

def gen_base(num=512, len_=5):
    B = [-1]
    for i in range(2, num):
        if isprime(i) and jacobi(num, i) == 1:
            B.append(i)
        if len(B) == len_:
            return B
    return B
```

```
def get_factor_base(base, num, m):
    res = []
    res_pows = []
    if num < 0:
        res.append(1)
        res_pows.append((-1, 1))
    else:
        res.append(0)
        res_pows.append((-1, 0))
    num = abs(num)
    for p in base[1:]:
        count = 0
        while num % p == 0:
            num //= p
            count += 1
        res.append(count % 2)
        res_pows.append((p, count))
    while len(res) != len(base):
        res.append(0)
    if num != 1:
        return None, None
    return res, res_pows

def abs_(p, n, q):
    return pow(p, 2) - n * pow(q, 2)


def xor_vectors(vectors):
    result = [0] * len(vectors[0])
    for vector in vectors:
        result = [result[i] ^ vector[i] for i in range(len(result))]
    return result

def find_zero_vector_combination(matrix):
    zero_vector = [0] * len(matrix[0])
    results = []
    for r in range(2, len(matrix) + 1):
        for combo in combinations(range(len(matrix)), r):
            selected_vectors = [matrix[i] for i in combo]
            if xor_vectors(selected_vectors) == zero_vector:
                results.append(combo)
    return results

def find_sum(comb, pows, i):
    sum_ = 0
    for index in comb:
        sum_ += pows[index][i][1]
    return sum_ // 2
```

```
def CFRAC(n, len_base=20):
    chain = sqrt_to_chain_(n, len_base)
    P, Q = suitable_fractions_chain(chain, len_base)
    base = gen_base(n, len_base)
    h = len(base)
    system = []
    P_k = [] # Числа которые подошли
    pows_ = [] # Для найденных числе запишем показатели чисел
    for i, p in enumerate(P):
        if len(system) != h + 2:
            min_sub = abs_(p, n, Q[i])
            factor, pows = get_factor_base(base,  min_sub, n)
            if factor != None:
                system.append(factor)
                P_k.append(p)
                pows_.append(pows)
        else:
            break
    combinations = find_zero_vector_combination(system)
    for comb in combinations:
        x = 1
        for i in comb:
            x *= P_k[i]
        x %= n
        y = 1
        i = 1
        for p in enumerate(base[1:]):
            sum_ = find_sum(comb, pows_, i)
            y *= pow(base[i], sum_)
            i += 1
        y %= n
        p = 1
        # print(x, y)
        if x % n != y % n and x % n != -y % n:
            return gcd(x - y, n)
    return CFRAC(n, len_base + 10)

def is_perfect_power(n):
    if n < 2:
        return False

    # Проверяем для всех возможных степеней k, начиная с 2
    for k in range(2, int(math.log(n, 2)) + 1):
        x = int(round(n ** (1 / k)))
        if x ** k == n:
            return True, x, k

    return False, None, None

if __name__ == "__main__":
```

```
    type_ = """Введите тип факторизации: \n
1 - ρ-метод Полларда
2 - (p-1)-метод Полларда
3 - Метод цепных дробей
4 - Использовать сразу все методы
5 - Выход\n"""
    print(type_)
    param = None
    while param not in ["1", "2", "3", "4", "5"]:
        param = input(":>")
        match param:
            case "1":
                print("Выбран ρ-метод Полларда")
                a = input("Введите число a для разложения: ")
                k = input("Укажите e (0 < e < 1): ")
                lst_factors = pollard_ro(int(a), float(k))
                print(f"Делители числа {a}: ", lst_factors)
                print("Проверка:")
                for i, num in enumerate(lst_factors):
                    print(f"{i + 1}) {int(a) / num} * {num} = {int(a) /
num * num}")
                param = None
            case "2":
                print("Выбран (p-1)-метод Полларда")
                a = input("Введите число a для разложения: ")
                k = input("Укажите B для построения базы: ")
                lst_factors = pollard_p(int(a), int(k))
                print(f"Делители числа {a}: ", lst_factors)
                for i, num in enumerate(lst_factors):
                    print(f"{i + 1}) {int(a) / num} * {num} = {int(a) /
num * num}")
                param = None
            case "3":
                print("Выбран Метод цепных дробей")
                a = input("Введите число a для разложения: ")
                if isprime(int(a)):
                    print("Число является простым!")
                    print(f"Результат: {a}")
                elif Decimal(int(a)).sqrt() ** 2 == a:
                    print(f"{a} квадрат числа
{int(Decimal(int(a)).sqrt())}")
                elif is_perfect_power(int(a))[0]:
                    print(f"Делитель: {is_perfect_power(int(a))[1]}")
                elif int(a) % 2 == 0:
                    print(f"Делитель: {int(a) // 2}")
                else:
                    res = CFRAC(int(a))
                    print(f"Делитель: {res}")
                    print(f"Проверка {int(a) / res} * {res} = {int(a) /
res * res}")
```

```
                param = None
            case "4":
                print("Выбраны сразу все методы")
                a = input("Введите число a для разложения: ")
                k = input("Количество итераций: ")
                res1 = pollard_ro(int(a), int(k))
                print(f"Делители числа {a}: ", res1)
                print("Проверка для ρ-метода Полларда")
                for i, num in enumerate(res1):
                    print(f"{i + 1}) {int(a) / num} * {num} = {int(a) /
num * num}")
                res2 = pollard_p(int(a), int(k))
                print(f"Делители числа {a}: ", res2)
                print("Проверка для (p-1)-метода Полларда")
                for i, num in enumerate(res2):
                    print(f"{i + 1}) {int(a) / num} * {num} = {int(a) /
num * num}")
                print("Проверка метода цепных дробей")
                if isprime(int(a)):
                    print("Число является простым!")
                    print(f"Результат: {a}")
                elif Decimal(int(a)).sqrt() ** 2 == a:
                    print(f"{a} квадрат числа
{int(Decimal(int(a)).sqrt())}")
                elif is_perfect_power(int(a))[0]:
                    print(f"Делитель: {is_perfect_power(int(a))[1]}")
                elif int(a) % 2 == 0:
                    print(f"Делитель: {int(a) // 2}")
                else:
                    res3 = CFRAC(int(a), int(k))
                    print(f"Проверка {int(a) / res3} * {res3} = {int(a) /
res3 * res3}")
                param = None
            case "5":
                param = "5"
```