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
Алексеева Александра Александровича

Преподаватель
профессор, д.ф.-м.н. ___________ В. А. Молчанов
подпись, дата

Саратов 2023

СОДЕРЖАНИЕ

1 Постановка задачи

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

В рамках данной лабораторной работы нужно выполнить следующие задачи:

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 Теория

Алгоритмы факторизации бывают:

1) экспоненциально зависящие от длины позиционной записи числа n;

2) субэкспоненциальные алгоритмы, имеющие оценку сложности вида

$$L_n(\gamma, c) = \exp(\left(c + o(1)\right) \log^{\gamma} n \, (loglogn)^{1-\gamma})$$

где $o(1)$ – б.м. при $n \to \infty$ и $0 < \gamma < 1$.

При $\gamma = 0$ величина $L_n(0, c) = (logn)^{c+o(1)}$ – степенная функция от $\log n$.

При $\gamma = 1$ величина $L_n(1, c) = n^{c+o(1)}$ – экспоненциальная функция от $\log n$.

Все современные алгоритмы факторизации субэкспоненциальны.

2.1 $\boldsymbol{\rho}$-метод Полларда

Этот алгоритм является экспоненциальным. Это вероятностный алгоритм факторизации целых чисел, с помощью которого разложено число $F_8 = 2^{2^8} + 1$.

С помощью случайного сжимающего отображения $f: \boldsymbol{Z}_n \to \boldsymbol{Z}_n$ (например, многочлена) строится рекуррентная последовательность $x_{i+1} = f(x_i)(mod\ n)$ со случайным начальным условием $x_0 \in \boldsymbol{Z}_n$ и проверяется

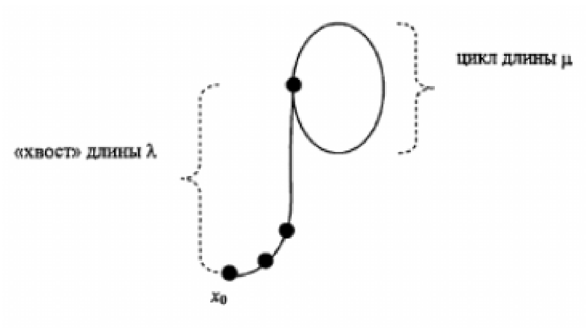
$$1 < \text{НОД}(x_i - x_k, n) < n.$$

Так как составное число n имеет простой делитель $p < \sqrt{n}$, то последовательность $\{x_i\}$ имеет период $\leq n$ и последовательность $\{x_i(mod\ p)\}$ имеет период $\leq p$. Значит, с большой вероятностью найдутся такие значения последовательности x_i, x_k, для которых

$$x_i \equiv x_k(mod\ p), x_i \not\equiv x_k(mod\ n)$$

и, значит, $1 < \text{НОД}(x_i - x_k, n) < n$.

Графически члены последовательности $\{x_i\}$ изображаются так, что сначала образуется конечный «хвост», а затем — цикл конечной длины $\leq p$. Из-за такой фигуры метод называется ρ-методом.



Алгоритм:

Вход. Составное число n и значение $0 < \varepsilon < 1$.

Выход. Нетривиальный делитель d числа n, $1 < d < n$ с вероятностью не менее $1 - \varepsilon$.

Шаг 1. Вычислить $T = \left\lceil \sqrt{2\sqrt{n}\ln\frac{1}{\varepsilon}} \right\rceil + 1$ и выбрать случайный многочлен $f \in \boldsymbol{Z}_n[x]$ (например, $f(x) = x^2 + 1$).

Шаг 2. Случайно выбрать $x_0 \in \boldsymbol{Z}_n$ и, последовательно вычисляя значения $x_{i+1} = f(x_i)(mod\, n), 0 \leq i \leq T$, проверять тест на шаге 3.

Шаг 3. Для каждого $0 \leq k \leq i$ вычислить $d_k = \text{НОД}(x_{i+1} - x_k, n)$ 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 $0 \leq k \leq i$, то перейти к выбору следующего значения последовательности на шаге 2. . Если найдется $d_k = n$ для некоторого $0 \leq k \leq i$, то перейти к выбору нового значения $x_0 \in \boldsymbol{Z}_n$ на шаге 2.

Число шагов алгоритма можно ограничить значением $T = \left\lceil \sqrt{2\sqrt{n}\ln\frac{1}{\varepsilon}} \right\rceil + 1$ и получаем экспоненциальную общую сложность вычислений

$$O(k^2 \log^2 n) = O(\sqrt{n}\ln\frac{1}{\varepsilon}\log^2 n).$$

Теорема («парадокс дней рождения»). Пусть $\lambda > 0$ и $k = \lceil\sqrt{2\lambda n}\rceil$. Для случайной выборки объема $k + 1$ из n элементов вероятность $P_{n,k}$ того, что все элементы попарно различны удовлетворяет условию $P_{n,k} < e^{-\lambda}$.

Замечание 1. Емкостная сложность алгоритма значительно упрощается за счёт его модификации (предложенной Флойдом) – параллельно вычисляются пары членов последовательности (x_i, x_{2i}) до тех пор, пока не найдётся такое k, что $x_k = x_{2k}$. Здесь экспоненциальная сложность вычислений $O(\sqrt{n})$.

Замечание 2. Алгоритм значительно ускоряется за счет модификации шага 3: для $2^h \le i < 2^{h+1}$ вычислять $d_k = \text{НОД}(x_{i+1} - x_k, n)$ для $k = 2^{h-1}$. Получаем экспоненциальную общую сложность вычислений $O(\sqrt[4]{n}\ln\frac{1}{\varepsilon}\log^2 n)$.

Псевдокод ρ-метода Полларда

```
Функция ро_Поллард(n, eps)
        T = корень(2*корень(n) * log(1 / eps)) + 1;
        Бесконечный цикл:
            xi = случайное число от 1 до n;
            xk = xi;
            Цикл для i от 0 до T:
                xi = f(xi) mod n;
                dk = НОД(xi - xk, n);
                Если dk = 1 {
                    Продолжить цикл;
                Если dk = n:
                    Выйти из цикла;
                d[0] = dk;
                d[1] n /dk;
                Вернуть d[0], d[1];
            Конец цикла
            Если i <= T
                Продолжить цикл;
            Прекратить цикл;
Конец функции
```

2.2 $(p - 1)$-метод Полларда

Пусть n – составное число. Фиксируется параметр метода – число $B > 0$, (для больших чисел n, как правило, $10^5 < B \le \sqrt{n}$).

Будем называть B – гладкими те числа, у которых все простые множители не превосходят B.

Рассматривается множество простых чисел $\{q_1, .., q_{\pi(B)}\}$ – факторная база и значения

$$k_i = \left[\frac{\ln n}{\ln q_i}\right] (\text{чтобы } q_i^{k_i} \leq n), T = \prod_{i=1}^{\pi(B)} q_i^{k_i}.$$

<u>Алгоритм:</u>

Вход. Составное число n, число $B > 0$ и значение $T = \prod_{i=1}^{\pi(B)} q_i^{k_i}$.

Выход. Разложение числа n на нетривиальные делители.

<u>Шаг 1</u>. Случайно выбрать $a \in \boldsymbol{Z}_n$ и вычислить $d = \text{НОД}(a, n)$. Если $1 < d < n$, то найден нетривиальный делитель d числа n. Если $d = 1$, то вычислить $b \equiv a^T - 1 \ (mod\ n)$.

<u>Шаг 2</u>. Вычислить $n_1 = \text{НОД}(b, n)$. Если $1 < n_1 < n$, то найден нетривиальный делитель n_1 числа n. Если $n_1 = 1$, то увеличить B. Если $n_1 = n$, то перейти к шагу 1 и выбрать новое значение $a \in \boldsymbol{Z}_n$. Если для нескольких значений $a \in \boldsymbol{Z}_n$ выполняется $n_1 = n$, то уменьшить B.

Сложность вычисления $a^T \equiv 1 \ (mod\ n)$ равна $O(\log T) = O(\pi(B) \log n)$, сложность вычисления $\text{НОД}(b, n)$ равна $O(\log^2 n)$ и общая алгоритма равна $O(\pi(B) \log^3 n)$. Сложность алгоритма при малых B полиномиальная и при $B \approx \sqrt{n}$ экспоненциальная.

<u>Псевдокод (p − 1)-метода Полларда</u>

```
Функция Генерация_базы()
        this.T = BigInteger.ONE;
        для всех простых p от 2 до B:
            В массив базы положить qi = p;
            ki = (целая часть) log(n) / log(qi));
            T = (T * qi)^ki;
Конец функции

Функция p-1_Поллард (n, B)
        Генерация_базы();
```

```
        flag = ложь;
        Бесконечный цикл
            a = случайное число от 1 до n;
            d = НОД(a, n);
            Если d > 1 И d < n:
                d[0] = d;
                d[1] = n /d;
                Вернуть d[0], d[1];
            Если d = 1
                b = (a^T mod n) -1;
                n1 = НОД(b, n);
                Если n1 == 1
                    Увеличить факторную базу
                    Продолжить цикл;
                Если n1 = n
                    Если flag
                        Уменьшить факторную базу;
                        flag = ложь;
                        Продолжить цикл;
                    flag = истина;
                    Продолжить цикл;
                d[0] = n1;
                d[1] = n / n1;
                Вернуть d[0], d[1]
Конец Функции
```

2.3 Алгоритм Бриллхарта-Моррисона

Обозначения:

$$L_n[\gamma, c] = \exp((c + o(1)) \log^{\gamma} n \, (\log\log n)^{1-\gamma}),$$

где $o(1)$ — бесконечно малая при $n \to \infty$ и $0 < \gamma < 1$.

Для фиксированного $\gamma = \frac{1}{2}$ положим

$$L_n[c] = L_n\left[\frac{1}{2}, c\right] = \exp((c + o(1))(\log n \log\log n)^{\frac{1}{2}}) = L^{c+o(1)},$$

где $L = \exp((\log n \log\log n)^{\frac{1}{2}})$.

Пусть n – составное число (что установлено с помощью вероятностных алгоритмов простоты), которое не имеет небольших простых делителей (что проверяется пробными делениями).

Общая идея Лагранжа: найти решения сравнения $x^2 \equiv y^2 (mod\, n)$, удовлетворяющие условию $x \not\equiv \pm\, y \ (mod\, n)$, и, значит,

$$(x - y)(x + y) \equiv 0 \ (mod\, n)$$

влечет, что один делитель p числа n делит $x - y$ и другой делитель q числа n делит $x + y$. Для этого проверяются два условия $1 <$ НОД$(x - y, n) < n$, $1 <$ НОД$(x + y, n) < n$.

Общая схема субэкспоненциальных алгоритмов факторизации:

1. Создаются наборы сравнений $u \equiv v (mod\ n)$ с небольшими u, v.

2. Факторизуются числа u, v.

3. Перемножаются сравнения из набора с целью получения сравнения $x^2 \equiv y^2 (mod\ n)$ с условием $x \not\equiv \pm y\ (mod\ n)$.

4. Вычисляются НОД$(x - y, n)$, НОД$(x + y, n)$.

Известно, что для случайной пары $x, y \in \boldsymbol{Z}_n^*$, удовлетворяющей условию $x^2 \equiv y^2 (mod\ n)$, вероятность

$$P_0 = P[1\ <\ \text{НОД}(x \pm y, n) < n] \geq \frac{1}{2}.$$

Алгоритм Диксона:

Пусть $0 < a < 1$ – некоторый параметр и B – факторная база всех простых чисел, не превосходящих $L^a, k = \pi(L^a)$.

$Q(m) \equiv m^2 (mod\ n)$ – наименьший неотрицательный вычет числа m^2.

Шаг 1. Случайным выбором ищем $k + 1$ чисел $m_1, .., m_{k+1}$, для которых $Q(m_i) = p_1^{\alpha_{i1}} \ldots p_k^{\alpha_{ik}}$, обозначаем $\overline{v_i} = (\alpha_{i1}, .., \alpha_{ik})$.

Шаг 2. Найти ненулевое решение $(x_1, .., x_{k+1}) \in \{0,1\}^{k+1}$ системы k линейных уравнений с $k + 1$ неизвестными

$$x_1 \overline{v_1} + .. + x_{k+1} \overline{v_{k+1}} = \overline{0}\ (mod\ 2).$$

Шаг 3. Положить

$$X \equiv m_1^{x_1} .. m_{k+1}^{x_{k+1}}\ (mod\ n), Y \equiv \prod_{j=1}^{k} p_j^{\frac{\sum x_i \alpha_{ij}}{2}}\ (mod\ n),$$

для которых

$$X^2 \equiv p_1^{\sum_{i=1}^{k+1} x_i \alpha_{i1}} .. p_k^{\sum_{i=1}^{k+1} x_i \alpha_{ik}} \equiv Y^2\ (mod\ n).$$

Проверить условие $1 <$ НОД$(X \pm Y, n) < n$. Если выполняется, то получаем собственный делитель числа n (с вероятностью успеха $P_0 \geq \frac{1}{2}$). В противном случае возвращаемся на шаг 1 и выбираем другие значения $m_1, .., m_{k+1}$.

Сложность алгоритма минимальна при $a = \frac{1}{2}$ и равна

$$L_n\left[\frac{1}{2}, 2\right] = L^{2+o(1)} \text{ для } L = exp((lognloglogn)^{\frac{1}{2}}).$$

<u>Алгоритм Бриллхарта-Моррисона</u> отличается от алгоритма Диксона только способом выбора значений $m_1, .., m_{k+1}$ на шаге 1: случайный выбор заменяется детерминированным определением этих значений с помощью подходящих дробей для представления числа $\sqrt{n}$ цепной дробью.

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

Разложение числа $\sqrt{n}$ в цепную дробь с помощью только операции с целыми числами и нахождения целой части чисел вида $\frac{\sqrt{D}-u}{v}$ может быть найдено по следующей теореме.

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

Таким образом, в алгоритме Диксона возможен выбор $m_i = P_i, Q(m_i) \equiv m_i^2 = P_i^2 \equiv P_i^2 - nQ_i^2\ (mod\ n), Q(m_i) = P_i^2 - nQ_i^2$ и факторная база сужается $B = \{p_0 = -1\} \cup \{p - \text{простое число}: p \leq L^a \text{ и } n \in QR_p\}$.

Сложность алгоритма минимальна при $a = \frac{1}{\sqrt{2}}$ и равна $L_n[\frac{1}{2}, \sqrt{2}]$.

Псевдокод алгоритма Бриллхарта-Моррисона

```
Функция Бриллхар_Моррисон(n, a)
        L=e^((logn*log(logn))^a)
        Сгенерировать факторную базу, первый элемент это -1, затем все простые числа pi <= L такие, что Якоби(pi, n) != -1;
        k = размер базы
        вычислить числители и знаменатели подходящих дробей корня из n
        Бесконечный цикл:
            Qmi = Pi^2 – nQi^2
            Вычислить k+1 массивов:
                vi = (ai0, …, aik)
                ei = (ai0 % 2, …, aik % 2)
            x = решение СЛУ (k уравнений, k+1 неизвестных) x1v1 + … + xk+1*vk+1= 0 (mod 2);
            Если x пусто:
                Увеличить базу;
                Продолжить цикл;
            X = 1;
            Y = 1;
            Для i от 0 до k
                X = X*P[i]^x[i] mod n;
            Для j от 0 до k-1
                step = 0;
                Для i от 0 до размера решения x
                    step += x[i] * vStep[i][j];
                step /= 2;
                Y = Y*p[j]^step mod n;
            Если X^2 mod n != Y^2 mod n
                Продолжить цикл
            gcd1 = НОД(X+Y, n);
            gcd2 = НОД(X-Y, n)
            Если gcd1 ∈ (0, n) ИЛИ gcd2 ∈ (0, n):
                d[0] = gcd1 или gcd2;
                d[1] = n / d[0]
                Вернуть d[0], d[1]
            Иначе:
                Продолжить цикл
Конец Функции
```

3 Результаты работы

3.1 Тестирование программы

```
        Факторизация целых чисел
1 - Факторизация p-методом Полларда
2 - Факторизация (p-1)-методом Полларда
3 - Факторизация методом цепных дробей
1

Составное число n: 221
Значение eps, 0 < eps < 1: 0.1

221 = 13 * 17
```

```
        Факторизация целых чисел
1 - Факторизация p-методом Полларда
2 - Факторизация (p-1)-методом Полларда
3 - Факторизация методом цепных дробей
2

Составное число n: 493
База B: 5

493 = 17 * 29
```

```
        Факторизация целых чисел
1 - Факторизация p-методом Полларда
2 - Факторизация (p-1)-методом Полларда
3 - Факторизация методом цепных дробей
3

Составное число n: 7839991

Pk: { 1 2799 2800 1741599 6969196 870804 9 871416 871425 1742841 4357107 6099948 2617064 877021 3494085 4371106 25200
4547506 5517862 2225377 7743239 193585 96833 871416 }
Base: { -1 2 3 5 11 13 17 43 71 167 179 653 }

(1 1 0 1 0 1 0 1 0 0 0 0 )
(0 0 2 0 0 0 0 0 0 0 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 0 1 1 0 0 0 0 0 1 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 0 4 0 0 0 0 0 0 0 0 0 )
(1 1 2 0 0 0 0 0 0 0 1 0 )
(0 0 0 1 1 0 0 1 0 0 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 0 1 0 0 1 0 0 1 0 0 0 )
(1 1 0 0 0 0 1 0 1 0 0 0 )
(0 0 1 1 0 0 0 0 0 1 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 1 2 3 0 0 0 0 0 0 0 0 )
(1 0 0 0 1 0 2 0 0 0 0 0 )
(0 0 6 0 0 0 0 0 0 0 0 0 )
(1 1 0 0 0 0 0 0 0 0 1 0 )
(0 0 1 0 0 1 0 0 1 0 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 1 0 3 0 0 0 0 0 0 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 0 )
(0 0 0 0 0 0 0 0 0 0 0 1 )
(1 1 2 0 0 0 0 0 0 0 1 0 )

7839991 = 2803 * 2797
```

3.2 Код программы

```
#include <iostream>
#include <cmath>
#include <vector>
#include <string>
#include <set>
#include "map"
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include "Pattern.cpp"

using namespace std;
using namespace boost::multiprecision;

class Pattern {
private:
	static vector <cpp_int> deg2(cpp_int el, cpp_int n) {//Раскладываем 
число на степени двойки
		vector <cpp_int> res;
		while (n != 0) {
			if (n / el == 1) {
				res.push_back(el);
				n -= el;
				el = 1;
			}
			else
				el *= 2;
		}
		return res;
	}


	static cpp_int multMod(cpp_int n, cpp_int mod, vector <pair <cpp_int, 
cpp_int>> lst) {//Умножаем число по модулю
		if (lst.size() == 1) {
			cpp_int res = 1;
			for (int i = 0; i < lst[0].second; i++)
				res = res * lst[0].first % mod;
			return res;
		}
		else if (lst[0].second == 1) {
			cpp_int el = lst[0].first;
			lst.erase(lst.begin());
			return (el * multMod(n, mod, lst)) % mod;
		}
		else {
			for (int i = 0; i < lst.size(); i++)
				if (lst[i].second > 1) {
					lst[i].first = (lst[i].first * 
lst[i].first) % mod;
					lst[i].second /= 2;
				}
			return multMod(n, mod, lst);
		}
	}


	static int partition(vector <cpp_int>& a, int start, int end) {
		cpp_int pivot = a[end];
		int pIndex = start;

		for (int i = start; i < end; i++) {
```

```
                        if (a[i] <= pivot) {
                                swap(a[i], a[pIndex]);
                                pIndex++;
                        }
                }

                swap(a[pIndex], a[end]);
                return pIndex;
        }
public:
        static cpp_int powClosed(cpp_int x, cpp_int y, cpp_int mod)
{//Возводим число в степени по модулю
                if (y == 0)
                        return 1;

                vector <cpp_int> lst = deg2(1, y);
                vector <pair <cpp_int, cpp_int>> xDegs;
                for (int i = 0; i < lst.size(); i++)
                        xDegs.push_back(make_pair(x, lst[i]));

                cpp_int res = multMod(x, mod, xDegs);
                return res;
        }


        //Возведение в степень
        static cpp_int pow(cpp_int x, cpp_int y) {
                cpp_int res = 1;
                for (int i = 0; i < y; i++)
                        res *= x;
                return res;
        }


        //Символ Лежандра через критерий Эйлера
        static cpp_int symbolLegendre(cpp_int a, cpp_int p) {
                if (a == 0)
                        return 0;
                cpp_int res = powClosed(a, (p - 1) / 2, p);
                return res == 1 ? 1 : -1;
        }


        //Символ Якоби
        static cpp_int symbolJacobi(cpp_int  a, cpp_int  b) {
                if (usualEuclid(a, b) != 1)
                        return 0;

                cpp_int  r = 1;
                if (a < 0) {
                        a = abs(a);
                        if (b % 4 == 3)
                                r = -r;
                }

                while (a != 0) {
                        cpp_int t = 0;
                        while (a % 2 == 0) {
                                t++;
                                a /= 2;
                        }
                        if (t % 2 == 1 && (b % 8 == 3 || b % 8 == 5))
                                r = -r;
```

```
			if (a % 4 == 3 && b % 4 == 3)
				r = -r;

			cpp_int c = a;
			a = b % c;
			b = c;
		}
		return r;
}


//Из бинарной формы в десятичную
static cpp_int decForm(string x) {
		cpp_int res = 0, deg = 1;
		if (!x.empty() && x.back() == '1')
			res += 1;
		for (short i = x.length() - 2; i >= 0; i--) {
			deg = deg * 2;
			if (x[i] == '1')
				res += deg;
		}
		return res;
}


//Из десятчной формы в бинарную
static string binForm(cpp_int x) {
		string bitter = "";
		while (x != 0) {
			bitter = (x % 2 == 0 ? "0" : "1") + bitter;
			x = x / 2;
		}
		if (bitter == "")
			return "0";
		return bitter;
}


//Тест на простоту Миллера-Рабина
static bool miller_rabin(cpp_int n, int k = 10) {
		if (n == 0)
			return false;
		else if (n == 1 || n == 2 || n == 3)
			return true;

		cpp_int d = n - 1;
		cpp_int s = 0;
		while (d % 2 == 0) {
			s++;
			d = d / 2;
		}

		cpp_int nDec = n - 1;
		for (int i = 0; i < k; i++) {
			cpp_int a = rand() % nDec;
			if (a == 0 || a == 1)
				a = a + 2;

			cpp_int x = powClosed(a, d, n);
			if (x == 1 || x == nDec)
				continue;
```

```
                        bool flag = false;
                        for (int j = 0; j < s; j++) {
                                x = (x * x) % n;
                                if (x == nDec) {
                                        flag = true;
                                        break;
                                }
                        }
                        if (!flag)
                                return false;
                }

                return true;
        }


        //Обычный алгоритм Евклида
        static cpp_int usualEuclid(cpp_int a, cpp_int b) {
                if (a < b)
                        swap(a, b);
                if (a < 0 || b < 0)
                        throw string{ "Выполнение невозможно: a < 0 или b < 0"
};
                else if (b == 0)
                        return a;

                cpp_int r = a % b;
                return usualEuclid(b, r);
        }


        //Бинарный алгоритм Евклида
        static int binaryEuclid(int a, int b) {
                if (a < 0 || b < 0)
                        throw string{ "Выполнение невозможно: a < 0 или b < 0"
};

                if (a == 0)
                        return b;
                else if (b == 0 || a == b)
                        return a;
                else if (a == 1 || b == 1)
                        return 1;
                else if ((a & 1) == 0 && (b & 1) == 0)
                        return binaryEuclid(a >> 1, b >> 1) << 1;
                else if ((a & 1) == 0 && (b & 1) == 1)
                        return binaryEuclid(a >> 1, b);
                else if ((a & 1) == 1 && (b & 1) == 0)
                        return binaryEuclid(a, b >> 2);
                else {
                        if (b > a)
                                return binaryEuclid((b - a) >> 1, a);
                        else
                                return binaryEuclid((a - b) >> 1, b);
                }
        }


        //Расширенный алгоритм Евклида
        static pair <cpp_int, cpp_int> advancedEuclid(cpp_int a, cpp_int b) {
                if (a < 0 || b < 0)
                        throw string{ "Выполнение невозможно: a < 0 или b < 0"
};
```

```
        cpp_int q, aPrev = a, aCur = b, aNext = -1;
        cpp_int xPrev = 1, xCur = 0, xNext;
        cpp_int yPrev = 0, yCur = 1, yNext;
        while (aNext != 0) {
                q = aPrev / aCur;
                aNext = aPrev % aCur;
                aPrev = aCur; aCur = aNext;

                xNext = xPrev - (xCur * q);
                xPrev = xCur; xCur = xNext;

                yNext = yPrev - (yCur * q);
                yPrev = yCur; yCur = yNext;
        }

        return make_pair(xPrev, yPrev);
}


//Функция Эйлера
static cpp_int funEuler(cpp_int n) {
        cpp_int res = 1;
        for (int i = 2; i < n; i++)
                if (usualEuclid(n, i) == 1)
                        res++;
        return res;
}


//Хэш-функция
static string hashing(string str) {
        string res;
        hash <string> hashStr;
        return to_string(hashStr(res));
}


//Генерация простого числа и его первообразного корня
static pair <cpp_int, cpp_int> generateGN() {
        cpp_int q = rand() % 1000;
        while (funEuler(q) != q - 1)
                q++;

        cpp_int s, n = 2, nDec;
        while (!miller_rabin(n)) {
                string sBin = "";
                int sBinSize = rand() % 50 + 1;
                for (int i = 0; i < sBinSize; i++)
                        sBin = sBin + to_string(rand() % 2);
                s = decForm(sBin);

                n = (q * s) + 1;
                nDec = n - 1;
        }

        cpp_int a = 2;
        while (nDec > a) {
                cpp_int g = powClosed(a, nDec / q, n);
                if (g == 1) {
                        a++;
                        continue;
                }
```

```
				return make_pair(g, n);
			}
			return make_pair(0, 0);//Строка для обхода warning'a в Linux
		}


		//Квадратный корень в Zp
		static cpp_int sqrtFromZp(cpp_int a, cpp_int p) {
			a = a % p;
			cpp_int m = 0, q = p - 1;
			while (q % 2 != 1) {
				m++;
				q /= 2;
			}

			cpp_int b = rand() % p;
			while (symbolLegendre(b, p) != -1)
				b = (b + 1) % p;

			vector <cpp_int> kArr;
			for (int i = 1;; i++) {
				cpp_int k = 0;
				while (powClosed(a, pow(2, k) * q, p) != 1)
					k++;
				kArr.push_back(k);
				if (k == 0)
					break;
				a = (a * pow(b, pow(2, m - kArr.back()))) % p;
			}

			cpp_int r = powClosed(a, (q + 1) / 2, p);
			for (int i = kArr.size() - 2; i >= 0; i--)
				r = (r * advancedEuclid(pow(b, pow(2, m - kArr[i] -
1)), p).first) % p;

			return r;
		}


		static cpp_int generateSimpleNum(unsigned short k) {
			cpp_int q = rand() % 1000;
			while (funEuler(q) != q - 1)
				q++;

			cpp_int s, n = 2, nDec;
			while (!miller_rabin(n)) {
				string sBin = "1";
				int sBinSize = rand() % (k / 2) + k / 2;
				for (int i = 0; i < sBinSize; i++)
					sBin = sBin + to_string(rand() % 2);
				s = decForm(sBin);

				n = (q * s) + 1;
				nDec = n - 1;
			}

			return n;
		}


		//Быстрая сортировка
		static void quicksort(vector <cpp_int>& a, int start, int end) {
```

```
		if (start >= end) {
			return;
		}

		int pivot = partition(a, start, end);
		quicksort(a, start, pivot - 1);
		quicksort(a, pivot + 1, end);
	}
};



class Factorization
{
private:
	static cpp_int baseForPollard(cpp_int n, int B)
	{
		cpp_int res = 1;

		if (B < 2)
			throw string{ "База B должна быть > 1!" };
		if (B >= 2)
			res *= Pattern::pow(2, cpp_int(log(cpp_dec_float_50(n))
/ 0.69314718));
		if (B >= 3)
			res *= Pattern::pow(3, cpp_int(log(cpp_dec_float_50(n))
/ 1.09861228));

		for (int i = 5; i <= B; i++)
			if (Pattern::miller_rabin(i))
				res *= Pattern::pow(i,
cpp_int(log(cpp_dec_float_50(n)) / log(cpp_dec_float_50(i))));
		return res;
	}


	static vector <cpp_int> factorsNum(cpp_int n)
	{
		vector <cpp_int> res;
		if (n < 0)
		{
			res.push_back(-1);
			n = -n;
		}

		while (!Pattern::miller_rabin(n))
		{
			cpp_int factor = roMethodPollarda(n, 0.1);
			res.push_back(factor);
			n /= factor;
		}

		res.push_back(n);
		return res;
	}


	static set <cpp_int> baseForBM(vector <vector <cpp_int>> factorsP)
	{
		set <cpp_int> res;

		map <cpp_int, unsigned short> countFactors;
		for (unsigned short i = 0; i < factorsP.size(); i++)
```

```
		{
			if (countFactors.find(factorsP[i][0]) ==
countFactors.end())
				countFactors.insert(make_pair(factorsP[i][0],
1));

			unsigned short deg = 1;
			for (unsigned short j = 1; j < factorsP[i].size(); j++)
			{
				if (factorsP[i][j] == factorsP[i][j - 1])
					deg++;
				else
				{
					if (deg % 2 == 0)
						countFactors[factorsP[i][j - 1]]
+= 2;
					else
						countFactors[factorsP[i][j - 1]]
+= 1;
					deg = 1;
				}
			}

			if (deg % 2 == 0)
				countFactors[factorsP[i].back()] += 2;
			else
				countFactors[factorsP[i].back()] += 1;
		}

		for (auto i = countFactors.begin(); i != countFactors.end();
i++)
			if (i->second >= 2)
				res.insert(i->first);

		if (res.find(1) != res.end())
			res.erase(1);
		return res;
	}


	static vector <vector <unsigned int>> getValueEs(vector <cpp_int> P,
set <cpp_int> base, cpp_int n)
	{
		P.erase(P.begin());
		vector <vector <unsigned int>> es;
		for (unsigned short i = 0; i < P.size(); i++)
		{
			cpp_int Pdeduct = P[i] * P[i] % n;
			Pdeduct = Pdeduct > n / 2 ? Pdeduct -= n : Pdeduct;

			vector <unsigned int> e;
			e.push_back(Pdeduct < 0 ? 1 : 0);
			Pdeduct = abs(Pdeduct);
			for (auto j = ++base.begin(); j != base.end(); j++)
			{
				unsigned int deg = 0;
				while (Pdeduct % *j == 0)
				{
					deg++;
					Pdeduct /= *j;
				}
				e.push_back(deg);
			}
```

```
                        if (Pdeduct == 1)
                                es.push_back(e);
                        else
                                es.push_back(vector <unsigned int>
(base.size(), 0));
                }

                return es;
        }


        static cpp_int xorRaws(vector <unsigned int> a, vector <unsigned int>
b)
        {
                unsigned int res = 0;
                for (unsigned short i = 0; i < a.size(); i++)
                        res += a[i] + b[i];
                return res % 2;
        }
public:
        static cpp_int roMethodPollarda(cpp_int n, cpp_dec_float_50 eps)
        {
                cpp_int T = sqrt(2 * sqrt(n) * cpp_int(log(1 / eps))) + 1;
                vector <cpp_int> xs;

        gen_x0:
                cpp_int xCur = rand() % (n - 1) + 1, xNext;
                xs.clear();
                xs.push_back(xCur);

                for (unsigned short i = 1; i <= T; i++)
                {
                        xNext = (xCur * xCur + 1) % n;
                        cpp_int dk;

                        for (unsigned short k = 0; k < xs.size(); k++)
                        {
                                dk = Pattern::usualEuclid((xNext - xs[k] + n) %
n, n);

                                if (1 < dk && dk < n)
                                        return dk;
                                else if (dk == n)
                                        goto gen_x0;
                        }

                        xCur = xNext;
                        xs.push_back(xNext);
                }

                goto gen_x0;
                return 0;
        }


        static cpp_int roDecMethodPollarda(cpp_int n, int B)
        {
                if (Pattern::miller_rabin(n))
                        throw string{ "Число " + to_string(n) + " - простое!"
};

        step0:
                cpp_int T = baseForPollard(n, B);
```

```
		unsigned short countA = 0;
	step1:
		cpp_int a = rand() % (n - 2) + 2;
		countA++;
		cpp_int d = Pattern::usualEuclid(a, n);
		if (1 < d && d < n)
			return d;

		cpp_int b = Pattern::powClosed(a, T, n) - 1;
		cpp_int n1 = Pattern::usualEuclid(b, n);
		if (1 < n1 && n1 < n)
			return n1;
		else if (n1 == 1)
		{
			B++;
			goto step0;
		}
		else if (n1 == n)
		{
			if (countA == 10)
			{
				countA = 0;
				B--;
				goto step0;
			}
			goto step1;
		}
	}


	static cpp_int BrillhartMorrison(cpp_int n)
	{
		if (Pattern::miller_rabin(n))
			throw string{ "Число " + to_string(n) + " - простое!" 
};
		else if (sqrt(n) * sqrt(n) == n)
			throw string{ "Число " + to_string(n) + " является 
квадратом некоторого числа!" };

		vector <cpp_int> P{ 1, sqrt(n) };
		cpp_int a = sqrt(n);
		cpp_dec_float_50 x = sqrt(cpp_dec_float_50(n)) - 
cpp_dec_float_50(a);
		vector <vector <cpp_int>> factorsP;

		cpp_int Pdeduct = P.back() * P.back() % n;
		Pdeduct = Pdeduct > n / 2 ? Pdeduct -= n : Pdeduct;
		factorsP.push_back(factorsNum(Pdeduct));
		Pattern::quicksort(factorsP.back(), 0, factorsP.back().size() 
- 1);

		for (unsigned short k = 1; k <= log2(cpp_dec_float_50(n)); 
k++)
		{
			a = cpp_int(1 / x);
			x = 1 / x - cpp_dec_float_50(a);
			P.push_back((cpp_int(a) * P.back() + P[P.size() - 2]) % 
n);

			Pdeduct = P.back() * P.back() % n;
			Pdeduct = Pdeduct > n / 2 ? Pdeduct -= n : Pdeduct;
			factorsP.push_back(factorsNum(Pdeduct));
```

```
                        Pattern::quicksort(factorsP.back(), 0,
factorsP.back().size() - 1);
                }

                cout << "\nPk: { ";
                for (unsigned short i = 0; i < P.size(); i++)
                        cout << P[i] << " ";
                cout << "}";

                set <cpp_int> base = baseForBM(factorsP);
                cout << "\nBase: { ";
                for (auto i = base.begin(); i != base.end(); i++)
                        cout << *i << " ";
                cout << "}";

                vector <vector <unsigned int>> es = getValueEs(P, base, n);
                cout << endl;
                for (unsigned short i = 0; i < es.size(); i++)
                {
                        cout << "\n(";
                        for (unsigned short j = 0; j < es[i].size(); j++)
                                cout << es[i][j] << " ";
                        cout << ")";
                }

                P.erase(P.begin());
                for (unsigned short i = 0; i < es.size(); i++)
                        for (unsigned short j = i + 1; j < es.size(); j++)
                                if (xorRaws(es[i], es[j]) == 0)
                                {
                                        cpp_int s = P[i] * P[j] % n;
                                        cpp_int t = 1;

                                        unsigned short iter = 1;
                                        for (auto k = ++base.begin(); k !=
base.end(); k++, iter++)
                                        {
                                                unsigned int gamma =
(es[i][iter] + es[j][iter]) / 2;
                                                t *= Pattern::pow(*k, gamma);
                                        }

                                        cpp_int q = Pattern::usualEuclid(s + t,
n);
                                        if (q != n && q != 1)
                                                return q;
                                        q = Pattern::usualEuclid(s - t, n);
                                        if (q != n && q != 1)
                                                return q;
                                }

                return roMethodPollarda(n, 0.1);
        }
};



int main()
{
        setlocale(LC_ALL, "ru");
        srand(time(NULL));
```

```
	for (;;)
	{
		cout << "\tФакторизация целых чисел \n1 - Факторизация p-
методом Полларда \n";
		cout << "2 - Факторизация (p-1)-методом Полларда \n3 -
Факторизация методом цепных дробей \n";
		unsigned short x;
		cin >> x;

		if (x == 1)
		{
			cpp_int n;
			cpp_dec_float_50 eps;
			cout << "\nСоставное число n: ";
			cin >> n;
			cout << "Значение eps, 0 < eps < 1: ";
			cin >> eps;

			try
			{
				cpp_int res =
Factorization::roMethodPollarda(n, eps);
				cout << endl << n << " = " << res << " * " << n
/ res;
			}
			catch (string& error)
			{
				cout << endl << error;
			}
		}
		else if (x == 2)
		{
			cpp_int n;
			int B;
			cout << "\nСоставное число n: ";
			cin >> n;
			cout << "База B: ";
			cin >> B;

			try
			{
				cpp_int res =
Factorization::roDecMethodPollarda(n, B);
				cout << endl << n << " = " << res << " * " << n
/ res;
			}
			catch (string& error)
			{
				cout << endl << error;
			}
		}
		else if (x == 3)
		{
			cpp_int n;
			cpp_dec_float_50 a;
			cout << "\nСоставное число n: ";
			cin >> n;

			try
			{
				cpp_int res =
Factorization::BrillhartMorrison(n);
```

```
                                cout << "\n\n" << n << " = " << res << " * " <<
n / res;
                        }
                        catch (string& error)
                        {
                                cout << endl << error;
                        }
                }

                cout << "\n\n";
        }
}
```