Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Лабораторная работа №5

студента 5 курса 531 группы
специальности 10.05.01 «Компьютерная безопасность»
факультета компьютерных наук и информационных технологий
Енца Михаила Владимировича

Преподаватель
профессор _____ В. А. Молчанов
подпись, дата

Заведующий кафедрой
д.ф.-м.н., доцент _____ М. Б. Абросимов
подпись, дата

Саратов 2018

1. Постановка задачи.

Изучение основных методов факторизации целых чисел и их программная реализация.

2. Теоретические сведения по рассмотренным темам с их обоснованием.

Определение. Задача разложения составного числа на множители формулируется так: для данного положительного целого числа n найти его каноническое разложение $n = p_1^{a^1} p_2^{a^2} \dots p_3^{a^3}$, где p_i – попарно различные простые числа, $a_i \geq 1$.

p-Метод Полларда

Пусть n – нечетное составное число, $S = \{0, 1, \dots, n-1\}$ и $f: S \to S$ – случайное отображение, обладающее сжимающими свойствами, например $f(x) \equiv x^2 + 1 \ (mod\ n)$. Основная идея метода состоит в следующем. Выбираем случайный элемент $x_0 \in S$ и строим последовательность $x_0 x_1 x_2 \dots$, определяемую рекуррентным 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» $x_0, x_1, \dots, x_{i-1}$ длины $O(\sqrt{\pi n/8})$ и цикла $x_i = x_j, x_{i+1}, \dots, x_{j-1}$ той же длины.

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

Псевдокод p-Метод Полларда.

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

Выход. Нетривиальный делитель p числа n.

1. Положить $a = c, b = c$.

2. Вычислить $a = f(a)(mod\ n), b = f\big(f(b)(mod\ n)\big)\ (mod\ n)$

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.

(p-1)-Метод Полларда.

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.

Пусть $n = pq$ и пусть каноническое разложение числа $p - 1$ имеет вид $p - 1 = p_1^{a_1} p_2^{a_2} \dots p_s^{a_s}$. Найдем максимальные показатели $l_1 l_2 \dots l_n$, для которых $p_i^{l_i} \le n$. Прологарифмируем обе части этого неравенства: $l_i \ln p_i \le \ln n$, откуда $l_i \le \left[\frac{\ln n}{\ln p_i}\right]$. Вычислим $M = p_1^{\left[\frac{\ln n}{\ln p_1}\right]} p_2^{\left[\frac{\ln n}{\ln p_2}\right]} \dots p_s^{\left[\frac{\ln n}{\ln p_s}\right]}$, тогда $M = (p-1) * z$ для некоторого целого числа z.

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

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

Псевдокод (p-1)-Метод Полларда.

Вход. Составное число n.

Выход. Нетривиальный делитель p числа n.

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

2. Выбрать случайное целое a, $2 \le a \le n-2$, и вычислить $d = \text{НОД}(a, n)$. При $d \ge 2$ положить $p = d$ и результат: p.

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

3.1. Вычислить $l = [(\ln n)/(\ln p_i)]$.

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

4. Вычислить $d = \text{НОД}(a-1, n)$.

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

Метод непрерывных дробей

Метод факторизации с помощью непрерывных дробей является оптимизацией метода Диксона, который в свою очередь основывается на методе квадратов.

Теорема Ферма о разложении: $\forall n \in Z, n > 0, n-$ нечетное существует взаимно однозначное соответствие между множеством делителей числа n, не меньших $\sqrt{n}$, и множеством пар $\{s,t\}$ таких неотрицательных целых чисел, что $n = s_2 - t_2$.

Если $n = pq$, где числа p и q достаточно близки друг к другу, то число t мало, а значит, s намного больше, чем $\sqrt{n}$. В этом случае можно найти p и q, последовательно перебирая числа $s = [\sqrt{n}] + 1, [\sqrt{n}] + 2, \dots$ до тех пор, пока не найдется такое s, что разность $s_2 - n$ является полным квадратом, то есть равна t_2.

Далее будем считать, что число n не является полным квадратом.

На практике для разложения числа n достаточно найти такие целые числа s, t, что $s_2 \equiv t_2 \ (mod\ n)$, то есть $(s+t)(s-t) \equiv 0 \ (mod\ n)$. Если $s \neq \pm t \ (mod\ n)$, то число $n|(s+t)(s-t)$, но не делит ни один из сомножителей.

Значит, один делитель числа n ($p = \text{НОД}(s-t, n)$) делит разность $s-t$, а другой делитель $q = np$ делит сумму $s+t$.

Метод Диксона

Вход: составное число n.

Выход: нетривиальный делитель p числа n.

1. Построить базу разложения $B = \{p_0, p_1, \dots, p_h\}$, где $p_0 = -1$ и $p_1, \dots, p_h$ - попарно различные простые числа.

2. Найти $h+2$ целых чисел s_i для каждого из которых абсолютно наименьший вычет $s_i^2 \ (mod\ n)$ является B-гладким: $s_i^2 (mod\ n) = \prod_{j=0}^{h} p_j^{\alpha_j(s_i)}$,

где $\alpha_j(s_i) \geq 0$, и каждому числу s_i сопоставить вектор показателей $(\alpha_0(s_i), \alpha_1(s_i), \dots, \alpha_h(s_i))$.

3. Найти (например, методом гауссова исключения) такое непустое множество $K \subseteq \{1,2, \dots, h+1\}$, что, для $k \in K \quad \bigoplus e_k = 0$, где $e_k = (e_1^{(k)}, e_2^{(k)}, \dots, e_h^{(k)}), e_j^{(k)} \equiv \alpha_j(s_k)(mod\ 2), 0 \leq j \leq h$.

4. Положить $s = \prod_{k \in K} s_k (mod\ n)$, $t = \prod_{i=1}^{h} p_j^{\frac{1}{2}\sum_{k \in K} a_i(s_k)} \ (mod\ n)$.

Тогда $s^2 \equiv t^2 \ (mod\ n)$.

5. Если $s \neq \pm t \ (mod\ n)$, то положить $p = \text{НОД}(s-t, n)$ и результат: p. В противном случае вернуться на шаг 3 и поменять множество K (на практике обычно есть несколько вариантов выбора множества K и при одной и той же базе разложения B. Если все возможности исчерпаны, то следует увеличить базу разложения).

Метод непрерывных дробей является оптимизацией метода Диксона (опубликован Дж. Брилхартом и обычно данный алгоритм называют его именем). В качестве чисел s_i выбираются числители P_k подходящих дробей к обыкновенной непрерывной дроби, выражающей число $\sqrt{n}$. Малый размер чисел $s_i^2 \ (mod\ n)$ обеспечивается тем, что для всех $k = 1,2, \dots$ справедливо $|P_k^2 - \alpha^2 Q_k^2| < 2\alpha$, где $\frac{P_k}{Q_k}$ – k-ая подходящая дробь к числу $\alpha > 1$.

Таким образом, в алгоритме Диксона в качестве s_i можно брать числители подходящих дробей к $\sqrt{n}$. Кроме того, из базы B можно исключить те простые p_i, по модулю которых n является квадратичным невычетом.

Формула для непрерывной дроби $\sqrt{n} = (r_0, r_1, r_2, \dots)$:

1. Положить $a_0 = \sqrt{n}$ (действительное число), $r_0 = \lfloor a_0 \rfloor, rat_0 = 1, n_0 = 0$.

2. Для всех $i = 1,2, \dots$ имеем:

$n_{i+1} = r_i * rat_i - n_i$, $rat_{i+1} = n - \frac{n_{i+1}^2}{rat_i}$ (целочисленное деление), $r_{i+1} = \lfloor a_0 + \frac{n_{i+1}}{rat_{i+1}} \rfloor$.

Коды программ, реализующей рассмотренные алгоритмы

Программа реализована на языке Python (версия интерпретатора 3.6).

factorization.py

```
import sys
import time
from math import log, sqrt, exp, gcd
import random
import functools
import sympy

import utils
import gaussian
from chain_fractions import *

B_DEFAULT = (2, 3, 5)
N_DEFAULT = 1728239

def compressor(x):
    return x * x + 1

def compressor2(x):
    return x * x + 31

def ppollard(n, c=1, f=compressor2):
    a = c
    b = c
    while True:
        a, b = f(a) % n, f(f(b) % n) % n
```

```
        d = gcd(a - b, n)
        if 1 < d < n:
            return d
        elif d == n:
            return

def p1pollard(n):
    b = p1pollard_base(n)
    a = random.randint(2, n - 2)
    d = gcd(a, n)
    if d >= 2:
        return d
    for pi in b:
        li = int(log(n) / log(pi))
        a = pow(a, pow(pi, li), n)
    d = gcd(a - 1, n)
    if 1 < d < n:
        return d

def p1pollard_base(n):
    base_size = int(n ** (1 / 6))
    return utils.generate_base(min(base_size, utils.MAX_B_SIZE))


def is_b_smooth(p, b):
    alpha = []
    for bi in b:
        k = 0
        while p % bi == 0 and not bi < 0 < p and p != 1:
            p //= bi
            k += 1
        alpha.append(k)
    return p == 1, alpha, [al % 2 for al in alpha]


def dixon(n):
    base = dixon_base(n)
    base = [-1] + list(filter(lambda bi: utils.legendre(n, bi) == 1, base))
    h = len(base) - 1

    ps = []
    alphas = []
    es = []
    convergent = gen_convergent(gen_square_chain_fraction(n))
    while len(ps) < h + 2:
```

```
            try:
                pi, qi = next(convergent)
            except ValueError:
                return None

            pi2 = pi ** 2 % n
            if n - pi2 < pi2:
                pi2 = -(n - pi2)
            smooth, alpha, e = is_b_smooth(pi2, base)
            if smooth:
                ps.append(pi)
                alphas.append(alpha)
                es.append(e)

        for ks in gaussian.gen_gaussian(es):
            s = 1
            for k in ks:
                s = (s * ps[k]) % n
            t = 1
            for b_idx, b in enumerate(base):
                t = (t * pow(b, functools.reduce(int.__add__, (alphas[k][b_idx] for k
in ks)) // 2, n)) % n
            # проверка, что ks - не решение системы
            assert pow(s, 2, n) == pow(t, 2, n)

            if s != t and s != n - t:
                p = gcd((s - t) % n, n)
                return p


    def dixon_base(n):
        base_size = int(sqrt(exp(2 / 3 * sqrt(log(n) * log(log(n))))))
        return utils.generate_base(min(base_size, utils.MAX_B_SIZE))


    FUNCS = [('p-метод Полларда', ppollard),
             ('(p-1)-метод Полларда', p1pollard),
             ('Метод непрерывных дробей', dixon)]


    def test_accuracy(n):
        print('Тест корректности числа {}'.format(n))
        for (func_name, func) in FUNCS:
            divisor = func(n)
```

```
            print(func_name + ':', 'Делитель не найден' if divisor is None else divisor)


        def test_speed(n):
            print('Тест скорости работы алгоритмов (10000 запусков) на числе {}:'.format(n))
            divisors = [str(func(n)) for (func_name, func) in FUNCS]
            times = []
            for (func_name, func) in FUNCS:
                start = time.time()
                for _idx in range(10000):
                    func(n)
                times.append(time.time() - start)

            print(' ' * 10, 'p-метод Полларда  ', '(p-1)-метод Полларда  ', 'Метод непрерывных дробей')
            print('{:>9}  {:>15}  {:>20}  {:>23}'.format('Делитель', *divisors))
            print('{:>9}    {:>13.3f} с    {:>18.3f} с    {:>21.3f} с'.format('Время', *times))


        if __name__ == '__main__':
            arg = sys.argv[1]
            test_n, k = utils.fac2k(int(sys.argv[2]))
            print(test_n)
            if sympy.isprime(test_n):
                print('{} - простое'.format(test_n))
                exit(0)
            if arg == '-a':
                test_accuracy(test_n)
            elif arg == '-s':
                test_speed(test_n)

        import math


        def gen_chain_fraction(p, q):
            a = int(p / q)
            yield a

            while p != q:
                p, q = q, p - q * a
                a = int(p / q)
                yield a
```

```
def gen_square_chain_fraction(n):
    a0 = math.sqrt(n)
    r0 = int(a0)
    yield r0
    ratio0 = 1
    numenator0 = 0

    while True:
        numenator1 = r0 * ratio0 - numenator0
        ratio1 = (n - numenator1 * numenator1) // ratio0
        if ratio1 == 0:
            raise ValueError('Число является полным квадратом')
        r1 = int((a0 + numenator1) / ratio1)

        yield r1
        r0, ratio0, numenator0 = r1, ratio1, numenator1


def gen_convergent(generator):
    p0, p1 = 0, 1
    q0, q1 = 1, 0

    while True:
        ai = next(generator)
        pi = ai * p1 + p0
        qi = ai * q1 + q0
        yield pi, qi
        p0, p1, q0, q1 = p1, pi, q1, qi


if __name__ == '__main__':
    pass
```

Оценки сложности рассмотренных алгоритмов

Сложность алгоритма p-Метод Полларда равна $O(\sqrt[4]{n})$.

Сложность алгоритма $(p-1)$-Метод Полларда равна $O(P \log P \, (\log n)^2)$.

Временная сложность алгоритма Брилхарта, равна $O(\exp(2\sqrt{\ln n \ln \ln n}))$.

Результаты тестирования программ

В ходе работы были рассмотрены следующие методы разложения целых числе на множители: метод непрерывных дробей, p-метод Полларда, $(p-1)$-метод Полларда. Так как в общем случае проверка числа на простоту является менее трудоемкой, чем любой из алгоритмов разложения числа на множители, то имеет смысл каждое число перед разложением на множители проверить на простоту.

Пример работы алгоритмов (анализ корректности)

	Делитель		
Число	p-Метод Полларда	$(p-1)$-Метод Полларда	Метод непрерывных дробей
63169	181	181	Делитель не найден
1001	Число прошло тест на простоту!		
25414271907951613	179425457	141642509	179425457

Примеры работы алгоритмов (анализ эффективности)

Для сравнительно анализа алгоритмов Полларда целесообразно воспользоваться тестом, аналогичным одному из тестов модификаций алгоритма Евклида.

Для фиксированного числа оцениваем время работы алгоритмов (на 1000 запусков). При такой оценки многое зависит как от вида числа, так и от свойств выбранного отображения. В данном случае оно представляло собой функцию $x^2+1\ (mod\ n)$. Эту функцию достаточно легко вычислить. Очевидно, такая реализация p-метода Полларда будет выигрывать по сравнению с реализацией $(p-1)$-метода хотя бы потому, что последняя требует (помимо операций умножения и деления) вычисления экспоненты и трехкратного вычисления логарифма на каждой итерации. Действительно, p-

метод Полларда превосходит $(p-1)$-метод более чем в 10 раз для числа любого вида.

	p-метод Полларда		$(p-1)$-Метод Полларда	
Число	Делитель	Время	Делитель	Время
11021	103	0.264	Не найден	0.523
63169	181	0.782	181	1.103

Несмотря на эффективность, p-метод Полларда имеет существенный недостаток: если случайное отображение фиксировано, то для некоторых чисел он не работает. В этом случае требуется поменять отображение или воспользоваться иным методом разложения на множители.