МИНОБРНАУКИ РОССИИ
ФГБОУ ВО «СГУ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ДИСКРЕТНОЕ ЛОГАРИФМИРОВАНИЕ В КОНЕЧНОМ ПОЛЕ

ЛАБОРАТОРНАЯ РАБОТА

студента 5 курса 531 группы
направления 100501 — Компьютерная безопасность
факультета КНиИТ
Улитина Ивана Владимировича

Проверил
профессор ____________ В. А. Молчанов

Саратов 2023

1 Постановка задачи

Цель работы - изучение основных методов дискретного логарифмирования в конечном поле и их программная реализация.

Порядок выполнения работы:

1. Рассмотреть метод Гельфонда-Шенкса вычисления дискретного логарифма и привести его программную реализацию;
2. Рассмотреть (ρ)-метод Полларда вычисления дискретного логарифма и привести его программную реализацию;
3. Рассмотреть метод вычисления дискретного логарифма в конечных полях;

2 Теоретические сведения

2.1 Дискретный логарифма

Пусть $G = \langle a \rangle$ — конечная циклическая группа порядка m, т.е.

$$G = \{a^0 = 1, a^1 = a, a^2, \ldots, a^{m-1}\}.$$

Определение: Дискретным логарифмом элемента $b \in G$ называется число $x \in \{0, 1, \ldots, m-1\}$, для которого

$$a^x = b.$$

Обозначение $x = \log_a b$.

Задача нахождения дискретного логарифма имеет большую сложность вычислений.

Алгоритм Гельфонда-Шенкса вычисления дискретного логарифма осуществляется в произвольной циклической группе, элементы которой линейно упорядоченны.

2.2 ρ-метод Полларда

Дана конечная циклическая группа $G = \langle a \rangle$ порядка m и элемент $b \in G$. Причем группа разбита на три примерно равные части U_1, U_2, U_3 с простым алгоритмом проверки вхождения элементов в эти части.

Определяется преобразование $f : G \to G$ для элементов $x \in G$ по формуле:

$$f(x) = \begin{cases} bx, \text{ если } x \in U_1, \\ x^2, \text{ если } x \in U_2, \\ ax, \text{ если } x \in U_3. \end{cases}$$

Для случайно выбранного значения $s \in Z_m$ рассматривается рекуррентная последовательность:

$$y_i = f(y_{i-1}), i \geq 1, y_0 = a^s.$$

Тогда $y_i = a^{\alpha_i} b^{\beta_i}$ для рекуррентно заданных последовательностей:

$$\alpha_0 = s,\ \alpha_{i+1} = \begin{cases} \alpha_i \pmod m, \text{ если } y_i \in U_1, \\ 2\alpha_i \pmod m, \text{ если } y_i \in U_2, \\ \alpha_i + 1 \pmod m, \text{ если } y_i \in U_3; \end{cases}$$

$$\beta_0 = 0,\ \beta_{i+1} = \begin{cases} \beta_i + 1 \pmod m, \text{ если } y_i \in U_1, \\ 2\beta_i \pmod m, \text{ если } y_i \in U_2, \\ \beta_i \pmod m, \text{ если } y_i \in U_3. \end{cases}$$

Так как при этом

$$y_i = a^{\alpha_i} b^{\beta_i} = a^{\alpha_i} (a^x)^{\beta_i} = a^{\alpha_i + \beta_i x},$$

то выполняется

$$\log_a y_i = \beta_i x + \alpha_i \pmod m.$$

2.3 Индекс-метод дискретного логарифмирования в конечном простом поле

Если $h = p_1^{k_1} \dots p_r^{k_r}$, то $\log_a h = k_1 \log_a p_1 + \dots + k_r \log_a p_r$.

Даны a — образующий элемент группы $GF(p)^*$ и $h \in GF(p)^*$. Требуется найти $x = \log_a h$. Будем считать, что $GF(p) = Z_p$. Пусть B — некоторое натуральное число - параметр метода, который определяет факторную базу $S_B = \{2, 3, 5, \dots, q\}$ — множество первых простых чисел, не превосходящих B, $|S_B| = \pi(B)$. Значение параметра B выбирается таким образом, чтобы минимизировать сложность алгоритма.

Для всех $1 \leq i \leq \pi(B)$ обозначим $x_i = \log_a q_i \in Z_{p-1}$, где $q_i \in S_B$.

Алгоритм аналогичен субэкспоненциальным алгоритмам факторизации, но соответствующие алгоритмы дискретного логарифмирования обладают одной особенностью – эти алгоритмы условно можно разделить на два этапа:

1. сначала по заданному p надо выбрать факторную базу и определить логарифмы элементов факторной базы (при фиксированном p этот этап надо проделать только один раз);
2. на втором этапе с использованием известных логарифмов элементов факторной базы по заданному $h \in Z_p^*$ необходимо найти $\log_a h$ (этот этап может производиться неоднократно).

3 Результаты работы

3.1 Описание и псевдокод алгоритмов факторизации целых чисел

Алгоритм 1 - Метод Гельфонда-Шенкса

Вход: Конечная линейно упорядоченная группа $G = \langle a \rangle$, верхняя оценка порядка группы $|G| \leq B$ и $b \in G$.

Выход: $x = \log_a b$.

Шаг 1. Вычислить $r = [\sqrt{B}] + 1$. Вычислить элементы $a, a^2, \ldots, a^{r-1}$ и упорядочить по второй координате множество пар (k, a^k), $1 \leq k \leq r - 1$.

Шаг 2. Вычислить $a_1 = a^{-r}$. Для каждого $0 \leq i \leq r - 1$ вычислить a_1^i и проверить, является ли элемент $a_1^i b$ второй координатой какой-нибудь пары из упорядоченного множества, построенного на шаге 1. Если $a_1^i b = a^k$, то $a^{-ri}b = a^k, b = a^k a^{ri} = a^{k+ri}$ запомнить $k + ri$.

Шаг 3. Найти число x, равное наименьшему значению среди чисел $k + ri$, вычисленных на предыдущем шаге. В результате получаем $x = \log_a b$.

Псевдокод:

```
Функция Полларда_Ро(число, эпсилон):
    Если n == 1:
        Вернуть 1
    Если n четное:
        Вернуть 2

    Пусть T = вычислить_T_(эпсилон)
    Пусть rng - генератор случайных чисел
    Пусть x = случайное_Число_Из_Диапазона(2..n)
    Пусть y = x
    Пусть d = 1

    Функция f(x):
        Вернуть (x * x + 1) % n

    Пока d == 1 и количество_повторений < T:
        x = f(x)
        y = f(f(y))
        Если x > y:
            d = НОД(x - y, n)
        Иначе:
            d = НОД(y - x, n)
```

```
            T = T - 1

        Вернуть d
```

Трудоемкость алгоритма $O(\sqrt{B}\log B)$.

Алгоритм 2 - ρ-метод Полларда вычисления дискретного логарифма в прооизвольной циклической группе

Вход: конечная группа $G = \langle a \rangle$ порядка m, элемент $b \in G$, определенная выше функция $f : G \to G$ и число $\varepsilon > 0$.

Выход: $x = \log_a b$ с вероятностью не менее $1 - \varepsilon$.

Шаг 1. Вычислить $k = [\sqrt{2\sqrt{m}\ln\frac{1}{\varepsilon}}] + 1$.

Шаг 2. Положить $i = 1$, выбрать случайное значение $s \in Z_m$ и вычислить $y_0 = a^s, y_1 = f(y_0)$. Запомнить две тройки $(y_0, \alpha_0, \beta_0), (y_1, \alpha_1, \beta_1)$ и перейти к шагу 3.

Шаг 3. Положить $i = i+1$, вычислить $y_i = f(y_{i-1}), y_{2i} = f(f(y_{2i-2}))$, запомнить две тройки $(y_i, \alpha_i, \beta_i), (y_{2i}, \alpha_{2i}, \beta_{2i})$ и перейти к шагу 4.

Шаг 4. Если $y_i \neq y_{2i}$, то проверить условие $i < k$. Если это условие выполнено, то перейти к шагу 3. Иначе закончить вычисления и сообщить, что значение $x = \log_a b$ вычислить не удалось. Если же $y_i = y_{2i}$, то

$$\log_a y_i = \beta_i x + \alpha_i = \log_a y_{2i} = \beta_{2i} x + \alpha_{2i} \pmod{m},$$

$$\alpha_{2i} - \alpha_i \equiv (\beta_i - \beta_{2i})x \pmod{m}.$$

и для решения сравнения перейти к шагу 5.

Шаг 5. Вычислить НОД$(\beta_i - \beta_{2i}, m) = d$. Если $\sqrt{m} < d < m$, то перейти на шаг 2 и выбрать новое значение $s \in Z_m$. В противном случае решить сравнение $\alpha_{2i} - \alpha_i \equiv (\beta_i - \beta_{2i})x \pmod{m}$. Если $d = 1$, то единственное решение последнего сравнения равно значению $\log_a b$. Если $1 < d \leq \sqrt{m}$, то последнее сравнение имеет d различных решений по модулю m. Для каждого из этих решений проверить выполнимость равенства $a^x = b$ и найти истинное значение $x = \log_a b$.

Псевдокод:

```
функция Полларда_Ро_минус_1(число, база):
    Пусть rng - генератор случайных чисел
```

```
Пусть a = случайное_Число_Из_Диапазона(2..n)

Пусть power = база
Пусть d = 1
Пусть верхняя_Граница = 18446744073709551615 / 1000

Пока d == 1:
    power *= 2
    Пусть a_powered = в_степень_по_модулю(a, power, n)
    d = НОД(a_powered - 1, n)

    Если power > верхняяГраница:
        Если база == 2:
            Вернуть n
        Иначе:
            база = база - 1
            power = база

    Если d != 1:
        Прервать цикл

Вернуть d
```

Трудоемкость алгоритма: $O(\sqrt{m}\sqrt{\ln\frac{1}{\varepsilon}})$ операций в группе G.

<u>Алгоритм 3 - индекс-метод логарифмирования в конечном простом поле</u>

Вход: простое нечетное число p, $Z_p^* = \langle a \rangle, h \in Z_p^*$.

Выход: значение $x = \log_a h$.

<u>Шаг 1.</u> Выбрать значение параметра B. Построить факторную базу S_B.

<u>Шаг 2.</u> Выбрать случайное m, $0 \le m \le p-2$, найти вычет $b \in Z_p^*$, $b \equiv a^m \pmod p$.

<u>Шаг 3.</u> Проверить число b на B-гладкость. Если b является B-гладким, то вычислить его каноническое разложение $b = \prod_{i=1}^{\pi(B)} q_i^{l_i}$ и запомнить строку $(l_1, l_2, \ldots, l_{\pi(B)})$.

Из соотношений

$$\begin{cases} b = \prod_{i=1}^{\pi(B)} q_i^{l_i} = a^{\sum_{i=1}^{\pi(B)} l_i \log_a q_i} \pmod p \\ b \equiv a^m \pmod p \end{cases}$$

вытекает сравнение

$$m \equiv \sum_{i=1}^{\pi(B)} l_i x_i \pmod{(p-1)},$$

где $x_i = \log_a q_i$.

Повторять шаги 2 и 3 до тех пор, пока число найденных строк не превысит $N = \pi(B) + \delta$, где δ — некоторая небольшая константа. В результате будет построена система линейных уравнений над кольцом Z_{p-1} относительно неизвестных $x_i = \log_a q_i$, $q_i \in S_B$:

$$m_j \equiv \sum_{i=1}^{\pi(B)} l_{ji} x_i \pmod{(p-1)},\ 1 \le j \le N.$$

Полученная система заведомо совместна.

<u>Шаг 4.</u> Решить полученную на предыдущем шаге систему линейных уравнений над кольцом Z_{p-1}. Если система имеет более одного решения, то вернуться на шаг 2 и получить несколько новых линейных соотношений. Затем вернуться к шагу 4.

<u>Шаг 5.</u> (Вычисление индивидуального логарифма.) Выбрать случайное m, $0 \le m \le p-2$, найти вычет $b \equiv ha^m \pmod{p}$, $b \in Z_p^*$. Проверить число b на B-гладкость. Если b является B-гладким, то

$$\begin{cases} b = \prod_{i=1}^{\pi(B)} q_i^{r_i} = a^{\sum_{i=1}^{\pi(B)} r_i \log_a q_i} \pmod{p} \\ b \equiv ha^m = a^x a^m = a^{x+m} \pmod{p} \end{cases}$$

и, следовательно,

$$x \equiv -m + \sum_{i=1}^{\pi(B)} r_i x_i \pmod{(p-1)},\ \text{где } x_i = \log_a q_i.$$

<u>Псевдокод:</u>

```
Функция Бриллхарт_Моррисон_метод(n):
    Пусть max_iterations = 100
    Пусть result = Пустой_Список
    Пусть cf_expansion = непрерывная_дробь_кв_корня(n)
    Для каждого i в диапазоне от 0 до max_iterations:
        Пусть a_i = cf_expansion.удалить_первый_элемент()
```

```
            Пусть gcd_result = НОД(a_i, n)
            Если gcd_result не равно 1 и gcd_result не равно n:
                result.добавить(gcd_result)

                // Если число разложено полностью
                Если n == gcd_result:
                    Вернуть result

                Пусть factor2 = n / gcd_result
                result.добавить(factor2)
                Вернуть result
        Вернуть result
```

При $p \to \infty$ оптимальное значение $B = L_p[\frac{1}{2}]$ и сложность всего алгоритма оценивается величиной $L_p[2]$, где

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

для $L = \exp\left((\log p \log\log p)^{\frac{1}{2}}\right)$.

3.2 Код программы, реализующей рассмотренные алгоритмы

```
use std::io;
use std::num;
// use rand::Rng;
use std::f64;
// use quadratic::jacobi;
// use rand::distributions::Uniform;


fn read_integer() -> i128 {
    let mut n = String::new();
    io::stdin()
        .read_line(&mut n)
        .expect("failed to read input.");
    let n: i128 = n.trim().parse().expect("invalid input");
    n
}
```

```


fn read_float() -> f32 {
    let mut num:f32=0.0;
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Not a valid string");
    num = input.trim().parse().expect("Not a valid number");
    num
}


fn euclid_gcd_extended(a: i128, b: i128, _x: i128, _y: i128) -> (i128, i128, i128, i128) {
    if a == 0 {
        return (a, b, 0, 1);
    }
    else {
        let (_, d, x1, y1) = euclid_gcd_extended(b % a, a, 0, 0);
        let division = b / a;
        let x = y1 - division * x1;
        let y = x1;
        return (0, d, x, y)
    }
}


fn continued_fraction(mut a: i128, mut b: i128) -> Vec<i128> {
    let mut fraction: Vec<i128> = Vec::new();
    while b != 0 {
        fraction.push(a / b);
        let c = a;
        a = b;
        b = c % b;
    }
    fraction
}


fn print_array(arr: Vec<i128>) {
    print!("[");
    for i in 0..arr.len() {
```

```
        print!("{}", arr[i]);
        if i < arr.len() - 1 {
            print!(", ");
        }
    }
    println!("]");
}



fn mod_inverse(a: i128, n: i128) -> i128 {
    let fraction: Vec<i128> = Vec::new();
    let (_, g, x, _) = euclid_gcd_extended(a, n, 0, 0);
    if g != 1 {
        return -1;
    } else {
        let result = (x % n + n) % n;
        return result;
    }
}


fn is_prime(n: i128) -> bool {
    if n <= 1 {
        return false;
    }
    if n == 2 || n == 3 {
        return true;
    }
    if n % 2 == 0 || n % 3 == 0 {
        return false;
    }

    let mut i = 5;
    while i * i <= n {
        if n % i == 0 || n % (i + 2) == 0 {
            return false;
        }
        i += 6;
    }

```

```
    true
}


fn power_mod(base: u64, exponent: u64, modulus: u64) -> u64 {
    if modulus == 1 {
        return 0;
    }
    let mut result = 1;
    let mut base = base.rem_euclid(modulus);
    let mut exp = exponent;

    while exp > 0 {
        if exp % 2 == 1 {
            result = (result * base).rem_euclid(modulus);
        }
        exp = exp >> 1;
        base = (base * base).rem_euclid(modulus);
    }

    result.rem_euclid(modulus)
}


fn jacobi_symbol(mut a: i128, mut n: i128) -> i128 {
    let mut t = 1;
    while a != 0 {
        while a % 2 == 0 {
            a /= 2;
            let n_mod_8 = n % 8;
            if n_mod_8 == 3 || n_mod_8 == 5 {
                t = -t;
            }
        }

        std::mem::swap(&mut a, &mut n);
        if a % 4 == 3 && n % 4 == 3 {
            t = -t;
        }

        a %= n;
```

```
    }

    if n == 1 {
        return t;
    } else {
        return 0;
    }
}


// fn baby_step_giant_step() -> () {
//     println!("Введите a: ");
//     let mut a = read_integer();
//     println!("Введите B: ");
//     let mut p = read_integer() as f64;
//     println!("Введите b: ");
//     let mut b = read_integer() as f64;

//     let big_b = p - 1.0;
//     let r = (big_b.sqrt()) as i128 + 1;
//     // println!("{}", r);
//     let mut pairs: Vec<(i128, i128)> = vec![];
//     for k in 1..r {
//         pairs.push((k, power_mod(a as u64, k as u64, p as u64) as i128))
//     }
//     pairs.sort_by(|a, b| a.1.cmp(&b.1));
//     // println!("{:?}", pairs);
//     let inv_powered_a = mod_inverse(power_mod(a as u64, r as u64, p as u64) as i128,
//                                     p as i128);
//     for i in 0..r {
//         let mut result = power_mod(inv_powered_a as u64, i as u64, p as u64) * (b as u64);
//         result = result.rem_euclid(p as u64);
//         for k in 0..(r - 1) {
//             if pairs[k as usize].1 == result as i128 {
//                 if power_mod(a as u64, (k + r * i) as u64, p as u64) == b as u64 {
//                     println!("Значение x = {}", k + r * i);
//                     return ();
//                 }
```

```
//                  else if power_mod(a as u64, (k + 1 + r * i) as u64, p as u64) == b as u64 {
//                      println!("Значение x = {}", k + 1 + r * i);
//                      return ();
//                  }
//              }
//          }
//      }
//      println!("Значение a не является образующим элементом.");
// }


fn baby_step_giant_step() -> () {
    println!("Введите a: ");
    let mut alpha = read_integer() as i64;
    println!("Введите B: ");
    let mut n = read_integer() as i64;
    println!("Введите b: ");
    let mut beta = read_integer() as i64;

    let m = (n as f64).sqrt() as i64 + 1;

    let alpha_inv_m = power_mod(alpha as u64, (n - m - 1) as u64, n as u64);

    let mut hash_table = std::collections::HashMap::new();

    for j in 0..m {
        let value = power_mod(alpha as u64, j as u64, n as u64);
        hash_table.insert(value, j);
    }

    for i in 0..m {
        let value = (beta * (power_mod(alpha_inv_m as u64, i as u64, n as u64)) as i64).rem_euclid(n);

        if let Some(j) = hash_table.get(&(value as u64)) {
            let mut x = i * m + *j;
            if x != 0
            {
                println!("Значение x = {}", i * m + *j);
                return ()
```

```
            }
        }
    }

    println!("Значение a не является образующим элементом.");
    return ()
}


fn rho_method() -> () {
    println!("Введите a: ");
    let mut a = read_integer();
    println!("Введите m: ");
    let mut p = read_integer() as f64;
    println!("Введите b: ");
    let mut b = read_integer() as f64;


}


fn main() {
    println!("Выберите метод разложения:");
    println!("1. метод Гельфонда-Шенкса");
    println!("2. р-метод Полларда");
    println!("3. Индекс-метод");

    let chosen_method = read_integer();
    match chosen_method {
        1 => baby_step_giant_step(),
        2 => rho_method(),
        _ => println!("Введено неверное число!"),
    }

}
```

3.3 Результаты тестирования программ

ЗАКЛЮЧЕНИЕ

В данной лабораторной работе были изучены теоретические сведения о методах дискретного логарифмирования (метод Гельфонда-Шенкса, ρ-метод

Полларда, метод вычисления дискретного логарифма в конечных полях). На их основе были рассмотрены соответствующие алгоритмы. Была произведена оценка сложности созданных алгоритмов. Они послужили фундаментом для программной реализации, которая впоследствии успешно прошла тестирование, результаты которого были прикреплены к отчету вместе с листингом программы, написанной на языке Rust с использованием стандартных библиотек языка.