Отчёт по лабораторной работе №7.

Дискретное логарифмирование в конечном поле

Коне Сирики

20 декабря 2024

Российский университет дружбы народов, Москва, Россия

Объединённый институт ядерных исследований, Дубна, Россия

Информация

Докладчик

- Коне Сирики
- Студент физмат
- профессор кафедры прикладной информатики и теории вероятностей
- Российский университет дружбы народов
- konesirisil@yandex.ru
- https://github.com/skone19

Цели и задачи работы

Целью данной лабораторной работы является краткое ознакомление с задачей дискретного логарифмирования и ρ-методом Полларда для её решения, а также его последующая программная реализация.

Задачи: Рассмотреть и реализовать на языке программирования Python ρ-метод Полларда для задачи дискретного логарифмирования.

Теоретическое введение

Для конечного поля $\mathbb{F}_p$ (в частности, в простейшем и важнейшем случае $\mathbb{Z}_p^*$, где p – большое простое число) *задача дискретного логарифмирования* определяется следующим образом: при заданных ненулевых $a, b \in \mathbb{F}_p$ найти такое целое x, что:

$$a^x \equiv b \in \mathbb{F}_p, \text{ или } a^x \equiv b \pmod{p}.$$

Пусть число a также имеет порядок r, то есть $a^r \equiv 1 \pmod{p}$.

ρ-метод Полларда для дискретного логарифмирования

Идея ρ-метода Полларда, как и аналогичного метода факторизации, в построении последовательности итеративных значений функции f, в которой требуется найти цикл.

Для этого, как и ранее, используем алгоритм “черепахи и зайца” Флойда: к одному значению, c, на каждом шаге будем применять функции единожды, к другому, d, – дважды, пока их значения не совпадут и мы не сможем их приравнять.

ρ-метод Полларда. Шаг 1

- Зададим начальные значения c и d. Пусть $c = d \equiv a^{u_0} b^{v_0} \pmod{p}$, где u_0, v_0 случайные целые числа.
- Поскольку по условию $b \equiv a^x \pmod{p}$, можно записать $c \equiv a^{u_0} (a^x)^{v_0} \pmod{p} \equiv a^{u_0 + v_0 x} \pmod{p}$. Тогда $\log_a c \pmod{p} = u_0 + v_0 x$.
- Таким образом, логарифмы c и d по основанию a могут быть представлены линейно.

Зададим отображение f, обладающее а) сжимающими свойствами и б) вычислимостью логарифма.

$$f(c) = \begin{cases} ac, & \text{при } c < \frac{p}{2} \\ bc, & \text{при } c \geq \frac{p}{2} \end{cases}$$

1) $f(c) \equiv (a^u b^v)a \pmod p \equiv\ a^{(u+1)+vx} \pmod p$, и тогда $\log_a f(c) = (u+1) + vx = \log_a c + 1$.

2) $f(c) \equiv (a^u b^v)b \pmod p \equiv\ a^{u+(v+1)x} \pmod p$, и тогда $\log_a f(c) = u + (v+1)x = \log_a c + x$.

ρ-метод Полларда. Шаг 2

- Выполнять $c \leftarrow f(c) \pmod p, d \leftarrow f(f(d)) \pmod p$, вычисляя при этом $\log_a c$ и $\log_a d$ как линейные функции от x по модулю r, пока не получим равенство $c \equiv d \pmod p$.

Приравниваем линейные представления логарифмов и получаем: $u_i^c + v_i^c x \equiv u_i^d + v_i^d x \pmod{r}$.

Приведем подобные слагаемые: $vx \equiv u \pmod{r}$, где $v = v_i^c - v_i^d$, $u = u_i^d - u_i^c$.

Чтобы решить такое сравнение, нужно найти *обратный элемент* v^{-1} *по модулю* r и умножить на него левую и правую части сравнения: $x \equiv uv^{-1} \pmod{r}$.

Обратный элемент будет существовать, если НОД$(v, r) = 1$. В этом случае для его поиска можно воспользоваться *расширенным алгоритмом Евклида*.

С помощью расширенного алгоритма Евклида получим линейное представление НОД, равного единице: $e_v v + e_r r = 1$, что эквивалентно $e_v v - 1 = -e_r r$, или $(e_v v - 1) \mid r$, или $e_v v \equiv 1 \pmod r$. Таким образом $v^{-1} = e_v$.

Если же НОД не равен единице, то мы предполагаем, что $gcd =$ НОД$(v, r) =$ НОД(v, u, r). Тогда сравнение можно поделить на gcd, и получим $\frac{v}{gcd} x \equiv \frac{u}{gcd} \pmod{\frac{r}{gcd}}$.

Ход выполнения и результаты

Реализация (1 / 4)

```python
import math; import numpy as np
def multiplicative_order(a, n):
    k = 1; flag = True # начнем перебор с единицы
    while flag:
        if (a ** k - 1) % n == 0: flag = False
        else: k += 1
    return k
```

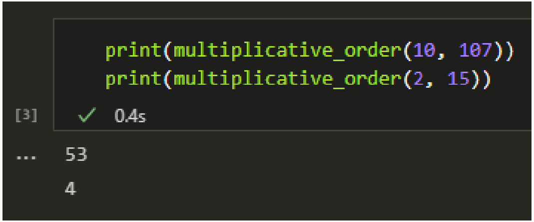

Рис. 1: Примеры нахождения порядка числа a по модулю n

```
def euclidean_algorithm_extended(a, b): <...> return (d, x_r, y_r)

def solve_congruence(c, d, p):
    (k_1, b_1) = c; (k_2, b_2) = d # получаем коэффициенты
    k = k_1 - k_2; b = b_2 - b_1 # kx = b (mod p)
    (gcd, k_inverse, _) = euclidean_algorithm_extended(k, p)
    if gcd == 1: # если k и p - взаимно простые..
        return (b * k_inverse) % p
    else: # иначе
        k = int(k / gcd); b = int(b / gcd) # делим сравнение на gcd
        (_,k_inverse,_) = euclidean_algorithm_extended(k, int(p/gcd))

        return (b * k_inverse) % p
```

Реализация (3 / 4)

```python
def pollard_rho_dlog(a, b, p, def0 = True, to_print = False):
    r = multiplicative_order(a, p) # порядок числа a
    half_p = math.floor(p / 2) # p / 2
    f = "({a} * x % {p}) if x < {half} else ({b} * x % {p})"
                   .format(a = a, p = p, half = half_p, b = b)
    (u, v) = (2, 2) if def0 else (np.random.randint(1, half_p),
                                  np.random.randint(1, half_p))
    c = ((a ** u) * (b ** v)) % p    #
    d = c                             # шаг 1
    (k_c, l_c) = (k_d, l_d) = (u, v) #

    if to_print: <...>
      print("{:^10} | {:^3} + {:^3}x | {:^10} | {:^3} + {:^3}x"
                               .format(c, l_c, k_c, d, l_d, k_d))
```

```python
while True:
    x = c
    if x < half_p: l_c += 1
    else: k_c += 1
    c = eval(f); x = d
    if x < half_p: l_d += 1
    else: k_d += 1
    x = eval(f)
    if x < half_p: l_d += 1
    else: k_d += 1
    d = eval(f) <...>
    if c == d: # шаг 3
        result = solve_congruence((k_c, l_c), (k_d, l_d), r)
        if (a ** result - b) % p == 0: return result
```

Результаты (1 / 2)

```
pollard_rho_dlog(10, 64, 107, True, True)
```

[6] ✓ 0.6s

```
     c     |   log_c   |     d     |   log_d
---------- | --------- | --------- | ----------
     4     | 2  +  2 x |     4     |  2  +  2 x
     40    | 3  +  2 x |     79    |  4  +  2 x
     79    | 4  +  2 x |     56    |  5  +  3 x
     27    | 4  +  3 x |     75    |  5  +  5 x
     56    | 5  +  3 x |     3     |  5  +  7 x
     53    | 5  +  4 x |     86    |  7  +  7 x
     75    | 5  +  5 x |     42    |  8  +  8 x
     92    | 5  +  6 x |     23    |  9  +  9 x
     3     | 5  +  7 x |     53    | 11  +  9 x
     30    | 6  +  7 x |     92    | 11  + 11 x
     86    | 7  +  7 x |     30    | 12  + 12 x
     47    | 7  +  8 x |     47    | 13  + 13 x

20
```

Результаты (2 / 2)

```
[10]  pollard_rho_dlog(5, 3, 23, False, True)
      ✓ 0.3s

...   (u, v) = (7, 3)
```

c	log_c	d	log_d
22	3 + 7 x	22	3 + 7 x
20	3 + 8 x	14	3 + 9 x
14	3 + 9 x	11	3 + 11 x
19	3 + 10 x	4	4 + 12 x
11	3 + 11 x	14	5 + 13 x
10	3 + 12 x	11	5 + 15 x
4	4 + 12 x	4	6 + 16 x

```
      16

[11]  pollard_rho_dlog(29, 479, 797, True, False)
      ✓ 0.8s

...   3
```

Заключение

Таким образом, была достигнута цель, поставленная в начале лабораторной работы: было проведено краткое знакомство с задачей дискретного логарифмирования и с алгоритмом, реализующим ρ-метод Полларда для её решения, после чего алгоритм был успешно реализован на языке программирования **Python**.

Спасибо за внимание