Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики, кафедра
Дослідження Операцій

Курсова робота на тему:

Основні алгоритми теорії чисел та криптографії

Виконала:
студентка групи ДО-3
Захарченко Анастасія Ігорівна

Керівник:
професор, доктор фізико-математичних наук
Маринич Олександр Віталійович

Київ 2019

Зміст

0 Вступ

Метою роботи було розглянути декілька алгоритмів з теорії чисел, які знаходять своє застосування в тому числі в криптографії.
Кожен із зазначених алгоритмів реалізувати на мові програмування python.

При побудові сучасних криптосистем потрібні дуже великі прості числа. Наприклад, в схемі шифрування з відкритим ключем, відомій під назвою RSA, яка використовується у величезній кількості сучасних протоколів захисту інформації таких, як PGP, S=MIME, і різних системах, заснованих на завданні дискретного логарифмування в кінцевих полях, використовуються «випадкові» прості числа, записи яких в десятковій системі числення складаються з сотень цифр.

Криптографічна стійкість системи RSA базується на складності вирішення проблеми RSA, яка, в свою в чергу, базується на складності проблеми факторизації великих чисел.
В ході роботи ми розглянемо одні з найбільш швидких алгоритмів пошуку розв'язків таких задач. Ми розглянемо наступні алгоритми:

1. ρ-алгоритм Полларда факторизації довгих цілих чисел.
2. ρ-алгоритм Полларда знаходження дискретного логарифма.
3. Обчислення функцій Ейлера та Мьобіуса.
4. Обчислення символів Лежандра та Якобі.
5. Алгоритм Чіполи знаходження дискретного квадратного кореня.
6. Один алгоритм перевірки чисел на простоту на вибір: алгоритм Соловея-Штрассена або алгоритм Міллера-Рабіна.

1 Алгоритм факторизації довгих цілих чисел: ρ-алгоритм Полларда

Метод був розроблений Джоном Поллардом в 1975 р. Нехай n – число, яке потрібно розкласти наступним чином:

1. Обираємо невелике число x_0 та будуємо послідовність $\{x_n\}$, $n = 0, 1, 2, \ldots$, визначаючи кожне наступне x_{n+1} за формулою $x_{n+1} = (x_n^2 - 1) \pmod n$
2. Одночасно на кожному кроці рахуємо Н.С.Д. d числа n та можливих різниць $|x_i - x_j|$, де $j < i$.
3. Коли $d = \gcd(n, |x_i - x_j|)$, відмінний від 1, рахування припиняємо. Знайдене d є дільником n. Якщо n/d не є простим числом, продовжуємо рахувати, взявши замість n число n/d.

Замість функції $F(x) = (x^2 - 1) \pmod n$ в підрахунку x_{n+1} можна взяти інший довільний многочлен 2-го степеня.

Недоліком цього варіанту методу є необхідність зберігання великого числа попередніх значень x_j. Якщо $(x_j - x_i) \equiv 0 \pmod p$, то $(f(x_j) - f(x_i)) \equiv 0 \pmod p$. Тому, якщо пара (x_i, x_j) дає нам розв'язок, то розв'язок дасть будь-яка пара (x_{i+k}, x_{j+k}). Тому нема необхідності перевіряти всі пари, можна лише обмежитись тими, де $j = 2^k$, та $k = 1, 2, 3, \ldots$, а $i \in [2^k + 1; 2^{k+1}]$.

У варіанті алгоритму написання коду в додатку в пам'яті зберігаємо лише 3 змінні: n, x, y.

```
def _rho_pollard(number):
    answer = set()
    for d in range(2, min(int(sqrt(sqrt(number))) + 10, number)):
        if number % d == 0:
            answer.add(d)
        while number % d == 0:
            number //= d

    if number == 1:
        return answer

    x_fixed = 2
    cycle_size = 2
```

```
    x = 2
    factor = 1

    while factor == 1:
        count = 1
        while count <= cycle_size and factor <= 1:
            x = (x * x + 1) % number
            factor = gcd(x - x_fixed, number)
            count += 1
        cycle_size *= 2
        x_fixed = x

    if factor == number:
        return {factor} | answer
    else:
        return {factor} | rho_pollard(number // factor) | answer


def rho_pollard(number):
    factors = _rho_pollard(number)

    factor_powers = {}

    for factor in factors:
        while number % factor == 0:
            factor_powers[factor] = factor_powers.get(factor, 0) + 1
            number //= factor

    sl = []
    for factor, power in factor_powers.items():
        sl.append(f{factor}^{power})
    print (factors)
    return factor_powers,  * .join(sl)
```

2 Алгоритм знаходження дискретного логарифма на вибір: ρ-алгоритм Полларда

2.1 Розширений алгоритм Евкліда

Розширений алгоритм Евкліда використовується у багатьох криптографічних і теоретико-числових алгоритмах. Він складається з двох частин. У першій частині алгоритму для заданих чисел A і B обчислюється їхній найбільший спільний дільник (eng. *greatest common divisor*) d. Обчислення Н.С.Д. чисел A і B відбувається за рекурентною формулою:

$$\gcd(A, B) = \gcd(B, A\%B), \tag{1}$$

де $A\%B$ позначає операцію обчислення залишку при діленні A на B. Виконується послідовне використання цієї формули, допоки залишок від ділення першого операнду на другий не стане рівним 0. Останнє ненульове значення другого операнду і є шуканий найбільший спільний дільник.

Для розв'язування рівнянь вигляду $Ax + By = d$, де A, B — задані числа, а d — їхній найбільший спільний дільник, використовується *розширений* алгоритм Евкліда. Перша часина розширеного алгоритму Евкліда, у результаті якої ми знаходимо найбільший спільний дільник d виконується так же, як описано вище. Значення A, B, а також цілу частину від ділення A на B зберігаються у таблиці, що містить 4 стовпчика.

У другій частині роботи алгоритму до таблиці додаються два нових стовпчики, xі y. Помістимо у останній рядок цих стовпчиків значення 0 і 1. Далі, вважаючи значення x_{i+1} і y_{i+1} відомими, послідовно обчислюємо значення x_i, y_i, $i \geq 0$ за формулами:

$$x_i = y_{i+1}, \quad y_i = x_{i+1} - y_{i+1} \cdot \left\lfloor \frac{A}{B} \right\rfloor_i, \tag{2}$$

де $\lfloor A/B \rfloor_i$ — значення у i-ому рядку стовпчика цілих частин від ділення A на B.

Зауважимо, що мова програмування python дозволяє виконати усі ці обчислення доволі ефективно:

```
def extended_euclid(a: int, b: int) -> Tuple[int, int, int]:
    if a == 0:
        return b, 0, 1
    g, x1, y1 = extended_euclid(b % a, a)
    return g, y1 - b // a * x1, x1
```

2.2 ρ-алгоритм Полларда знаходження дискретного логарифма

Проблема дискретного логарифма (Discrete Logarithm Problem DLP) полягає в обчисленні утворюючої g для довільного елемента t найменшого числа k такого, що $g^k = t$. Хоча ця проблема не пов'язана безпосередньо з проблемою факторизації цілих чисел, вона грає важливу роль в криптографії. При довжині ключа L проблема DLP має таку ж складність розв'язання, як і проблема факторизації числа довжини L, тому на проблемі обчислення DLP побудовано багато криптографічних протоколів, в тому числі, відомі протоколи Діффі-Хелмана обчислення загального секретного ключа і Ель-Гамаля електронного цифрового підпису.

Існує велика кількість різних методів для розв'язання цієї задачі. Розглянемо ρ-метод Полларда для DLP. Групу по множенню поля F_p^*, p — просте число, позначимо через $F_p^* = 1, 2, \ldots, p-1$. Нагадаємо, що елемент $g \in F_p^*$ називається генератором поля, якщо кожен елемент $t \in F_p^*$ дорівнює деякому ступеню елемента $g : t = g^k$. Нехай g - (який-небудь) генератор цієї групи, і нехай t - довільний елемент F_p^*.

Для знаходження невідомого показника k такого, що $g^k = t$, будемо будувати послідовність пар $(a_i; b_i)$ чисел по модулю $p-1$ і послідовність x_i чисел по модулю p таку що $x_i = t^{a_i} g^{b_i}$. визначимо початкові значення $a_0 = b_0 = 0, x_0 = 1$. Обчислення наступних членів послідовностей будемо виконувати за наступними формулами:

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i + 1, b_i) \pmod{(p-1)}, & 0 < x_i < p/3, \\ (2a_i, 2b_i) \pmod{(p-1)}, & p/3 < x_i < 2p/3, \\ (a_i, b_i + 1) \pmod{(p-1)}, & 2p/3 < x_i < p. \end{cases} \tag{3}$$

Та, відповідно,

$$x_{i+1} = \begin{cases} t\, x_i \pmod{p}, & 0 < x_i < p/3, \\ x_i^2 \pmod{p}, & p/3 < x_i < 2p/3, \\ g x_i \pmod{p}, & 2p/3 < x_i < p. \end{cases} \tag{4}$$

Ця послідовність рахується, поки не з'являться номери i, j такі, що $x_i = x_j$. Тоді, $t^{a_i} g^{b_i} = t^{a_j} g^{b_j}$, звідки,

$$(a_j - a_i)k \equiv b_i - b_j \pmod{(p-1)}$$

Якщо $\gcd(a_j - a_i, p-1) = 1$, тоді множник k може бути знайдений з використанням узагальненого алгоритму Евкліда, розв'язав рівняння в цілих

числах

$$x(a_j - a_i) + y(p-1) = b_i - b_j$$

відносно x, y та визначаючи $k = x \pmod{(p-1)}$.

Якщо ж $\gcd(a_j - a_i; p-1) = d > 1$, тоді, рівняння, як і раніше, має розв'язок, але дає розв'язок з точністю до доданка кратного $(p-1)/d$, тобто розв'язання має вигляд $x = x_0 + m(p-1)/d$, де $m \in [0; d-1]$ — ціле число. Якщо множник d - малий, то рішення буде знайдено підстановкою чисел в рівняння $g^X \equiv t \pmod{t}$.

Так само, як в ρ-методі факторизации, в цьому алгоритмі можна використовувати модифікацію Флойда, обчислюючи на i-му кроці одночасно трійку $(a_i; b_i; x_i)$ і трійку $(a_{2i}; b_{2i}; x_{2i})$, поки не дійдемо до кроку i, на якому $x_i = x_{2i}$. У цьому варіанті знову не треба зберігати в пам'яті на кроці i всі трійки $(a_j; b_j; x_j)$ для ji, а досить зберігати дві трійки $(a_i; b_i; x_i)$ і $(a_{2i}; b_{2i}; x_{2i})$

Наведемо реалізацію описаного вище алгоритму на мові програмування python.

```
def rho_pollard_discrete_logarithm(g: int, p: int, t: int) -> int:
    a_0, b_0, x_0 = 0, 0, 1

    def step(a_i: int, b_i: int, x_i: int) -> Tuple[int, int, int]:
        if 0 <= x_i < p / 3:
            return (a_i + 1) % (p - 1), b_i % (p - 1), (t * x_i) % p
        if p / 3 <= x_i < 2 * p / 3:
            return (a_i << 1) % (p - 1), (b_i << 1) % (p - 1), (x_i * x_
        if 2 * p / 3 <= x_i < p:
            return a_i % (p - 1), (b_i + 1) % (p - 1), (g * x_i) % p

    a_i, b_i, x_i = step(a_0, b_0, x_0)
    a_2i, b_2i, x_2i = step(*step(a_0, b_0, x_0))

    while x_i != x_2i:
        a_i, b_i, x_i = step(a_i, b_i, x_i)
        a_2i, b_2i, x_2i = step(*step(a_2i, b_2i, x_2i))

    # (a_2i - a_i) * k = b_i - b_2i (mod p - 1)

    d, x, y = extended_euclid(a_2i - a_i, p - 1)

    if d == 1:
```

```
    return x % (p - 1)
else:
    x_0 = x * d
    for m in range(d):
        if bin_pow_mod(g, x_0 + m * (p - 1) // d, p) == t:
            return x_0 + m * (p - 1) // d
```

3 Обчислення функцій Ейлера та Мьобіуса

3.1 Функція Ейлера

Нехай $m \in \mathbb{N}$. Визначимо $\varphi(m)$ — число (натуральних) чисел, не більших за m та взаємно-простих з m. Іншими словами, $\varphi(m)$ є кількість цілих k, таких, що $1 \leq k \leq m$, $(k, m) = 1$. Функцію $\varphi(m)$ називають функцією Ейлера.

Властивості (функції Ейлера)**.**

1. Мультиплікативність: якщо $(a, b) = 1$, то $\varphi(ab) = \varphi(a) \cdot \varphi(b)$
2. Якщо $p \in \mathbb{P}$, а $a \in \mathbb{N}$, то $\varphi(p^a) = p^{a-1} \cdot (p - 1)$
3. Якщо $m = p_1^{\lambda_1} \cdot p_2^{\lambda_2} \dots p_n^{\lambda_n}$, то $\varphi(m) = p_1^{\lambda_1 - 1} \cdot p_2^{\lambda_2 - 1} \dots p_n^{\lambda_n - 1} \cdot (p_1 - 1) \cdot (p_2 - 1) \dots (p_n - 1)$
4. Якщо $p > 1$, а $m \in \mathbb{N}$, то $\varphi(m) = m \cdot \prod\limits_{p|m} \left(1 - \dfrac{1}{p}\right)$
5. Адитивність: $\sum\limits_{d|m} \varphi(d) = m$

Теорема (Ейлера)**.** $\forall m \in \mathbb{N}$, *і* $a \in \mathbb{N}$, *для яких* $(a, m) = 1$ *має місце конгруенція* $a^{\varphi(m)} \equiv 1 \pmod{m}$.

Для повноти опису наведемо також реалізацію функції Ейлера на мові програмування python:

```
def euler(n: int) -> int:
    phi = 1
    for d in range(2, int(sqrt(n)) + 1):
        if n % d == 0:
            n //= d
            phi *= d - 1
            while n % d == 0:
                n //= d
                phi *= d
    if n > 1:
        return phi * (n - 1)
    else:
        return phi
```

3.2 Функція Мьобіуса

Функція Мьобіуса приймає значення -1, 0 та 1, та рахується вона наступним чином:

$$\mu(n) = \begin{cases} 1, & n = 1, \\ 0, & \exists p : p^2 | n, \\ (-1)^k, & k = \#p \in \mathbb{P} : p | n. \end{cases} \tag{5}$$

Тобто $\mu(n) = 0$, якщо в розкладі числа n на прості множники хоча б одне $\alpha_k > 1$.

Властивості (функції Мьобіуса)**.**

1. Мультиплікативність: $\mu(ab) = \mu(a) \cdot \mu(b)$
2. Функція Мьобіуса пов'язана з функцією Ейлера $\varphi(n)$ таким співвідношенням:
$$\varphi(n) = \sum_{d|n} \mu(d) \cdot \frac{n}{d} \tag{6}$$
3. $\sum_{d|m} \mu(d) = \begin{cases} 1, & n = 1 \\ 0, & n > 1 \end{cases}$
4. $\sum_{n=1}^{\infty} \frac{\mu(n)}{n^z} = \prod_p \left(1 - \frac{1}{p^z}\right) = \frac{1}{\zeta(z)}$

Таким чином, функція Мьобіуса має безпосереднє відношення до дзета-функції Рімана. Зокрема, гіпотеза Рімана про нулі дзета-функції еквівалентна оцінці

$$\sum_{n \le x} \mu(n) = O\left(x^{\frac{1}{2+\epsilon}}\right),$$

за будь-якого як завгодно малого ϵ.

Для повноти опису наведемо також реалізацію функції Мьобіуса на мові програмування python:

```
def möbius(n: int) -> int:
    mu = 1
    for d in range(2, int(sqrt(n)) + 1):
        if n % d == 0:
```

```
            n //= d
            mu *= -1
            if n % d == 0:
                return 0

    if n > 1:
        return -mu
    else:
        return mu
```

4 Обчислення символів Лежандра та Якобі

4.1 Символ Лежандра

Символ Лежандра $(\frac{a}{p})$ є функція, що приймає три значення: -1, 0 та 1. Її аргументами є просте непарне число p та ціле число a.

Умова того, що a є квадратичним лишком за простим модулем p, перевіряється наступним чином:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & (\exists x) x^2 \equiv a \pmod{p} \\ -1 & !(\exists x) x^2 \equiv a \pmod{p} \\ 0 & p \mid a \end{cases}$$

Підрахунок символа Лежандра може бути виконано за допомогою наступної формули, отриманої Леонардом Ейлером:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

Властивості (символа Лежандра)**.** Нехай a, b — довільні цілі числа. Тоді

1. Якщо $a \equiv b \pmod{p}$, то $(\frac{a}{p}) = (\frac{b}{p})$
2. $(\frac{a^2}{p}) = 1$
3. $(\frac{-1}{p}) = \begin{cases} 1, & p \equiv 1 \pmod{4} \\ -1, & p \equiv 3 \pmod{4} \end{cases}$
4. $(\frac{a_1 a_1 \cdots a_n}{p}) = (\frac{a_1}{p}) \cdot (\frac{a_2}{p}) \cdots (\frac{a_n}{p})$
5. $(\frac{1}{p}) = 1$
6. (Квадратичний закон взаємності). p, q — непарні, прості, тоді

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$$

Алгоритм підрахунку символа Лежандра:

1. Використовуючи властиві властивості №6 та №1, зменшуємо "чисельник"
2. Коли a дорівнює $1, -1$ або квадрат цілого числа, завершуємо цикл

3. Рахуємо значення символа за допомогою властивостей №5, №3 або №2 (відповідно)

4.2 Символ Якобі

Символ Лежандра визначений лише для пар цілих чисел, де друге число є простим непарним. Визначення можна розповсюдити на більшу множину пар цілих чисел, тобто побудувати функцію, яка є символом Якобі.

Символ Якобі — теоретико-числова функція двох аргументів, введена Карлом Якобі в 1837.

Нехай n є простим непарним числом, та $n = p_1 \cdots p_k$ — його розклад в добуток простих чисел (не обов'язково різних), тоді визначимо для кожного числа a символ Якобі рівністю:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right)\cdots\left(\frac{a}{p_k}\right),$$

де в правій частині є добуток звичайних символів Лежандра. Якщо n є простим числом то символ Якобі дорівнює символу Лежандра.

Властивості символа Якобі

Якщо $a = b \mod n$, то $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$

$\left(\frac{a}{n}\right) = 0$ тоді і тільки тоді, коли a і n не є взаємно простими

$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$

$\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right)$

$\left(\frac{1}{n}\right) = 1$

$\left(\frac{-1}{n}\right) = 1$, якщо $n = 1 \mod 4$

$\left(\frac{-1}{n}\right) = -1$, якщо $n = 3 \mod 4$

$\left(\frac{2}{n}\right) = 1$, якщо $n = 1 \mod 8$ або $n = 7 \mod 8$

$\left(\frac{2}{n}\right) = -1$, якщо $n = 3 \mod 8$ або $n = 5 \mod 8$

Узагальнений квадратичний закон взаємності:

$$\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)(-1)^{\frac{(m-1)(n-1)}{4}}$$

В алгоритмі знаходження символа Якобі беремо найменший додатній лишок. Далі ми використовуємо мультиплікативність символа Якобі,потім рахуємо символ Якобі:

$$(\frac{2}{b}) = (-1)^{(b^2-1)/8}$$

. Замість зведення в ступіть використовуємо перевірку залишків від ділення.

Складність алгоритму дорівнює $O(\log a \cdot \log b)$.

Далі наведемо приклад реалізації символів Лежандра та Якобі, використовуючи мову програмування python.

```
def legendre(a, b):

    leg = 5
    sign = 1
    while a != -1 and sqrt(a) ** 2 != a:
        if a % b == 0: leg = 0; break
        if a < b:
            a, b = b, a
            sign *= (-1) ** ((a - 1) / 2 * (b - 1) / 2)
        if (a % b) % 2 == 0:
            a -= b * (a // b + 1)
        else:
            a %= b

    # a = 1 или -1 or sqrt is int
    if a == 1 or isinstance(sqrt(a), int):
        leg = 1
    elif a == -1:
        leg = 1 if b % 4 == 1 else -1
```

```
    return(int(leg * sign))

def jacobi(a, b):
    p = []
    jac = 1
    for factor in rho_pollard(b):
        p.append(factor)

    for p_i in p:
        jac *= legendre(a, p_i)

    return jac
```

5 Алгоритм Чіпполи знаходження дискретного квадратного кореня

Криптосистеми, такі як система RSA або Рабіна, наприклад, базуються на складності задачі факторизації та проблемі квадратичних лишків.

Дешифрування зводиться до знаходження розв'язку рівняння $c \equiv x^2 \pmod{n}$. Для пошуку пари розв'язків такого рівняння використовуємо алгоритм, описаний в даному розділі.

Алгоритм Чіпполи є алгоритмом знаходження кореня рівняння вигляду

$$x^2 \equiv n \pmod{p},$$

де $x, n \in F_p, p$ є простим числом. F_p — скінченне поле з p елементами $\{0, 1, \ldots, p-1\}$.

Алгоритм знахождення x:

Дано: $a \in \mathbb{Z}, p$ – просте
Результат: x

1. Рахуємо $(\frac{a}{p})$
2. Якщо $(\frac{a}{p}) = -1$, то розв'язків немає
3. Якщо $(\frac{a}{p}) = 1$, шукаємо таке ціле $t\,(0 \leq t \leq p-1)$, що число $v := t^2 - a$ не є квадратичним лишком, тобто $(\frac{v}{p}) = -1$
4. Потім розглядаємо скінченне поле $F : (x + y \cdot \sqrt{v})$, $x, y \in F_p$, $0 \leq x, y \leq p-1$
   Де $(x_1, y_1) \cdot (x_2, y_2) = (x_1 \cdot x_2 + y_1 \cdot y_2,\ x_1 \cdot y_2 + x_2 \cdot y_1)$
5. Рахуємо відповідь у вигляді $x = (t + \sqrt{v})^{(\frac{p+1}{2})}$ за правилами з пункта 4

Далі наведемо приклад реалізації алгоритму Чіпполи знаходження дискретного квадратного кореня, використовуючи мову програмування python.

```
def cipolla(a: int, p: int) -> str:

    if legendre(a, p) == -1:
        return Немає розв*язку
    elif legendre(a, p) == 1:
        for t in range(1, p-1):
            v = t ** 2 - a
            if legendre(v, p) == -1:
                break
        y = 1
        t1, y1 = t, y
        for power in range(1, int((p+1)/2)):
            t1, y1 = (t1 * t + y1 * y * v) % p, (t1 * y + t * y1) % p
    return fx = {t1}
```

(На виході отримаємо одну з нескінченної кількості пар $x = \pm const$)

6 Алгоритм Соловея-Штрассена перевірки чисел на простоту

Тест Соловея-Штрассена є ймовірносним тестом простоти. Він спирається на теорему Ферма та властивості символа Якобі.

Основна перевага тесту полягає в тому, що він, на відміну від тесту Ферма, розпізнає числа Кармайкла як складені.

Алгоритм Соловея — Штрассена параметризується кількістю ітерацій k.

Власне алгоритм

1. У кожній ітерації випадковим чином вибирається число $a < n$
2. Якщо $\gcd(a, n) > 1$, то виноситься рішення, що n - складене
3. Інакше перевіряється справедливість порівняння $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$
4. Якщо воно не виконується, то виноситься рішення, що n — складене.
5. Якщо це порівняння виконується, то a є свідком простоти числа n
6. Далі вибирається інше випадкове a і процедура повторюється
7. Після знаходження k свідків простоти в k ітераціях виноситься висновок, що n є простим числом з імовірністю $1 - 2^{-k}$.

Наведемо варіант реалізації даного методу:

```
def solovay_strassen(n: int, k:int) -> str:
    for i in range(1, k + 1):
        a = int(randrange(2, n - 1))
        if extended_euclid(a, n)[0] > 1:
            return "Складене"
        elif not a ** ((n-1)/2) % n == sympy.ntheory.residue_ntheory.jac
            return "Складене"
        else:
            "Просте з ймовірністю 1 - 2^{-k}"
```

7 Висновки

В ході роботи були розглянуті 7 алгоритмів, кожен з яких ми перевірили, реалізувавши на мові програмування python.

Розглянуті задачі мають велику обчислювальну складність. Один з найбільш популярних методів криптографії з відкритим ключем є метод RSA, яких базується на складності задачі факторизації та проблемі квадратичних лишків.

Дешифрування зводиться до знаходження розв'язку рівняння $c \equiv x^2 \pmod{n}$. Розв'язання обох зазначених задач було запропоновано в роботі.

Література

[1] Маринич О. В. — "Алгебраїчні структури, криптографія та захист інформації. Конспект лекцій"

[2] Ишмухаметов Ш. Т. — "Методы факторизации натуральных чисел"

[3] Нестеренко Ю. В. — "Теория чисел"

[4] Орлов В.А., Медведев Н.В., Шимко Н.А., Домрачева А.Б. — "Теория чисел в криптографии"