МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА

(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій

(повне найменування інституту, назва факультету (відділення))

Комп'ютерної алгебри та дискретної математики

(повна назва кафедри (предметної, циклової комісії))

Дипломна робота

на здобуття ступеня вищої освіти рівня «бакалавр»

(освітньо-кваліфікаційний рівень)

на тему «Проблема дискретного логарифма в криптографії»
«The problem of discrete logarithm in cryptography »

Виконав: студент денної форми навчання
напряму підготовки Комп'ютерна інженерія.

(шифр і назва напряму підготовки, спеціальності)

Дімов Едуард Олегович

(прізвище, ім'я, по-батькові)

Керівник Варбанець П.Д.

(науковий ступінь, вчене звання, прізвище та ініціали, **підпис**)

Рецензент Федоровський С.В.

(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:

Протокол засідання кафедри

№___ від «___»_____ 2020 р.

Завідувач кафедри

_____ __

(підпис) (прізвище, ініціали)

Захищено на засіданні ЕК № ___

протокол № __ від «___» ____ 2021 р.

Оцінка _____/_____/_____

(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ __

(підпис) (прізвище, ініціали)

Одеса - 2021

АНОТАЦІЯ

Дана дипломна робота посвячена проблемі дискретного логарифма в криптографії, розв’язання задачі дискретного логарифма двома методами, оцінка складності використаних алгоритмів, розв’язання конкретних задач.

В роботі розглянуто алгоритм первісного кореня, алгоритм Сільвера Поліга Хеллмана, великих і малих кроків, та розроблено відповідний інтерфейс програми для ПК. Приведені приклади роботи застосунку.

Розробка програми виконувалась з використанням мови програмування C#

ABSTRACT

This thesis is devoted to the problem of discrete logarithm in cryptography, solving the problem of discrete logarithm by two methods, estimating the complexity of the algorithms used, solving specific problems.

The paper considers the algorithm of the original root, the algorithm of Silver Polyg Hellman, large and small steps, and developed the appropriate program interface for the PC. Examples of application operation are given.

Program development was performed using the C # programming language

АННОТАЦИЯ

Данная дипломная работа посвящена проблеме дискретного логорифма в криптографии, решение задачи дискретного логарифма двумя методами, оценка сложности использованных алгоритмов, решения конкретных задач.

В работе рассмотрен алгоритм первообразного корня, алгоритм Сильвера-Полига-Хеллмана, больших и малых шагов и разработан соответствующий интерфейс программы для ПК. Приведены примеры работы приложения.

Разработка программы выполнялась с использованием языка программирования C #

ВСТУП 6
1 АЛГОРИТМ ПЕРВІСНОГО КОРЕНЯ 7
2 КІНЦЕВІ ПОЛЯ 9
1.1 Приклади таблиць 10
3 ДИСКРЕТНИЙ ЛОГАРИМ ТА ФАКТОРИЗАЦІЯ 14
3.1 Умовні позначення і історія питання 15
3.2 Як факторизувати, повертаючи дискретні логаритми 16
3.3 Підйомні рішення по модулю, принцип потужності 17
2.4 Отримання регіонального рішення 19
3.5 Зауваження 19
3 АЛГОРИТМ ВЕЛИКИХ ТА МАЛИХ КРОКІВ 21
4 АЛГОРИТМ СІЛЬВЕРА-ПОЛІГА-ХЕЛЛМАНА 24
1.1Таблиця результатів порівняльного аналізу алгоритмів 25
5 ОПИС ТА ДЕМОНСТРАЦІЯ РОБОТИ 26
ВИСНОВКИ 29
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 30
ДОДАТОК А 31

ВСТУП

Для заданих g і h розв’язок x рівняння над полем Zp $g^x = h$ називається дискретним логарифмом елементу h по основі g. У разі, коли G є мультиплікативною групою кільця по модулю m, розв’язок називають також індексом числа h по основі g. Індекс числа h по основі g гарантовано існує, якщо g є первісним коренем по модулю p.

Рішення задачі дискретного логарифмування полягає в знаходженні деякого цілого позитивного числа x, що задовольняє рівнянню$g^x = h$. Якщо воно має розв’язок, у нього має бути хоча б одно натуральне рішення, що не перевищує порядок групи. Це відразу дає грубу оцінку складності алгоритму пошуку рішень, алгоритм повного перебору знайшов би рішення за число кроків не вище за порядок цієї групи.

В даній роботі ми розглянемо: дискретний логарифм та факторизацію, алгоритм первісного кореня, розв’язання дискретного логарифма двома методами (великих і малих кроків, Сільвера — Поліга — Хеллмана) приклади та їх реалізація на програмному рівні

1 АЛГОРИТМ ПЕРВІСНОГО КОРЕНЯ

Визначення:

Первісним коренем за модулем n називається таке число g, що всі його ступеня за модулем n пробігають по всім числам, взаємно простим з n. Математично це формується таким чином: якщо g є первісним коренем за модулем n, то для будь-якого цілого a такого, що gcd(a,n) = 1 знайдеться таке ціле k, що $g^k \equiv a \pmod{n}$

Зокрема, у випадку простого n ступеня первісної кореня пробігають по всім числам від 1 до n-1.

Існування:

Первісний корінь по модулю n існує тоді і тільки тоді, коли n є або ступенем непарного простого, або подвоєною ступенем простого, а також у випадках n = 1, n = 2, n = 4.

Приклад: використовуємо два числа 1008 и 1009

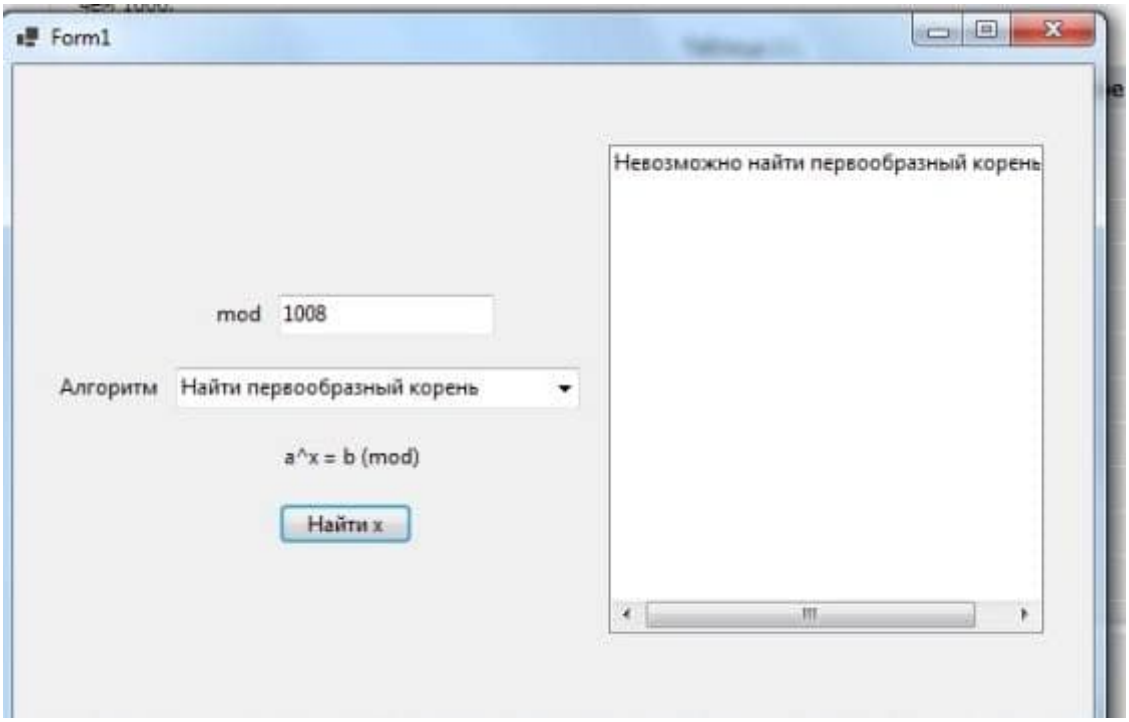

Рисунок 1 – реалізація алгоритма первісного кореня

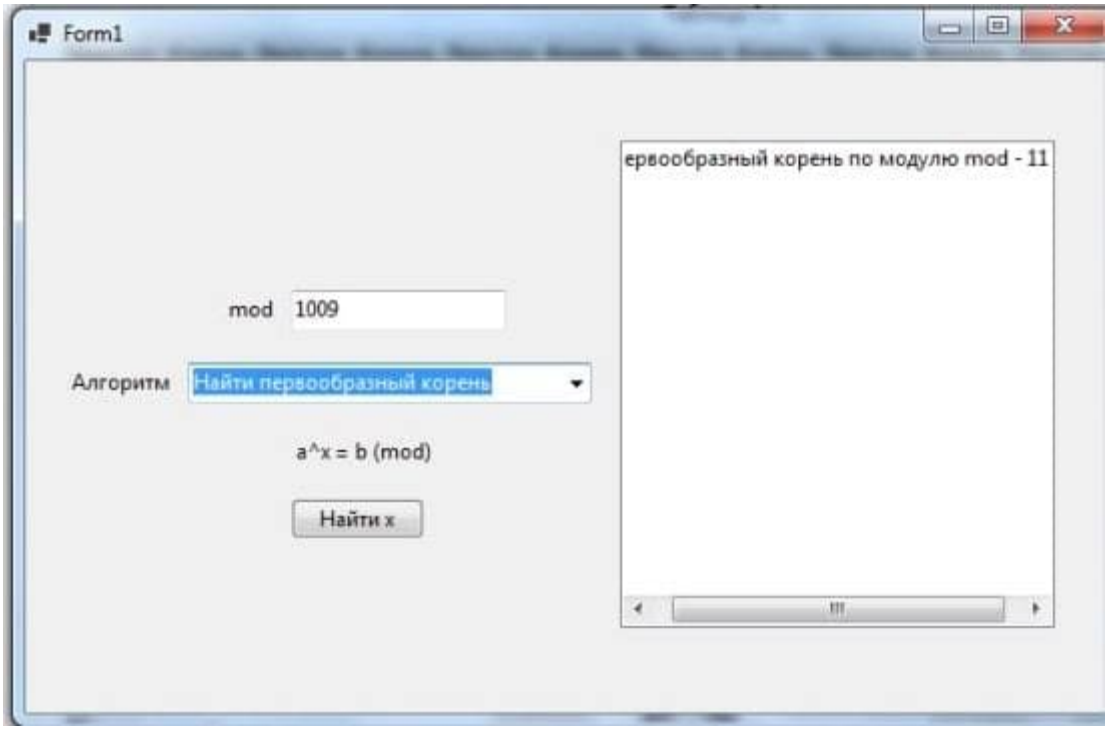

Рисунок 1.2 – знаходження первісного кореня

2 КІНЦЕВІ ПОЛЯ

Поле, що складається з кінцевого числа елементів, називається кінцевим.

Найбільш відомим прикладом кінцевого поля є поле класів відрахувань по простому модулю, тобто факторкольцо $\mathbb{Z}/(p)$, де p - просте число .
Кінцеве поле зазвичай позначається $\mathbb{F}_q$ або $\mathrm{GF}(q)$ (скорочення від *Galois field*) і називається полем Галуа порядку q , де q - число елементів поля. З точністю до ізоморфізму кінцеве поле повністю визначається його порядком, який завжди є ступенем якого-небудь простого числа , тобто $q = p^n$, де p - просте число, а n - будь- натуральне число . При цьому p буде характеристикою цього поля .
Поняття кінцевого поля використовується, зокрема, в теорії чисел , теорії груп , криптографії , в розробці криптографіческих стійких шифрів .

Характеристика кожного кінцевого поля є простим числом. Нехай F - кінцеве поле. Тоді воно складається з p^nелементів, де p - характеристика поля F, а натуральне число n - ступінь поля F над його простим підполем .
Згідно з теоремою про існування та єдність кінцевих полів, для кожного простого числа p і натурального числа n існує кінцеве поле з p^nелементів і будь-яке кінцеве поле з $q = p^n$елементів ізоморфно полю розкладання многочлена $x^q - x$ над полем $\mathbb{F}_p$. Дана теорема дозволяє говорити про цілком певному полі даного порядку q (тобто про *поле Галуа з* q елементів)

Поле $\mathbb{F}_{p^n}$ при n > 1 можна побудувати як факторкольцо $\mathbb{K} = \mathbb{F}_p[x]/(f(x))$, де $f(x)$ - не приводиться многочлен ступеня n над полем $\mathbb{F}_p$. Таким чином, для побудови поля з p^nелементів досить відшукати многочлен ступеня n, не приводиться над полем $\mathbb{F}_p$. Елементами поля $\mathbb{K}$ є класи відрахувань

многочленів ступеня меншою n з коефіцієнтами з $\mathbb{F}_p$ по модулю головного ідеалу , породженого многочленом $f(x)$.

Елемент $\alpha = x + (f(x)) \in \mathbb{F}_p[x]/(f(x))$ є коренем многочлена $f(x)$і поле $\mathbb{F}_p[x]/(f(x))$ породжується цим елементом над полем $\mathbb{F}_p$, тому перехід від поля $\mathbb{F}_p$до поля $\mathbb{F}_p[x]/(f(x))$ називається приєднанням до поля $\mathbb{F}_p$ кореня неприводимого многочлена $f(x)$

1.1 Приклади таблиць:

Поле $\mathbb{F}_2$ складається з двох елементів, але воно може бути задано різними способами в залежності від вибору елементів і визначення операцій додавання і множення на них:

- Як багато з двох чисел «0» і «1», на якому операції додавання і множення визначені як додавання і множення чисел з приведенням результату по модулю 2:

Таблиця 1

+	**0**	**1**	×	**0**	**1**
0	0	1	**0**	0	0
1	1	0	**1**	0	1

- Як багато з двох логічних об'єктів «БРЕХНЯ» (F) і «ІСТИНА» (T), на якому операції додавання і множення визначені як булеві операції «виключне або» і «і» відповідно:

Таблиця 1.1

+	**F**	**T**	×	**F**	**T**
F	F	T	**F**	F	F
T	T	F	**T**	F	T

Дані поля є ізоморфними одна одній, т. е. Це фактично два різні способи завдання одного і того ж поля.

Поле $\mathbb{F}_3 = \{0, 1, 2\}$. Додавання і множення визначені як додавання і множення чисел по модулю 3. Таблиці операцій $\mathbb{F}_3$ мають вигляд:

Таблиця 1.2

+	**0**	**1**	**2**
0	0	1	2
1	1	2	0
2	2	0	1

×	**0**	**1**	**2**
0	0	0	0
1	0	1	2
2	0	2	1

Поле $\mathbb{F}_4$ можна уявити як безліч $\{0, 1, \alpha, \alpha + 1\}$ (де α - корінь многочлена $f(x) = x^2 + x + 1$, тобто $\alpha^2 = -\alpha - 1 = \alpha + 1$). Таблиці операцій $\mathbb{F}_4$ мають вигляд:

Таблиця 1.3

+	**0**	**1**	α	$\alpha + 1$
0	0	1	α	$\alpha + 1$
1	1	0	$\alpha + 1$	α
α	α	$\alpha + 1$	0	1
$\alpha + 1$	$\alpha + 1$	α	1	0

×	**0**	**1**	α	$\alpha + 1$
0	0	0	0	0
1	0	1	α	$\alpha + 1$
α	0	α	$\alpha + 1$	1
$\alpha + 1$	0	$\alpha + 1$	1	α

Для побудови поля $\mathbb{F}_9 = \mathrm{GF}(3^2)$ досить знайти нормований многочлен ступеня 2, не приводиться над $\mathbb{F}_3$. Такими многочленами є:

$x^2 + 1$
$x^2 + x + 2$
$x^2 + 2x + 2$

Для x^2+1 поле є $\mathbb{F}_9 = \mathbb{Z}_3[x]/(x^2+1)$ (якщо замість x^2+1взяти інший многочлен, то вийде нове поле, изоморфное старому). У наведених нижче таблицях символ i позначає клас еквівалентності многочлена x в Факторкольцо $\mathbb{Z}_3[x]/(x^2+1)$, що задовольняє рівняння $i^2+1=0$. Таблиця додавання в $\mathbb{F}_9$ визначається, виходячи зі співвідношення $1+1+1=0$:

Таблица 1.4

+	0	1	2	i	$i+1$	$i+2$	$2i$	$2i+1$	$2i+2$
0	0	1	2	i	$i+1$	$i+2$	$2i$	$2i+1$	$2i+2$
1	1	2	0	$i+1$	$i+2$	i	$2i+1$	$2i+2$	$2i$
2	2	0	1	$i+2$	i	$i+1$	$2i+2$	$2i$	$2i+1$
i	i	$i+1$	$i+2$	$2i$	$2i+1$	$2i+2$	0	1	2
$i+1$	$i+1$	$i+2$	i	$2i+1$	$2i+2$	$2i$	1	2	0
$i+2$	$i+2$	i	$i+1$	$2i+2$	$2i$	$2i+1$	2	0	1
$2i$	$2i$	$2i+1$	$2i+2$	0	1	2	i	$i+1$	$i+2$
$2i+1$	$2i+1$	$2i+2$	$2i$	1	2	0	$i+1$	$i+2$	i
$2i+2$	$2i+2$	$2i$	$2i+1$	2	0	1	$i+2$	i	$i+1$

Таблиця множення в $\mathbb{F}_9$ визначається зі співвідношення $i^2 = -1$:

Таблиця 1.5

×	0	1	2	i	$i+1$	$i+2$	$2i$	$2i+1$	$2i+2$
0	0	0	0	0	0	0	0	0	0
1	0	1	2	i	$i+1$	$i+2$	$2i$	$2i+1$	$2i+2$
2	0	2	1	$2i$	$2i+2$	$2i+1$	i	$i+2$	$i+1$
i	0	i	$2i$	2	$i+2$	$2i+2$	1	$i+1$	$2i+1$
$i+1$	0	$i+1$	$2i+2$	$i+2$	$2i$	1	$2i+1$	2	i
$i+2$	0	$i+2$	$2i+1$	$2i+2$	1	i	$i+1$	$2i$	2
$2i$	0	$2i$	i	1	$2i+1$	$i+1$	2	$2i+2$	$i+2$
$2i+1$	0	$2i+1$	$i+2$	$i+1$	2	$2i$	$2i+2$	i	1
$2i+2$	0	$2i+2$	$i+1$	$2i+1$	i	2	$i+2$	1	$2i$

Можна перевірити, що елемент $i+1$ має порядок 8 і є примітивним. Елемент i не є примітивним, так як $i^4 = 1$

3 ДИСКРЕТНИЙ ЛОГАРИМ ТА ФАКТОРИЗАЦІЯ

Проблема дискретного логарифму така: задані цілі числа a і b, відносно прості до n, ми хочемо вирішити

$$a^z \equiv b \pmod{n}.$$

Загалом, знайти такий z за відомими методами досить повільно; справді, ми могли б врахувати n за рівну кількість часу. Шенкс представляє алгоритм, який буде обчислювати дискретний логарифм за модулем n приблизно за n кроків; це приблизно час, необхідний для врахування n методом грубої сили. Найшвидший відомий метод дискретних логарифмів - метод Адельмана ; він використовує алгоритм факторингу Моррісона-Бриллхарта і має однакову складність: очікуваний час роботи, який $O(\exp(c \cdot \sqrt{\log n \log\log n}))$. Ці спостереження змушують шукати взаємозв'язки між двома проблемами, і це є предметом даної роботи.

Щоб закріпити ідеї, спершу розглянемо аналогічну задачу розв’язування поліноміальних рівнянь за модулем n. Відомо наступне:

1) Розв’язування поліноміальних f (z mod n) настільки важке, як розкладання на множники n; це було доведено для квадратних многочленів Рабіном .

2) Якщо є рішення f (z mod p) для простого p можна підняти його до розв’язку за модулем будь-якої степені p за поліноміальний час; це випливає про лему Гензеля.

3) Якщо ми знаємо факторизацію n, то ми можемо взяти рішення f за модулем p для p I n і отримати рішення за модулем n за поліноміальний час, використовуючи китайську теорему залишку.

Всі аналогічні теореми справедливі для експоненціальних конгруенцій mod n. Точніше, ми доводимо:

a) Якщо ми можемо розв’язати при a = b за модулем n за поліноміальний час, то з великою ймовірністю ми можемо знайти належний коефіцієнт n за поліномний час. Це скорочення можна зробити детермінованим, якщо розширена гіпотеза Рімана відповідає дійсності.

b) Вся складність у розв’язанні cz = b за модулем простої потужності полягає у її вирішенні за модулем простого рівня; якщо рішення існує за

модулем pc, ми можемо отримати його за поліномний час із рішення за модулем p.

c) Якщо ми можемо врахувати поліноміальний час, то для швидкого розв'язання за модулем n все, що нам потрібно, це рішення за модулем прості дільники n.

Підводячи підсумок: розв'язання задачі дискретного логарифму для складеного модуля настільки ж важке, як розкладання на множники та розв'язання його простих чисел.

3.1 Умовні позначення і історія питання

Поліноміальний час означає час, обмежений поліномом у кількості бітів, необхідних для представлення вхідних даних; на практиці це означає, що алгоритм, який працює за модулем n, потребує часу, обмеженого фіксованою потужністю r logn, де r - кількість вхідних значень.

Z - цілі числа, а zn - кільце цілих чисел за модулем n. Аддитивна група zn записується, а її мультиплікативна група одиниць z. Коли n є простим, це циклічна група; він також циклічний, якщо n - ступінь простого числа, більшого за 2. Ейлерова збірка é (n) визначається як кількість елементів у z.

Якщо n має просту факторизацію

$$n = P11 \cdots p_r$$

тоді структура zn задається китайською теоремою залишку:

$$\mathbb{Z}_n \cong \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_r^{e_r}}.$$

Обидва напрямки цього ізоморфізму можна обчислити за поліноміальний час. Для проектування z на i-й множник, ми зменшуємо z за модулем p. Щоб піти іншим шляхом, ми використовуємо таку теорему: з урахуванням будь-якої множини лінійних конгруенцій

$$a_i \cdot x \equiv b_i \pmod{n_i}, i = 1, \ldots, k,$$

ми можемо вирішити, чи вони узгоджуються, і якщо так, обчислимо рішення, все за поліноміальний час.

p завжди буде позначати просте число. Теорема простих чисел гарантує, що їх достатньо визначаємо

$$\nu_p(x) = \max\{k : p^k \mid x\}.$$

vp (O) можна прийняти за $+\infty$.

Для дійсних чисел z Lz позначає найбільше ціле число <z, a logz означає log2 (z).

3.2 Як факторизувати, повертаючи дискретні логаритми

Припустимо, що n - непарне ціле натуральне число, яке не є простим ступенем (усі ці умови перевіряються у випадковому поліноміальному часі); нехай буде його первинною факторизацією

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

Тоді

$$\mathbb{Z}_n^* \cong \mathbb{Z}_{p_1^{e_1}}^* \times \cdots \times \mathbb{Z}_{p_k^{e_k}}^*.$$

Кожен прямий фактор є циклічним порядком

$$\phi_i = (Pt\text{-}l)\ ps\ t,$$

і тому кожен елемент цієї групи має порядок,

що ділить X = lcm

(él,.. . , ϕ_k)

Наступна теорема була доведена (по суті) Міллером, припустимо, що ми можемо розв'язати збіжність k = mod n для деяких за час полінома (такий z називається показником для a). Тоді, якщо ми виберемо n випадковим чином, з імовірністю > 1k, ми можемо використовувати a, щоб знайти правильний коефіцієнт n за поліноміальний час.

Ось як це зробити: нехай

$$K = \{a \in \mathbb{Z}_n^* : a^{\lambda/2} \equiv \pm \text{I} \pmod{n}\}.$$

для всіх i, тоді нехай a має порядок за модулем i порядок 48/2 за модулем для інших простих степенів. Зараз ми побудували один cfK, але за теорією груп ми знаємо більше: якщо ми виберемо випадковий елемент z, шанси становитимуть принаймні що він знаходиться поза K.

Якщо a - такий елемент, а z - показник ступеня для нього, то для деякого k, $0 < k <$

$$\neq \pm 1 \pmod{n}$$

але

$$(a^{z/2^k})^2 \equiv 1 \pmod{n}.$$

Щоб побачити це, нехай a - порядок a у Z, який ми можемо записати $\lambda = \alpha \cdot \beta_0$ and $z = \alpha \cdot \beta_1 \cdot 2^\nu$ для непарних чисел ро . Потім $a^{\lambda/2} \equiv a^{\beta_0 \lambda/2} \equiv a^{\alpha/2} \pmod{n}$,

і результат випливає, приймаючи k = V + 1.

Звідси випливає, що

$$(a^{z/2^r} + 1) \bmod n)$$

є належним коефіцієнтом n. Більше того, якщо z обчислюється в поліноміальному часі, він не може бути занадто великим; тому ми можемо швидко отримати свій фактор.

Тепер ми покажемо, як знайти показник степеня для a, розв'язуючи mod n); зверніть увагу, що це не допоможе нам, як завжди рішення. Спочатку нам потрібне просте (n); ми не знаємо 4 (n) апріорі, але ми знаємо, що це менше n, і тому серед перших logn + I простих чисел повинен бути один, який буде працювати. Тоді такий ap - одиниця mod 4, а отже

$$(a^p)^y \equiv a \pmod{n}$$

має рішення y. Якщо ми ставимо

z = py-l/

Слід зазначити дві речі. По-перше, якщо ERH відповідає дійсності, вищезазначене скорочення можна зробити детермінованим: ми враховуємо n, знаходячи показник ступеня для cÊK, a ERH передбачає, що найменшим елементом поза K є O (logn) 2

По-друге, аргумент Міллера працює, якщо замінити Z поліноміальним кільцем k [Xl, для деякого скінченного поля k. Якщо fek [Xl має ступінь r, a k має q елементів, то $\prod_{i=1}^{r}(q^i-1)$ є показником

Таким чином, майже без зайвої роботи ми також маємо доказ того, що поліноми над k можуть бути розкладені на множники за випадковий багаточлен

3.3 Підйомні рішення по модулю, принцип потужності

У цьому розділі припустимо, що a(mod p) для $a, b \in \mathbb{Z}_p^*$; ми покажемо, як обчислити z таким, що mod p^e) (якщо такий існує) за поліноміальний час. На даний момент припустимо, що p> 3; алгоритм для p = 2 дещо відрізняється, і модифікації будуть вказані пізніше.

По-перше, оскільки z циклічна, ми це знаємо

$$\mathbb{Z}_{p^e}^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_{p^{e-1}}^+.$$

Проекція на z задається редукційним mod p, і пізніше буде показано, що проекція e на Z є поліноміальним часом, що обчислюється. Припускаючи цей останній факт, ми обчислюємо z2 так, що $\equiv \theta(b) \pmod{p}$;
ми вже знаємо таке, що

$$a^{z_1} \equiv b \pmod{p}.$$

Тоді ми знаходимо ціле число z, яке

задовольняє

$x \equiv x_1 \pmod{p-1}$,

$x_2 \pmod{p^c}$;

Залишилося показати, як обчислити e. У нас є факторизація

$$\mathbb{Z}_{p^e}^* \cong \mathbb{Z}_p^* \times U,$$

де $U = \{x \in \mathbb{Z}_{p^e}^* : x \equiv 1 \pmod{p}\}$.

Проекція r на U дається шляхом підняття елемента до p - l-ї степені, і тому ми закінчимо, якщо маємо обчислюваний ізоморфізм поліноміального часу $\lambda : U \to \mathbb{Z}_{p^t}$ -1 Якщо ми визначимо X тоді ми можемо обчислити його за поліноміальний час, обчислюючи чисельник у дужках за модулем

2e-1

Тепер ми покажемо, що X має правильні алгебраїчні властивості. По-перше, якщо a, beZ, $\nu_p(a-b) \geq 1$, і k> l, тоді

індукцією по k. Зокрема, тоді, ре I a mod p, так що X має сенс для цілочисельних аргументів. Це добре визначено як функція на U, для if mod ре, тоді від використання ідентичності

$$xy - 1 = (x-1) + (y-1) + (z-1)(y-1)$$

ми можемо довести, що це гомоморфізм від U до Z + 1. Нарешті, довести, що це a iso-, вибирайте aeZ так, щоб $\nu_p(a-1) = 1$.

Це означає, що X знаходиться отже, має бути ізоморфізмом.

Коротко, ось модифікації, необхідні для p = 2. Припустимо, що e> 3,

тоді Z2'e = SX U,

де l та 4. Проекції на S та U посилають z на ± 1 та ± z відповідно, приймаючи знак +, коли mod 4.

Цей алгоритм був представлений в алгебраїчному відношенні, але інтуїція його випливає з аналізу . Скажімо, що цілі числа z та y "близькі", якщо вони однакові за модулем степеня p; чим вища потужність, тим ближче вони Тоді ми шукаємо наближення

cZ — b

Як би ми мали правильну функцію логарифму X, то ми очікували б, що

наближення до X (a)

має бути необхідний z. Дивно, але цей "p-адичний чисельний аналіз" працює; апроксимуючи p-адичний логарифм, визначений у , ми отримуємо швидкий алгоритм

2.4 Отримання регіонального рішення

У цьому розділі ми припускаємо, що доступні всі необхідні прості факторизації. Нехай простим факторизацією n буде

n —- P1 · · · p_ri

нехай $a, b \in \mathbb{Z}_n^*$.Ми хочемо взяти "локальні"

рішення z, до a s'— b (mod p,)

і створити "регіональне" рішення z t o

$$a^z \equiv b \pmod{n}.$$

За результатами останнього розділу ми можемо припустити, що

$$a^{z_i} \equiv b \pmod{p_i^{e_i}},$$

і тому ми вибираємо z, що відповідає z, за модулем порядок a у Z • e, для i = l, . . . , r.

Залишилось лише показати, як обчислити порядок за модулем дільника простої потужності P з n. Це є

$$\prod_{\substack{q \mid \phi(P) \\ q \text{ prime}}} q^{\nu_q(\text{order}(a))},$$

і $\nu_q(\text{order}(a)) = \min\{k : a^{\phi(P)/q^k} \equiv 1 \pmod{p}\}$.

3.5 Зауваження

Ми довели, що факторизація n необхідна для розв'язку дискретних зондів логарифму лем за модулем n; можна також запитати, чи потрібно розкладання множників на фактори для вирішення дискретних задач логарифму за модулем p. І навпаки, можна попросити швидкий алгоритм задач, припускаючи всі необхідні факторизації. Обидва ці питання залишаються без відповіді.

Традиційна теоретично-теоретична аналогія полягає в тому, що "числові поля" (наприклад, Q) подібні до "функціональних полів" (наприклад, Z p

(X)); результати першого розділу свідчать про те, що нам було б добре дослідити цю відповідність з алгоритмічної точки зору.

3 АЛГОРИТМ ВЕЛИКИХ ТА МАЛИХ КРОКІВ

Детермінований алгоритм дискретного логарифмування в мульпликативной групі кільця вирахувань по модулю простого числа. Алгоритм був запропонований Дэниэлем Шенксом в 1972 році. Відноситься до методів зустрічі посередині. Для модулів спеціального виду алгоритм Шенкса є поліноміальним. Завдання дискретного логарифмування представляє великий інтерес в області криптосистем з відкритим ключем. Більшість існуючих алгоритмів знаходження дискретного логарифма грунтована на припущенні про надзвичайно високу обчислювальну складність(питання про рішення задачі дискретного логарифма за поліноміальний час на персональному комп'ютері залишається відкритим досі); чим складніше її вирішити, тим криптобезопасной є пересилка інформації. Таким чином, одним із способів підвищити складність завдання дискретного логарифмування являється створення криптосистеми, грунтованої на групі з великим порядком.

Ідея алгоритму полягає у виборі оптимального співвідношення часу і пам'яті, а саме у вдосконаленому пошуку показника міри.

Рішення рівняння $gx \equiv h$ для заданих g і h називається дискретним логарифмом елементу h по основі g. Якщо G є мультиплікативною групою кільця вирахувань по модулю m, то рішення можна також назвати індексом числа a по основі g. Індекс числа a по основі g обов'язково існуватиме, якщо g є первісним коренем по модулю m. Нехай задано в деякій кінцевій мультиплікативній групі рівняння $gx \equiv h$. Рішення задачі дискретного логарифмування полягає в знаходженні деякого цілого позитиного числа, що задовольняє рівнянню. Якщо воно вирішуване, у нього буде хоч би одно натуральне рішення, яке не перевищує порядок групи. Це дає первинну оцінку складності алгоритму пошуку рішень згори. Алгоритм "грубої сили" відшукав би рішення за число кроків не вище за порядок цієї групи. Зазвичай розглядається випадок, коли група циклічна, тоді рівняння завжди має рішення. Якщо група довільна, то питання про вирішувану завдання дискретного логарифмування вимагає окремого розгляду.

Приклад:

Таблиця 1

Приклад дискретного логарифма

a = 114; b= 24; mod 137

1:81
2:122
3:18
4:88
5:4
6:50
7:77
8:72
9:78
10:16
11:63
12:34

Нехай ми маємо рівняння: ax= b(mod m), де a і m взаємно прості. Перетворимо рівняння, позначивши x = np - q, де n - це вибрана константа. P зазвичай називають "giant step", оскільки збільшення p на одиницю збільшує x на n, а q відповідно називають "baby step". Будь-яке x з проміжку [0; m) можна представити в цій формі, при цьому буде досить значень: p належить [1; [m / n]], де q належить [1; n]. Рівняння набере вигляду: anp - q= b(mod m), а оскільки a і m взаємно прості, отримуємо: anp=baq(mod m). Щоб вирішити це рівняння, треба знайти відповідні значення p і q, щоб значення лівої і правої частин співпали, іншими словами треба вирішити рівняння: f1(p) = f2(q). Це завдання вирішується за допомогою методу "зустріч по середині". Перша частина алгоритму : вважаються значення функції f1 для усіх значень аргументу p і після цього ці значення сортуються. Друга частина алгоритму : перебираються значення другої змінної q і обчислюється друга функція f2, і шукається. Швидкість роботи алгоритму в середньому виконується 1.5 $\sqrt{n}$ операцій у гіршому випадку потрібно 2 $\sqrt{n}$операцій.

Реалізація на программному рівні виглядає так:

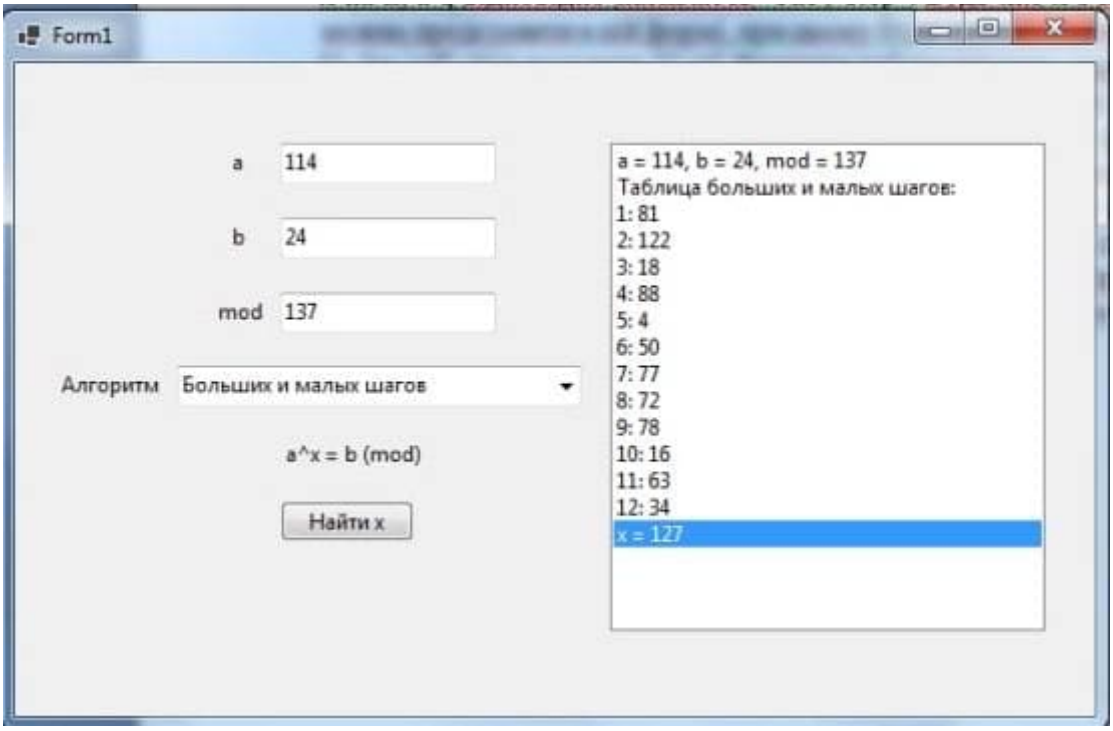

Рисунок 2 – алгоритм великих та малих кроків

Вхідні значення A = 114, B = 24, M = 137, результат обчислення X = 127

4 АЛГОРИТМ СІЛЬВЕРА-ПОЛІГА-ХЕЛЛМАНА

Цей алгоритм був придуманий американським математиком Роландом Сильвером(англ. Roland Silver), але уперше був опублікований іншими двома американськими математиками Стівеном Полигом(англ. Stephen Pohlig) і Мартіном Хеллманом в 1978 році в статті "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance", які незалежно від Роланда Сильвера розробили цей алгоритм.

Детермінований алгоритм дискретного логарифмування в кільці вирахувань по модулю простого числа. Одній з особливістю алгоритму є те, що для простих чисел спеціального виду можна знаходити дискретний логарифм за поліноміальний час.

Ідея алгоритму:

Суть алгоритму в тому, що досить знайти x по модулях $q_i^{\alpha_i}$ для усіх i, а потім рішення початкового порівняння можна знайти за допомогою китайської теореми про залишки. Щоб знайти x по кожному з таких модулів, треба вирішити порівняння:

$$(a^{\frac{p-1}{q_i^{\alpha_i}}})^x \equiv b^{\frac{p-1}{q_i^{\alpha_i}}} \pmod{p}.$$

Це порівняння вирішується за поліноміальний час у разі, якщо qi - невелике, де s - деяка константа)

Основний опис:

Скласти таблицю значень {ri, j}, де:

$$r_{i,j} = a^{j\frac{p-1}{q_i}},$$
$$i \in \{1, \ldots, s\},$$
$$j \in \{0, \ldots, q_i - 1\}.$$

Для i от 1 до s:
Нехай
$$x \equiv \log_a b \equiv x_0 + x_1 q_i + \ldots + x_{\alpha-1} q_i^{\alpha-1} \pmod{q_i^{\alpha_i}},$$
Де
$$0 \le x_i \le q_i - 1.$$
Тоді слідкуе, що
$$b^{\frac{p-1}{q_i}} \equiv a^{x_0 \frac{p-1}{q_i}} \pmod{p}$$

За допомогою таблиці, складеної на кроці 1, знаходимо x0.
Для j от 1 до α − 1
Розглядаємо порівняння

$$(ba^{-x_0-x_1q_i\ldots-x_{j-1}q_i^{j-1}})^{\frac{p-1}{q_i^{j+1}}} \equiv a^{x_i\frac{p-1}{q_i}} \pmod p$$

Рішення знову ж таки знаходиться по таблиці кінець циклу по і кінець циклу по j
Знайшовши $\log_a b \bmod q_i^{\alpha_i}$ для всіх і, знаходимо $\log_a b \bmod (p-1)$

1.1Таблиця результатів порівняльного аналізу алгоритмів.

Алгоритм	Плюси:	Мінуси:
Гельфонда-Шенкса	• простий в реалізації • детермінований	• повільний • величезні витрати по пам'яті
Поліга-Хеллмана	• Поліміальний коли qi в $p-1=\Pi_{i=1}^{k} Qi^a$ мали	• марний, коли qi великі

5 ОПИС ТА ДЕМОНСТРАЦІЯ РОБОТИ

Програма була реалізована на мові C#. Розширюваність системи, швидкість роботи, зручність розробки, захищеність і контроль версій алгоритмів, що підключаються

Головне вікно програми має досить простий вигляд. Маємо два числа, модуль, список алгоритмів, кнопку пошуку та вікно обчислення.

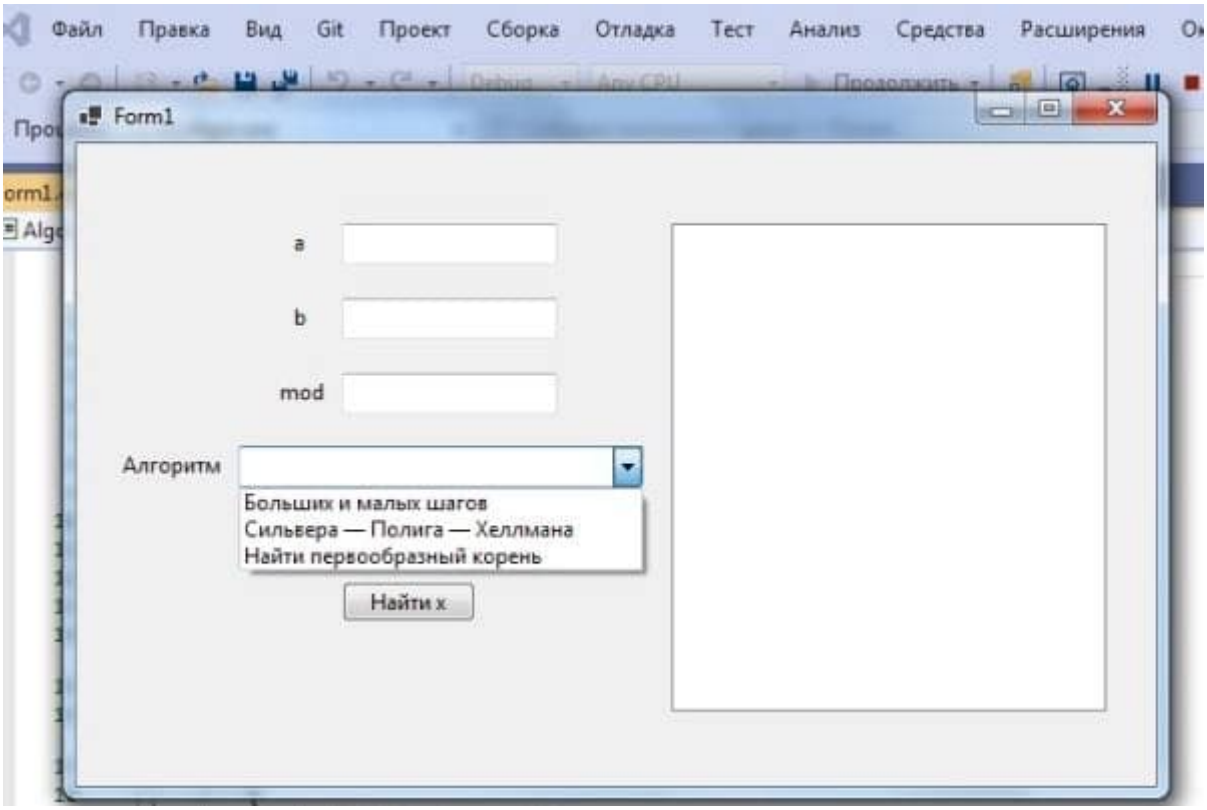

Рисунок 3 - реалізація програми

Вводяться дані після чого проходить обчислення алгоритму.

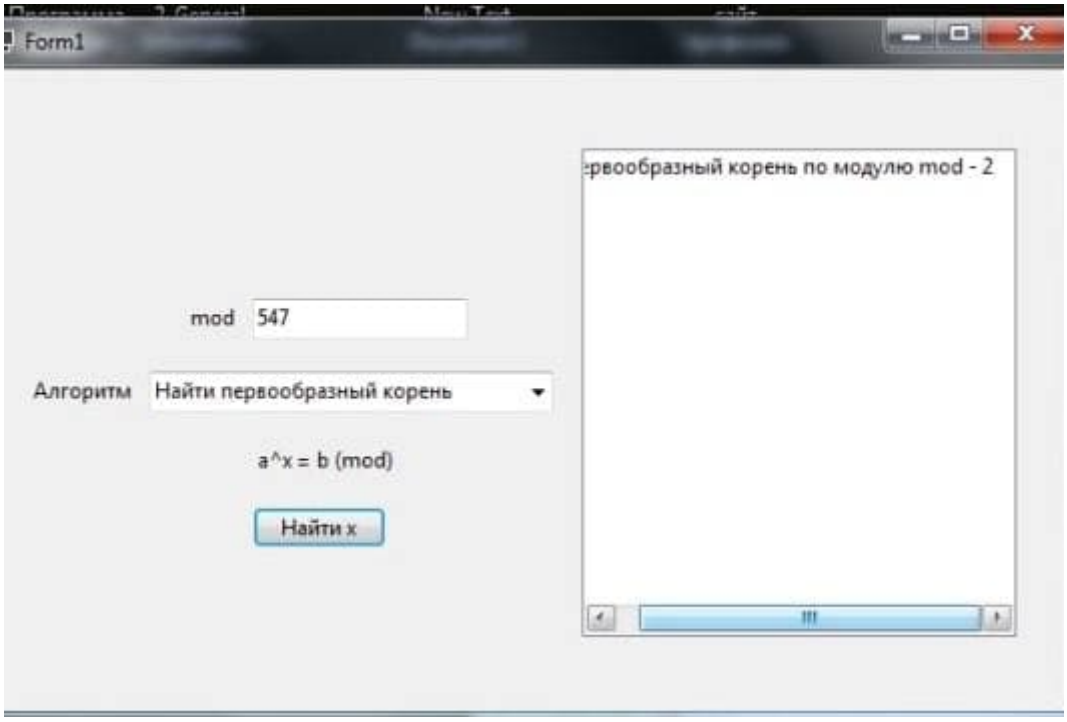

Рисунок 3.1 Спочатку ми перевіряємо чи є число 547 простим

Далі вирішуємо задачу за допомогою алгоритма Сільвера-Поліга-Хеллмана

Вхідні значення A = 2, B = 17, M = 547, результат обчислення X = 151

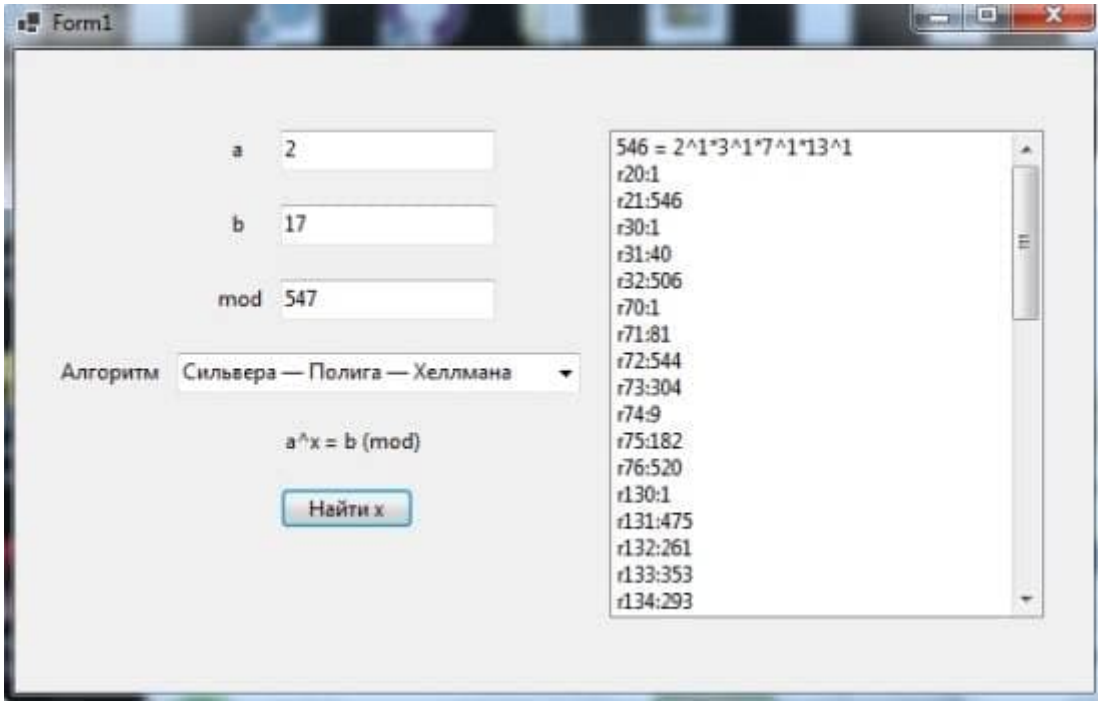

Рисунок 3.2 Реалізація алгоритма

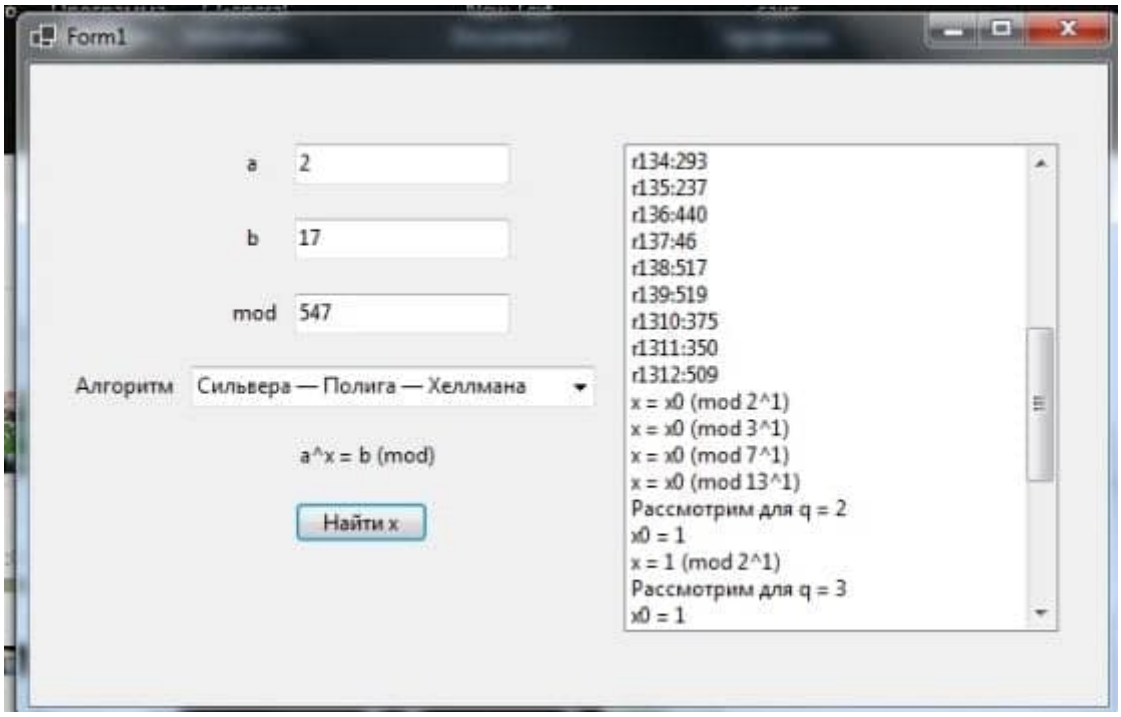

Рисунок 3.3 Процес обчислення

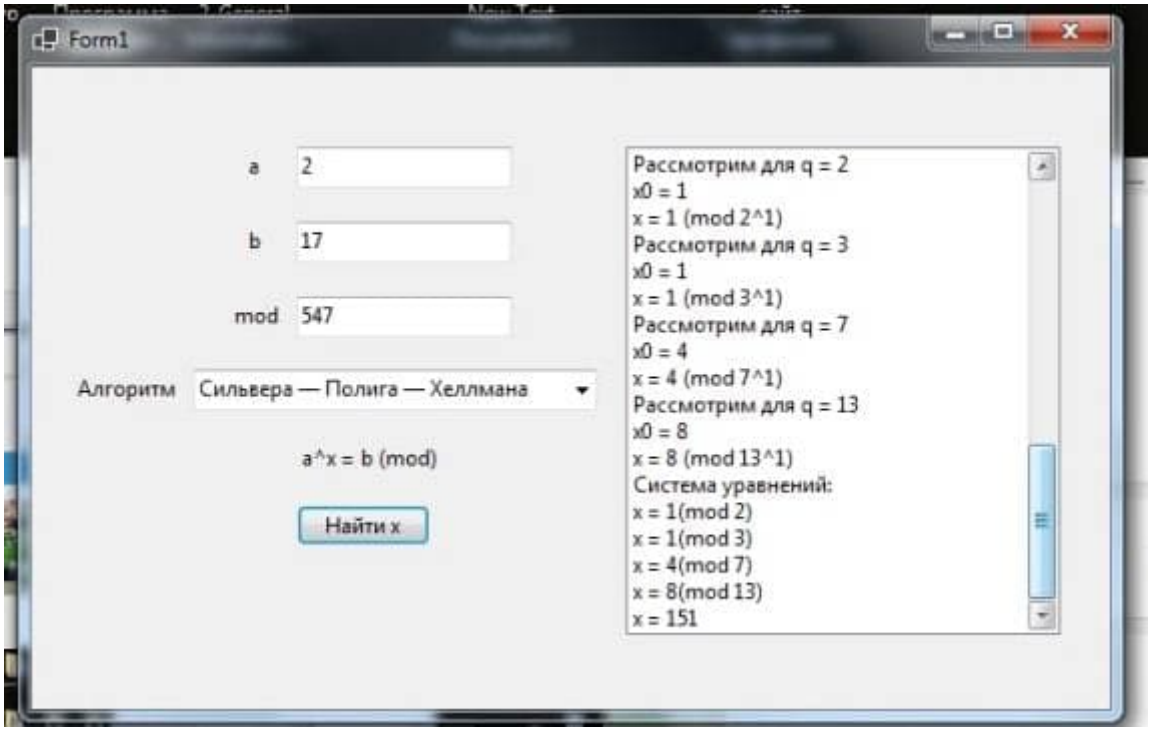

Рисунок 3.4 Кінцевий результат

ВИСНОВКИ

У представленій дипломній роботі розглянута проблема дискретного логорифма в криптографії, розв’язання дискретного логарифма двома методами (великих та малих кроків, Сільвера-Поліга-Хеллмана) реалізація на програмну рівні на мові C#. Основним показником складності алгоритму є час, необхідний для вирішення задачі і обсяг необхідної пам'яті.

Також при аналізі складності для класу задач визначається деяке число, що характеризує певний обсяг даних - розмір входу.

Отже, можемо зробити висновок, що складність алгоритму - функція розміру входу.

Складність алгоритму може бути різною при одному і тому ж розмірі входу, але різних вхідних даних.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Теоретико-числові алгоритми в криптографії. Автор: Василенко О.М.
2. Курс теорії чисел і криптографії. Деталі. Автор: Коблиц
3. Бабаш, А. В. Історія криптографії. Частина I / А.В. Бабаш, Г.П. Шанкин. - М.: Гелиос АРВ, 2016. - 240 с..
4. Криптографія на C# и C++ в действии. Учбовий посібник / М. Вельшенбах.
5. Шнайер, Б. Прикладна криптографія. Протоколи, алгоритми, вхідні тексти на мові С / Б. Шнайер. - М.: Триумф, 2012. - 518 с.
6. S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance (англ.) // IEEE Transactions on Information Theory. — 1978. — Vol. 1, no. 24. — P. 106-110.
7. A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance (англ.) // T.Beth, N.Cot, I.Ingemarsson Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques. — NY, USA: Springer-Verlag New York, 1985. — P. 224-314.
8. J. Hoffstein, J. Pipher, J. H. Silverman. An Introduction to Mathematical Cryptography (англ.). — Springer, 2008. — 524 p.

ДОДАТОК А

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Numerics;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Algos
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {

        }

        private void button1_Click(object sender, EventArgs e)
        {
            listBox1.Items.Clear();
            if (method.SelectedIndex == 2 && mod.Text != "")
            {
                int Mod = Convert.ToInt32(mod.Text);
                primitive_root(Mod);
            }
            else if (a.Text == "" || b.Text == "" || mod.Text == "")
                MessageBox.Show("Введите все необходимые данные!!!");
            else
            {
                int A = Convert.ToInt32(a.Text);
                int B = Convert.ToInt32(b.Text);
                int Mod = Convert.ToInt32(mod.Text);
                switch (method.SelectedIndex)
                {
                    case 0: big_and_little(A, B, Mod); break;
                    case 1: polima_silvera_helman(A, B, Mod); break;
                    case 2: primitive_root(Mod); break;
                }
```

```
            }
        }
        void big_and_little(int a, int b, int mod)
        {
            int m = (int)Math.Ceiling(Math.Sqrt(mod));
            int y = ModPow(a,m,mod);
            Hashtable steps = new Hashtable();
            listBox1.Items.Add("a = " + a + ", b = " + b + ", mod = "
+ mod);
            listBox1.Items.Add("Таблица больших и малых шагов:");
            for (int i = 1; i <= m; i++)
            {
                if(!steps.ContainsKey(ModPow(y, i, mod)))
steps.Add(ModPow(y, i, mod), i);
                listBox1.Items.Add(String.Format("{0}: {1}", i,
ModPow(y, i, mod)));
            }
            bool isfind = false;
            for (int i = 1; i <= m - 1; i++)
            {
                int check = (ModPow(a, i, mod) * b) % mod;
                if (steps.ContainsKey(check))
                {
                    listBox1.Items.Add("x = " + ((int)steps[check]* m
- i));
                    isfind = true;
                }
            }
            if(!isfind) listBox1.Items.Add("Для числа a не существует
x, чтобы a^x=b(mod)");
        }
        void polima_silvera_helman(int a, int b, int mod)
        {
            Dictionary<int, int> nums = decomposition(mod - 1);
            string str = (mod - 1) + " = ";
            foreach (var it in nums)
                str += it.Key + "^" + it.Value+"*";
            str = str.Substring(0,str.Length-1);
            listBox1.Items.Add(str);
            Dictionary<int,int> r = new Dictionary<int, int>();
            Dictionary<int, int> system = new Dictionary<int, int>();
            foreach (var i in nums)
            {
                for (int j = 0; j < i.Key; j++)
                {
                    r[i.Key*10+j] = ModPow(i.Value, j * (mod - 1) /
i.Key, mod);
                }
            }
            foreach (var it in r)
            {
```

```
                listBox1.Items.Add("r" + it.Key+ ":" + it.Value);
            }

            foreach (var i in nums)
            {
                string xs = "x = x0";
                for (int j = 1; j < i.Value; j++)
                {
                    xs += " + x" + j + " * " + i.Key + "^" + j;
                }
                xs+=" (mod "+i.Key+"^"+i.Value+")";
                listBox1.Items.Add(xs);
            }
            foreach (var i in nums)
            {
                listBox1.Items.Add("Рассмотрим для q = " + i.Key);
                int num = ModPow(b, (mod - 1) / i.Key, mod);
                List<int> x = new List<int>();
                foreach (var j in r)
                {
                    if (j.Key / 10 == i.Key && j.Value == num)
                    {
                        x.Add(j.Key % 10);
                        listBox1.Items.Add("x0 = "+j.Key % 10);
                    }
                }
                if (x.Count == 0)
                {
                    listBox1.Items.Add("Невозможно найти дискретный
логорифм этим методом для этого числа!");
                }
                else
                {
                    int alfa = 0;
                    for (int j = 1; j < i.Value; j++)
                    {
                        alfa -= x[j - 1] * (int)Math.Pow(i.Key, j -
1);
                        int osnova = (int)(b * Math.Pow(i.Value,
alfa));
                        int stepen = (mod - 1) / (int)Math.Pow(i.Key,
j + 1);
                        int b1 = ModPow(osnova, stepen, mod);
                        foreach (var k in r)
                        {
                            if (k.Key / 10 == i.Key && k.Value == b1)
                            {
                                x.Add(k.Key % 10);
                                listBox1.Items.Add("x" + j + " = " +
k.Key % 10);
                            }
```

```
                    }
                }
                string xk = "x = " + x[0];
                int decision = x[0];
                for (int k = 1; k < x.Count; k++)
                {
                    decision += x[k] * (int)Math.Pow(i.Key, k);
                    xk += "+" + x[k] + "*" + i.Key + "^" + k;
                }
                xk += " (mod " + i.Key + "^" + i.Value + ")";
                system[decision] = (int)Math.Pow(i.Key, i.Value);
                listBox1.Items.Add(xk);
            }
        }
        listBox1.Items.Add("Система уравнений:");
        foreach (var i in system)
        {
            listBox1.Items.Add("x = "+i.Key+"(mod "+i.Value+")");
        }
        int answer = ChiResTheory(system);
        listBox1.Items.Add("x = "+answer);
    }
    void primitive_root(int mod)
    {
        int m = GetPRoot(mod);
        if (m == 0)
            listBox1.Items.Add("Невозможно найти первообразный
корень!!!");
        else
            listBox1.Items.Add("Первообразный корень по модулю mod
- "+m);
    }
    public int GetPRoot(int p)
    {
        for (int i = 0; i < p; i++)
            if (IsPRoot(p, i))
                return i;
        return 0;
    }

    public bool IsPRoot(int p, int a)
    {
        if (a == 0 || a == 1)
            return false;
        int last = 1;
        List<int> set = new List<int>();
        for (int i = 0; i < p - 1; i++)
        {
            last = (last * a) % p;
            if (set.Contains(last)) // Если повтор
                return false;
```

```
            set.Add(last);
        }
        return true;
    }
    public int ModPow(int baseNum, int exponent, int modulus)
    {
        int B, D;
        B = baseNum;

        B %= modulus;
        D = 1;
        if ((exponent & 1) == 1)
        {
            D = B;
        }

        while (exponent > 1)
        {
            exponent >>= 1;
            B = (B * B) % modulus;
            if ((exponent & 1) == 1)
            {
                D = (D * B) % modulus;
            }
        }
        return (int)D;
    }

    Dictionary<int, int> decomposition(int n)
    {
        Dictionary<int, int> nums = new Dictionary<int, int>();
        if (Simple(n)) nums[n] = 1;
        else
        {
            for (int i = 2; n > 1; i++)
            {
                if (n % i != 0) continue;
                nums[i] = 0;
                while (n % i == 0)
                {
                    nums[i]++;
                    n /= i;
                }
            }
        }
        return nums;
    }
    bool Simple(int n)
    {
        bool prost = true;
        for (int i = 2; i <= n / 2; i++)
```

```
            {
                if (n % i == 0)
                {
                    prost = false;
                    break;
                }
            }
            return prost;
        }
        int ChiResTheory(Dictionary<int, int> nums)
        {
            int x = 0, M = 1;
            List<int> Mi = new List<int>();
            List<int> Y = new List<int>();
            int x0=0;
            foreach (var it in nums)
            {
                M*=it.Value;
            }
            foreach (var it in nums)
            {
                Mi.Add(M / it.Value);
            }
            for (int i = 0; i < nums.Count; i++)
            {
                for (int j = 1; j < nums.ElementAt(i).Value; j++)
                {
                    if ((Mi[i] * j) % nums.ElementAt(i).Value - 1 ==
0) Y.Add(j);
                }
            }
            for (int i = 0; i < Mi.Count; i++)
            {
                x0+=Mi[i]*Y[i]*nums.ElementAt(i).Key;
            }
            return x0%M;
        }

        private void method_SelectionChangeCommitted(object sender,
EventArgs e)
        {
            if (method.SelectedIndex == 2)
            {
                a.Visible = false;
                b.Visible = false;
                label1.Visible = false;
                label2.Visible = false;
            }
            else
            {
                a.Visible = true;
```

```
                b.Visible = true;
                label1.Visible = true;
                label2.Visible = true;
            }
        }
    }
}
```