НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ МОСКОВСКИЙ ЭНЕРГЕТИЧЕСКИЙ ИНСТИТУТ

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Лабораторная работа №1

По дисциплине «Криптографические методы защиты информации»

«Матричные шифры. Шифр Хилла»

Группа: А-14м-24

Студент: Гороховский И. А.

Преподаватель: Фролов А. Б.

Москва 2025

Задание 1.1

Построить шифр Хилла y=Mx при m=2, n=p, где p − простое число длины 50-k десятичных символов, k − Ваш номер в журнале группы.

Примечание: алфавиты открытого текста и шифртекста A=B=Zn.

Ключ есть обратимая матрица K размеров 2 x 2 над Zn.

(K обратима ⬄ НОД(n, det (M))=1).

(В нашем случае это эквивалентно тому, что определитель det K не равен 0, так n выбрано как простое число).

1. Получение числа p

Номер в журнале – 1. Следовательно, p должно быть длиной в 49.

Для нахождения такого большого числа, была написана функция generatePrime, комбинирующая несколько методов (решето Эратосфена, проверка малой теоремы Ферма) с использованием случайной генерации чисел:

def getSieve(n):  
 *"""  
 Реализация решета Эратосфена для нахождения всех простых чисел до n.  
 Возвращает список простых чисел.  
 """  
 # Создаем массив isprime, где isprime[i] = True, если i предположительно простое число* isprime = [True for \_ in range(n)]  
 prime = []  
 spf = [None for \_ in range(n)]  
  
 *# Числа 0 и 1 не являются простыми* isprime[0] = isprime[1] = False  
  
 *# Проходим по всем числам от 2 до n-1* for i in range(2, n):  
 if isprime[i]: *# Если i предположительно простое* prime.append(i) *# Добавляем его в список простых чисел* spf[i] = i *# Наименьший простой делитель числа i — это само число i  
  
 # Обновляем массив isprime, помечая составные числа как False* j = 0  
 while j < len(prime) and i \* prime[j] < n and prime[j] <= spf[i]:  
 isprime[i \* prime[j]] = False  
 spf[i \* prime[j]] = prime[j]  
 j += 1  
  
 return prime *# Возвращаем список простых чисел*def is\_probably\_prime(n, sieve):  
 *"""  
 Проверяет, является ли число n вероятно простым, используя список простых чисел из решета.  
 Возвращает False, если n делится на любое из простых чисел из решета.  
 """* for x in sieve:  
 if n % x == 0:  
 return False  
 return True  
  
  
def generatePrime(n: int):  
 *"""  
 Генерирует простое число длиной n десятичных цифр.  
 Использует алгоритм, основанный на решете Эратосфена и тестах на простоту.  
 """* up\_limit = 10 \*\* n *# Верхний предел для генерации числа (10^n)* lower\_limit = up\_limit // 10 *# Нижний предел для генерации числа (10^n)* primes = getSieve(1000)  
 s = primes[-1] *# Начинаем с наибольшего простого числа из решета  
  
 # Основной цикл для генерации простого числа* while s < lower\_limit:  
 lo = (lower\_limit - 1) // s + 1 *# Минимальное значение r* hi = (up\_limit - 1) // s *# Максимальное значение r* while True:  
 *# Генерируем случайное число r и вычисляем кандидата на простое число n* try:  
 r = random.randint(lo, hi) << 1 *# r — четное случайное число* except ValueError:  
 print((up\_limit - 1), s, hi)  
 r = random.randint(lo, hi + 1)  
 cand = s \* r + 1 *# Формула для нового кандидата на простое число  
  
 # Проверяем, является ли n вероятно простым* if not is\_probably\_prime(cand, primes):  
 continue  
  
 *# Проводим дополнительную проверку на простоту с помощью теста Ферма* while True:  
 a = random.randint(2, cand - 1)  
 if pow(a, cand - 1, cand) != 1: *# Проверяем малую теорему Ферма* break  
  
 *# Проверяем НОД для дополнительной уверенности* d = math.gcd((pow(a, r, cand) - 1) % cand, cand)  
 if d != cand:  
 if d == 1: *# Если НОД равен 1, n вероятно простое* s = cand  
 break  
  
 if s == cand:  
 break  
  
 if s > up\_limit:  
 return generatePrime(n)  
 return s

p = generatePrime(49)  
print("p =", p)

Результат:

p = 2415497672807849861119881665830571110721571755069

1. Составление матрицы M

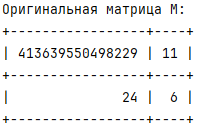
Для создания нужной матрицы были использованы методы получения случайных чисел. После её создания запускается проверка, что detM не равен 1, иначе матрица пересоздаётся, пока не выполнится это условие.

def get\_new\_matrix():  
 return [[random.randint(10 \*\* 10, 10 \*\* 15), random.randint(3, 30)],  
 [random.randint(3, 30), random.randint(3, 30)]]  
  
  
def determinant(matrix):  
 return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]

def print\_matrix(matrix, name):  
 print(name)  
 s = [[str(e) for e in row] for row in matrix]  
 lens = [max(map(len, col)) for col in zip(\*s)]  
 fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)  
 table = [fmt.format(\*row) for row in s]  
 print('\n'.join(table))  
 print()

matrix\_M = get\_new\_matrix()  
det\_M = determinant(matrix\_M)  
  
while det\_M == 1:  
 matrix\_M = get\_new\_matrix()  
 det\_M = determinant(matrix\_M)  
  
print\_matrix(matrix\_M, "Оригинальная матрица M:")

Результат:



1. Построить обратную матрицу M-1

Обратная матрица строится на основе оригинальной, с использованием расширенного алгоритма Евклида (функция mod\_inverse\_euclidian). После построения матрицы M-1, она перемножается с матрицей M, чтобы убедиться, что обратная матрица построена верно.

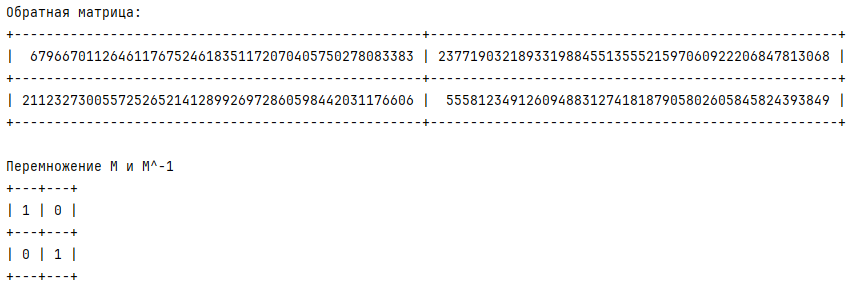
def extended\_gcd(a, b):  
 if b == 0:  
 return a, 1, 0  
 else:  
 g, x, y = extended\_gcd(b, a % b)  
 return g, y, x - (a // b) \* y  
  
  
def mod\_inverse\_euclidean(a, n):  
 g, x, y = extended\_gcd(a, n)  
 if g != 1:  
 return None *# Обратного элемента не существует* else:  
 return x % n *# Приводим результат к положительному значению*def inverse\_matrix(matrix, p):  
 det = determinant(matrix)  
 det\_inv = mod\_inverse\_euclidean(det, p)  
 if det\_inv is None:  
 raise ValueError("Матрица необратима")  
  
 *# Строим обратную матрицу* inv\_matrix = [  
 [(matrix[1][1] \* det\_inv) % p, (p - matrix[0][1] \* det\_inv) % p],  
 [(p - matrix[1][0] \* det\_inv) % p, (matrix[0][0] \* det\_inv) % p]  
 ]  
 return inv\_matrix

def multiply\_matrices(matrix1, matrix2, p):  
 *# Извлекаем элементы матриц* a11, a12 = matrix1[0][0], matrix1[0][1]  
 a21, a22 = matrix1[1][0], matrix1[1][1]  
  
 b11, b12 = matrix2[0][0], matrix2[0][1]  
 b21, b22 = matrix2[1][0], matrix2[1][1]  
  
 *# Вычисляем элементы результирующей матрицы* c11 = a11 \* b11 + a12 \* b21  
 c12 = a11 \* b12 + a12 \* b22  
 c21 = a21 \* b11 + a22 \* b21  
 c22 = a21 \* b12 + a22 \* b22  
  
 *# Возвращаем результирующую матрицу* return [[c11 % p, c12 % p], [c21 % p, c22 % p]]

matrix\_inv = inverse\_matrix(matrix\_M, p)  
print\_matrix(matrix\_inv, "Обратная матрица:")

print\_matrix(multiply\_matrices(matrix\_M, matrix\_inv, p), "Перемножение M и M^-1")

Результат:



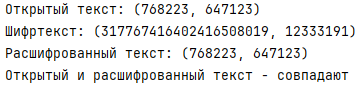
1. Зашифровать и расшифровать обратно выбранную шифрвеличину

Для шифрования и расшифрования была написана функция encode, которая принимает исходную шифрвеличину, матрицу и число p. В этой функции происходит перемножение шифрвеличины и матрицы, с дальнейшем получением остатка по модулю p. После проведения расшифрования, расшифрованный текст сравнивается с исходным.

def encode(ciphersize, matrix, p):  
 *# Извлекаем элементы* a11, a12 = ciphersize[0], ciphersize[1]  
  
 b11, b12 = matrix[0][0], matrix[0][1]  
 b21, b22 = matrix[1][0], matrix[1][1]  
 *# Вычисляем элементы результата* c11 = a11 \* b11 + a12 \* b21  
 c12 = a11 \* b12 + a12 \* b22  
  
 *# Возвращаем результирующую матрицу* return c11 % p, c12 % p

u1 = (random.randint(10 \*\* 2, 10 \*\* 6), random.randint(10 \*\* 2, 10 \*\* 6))  
c1 = encode(u1, matrix\_M, p)  
u\_dec = encode(c1, matrix\_inv, p)  
print("Открытый текст:", u1)  
print("Шифртекст:", c1)  
print("Расшифрованный текст:", u\_dec)  
  
if u1 == u\_dec:  
 print("Открытый и расшифрованный текст - совпадают")  
else:  
 print("Открытый и расшифрованный текст - не совпадают!")

Результат:



Задание 1.2

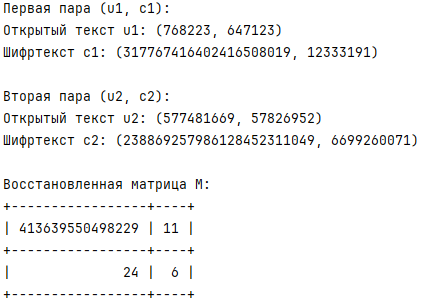
Выполнить криптоанализ по открытому тексту. Пусть известны две пары (u1, c1), (u2, c2), каждая из которых содержит два символа открытого текста и два символа соответствующего шифртекста (одну из них возьмите как вычисленную выше, а вторую вычислите аналогично для другого открытого текста). Восстановите матрицу M (составьте систему из четырех уравнений с неизвестными элементами матрицы, решив эту систему убедитесь, что решение составляют элементы Вашей матрицы M).

1. Провести криптоанализ

Для проведения криптоанализа и восстановления матрицы была составлена и решена система уравнений. Функция crypto\_analysis собирает значения элементов уравнений и передаёт их в функцию solve\_mod\_system, в которой решаются сами уравнения.

def solve\_mod\_system(a1, b1, c1, a2, b2, c2, p):  
 *"""  
 Решает систему двух линейных уравнений по модулю p:  
 a1 \* x + b1 \* y ≡ c1 (mod p),  
 a2 \* x + b2 \* y ≡ c2 (mod p).  
 Возвращает кортеж (x, y).  
 """  
 # Вычисляем определитель системы* det = (a1 \* b2 - a2 \* b1) % p  
 if det == 0:  
 raise ValueError("Система не имеет единственного решения")  
  
 *# Находим обратный элемент для определителя* det\_inv = mod\_inverse\_euclidean(det, p)  
 if det\_inv is None:  
 raise ValueError("Определитель необратим по модулю p")  
  
 *# Решаем систему* x = (det\_inv \* (c1 \* b2 - c2 \* b1)) % p  
 y = (det\_inv \* (a1 \* c2 - a2 \* c1)) % p  
  
 return x, y  
  
  
def crypto\_analysis(u1, c1, u2, c2, p):  
 *# Распаковываем пары* x11, x12 = u1  
 y11, y12 = c1  
 x21, x22 = u2  
 y21, y22 = c2  
  
 *# Решаем первую подсистему для m11 и m21* m11, m21 = solve\_mod\_system(  
 x11, x12, y11, *# Коэффициенты первого уравнения* x21, x22, y21, *# Коэффициенты второго уравнения* p *# Модуль* )  
  
 *# Решаем вторую подсистему для m12 и m22* m12, m22 = solve\_mod\_system(  
 x11, x12, y12, *# Коэффициенты первого уравнения* x21, x22, y22, *# Коэффициенты второго уравнения* p *# Модуль* )  
  
 *# Возвращаем восстановленную матрицу M* return [[m11, m12], [m21, m22]]  
  
  
*# Генерируем вторую пару (u2, c2)*u2 = (random.randint(10 \*\* 3, 10 \*\* 9), random.randint(10 \*\* 3, 10 \*\* 9))  
c2 = encode(u2, matrix\_M, p)  
  
print("Первая пара (u1, c1):")  
print("Открытый текст u1:", u1)  
print("Шифртекст c1:", c1)  
print()  
print("Вторая пара (u2, c2):")  
print("Открытый текст u2:", u2)  
print("Шифртекст c2:", c2)  
print()  
  
*# Выполняем криптоанализ*recovered\_matrix = crypto\_analysis(u1, c1, u2, c2, p)  
print\_matrix(recovered\_matrix, "Восстановленная матрица M:")

Результат:



1. Проверить, что восстановленная матрица соответствует исходной

Для проверки соответствия был использован оператор равенства:

if recovered\_matrix == matrix\_M:  
 print("Восстановленная матрица совпадает с исходной!")  
else:  
 print("Ошибка: восстановленная матрица не совпадает с исходной.")

Результат:



Задание 1.3

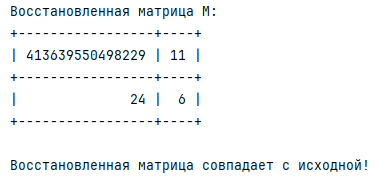
Осуществите атаку по выбираемому тексту.

1. Осуществить атаку по открытому тексту

Были зашифрованы два текста, на основе чего была восстановлена исходная матрица:

u1 = (0, 1)  
c1 = encode(u1, matrix\_M, p)  
u2 = (1, 0)  
c2 = encode(u2, matrix\_M, p)  
  
recovered\_matrix = [list(c2), list(c1)]  
print\_matrix(recovered\_matrix, "Восстановленная матрица M:")  
  
if recovered\_matrix == matrix\_M:  
 print("Восстановленная матрица совпадает с исходной!")  
else:  
 print("Ошибка: восстановленная матрица не совпадает с исходной.")

Результат:

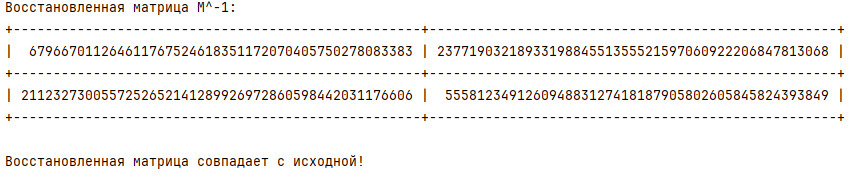


1. Осуществить атаку по Шифр тексту

Были расшифрованы два текста, на основе чего была восстановлена обратная матрица:

u1 = (0, 1)  
c1 = encode(u1, matrix\_inv, p)  
u2 = (1, 0)  
c2 = encode(u2, matrix\_inv, p)  
  
recovered\_matrix = [list(c2), list(c1)]  
print\_matrix(recovered\_matrix, "Восстановленная матрица M^-1:")  
  
if recovered\_matrix == matrix\_inv:  
 print("Восстановленная матрица совпадает с исходной!")  
else:  
 print("Ошибка: восстановленная матрица не совпадает с исходной.")

Результат:



Задание 1.4

По известному шифртексту восстановите неизвестный смысловой открытый текст, зашифрованный аффинным шифром: yi=axi+b mod 33, (a,33=1).

1. Общий код

Был написан перебор всех допустимых значений a и b. Для расшифровки использовалась формула (a\_inv \* (y - b)) % n, где a\_inv – число, обратное числу a, y – символ из шифртекста, n=33.

Для отсеивания некоторых значений в переборе, был составлен список невозможных сочетаний букв в русском языке.

*# Определяем алфавит*alphabet = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"  
n = len(alphabet)  
  
*# Функция для преобразования текста в числовую форму*def text\_to\_numbers(text):  
 return [alphabet.index(char) for char in text]  
  
*# Функция для преобразования чисел в текст*def numbers\_to\_text(numbers):  
 return "".join([alphabet[num] for num in numbers])  
  
*# Расширенный алгоритм Евклида для нахождения обратного элемента*def extended\_gcd(a, b):  
 if b == 0:  
 return a, 1, 0  
 else:  
 g, x, y = extended\_gcd(b, a % b)  
 return g, y, x - (a // b) \* y  
  
  
def mod\_inverse\_euclidean(a, n):  
 g, x, y = extended\_gcd(a, n)  
 if g != 1:  
 return None *# Обратного элемента не существует* return x % n *# Приводим результат к положительному значению  
  
# Функция расшифровки*def decrypt\_affine(ciphertext, a, b):  
 a\_inv = mod\_inverse\_euclidean(a, n)  
 if a\_inv is None:  
 raise ValueError("Обратный элемент для 'a' не существует")  
  
 ciphertext\_numbers = text\_to\_numbers(ciphertext)  
 plaintext\_numbers = [(a\_inv \* (y - b)) % n for y in ciphertext\_numbers]  
 return numbers\_to\_text(plaintext\_numbers)  
  
  
def check\_combinations(combs, text):  
 for c in combs:  
 if c in text:  
 return False  
 return True  
  
  
*# Подбор ключей a и b*def brute\_force\_decrypt(ciphertext):  
 impossible\_combinations = [n + "Ь" for n in "АГЕЁИЙОУЦЪЫЬЭЮЯ"]  
 impossible\_combinations += ["Ь" + n for n in "АЙЪЫЬЭ"]  
 impossible\_combinations += [n + "Ъ" for n in "АЕЁИЙОУЪЫЬЭЮЯ"]  
 impossible\_combinations += ["Ъ" + n for n in "АБГДЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭ"]  
 impossible\_combinations += [n + "Й" for n in "БВГДЖЗИЙКЛМНПРСТФХЦЧШЩЪЬ"]  
 for a in [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 23, 25, 26, 28, 29, 31, 32]:  
 for b in range(n):  
 try:  
 plaintext = decrypt\_affine(ciphertext, a, b)  
 if check\_combinations(impossible\_combinations, plaintext):  
 print(f"a = {a}, b = {b}: {plaintext}")  
 except ValueError:  
 continue

1. Первый текст

Был запущен перебор первого текста:

*# Запускаем подбор для первого шифртекста*ciphertext\_1 = "ШЭКЧРЧЦБКЭДЦНЦНЭЦБЧЪЗОЭЪЭШЭСКХ"  
print("Подбор ключей для первого шифртекста:")  
brute\_force\_decrypt(ciphertext\_1)

Осмысленный текст найден на a=10, b=6:



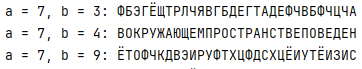
Текст гласит «ШИРОКОЕ ПРИМЕНЕНИЕ ПОЛУЧИЛИ ШИФРЫ»

1. Второй текст

Был запущен перебор второго текста:

*# Запускаем подбор для второго шифртекста*ciphertext\_2 = "СЙОЧЛУДЦФЁЬРЧЙЮЕЧДГЮЕСЁРЙСЁЯЁГ"  
print("\nПодбор ключей для второго шифртекста:")  
brute\_force\_decrypt(ciphertext\_2)

Результат был найден при a=7, b=4:



Текст гласит: «В ОКРУЖАЮЩЕМ ПРОСТРАНСТВЕ ПОВЕДЕН…»

Полный код программ доступен на Github:

Задания 1.1-1.3: <https://github.com/Zaicol/CryptoMPEI/blob/master/task_1.py>

Задание 1.4: <https://github.com/Zaicol/CryptoMPEI/blob/master/task_1.4.py>