**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**ОТЧЕТ**

по лабораторным работам

учебной дисциплины

«Математические и компьютерные основы защиты информации»

Вариант № 7

**Выполнила:**

Зуйкевич Лидия Анатольевна,

3 курс 7а группа, специальность «прикладная математика»

**Преподаватель:**

Лобач Сергей Викторович,

ассистент кафедры ММАД ФПМИ

Минск, 2023

Оглавление

[1. Лабораторная работа №1 «Классические шифры» 3](#_Toc122396302)

[Задание 3](#_Toc122396303)

[Теоретические сведения 3](#_Toc122396303)

[Листинг программы 4](#_Toc122396303)

[Результаты 6](#_Toc122396303)

[2. Лабораторная работа №2 «Блочно-итерационные криптосистемы» 6](#_Toc122396304)

[Задание 6](#_Toc122396303)

[Теоретические сведения 8](#_Toc122396303)

[Листинг программы 8](#_Toc122396303)

[Результаты 10](#_Toc122396303)

[3. Лабораторная работа №3 «Поточные криптосистемы» 10](#_Toc122396306)

[Задание 10](#_Toc122396303)

[Теоретические сведения 11](#_Toc122396303)

[Листинг программы 12](#_Toc122396303)

[Результаты 13](#_Toc122396303)

[4. Лабораторная работа №4 «Криптосистема RSA» 14](#_Toc122396308)

[Задание 14](#_Toc122396303)

[Теоретические сведения 14](#_Toc122396303)

[Листинг программы 15](#_Toc122396303)

[Результаты 16](#_Toc122396303)

[5. Лабораторная работа №5 «Схема Эль-Гамаля и хэш-функция SHA-2» 16](#_Toc122396308)

[Задание 16](#_Toc122396303)

[Теоретические сведения 17](#_Toc122396303)

[Листинг программы 18](#_Toc122396303)

[Результаты 20](#_Toc122396303)

Лабораторная работа № 1

«Классические шифры»

**Задание**

Необходимо реализовать зашифрование и расшифрование двумя классическими шифрами согласно своему варианту.

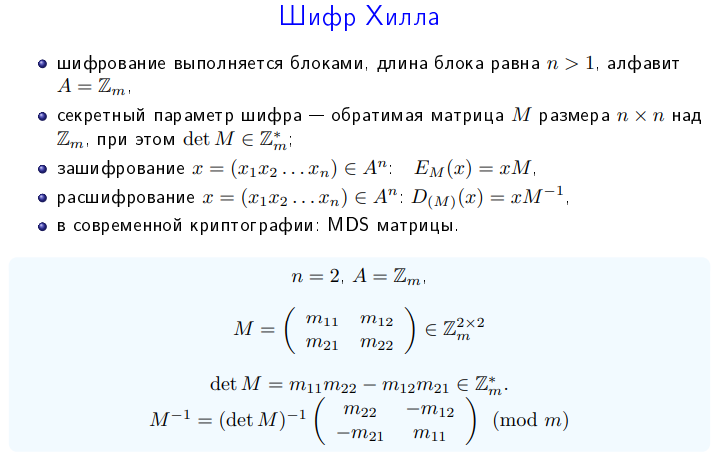
Зашифровать текст:

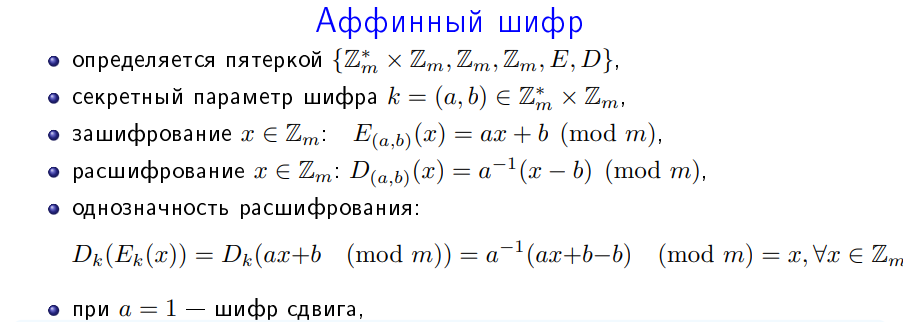
|  |  |  |
| --- | --- | --- |
| Шифр | Ключ | Текст |
| Хилла | 6 14 2 7 | статистика |

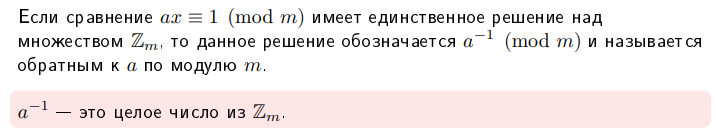
Расшифровать текст:

|  |  |  |
| --- | --- | --- |
| Шифр | Ключ | Текст |
| Аффинный | 17 27 | ЛТМЯВФТДУ |

**Теоретические сведения**







Для нахождения обратного по модулю числа использован расширенный алгоритм Евклида.

**Листинг программы**

HillCipher.py

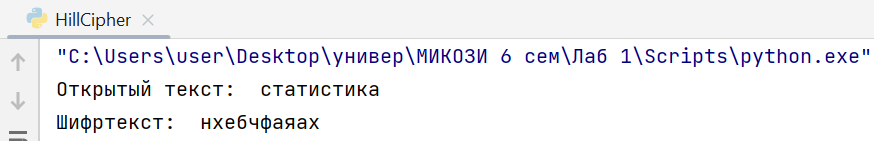
import numpy as np  
  
  
def create\_alphabet():  
 alphabet = []  
 for i in range(ord('а'), ord('е') + 1):  
 alphabet += chr(i)  
 alphabet += 'ё'  
 for i in range(ord('ж'), ord('я') + 1):  
 alphabet += chr(i)  
 return alphabet  
  
  
def get\_plain\_text():  
 str = "статистика"  
 *# str = input("Введите открытый текст, слово должно содержать четное количество букв: ")* print("Открытый текст: ", str)  
 plain\_text = list(str)  
 return plain\_text  
  
  
def get\_plain\_text\_codes(plain\_text, alphabet):  
 text\_len = len(plain\_text)  
 plain\_text\_codes = [0] \* text\_len  
 for i in range(text\_len):  
 plain\_text\_codes[i] = alphabet.index(plain\_text[i])  
 return plain\_text\_codes  
  
  
def encrypt(plain\_text\_codes, alphabet):  
 m = 33  
 text\_len = len(plain\_text\_codes)  
 key = [[6, 14], [2, 7]]  
 *# print("Введите ключ: обратимую по модулю 33 матрицу 2х2: ")  
 # key = []  
 # for i in range(2):  
 # row = input().split()  
 # for i in range(2):  
 # row[i] = int(row[i])  
 # key.append(row)* ciphertext\_codes = []  
 if text\_len % 2 == 0:  
 i = 0  
 while i < text\_len:  
 ciphertext\_codes[i:i+2] = np.dot(plain\_text\_codes[i:i+2], key)  
 i += 2  
 else:  
 print("Нечетное количество букв")  
 return  
  
 answer = ""  
 for i in range(len(ciphertext\_codes)):  
 answer += alphabet[ciphertext\_codes[i] % m]  
 print("Шифртекст: ", answer)  
  
  
alphabet = create\_alphabet()  
plain\_text = get\_plain\_text()  
plain\_text\_codes = get\_plain\_text\_codes(plain\_text, alphabet)  
encrypt(plain\_text\_codes, alphabet)

AffineCipher.py

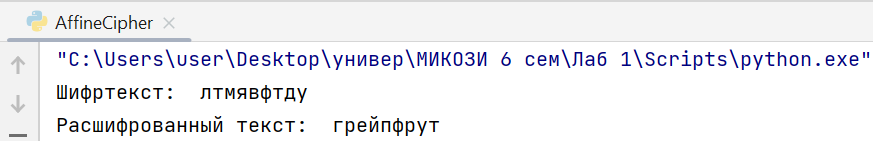
def create\_alphabet():  
 alphabet = []  
 for i in range(ord('а'), ord('е') + 1):  
 alphabet += chr(i)  
 alphabet += 'ё'  
 for i in range(ord('ж'), ord('я') + 1):  
 alphabet += chr(i)  
 return alphabet  
  
  
def gcd\_extended(a, b):  
 if a == 0:  
 return b, 0, 1  
 gcd, x1, y1 = gcd\_extended(b % a, a)  
 x = y1 - (b // a) \* x1  
 y = x1  
 return gcd, x, y  
  
  
def find\_inverse(a, m):  
 gcd, x, y = gcd\_extended(a, m)  
 if gcd == 1:  
 return (x % m + m) % m  
  
  
def get\_ciphertext(alphabet):  
 str = "лтмявфтду"  
 *# str = input("Введите шифртекст: ")* print("Шифртекст: ", str)  
 ciphertext = list(str)  
 ciphertext\_codes = [0] \* len(ciphertext)  
 for i in range(len(ciphertext)):  
 ciphertext\_codes[i] = alphabet.index(ciphertext[i])  
 return ciphertext\_codes  
  
  
def decrypt(ciphertext\_codes, a, b, alphabet):  
 m = 33  
 inv = find\_inverse(a, m)  
 answer = ""  
 for i in range(len(ciphertext\_codes)):  
 answer += alphabet[(inv \* (ciphertext\_codes[i] - b)) % m]  
 print("Расшифрованный текст: ", answer)  
  
  
alphabet = create\_alphabet()  
ciphertext\_codes = get\_ciphertext(alphabet)  
decrypt(ciphertext\_codes, 17, 27, alphabet)

**Результаты**

Задание 1. Шифр Хилла



Задание 2. Аффинный шифр



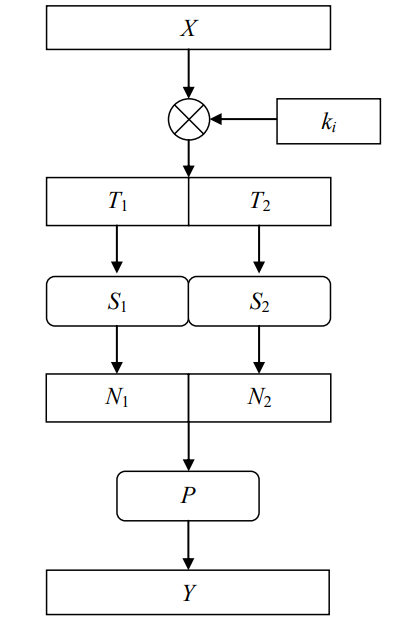
Лабораторная работа № 2

«Блочно-итерационные криптосистемы»

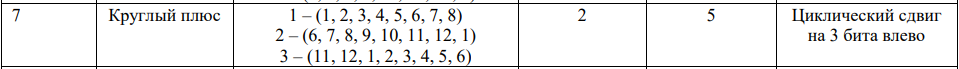
**Задание**

1. SP-подстановка.

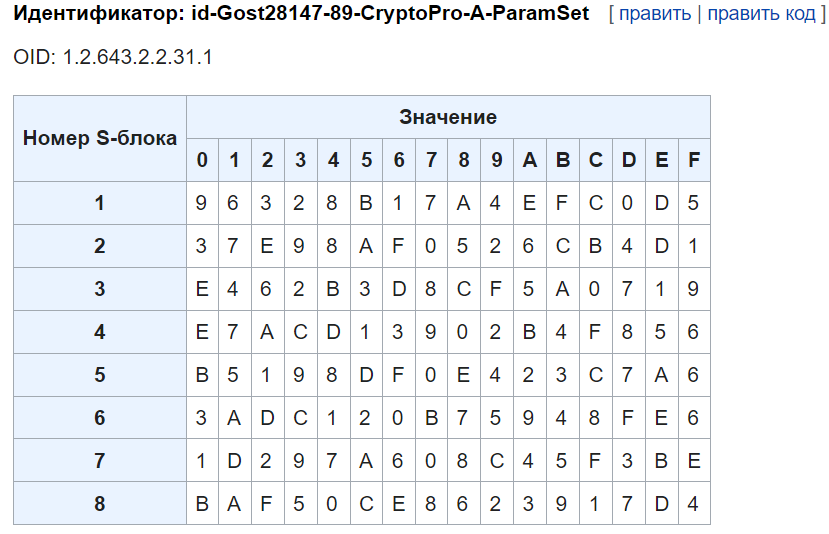
Необходимо реализовать (программно или вручную) блочно-итерационных шифр, состоящий из применения 3 итераций следующей упрощенной SP-подстановки:

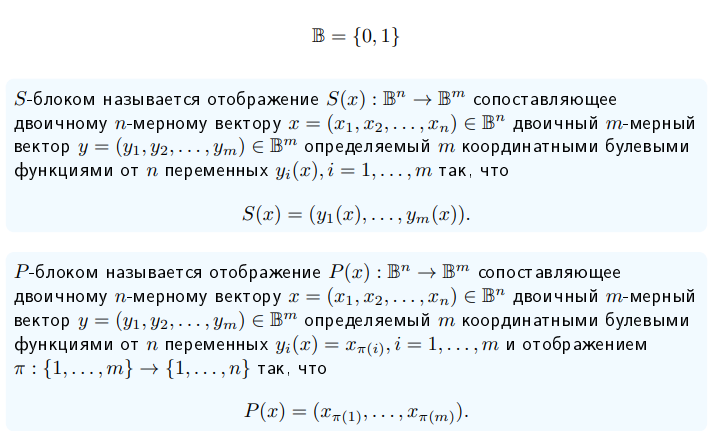


На вход поступает сообщение длины 8 бит, исходный ключ имеет длину 12 бит. Это сообщение «складывается» с раундовым ключом. Вид «сложения» и способ генерации раундовых ключей будет указан в описании варианта. Результат «сложения» разбивается на 2 блока по 4 бита каждый: . Каждый из получившихся блоков поступает на вход соответствующего S-блока. Сами S-блоки будут заданы в варианте задания. Результаты применения S-блоков снова собираются в один блок из 8 бит, к которому применяется операция перестановки бит (P-блок). P-блок будет задан в каждом варианте. Выход P-блока – это результирующий выход SP-подстановки. 2. Сначала X выбирается как , где N – номер студента в списке группы, запись означает, что число необходимо представить в двоичной системе счисления. Ключ k выбирается, как , где q – количество символов в имени, r – количество символов в фамилии. Для указанного сообщения X найти результат зашифрования. При этом выводить не только окончательный результат, но и промежуточные результаты после 1-ой и 2-ой итерации. Затем заменить один бит сообщения (любой на выбор). Провести зашифрование и посмотреть насколько сильно изменился результат на каждой из итераций (лавинный эффект). Поскольку параметры вариантов специально не выбирались, то лавинный эффект может не наблюдаться.



**Теоретические сведения**





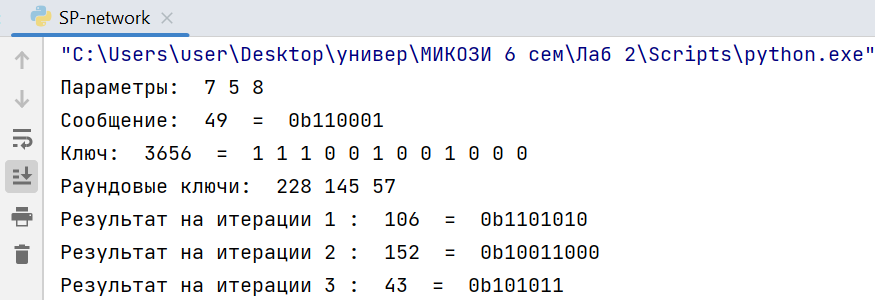
Круглый плюс – сложение по модулю 2.

**Листинг программы**

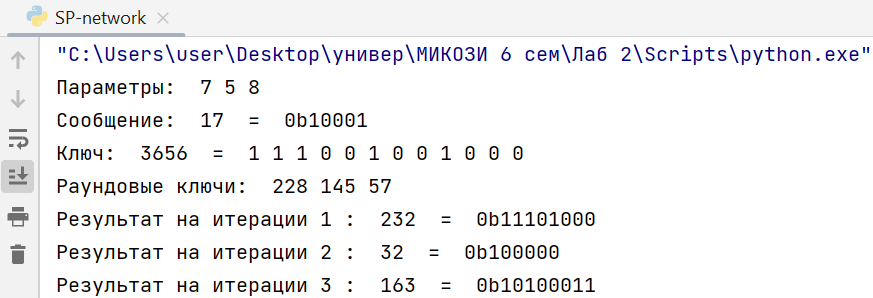
def get\_message\_and\_key():  
 *#n, q, r = input("Введите номер студента в списке, количество символов имени, количество символов фамилии: ").split(' ')  
 #n = int(n)  
 #q = int(q)  
 #r = int(r)* n = 7  
 q = 5  
 r = 8  
 print("Параметры: ", n, q, r)  
 x = 7 \* n  
 k = 4096 - 11 \* q \* r  
 *#message = x* message = 0b00110001  
 cipher\_key = to\_binary(k, 12)  
 print("Сообщение: ", message, " = ", bin(message))  
 print("Ключ: ", k, " = ", \*cipher\_key)  
 return message, cipher\_key,  
  
  
def key\_expansion(cipher\_key):  
 k1 = [1, 2, 3, 4, 5, 6, 7, 8]  
 k2 = [6, 7, 8, 9, 10, 11, 12, 1]  
 k3 = [11, 12, 1, 2, 3, 4, 5, 6]  
  
 round\_keys = [0] \* 3  
 fact = 1  
 i = 7  
 while i > -1:  
 round\_keys[0] += cipher\_key[k1[i] - 1] \* fact  
 round\_keys[1] += cipher\_key[k2[i] - 1] \* fact  
 round\_keys[2] += cipher\_key[k3[i] - 1] \* fact  
 i -= 1  
 fact \*= 2  
 print("Раундовые ключи: ", \*round\_keys)  
 return round\_keys  
  
  
def encrypt(x, round\_keys):  
 s\_box\_1 = [3, 7, 14, 9, 8, 10, 15, 0, 5, 2, 6, 12, 11, 4, 13, 1]  
 s\_box\_2 = [11, 5, 1, 9, 8, 13, 15, 0, 14, 4, 2, 3, 12, 7, 10, 6]  
  
 result = x  
 for i in range(3):  
 t = result ^ round\_keys[i]  
 t1 = t // 16  
 t2 = t % 16  
  
 n1 = s\_box\_1[t1]  
 n2 = s\_box\_2[t2]  
 result = p\_box(n1, n2)  
 print("Результат на итерации", i + 1, ": ", result, " = ", bin(result))  
  
  
def p\_box(n1, n2):  
 result = (n1 % 2) \* 128 + n2 \* 8 + (n1 // 2)  
 return result  
  
  
def to\_binary(num, length):  
 answer = [0] \* length  
 i = length - 1  
 while num > 0:  
 answer[i] = num % 2  
 num //= 2  
 i -= 1  
 return answer  
  
  
message, cipher\_key = get\_message\_and\_key()  
round\_keys = key\_expansion(cipher\_key)  
encrypt(message, round\_keys)

**Результаты**

Результат зашифрования исходного открытого текста:



Результат зашифрования после замены 3 бита исходного открытого текста:



Результаты на каждой итерации при изменении 1одного бита в исходном открытом тексте отличаются на 2 бита. Обычно под лавинным эффектом подразумевают, что при изменении одного бита открытого текста изменяется в среднем половина бит в шифртексте, в данном же случае при изменении 1 бита из 8 на входе мы получили лишь 2 измененных бита на выходе.

Лабораторная работа № 3

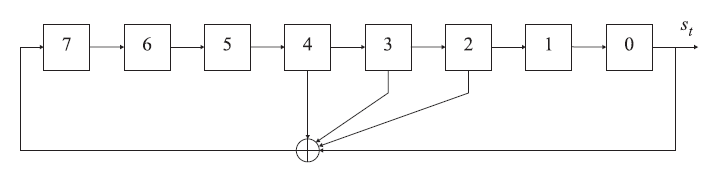
«Поточные криптосистемы»

**Задание**

В рамках данной лабораторной работы вам необходимо реализовать генератор Геффе с заданными характеристическими многочленами и начальными состояниями.

Шаг 1. Реализовать работу РСЛОС. На входе заданы число *n* – количество ячеек памяти, из которых состоит регистр, и две битовые последовательности: 1) *an*-1*an*-2...*a*1*a*0 – задает начальное заполнение, 2) *cn*-1*cn*-2...*c*1*c*0 – задает линейную обратную связь. *a*0 – задает состояние 0-ой ячейки памяти, *a*1 – перовой и т.д., *an*-1 – (*n*-1)-ой, таким образом задаем состояния всех *n* ячеек. Пусть *b*0, *b*1, ..., *bn*-1 – текущие значения в соответствующих ячейках памяти (в начальный момент *bi* = *ai*). За одну итерацию: 1) внешний выводе: регистр возвращает значение, хранящееся в 0-ой ячейке, т.е. *b*0; 2) изменение внутреннего состояния: вычисляется значение *r* = (*c*0\**b*0) + (*c*1\**b*1) +...+ (*cn*-1\**bn*-1); значение *i*-ой ячейки переходит в (*i*-1)-ую(*i*изменяется от 1 до *n*-1), т.е. *bi*-1 = *bi*; значение (*n*-1)-ой принимается равным *bn*-1 = *r*. На рисунке внизу представлен пример, для n = 8 и *c7c6*...*c*1*c*0=00011101.

На первом шаге необходимо реализовать 3 РСЛОС, заданные в вашем варианте. Для каждого регистра найти период выходной последовательности и сгенерированную последовательность, до начала зацикливания.



Шаг 2. Сгенерировать выходную последовательность *γi* генератора Геффе длительностью N = 10 000 элементов.

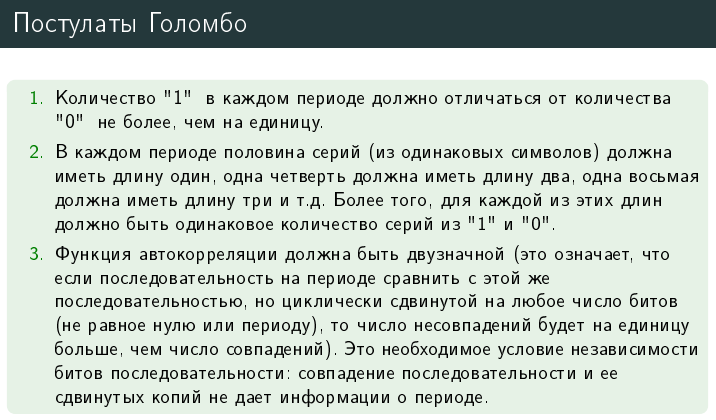
Шаг 3. Для сгенерированной на Шаге 2 последовательности вычислить следующие статистики:  
1) количество 0 и количество 1; 2)  для *i* от 1 до 5. Прокомментировать полученные результаты.

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **РСЛОС №1**  **(5)** | **РСЛОС №2**  **(7)** | **РСЛОС №3**  **(8)** |
| 7 | 00101  10101 | 0111001  1100111 | 10010100  11010011 |

**Теоретические сведения**

*Генератор Геффе.* Этот генератор один из вариантов комбинирующего генератора при . В нем t-ый элемент выходной последовательности может быть вычислен как , где – -ый элемент ЛРП, вырабатываемой -ым РСЛОС. Для достижения максимального периода выходной последовательности, длины РСЛОС должны быть попарно взаимно-простыми числами. Но даже в этом случае данный генератор уязвим к корреляционным атакам.

*Тестирование последовательностей.* Для проверки статистических свойств последовательностей, которые вырабатывают аппаратные и программные генераторы псевдослучайных чисел используются пакеты (батареи) статистических тестов. Один из самых известных пакетов – пакет тестов [NIST](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%8B_NIST).



**Листинг программы**

*# Зуйкевич Лидия, 3 курс, 7 группа  
# Лабораторная работа 3, вариант 7. Поточные криптосистемы. РЛОРС, генератор Геффе*

def lfsr(n, a, c):  
 print("Входные данные РСЛОС:")  
 print("a:", bin(a))  
 print("c:", bin(c))  
 answer = a  
 T = 0  
 s = [0]\*(pow(2, n) - 1)  
 while (answer != a or T == 0):  
 b = answer & c  
 s[T] = answer % 2  
 i = n  
 r = b & 1  
 for i in range (1, n):  
 r = r ^ (b & (1 << i)) >> i  
 answer = (answer >> 1) + (r << (n - 1))  
 T += 1  
 print("Период:", T)  
 print("Выходная последовательность:", \*s[:T])  
 return s, T  
  
def geffe\_generator(num, n1, a1, c1, n2, a2, c2, n3, a3, c3):  
 s1, T1 = lfsr(n1, a1, c1)  
 s2, T2 = lfsr(n2, a2, c2)  
 s3, T3 = lfsr(n3, a3, c3)  
  
 gamma = [0] \* num  
 gamma[0] = (s1[0] & s2[0]) ^ ((s1[0] ^ 1) & s3[0])  
 num\_of\_1 = gamma[0]  
 r = [0] \* 5  
 for i in range(1, num):  
 gamma[i] = (s1[i % T1] & s2[i % T2]) ^ ((s1[i % T1] ^ 1) & s3[i % T3])  
 num\_of\_1 += gamma[i]  
 r[0] += pow(-1, (gamma[i] ^ gamma[i - 1]))  
 for i in range(2, 6):  
 j = 0  
 while (j < num - i):  
 r[i - 1] += pow(-1, (gamma[j] ^ gamma[j + i]))  
 j += 1  
 num\_of\_0 = num - num\_of\_1  
 print()  
 print("Последовательность генератора Геффе:")  
 print(\*gamma)  
 print("Количество единиц:", num\_of\_1)  
 print("Количество нулей:", num\_of\_0)  
 for i in range(0, 5):  
 print("r", i + 1, "=", r[i])  
  
  
geffe\_generator(10000, 5, 0b00101, 0b10101, 7, 0b0111001, 0b1100111, 8, 0b10010100, 0b11010011)

**Результаты**

Входные данные РСЛОС:

a: 0b111001

c: 0b1100111

Период: 5

Выходная последовательность: 1 0 0 1 1

Входные данные РСЛОС:

a: 0b10010100

c: 0b11010011

Период: 62

Выходная последовательность: 0 0 1 0 1 0 0 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 1 1

Последовательность генератора Геффе (указаны только первые элементы):

1 0 0 0 1 0 0 0 1 1 1 0 0 0 1 1 1 0 1 1 0 1 0 1 1 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1 … Количество единиц: 4671

Количество нулей: 5329

r 1 = 759

r 2 = -2092

r 3 = -1917

r 4 = -764

r 5 = 2089

В сгенерированной последовательности 4671 единиц и 5329 нулей, т.е. их количество отличается на 658, первый постулат Голомба не выполняется. Значения статистики r, являющейся автокорреляционной функцией из 3 постулата Голомба, не являются двухзначными ни для одного из значений *i,* т.е. между битами последовательности есть зависимость, что плохо, поскольку по совпадению последовательности и ее сдвинутых копий будет возможно найти ее период.

Лабораторная работа № 4

«Криптосистема RSA»

**Задание**

0) Для выполнения всех заданий необходимо использовать "длинную" арифметику. Разрешается использовать любую библиотеку для этих целей или написать свою.

1) В условиях своего варианта для заданных чисел *p*, *q* и *e*, необходимо вычислить *d*. **Важно**: Использовать готовую реализацию расширенного алгоритма Евклида или вычисления обратного по модулю элемента нельзя! (Можно только для проверки своих результатов).

2) Для заданного сообщения *X*1, вычислить зашифрованное сообщение *Y*1, используя открытый ключ *e*.

3) Расшифровать сообщение *Y*1, использую закрытый ключ *d*, сравнить результат с исходным сообщением *X*1.

4) Для заданного шифртекста *Y*2, вычислить исходный открытый текст *X*2, используя закрытый ключ *d*.

| № Вар. | *p* | *q* | | *e* | |
| --- | --- | --- | --- | --- | --- |
| 7 | 578569278720973 | 976534805568533 | | 235108486320061234453015373083 | |
| *X*1 | | | *Y*2 | |
| 402700874043636335474593885222 | | | 416593343738152120120255791792 | |

**Теоретические сведения**

Формально как и для любой ассиметричной криптосистемы, для работы RSA необходимо реализовать 3 основные процедуры: генерацию ключей, функцию зашифрования и функцию расшифрования.

Генерация ключей:

1. Выбираются два больших простых числа *p* и *q*.

2. Вычисляется число *n* = *pq*; *n* – открытый параметр.

3. Открытый ключ *e* выбирается случайным образом из Zϕ(*n*) = {0, ..., ϕ(*n*)-1} таким образом, чтобы НОД(*e*, ϕ(*n*)) = 1. (Зная *p* и *q,* можно легко вычислить ϕ(*n*) = (*p*-1)(*q*-1)).

4. Закрытый ключ *d* из Zϕ(*n*); находится по формуле *d* = *e*-1 (mod ϕ(*n*)).

Замечание: Проверить, что случайно выбранное число *e* удовлетворяет условию НОД(*e*, ϕ(*n*)) = 1, и, если условие выполнено, найти закрытый ключ *d* = *e*-1 (mod ϕ(*n*)) можно с помощью расширенного алгоритма Евклида.

Сообщение *X*, которое необходимо зашифровать, представляется в виде некоторого числа из Z*n*.

Функция зашифрования: E*e*(*X*) = *Xe* (mod *n*), где *n* – открытый параметр, *e* – открытый ключ.

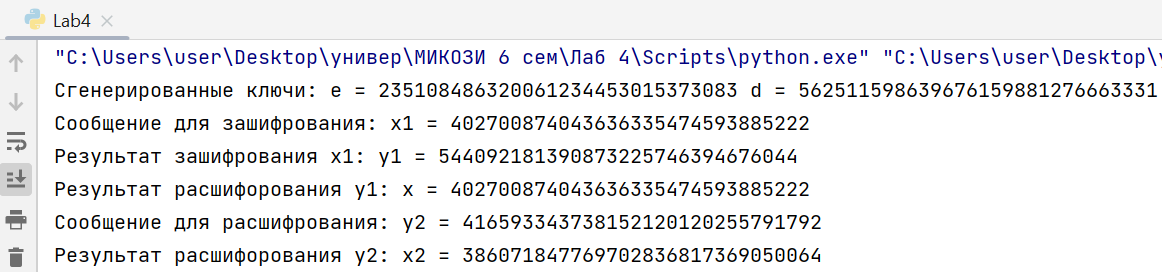
Замечание: Для возведения в степень рекомендуется использовать алгоритм быстрого возведения в степень. Вычислять остаток от деления необходимо после каждой операции умножения.

Функция расшифрования: D*d*(*Y*) = *Yd* (mod *n*), где *n* – открытый параметр, *d* – закрытый ключ.

**Листинг программы**

*# Зуйкевич Лидия, 3 курс, 7 группа  
# Лабораторная работа 4, вариант 7. Криптосистема RSA*def gcd\_extended(a, b):  
 if a == 0:  
 return b, 0, 1  
 gcd, x1, y1 = gcd\_extended(b % a, a)  
 x = y1 - (b // a) \* x1  
 y = x1  
 return gcd, x, y  
  
def find\_inverse(a, m):  
 gcd, x, y = gcd\_extended(a, m)  
 if gcd == 1:  
 return (x % m + m) % m  
  
  
def fast\_exponentation(base, exp, n):  
 res = 1  
 while(exp > 0):  
 if (exp & 1 == 1):  
 res = (res \* base) % n  
 base = (base \* base) % n  
 exp >>= 1  
 return res  
  
def generate\_keys(p, q, e):  
 n = p \* q  
 phi = (p - 1) \* (q - 1)  
 gcd, x, y = gcd\_extended(e, phi)  
 if gcd == 1:  
 d = find\_inverse(e, phi)  
 else:  
 print("gcd(e, phi) is not equal to 1")  
 return n, e, d  
  
  
def encrypt(x, e, n):  
 y = fast\_exponentation(x, e, n)  
 return y  
  
def decrypt(y, d, n):  
 x = fast\_exponentation(y, d, n)  
 return x  
  
  
def task():  
 p = 578569278720973  
 q = 976534805568533  
 e = 235108486320061234453015373083  
 x1 = 402700874043636335474593885222  
 y2 = 416593343738152120120255791792  
 n, e, d = generate\_keys(p, q, e)  
 print("Сгенерированные ключи:", "e =", e, "d =", d)  
 y1 = encrypt(x1, e, n)  
 print("Сообщение для зашифрования: x1 =", x1)  
 print("Результат зашифрования x1: y1 =", y1)  
 x = decrypt(y1, d, n)  
 print("Результат расшифорования y1: x =", x)  
 x2 = encrypt(y2, d, n)  
 print("Сообщение для расшифрования: y2 =", y2)  
 print("Результат расшифорования y2: x2 =", x2)  
  
task()

**Результаты**

****

Лабораторная работа № 5

«Схема Эль-Гамаля и хэш-функция SHA-2»

**Задание**

1) Для выполнения всех заданий необходимо использовать "длинную" арифметику. Разрешается использовать любую готовую библиотеку или написать свою. При этом учитывайте, что допустимые языки программирования {С/С++, C#, Java, Python, R}.

2) Ваша программа должна принимать на вход указание, какую операцию необходимо выполнить: Gen, Sign, Verify или Exit. В зависимости от операции программа должна считывать (из консоли, файла и т.п.) необходимые данные, выполнять шаги выбранного алгоритма и возвращать (в консоль, файл и т.п.) результат выполнения соответствующей операции. При вызове операции Exit программа не считывает входных данных, а просто завершает работу.

3) Программа должна иметь проверку корректности входных данных (d лежит в Zq для Sign; e лежит в Zp для Verify).

4) Программа должна предлагать пользователю выполнить очередную операцию до тех пор, пока пользователь не выберет вариант выхода из программы.

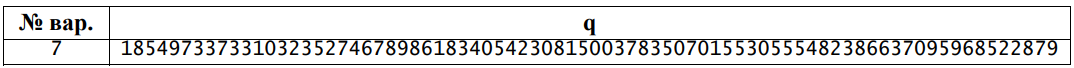
Отчет (условия по вариантам смотрите на последней странице):

1) В условиях своего варианта для заданного числа q необходимо выполнить алгоритм Gen.

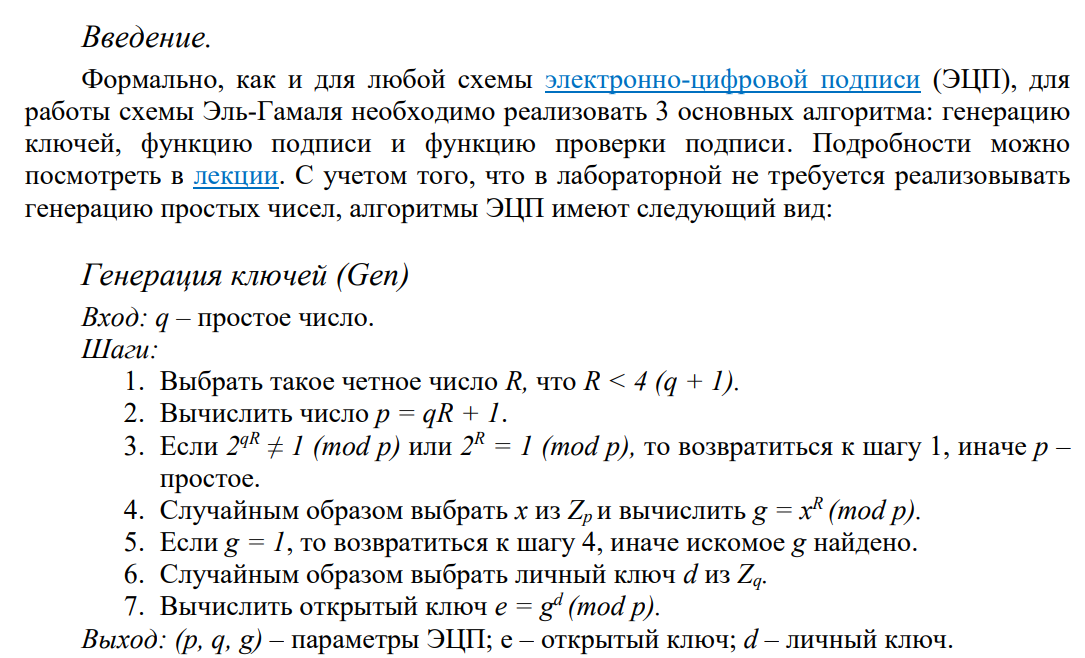
2) Для параметров, полученных с помощью Gen, и сообщения вида «I, Ivan Ivanov, love MiKOZI», где «Ivan Ivanov» – ваше имя и фамилия соответственно, выполните алгоритм Sign.

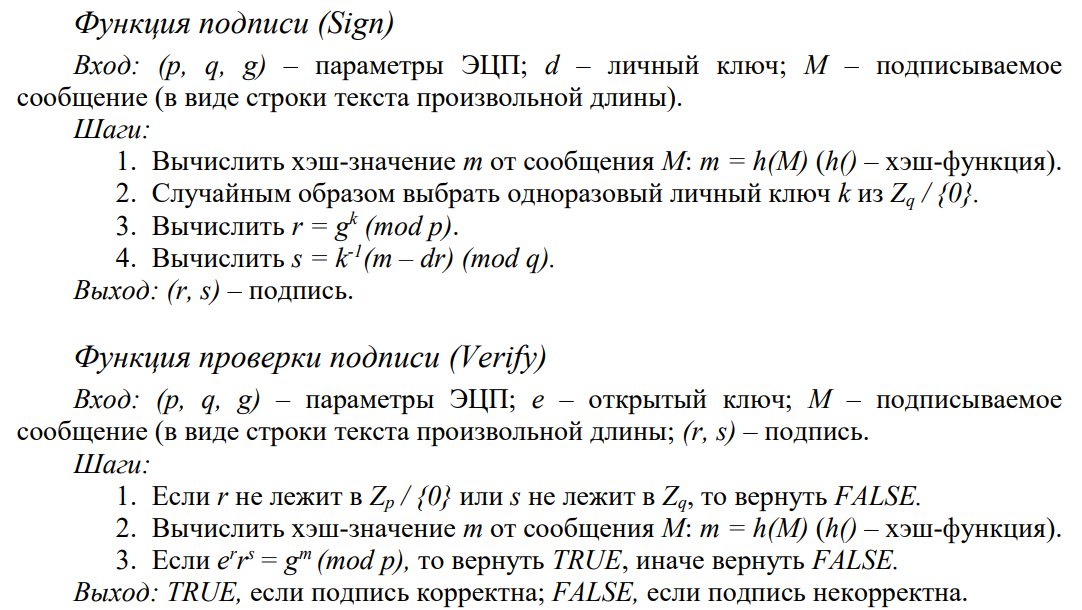
3) Проверьте результаты подписи из 2) пункта, выполнив алгоритм Verify.

4) Результаты шагов 1 – 3 необходимо записать в текстовый файл, назвав его «Report.txt»



**Теоретические сведения**





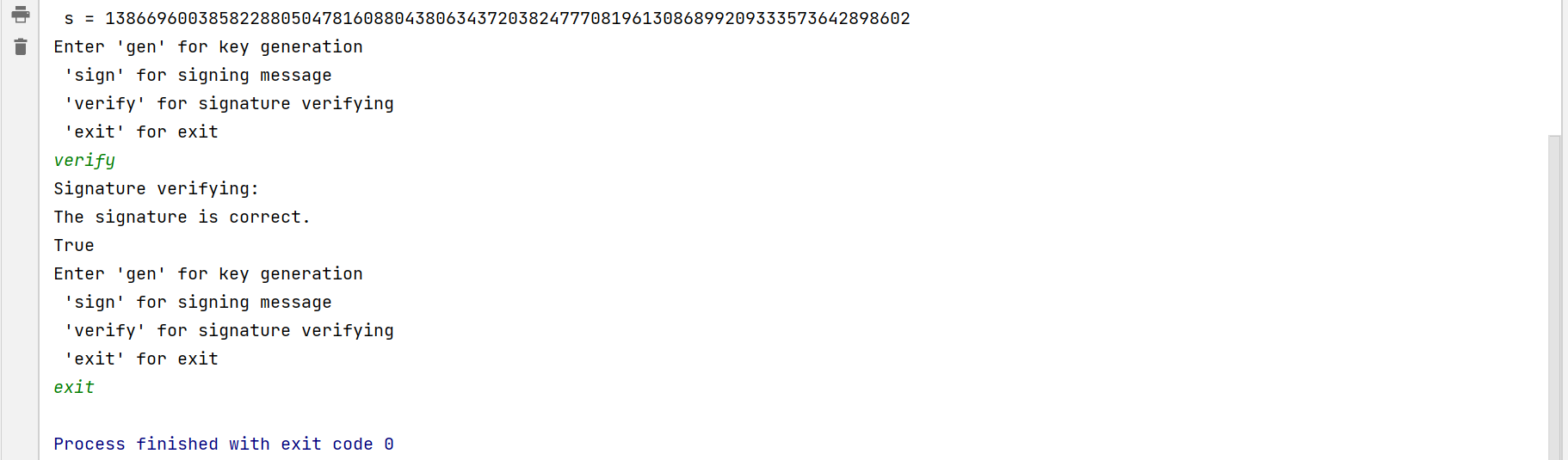
Для возведения чисел в степень используется алгоритм быстрого возведения в степень. Для нахождения обратного по модулю числа используется расширенный алгоритм Евклида. В качестве хэш-функции используется SHA256 (реализация - встроенная функция sha256 языка python).

**Листинг программы**

from random import randrange  
from hashlib import sha256  
  
def fast\_exponentation(base, exp, mod):  
 res = 1  
 while(exp > 0):  
 if (exp & 1 == 1):  
 res = (res \* base) % mod  
 base = (base \* base) % mod  
 exp >>= 1  
 return res  
  
  
def gcd\_extended(a, b):  
 if a == 0:  
 return b, 0, 1  
 gcd, x1, y1 = gcd\_extended(b % a, a)  
 x = y1 - (b // a) \* x1  
 y = x1  
 return gcd, x, y  
  
def find\_inverse(a, m):  
 gcd, x, y = gcd\_extended(a, m)  
 if gcd == 1:  
 return (x % m + m) % m  
  
  
def generate\_keys():  
 q = 185497337331032352746789861834054230815003783507015530555482386637095968522879  
 *#q = int(input("enter q:"))* r\_lim = 4 \* (q + 1) + 2  
 r = randrange(2, r\_lim, 2)  
 p = q \* r + 1  
 while(pow(2, q \* r, p) != 1 or pow(2, r, p) == 1):  
 r = randrange(2, r\_lim, 2)  
 p = q \* r + 1  
  
 x = randrange(0, p)  
 g = fast\_exponentation(x, r, p)  
 while(g == 1):  
 x = randrange(0, p)  
 g = fast\_exponentation(x, r, p)  
  
 d = randrange(0, q)  
 e = fast\_exponentation(g, d, p)  
 print("p =", p, "\n q = ", q, "\n g =", g, "\n e =", e, "\n d =", d)  
 return p, q, g, e, d  
  
  
def sign(p, q, g, d):  
 *#message = input("Please enter a message you would like to sign:")* message = "I, Lidia Zuikevich, love MIKOZI"  
 m = int(sha256(bytes(message, encoding="utf-8")).hexdigest(), 16)  
 k = randrange(1, q)  
 r = fast\_exponentation(g, k, p)  
 s = (find\_inverse(k, q) \* (m - d \* r)) % q  
 print("m =", m, "\n k =", k, "\n r =", r, "\n s =", s)  
 return r, s, m, message  
  
  
def verify(p, q, g, e, message, r, s):  
 if (r < 1 or r > p - 1):  
 print("r is out of range.")  
 return False  
 if (s < 0 or s > q - 1):  
 print("s is out of range.")  
 return False  
 m = int(sha256(bytes(message, encoding="utf-8")).hexdigest(), 16)  
 left = (fast\_exponentation(e, r, p) \* fast\_exponentation(r, s, p)) % p  
 right = fast\_exponentation(g, m, p)  
 if (left == right):  
 print("The signature is correct.")  
 return True  
 else:  
 print("The signature isn't correct.")  
 return False  
  
  
def run():  
 print("Please choose what would you like to do:")  
 p = 0  
 q = 0  
 g = 0  
 e = 0  
 d = 0  
 r = 0  
 s = 0  
 message = ""  
  
 while True:  
 action = input("Enter 'gen' for key generation \n 'sign' for signing message \n 'verify' for signature verifying \n 'exit' for exit\n")  
  
 if (action == "gen"):  
 print("Key generation:")  
 p, q, g, e, d = generate\_keys()  
  
 with open('Report.txt', 'w') as file:  
 file.write("Key generation:\n")  
 file.write(f"p = {p} \nq = {q} \ng = {g} \ne = {e} \nd = {d}\n")  
 continue  
  
  
 if (action == "sign"):  
 if (d != 0):  
 print("Signing message:")  
 r, s, m, message = sign(p, q, g, d,)  
 with open('Report.txt', 'a') as file:  
 file.write("Signing message:\n")  
 file.write(f"m = {m} \nr = {r} \ns = {s}\n")  
 continue  
 else:  
 print("Please do key generation first.")  
  
 if(action == "verify"):  
 if (e != 0 and r != 0):  
 print("Signature verifying:")  
 result = verify(p, q, g, e, message, r, s)  
 print(result)  
 with open('Report.txt', 'a') as file:  
 file.write("Signature verifying:\n")  
 file.write(f"result = {result}\n")  
 continue  
 else:  
 print("Please do key generation and signing first.")  
  
 if(action == "exit"):  
 return  
 else:  
 print("The command is wrong.")  
 continue  
  
  
run()

**Результаты**

****

****

Содержимое файла Report.txt после данного запуска программы:

Key generation:

p = 95601379180448697063138036037861573345915537932509001915917627253841631079565675843873809416914569091755729405734809617711509185520188300786708582029213623

q = 185497337331032352746789861834054230815003783507015530555482386637095968522879

g = 80674008501072461862339295523308776835665994273360113158704205862657478229265246465906388669872675893999298630022664177746838833015378771355252741057847383

e = 51974204220498190800144455300160695373835564797676365092651789304585957012296125944174800334622849172135182290668341431712920255975424140685110473640353018

d = 163786905210058198731145251192198168642581182363934578047456353391464298794919

Signing message:

m = 111119257775310160694651755682725193071482487146149799868697148406819657445410

r = 10288019725662856605841527359630997083064065270241372052427576129179739623205195220307346342601555383569717951701849653592218752403752690899550573462099709

s = 138669600385822880504781608804380634372038247770819613086899209333573642898602

Signature verifying:

result = True