МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра кібербезпеки

**Звіт до лабораторної роботи № 10-11**

на тему “ Програмна реалізація алгоритму RSA. Частина 1-2 ”

Виконав студент(ка) Борщ Дмитро

Група КБ-01

Перевірила Лаврик Т.В

**Суми 2022**

**ЗВІТ 10–11**

Реалізувати програму, яка буде виконувати такі дії:

**Завдання.** (**15 б.)**

1. Згенерувати два простих числа P і Q.

2. Обчислити значення N=P\*Q.

3. Обчислити функцію Ейлера.

4. Випадковим чином обрати відкритий ключ KA, що задовольняє умовам.

5. Згенерувати ключ KB з використанням одного з алгоритмів.

6. Зашифрувати за допомогою алгоритму RSA повідомлення (5 - 20 символів), яке вводиться користувачем з клавіатури.

7. Розшифровувати зашифроване за допомогою алгоритму RSA повідомлення (п. 6).

8. Виводити такі результати роботи програми на екран:

* алфавіт;
* ключі KA і KB;
* P, Q, N ;
* відкритий текст у числовому та символьному вигляді;
* зашифрований текст у числовому та символьному вигляді;
* розшифрований текст у числовому та символьному вигляді.

Програма повинна забезпечувати зручний інтерфейс користувача. Усі дані, що вводяться та виводяться повинні супроводжуватись чіткими та зрозумілими для користувача поясненнями.

Текст програми

**import** **random**

BIT\_SIZE = int(**8**)

first\_primes\_list = [

**2**, **3**, **5**, **7**, **11**, **13**, **17**, **19**, **23**, **29**,

**31**, **37**, **41**, **43**, **47**, **53**, **59**, **61**, **67**,

**71**, **73**, **79**, **83**, **89**, **97**, **101**, **103**,

**107**, **109**, **113**, **127**, **131**, **137**, **139**,

**149**, **151**, **157**, **163**, **167**, **173**, **179**,

**181**, **191**, **193**, **197**, **199**, **211**, **223**,

**227**, **229**, **233**, **239**, **241**, **251**, **257**,

**263**, **269**, **271**, **277**, **281**, **283**, **293**,

**307**, **311**, **313**, **317**, **331**, **337**, **347**, **349**

]

**def** **n\_bit\_random**(n):

**return** random.randrange(**2**\*\*(n-**1**)+**1**, **2**\*\*n - **1**)

**def** **get\_low\_level\_prime**(n):

**while** True:

pc = n\_bit\_random(n)

**for** divisor **in** first\_primes\_list:

**if** pc % divisor == **0** **and** divisor\*\***2** <= pc:

**break**

**else**: **return** pc

**def** **is\_miller\_rabin\_passed**(mrc):

max\_divisions\_by\_two = **0**

ec = mrc-**1**

**while** ec % **2** == **0**:

ec >>= **1**

max\_divisions\_by\_two += **1**

**assert**(**2**\*\*max\_divisions\_by\_two \* ec == mrc-**1**)

**def** **trial\_composite**(round\_tester):

**if** pow(round\_tester, ec, mrc) == **1**:

**return** False

**for** i **in** range(max\_divisions\_by\_two):

**if** pow(round\_tester, **2**\*\*i \* ec, mrc) == mrc-**1**:

**return** False

**return** True

number\_of\_rabin\_trials = **20**

**for** i **in** range(number\_of\_rabin\_trials):

round\_tester = random.randrange(**2**, mrc)

**if** trial\_composite(round\_tester):

**return** False

**return** True

**def** **gcd**(a, b):

**while** b != **0**:

a, b = b, a % b

**return** a

**def** **multiplicative\_inverse**(e, phi):

d = **0**

x1 = **0**

x2 = **1**

y1 = **1**

temp\_phi = phi

**while** e > **0**:

temp1 = temp\_phi/e

temp2 = temp\_phi - temp1 \* e

temp\_phi = e

e = temp2

x = x2- temp1\* x1

y = d - temp1 \* y1

x2 = x1

x1 = x

d = y1

y1 = y

**if** temp\_phi == **1**:

**return** d + phi

**def** **generate\_p\_q**():

p = q = int()

**while** **not** p:

prime\_candidate = get\_low\_level\_prime(BIT\_SIZE)

**if** is\_miller\_rabin\_passed(prime\_candidate): p = prime\_candidate

**while** **not** q:

prime\_candidate = get\_low\_level\_prime(BIT\_SIZE)

**if** is\_miller\_rabin\_passed(prime\_candidate) **and** prime\_candidate != p: q = prime\_candidate

**return** (p, q)

**def** **egcd**(a, b):

**if** a == **0**:

**return** (b, **0**, **1**)

**else**:

g, y, x = egcd(b % a, a)

**return** (g, x - (b // a) \* y, y)

**def** **modinv**(a, m):

g, x, y = egcd(a, m)

**if** g != **1**:

**raise** **Exception**('Модульна інверсія не існує, помилка генерування ключа!')

**else**:

**return** x % m

**def** **generate\_keypair**(p, q):

**if** **not** (is\_miller\_rabin\_passed(p) **and** is\_miller\_rabin\_passed(q)):

**raise** **ValueError**('Обидва числа мають бути простими.')

**elif** p == q:

**raise** **ValueError**('p та q не можуть бути однакові')

n = p \* q

**print**("P = ", p, "**\n**Q = ", q, "**\n**N = ", n)

phi = (p-**1**) \* (q-**1**)

**print**("Фі = ", phi)

e = random.randrange(**1**, phi)

g = gcd(e, phi)

**while** g != **1**:

e = random.randrange(**1**, phi)

g = gcd(e, phi)

d = modinv(e, phi)

**return** ((e, n), (d, n))

**def** **encrypt**(pk, plaintext):

key, n = pk

cipher = [(ord(char) \*\* key) % n **for** char **in** plaintext]

**return** cipher

**def** **decrypt**(pk, ciphertext):

key, n = pk

plain = [chr((char \*\* key) % n) **for** char **in** ciphertext]

**return** ''.join(plain)

**if** \_\_name\_\_ == '\_\_main\_\_':

**print**("RSA Шифрувальник/ Дешифрувальник")

p = int(input("Введіть просте число P (0 для автоматичної генерації): "))

**if** **not** p:

p, q = generate\_p\_q()

**else**:

q = int(input("Введіть просте число Q (не повинно співпадати з числом P): "))

**print**("Генеруємо публічний/приватний пару клоючів . . .")

public, private = generate\_keypair(p, q)

**print**("Ваш публічний ключ", public[**0**] ,"**\n**Та ваш приватний ключ", private[**0**])

message = input("Введіть повідомлення, яке буде зашифроване вашим приватним ключем: ")

encrypted\_msg = encrypt(private, message)

**print**("Зашифроване повідомлення: ")

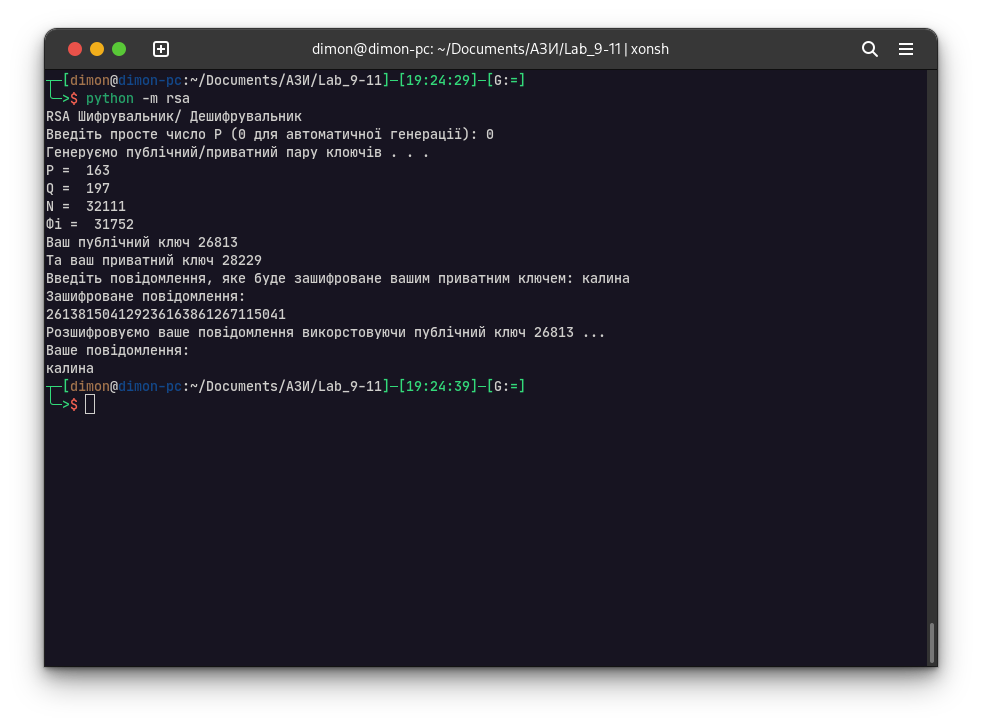
**print**(''.join(map(**lambda** x: str(x), encrypted\_msg)))

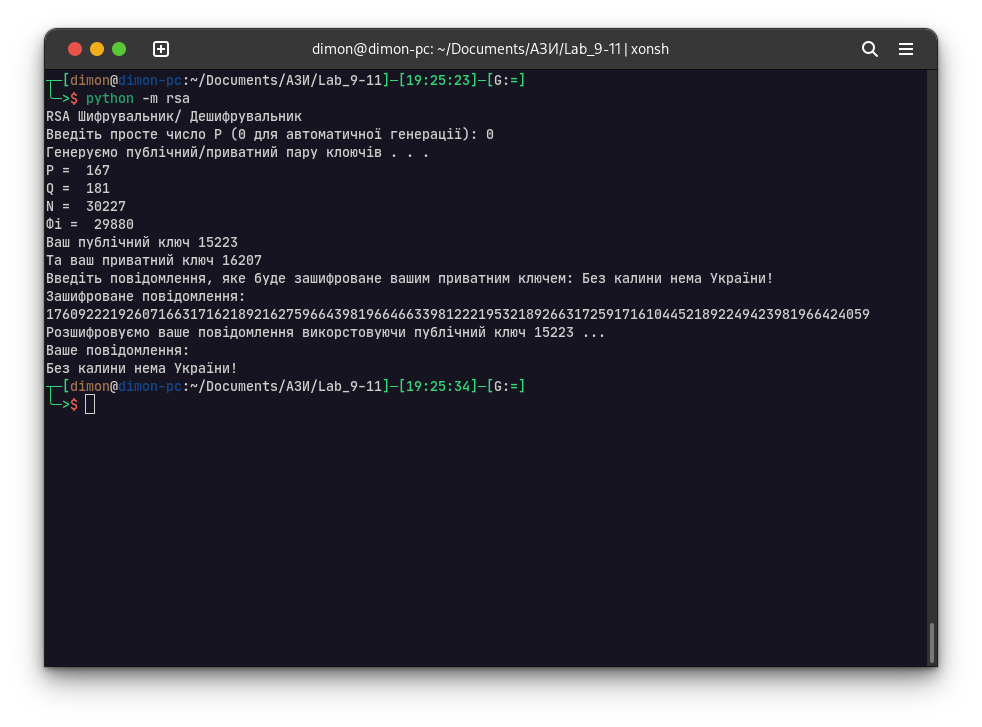
**print**("Розшифровуємо ваше повідомлення викорстовуючи публічний ключ", public[**0**] ,"...")

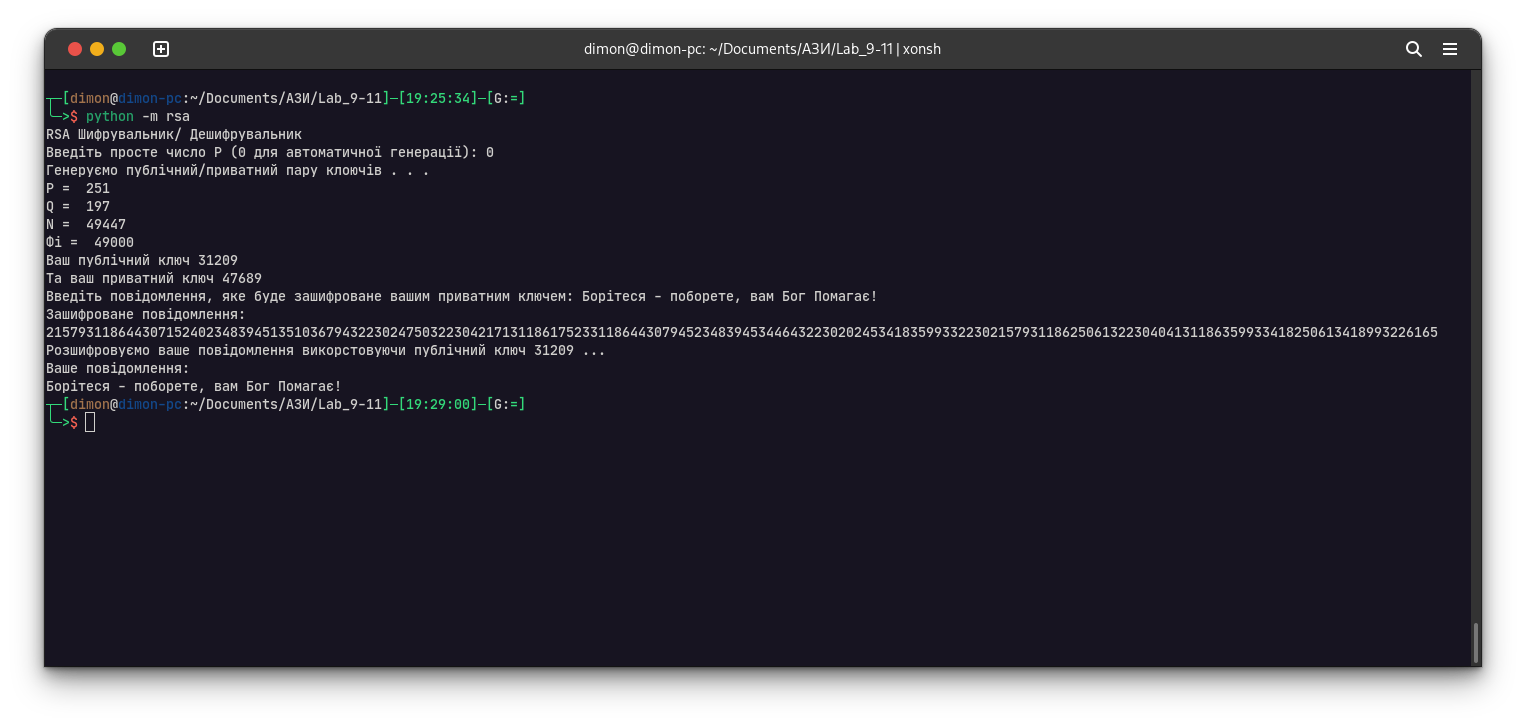
**print**("Ваше повідомлення:")

**print**(decrypt(public, encrypted\_msg))

Скріншоти з результатами роботи програми (мінімум 3 різні тексти)

Рис. 1 — Тест номер 1.

Рис. 2 — Тест номер 2.

Рис. 3 — Тест номер 3.