КОМП’ЮТЕРНИЙ ПРАКТИКУМ №4

Гоголєва Поліна ФБ-12

**Вивчення криптосистеми RSA та алгоритму електронного підпису; ознайомлення з методами генерації параметрів для асиметричних криптосистем**

1. Написати функцію пошуку випадкового простого числа з заданого інтервалу або заданої довжини, використовуючи датчик випадкових чисел та тести перевірки на простоту. В якості датчика випадкових чисел використовуйте вбудований генератор псевдовипадкових чисел вашої мови програмування. В якості тесту перевірки на простоту рекомендовано використовувати тест Міллера-Рабіна із попередніми пробними діленнями.

Як же боляче це було по цим вашим математичним поясненням у методичці… і хоч вже існуюча реалізація тесту Міллера-Рабіна, котру я знайшла в інтернеті чомусь була набагато коротша, я написала свою, котра косо-криво, але працює!

Намагалась робити покроково, як вказано у методичці.

От її код

import random

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

def mod\_exp(x, d, p):

    result = 1

    x = x % p

    while d > 0:

        if d % 2 == 1:

            result = (result \* x) % p

        d = d // 2

        x = (x \* x) % p

    return result

def trial\_division(n):

    if n < 2:

        return False

    for i in range(2, 100):

        if n % i == 0:

            return False

    return True

def miller\_rabin\_test(p, k=10):

    if not trial\_division(p):  # Перевірка на простоту за допомогою решти

        return False

    # Крок 0: Знаходимо розклад p - 1 = d \* 2^S

    d, s = p - 1, 0

    while d % 2 == 0:

        d //= 2

        s += 1

    # Лічильник

    count = 0

    is\_pseudo\_prime = False

    # Крок 3. Повторюємо k разів

    while count < k:

        # Крок 1: Вибираємо випадкове число x

        x = random.randint(2, p - 1)

        # Знаходимо gcd(x, p)

        g = gcd(x, p)

        # Якщо gcd(x, p) > 1, то p – складене число

        if g > 1:

            return False

        else:

        # Крок 2: Перевіряємо, чи є p сильно псевдопростим за основою x

            if mod\_exp(x, d, p) == 1 or mod\_exp(x, d, p) == p - 1:

                is\_pseudo\_prime = True

            else:

                for r in range(1, s):

                    xr = mod\_exp(x, d \* (2 \*\* r), p)

                    if xr == p - 1:

                        is\_pseudo\_prime = True

                        break

                    elif xr == 1:

                        continue

                    else:

                        return False

        if is\_pseudo\_prime:

            count += 1

    return True

# Тестування

p = 2659  # Приклад простого числа

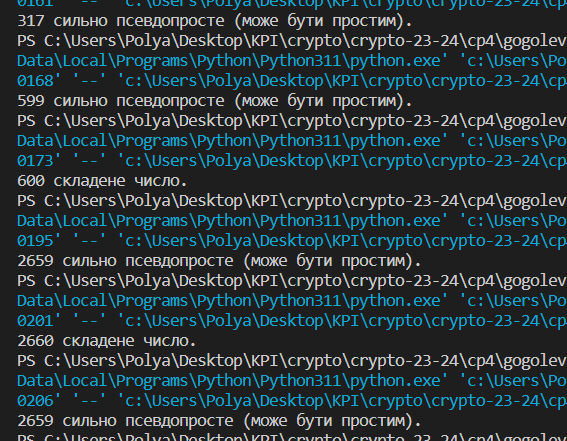
result = miller\_rabin\_test(p)

if result:

    print(f"{p} сильно псевдопросте (може бути простим).")

else:

    print(f"{p} складене число.")



Тепер додамо сюди ф-ю генерації випадкового простого числа з заданого інтервалу.

def generate\_random\_prime(start, end, k=10):

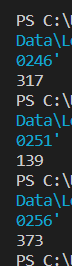
    while True:

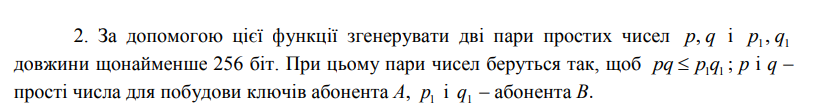
        p = random.randint(start, end)

        if trial\_division(p) and miller\_rabin\_test(p, k):

            return p

print (generate\_random\_prime(80,580))





Згенеруємо 4 рандомних числа, порівняємо їх, двом більшим призначимо p1 та q1, меншим p i q.

def generate\_key\_pairs():

    bit\_length = 256

    # Generate 4 random prime numbers of 256 bits

    primes = [generate\_random\_prime(2\*\*(bit\_length-1), 2\*\*bit\_length - 1) for \_ in range(4)]

    # Sort the primes

    primes.sort()

    p, q = primes[:2]

    p1, q1 = primes[2:]

    return p, q, p1, q1

p, q, p1, q1 = generate\_key\_pairs()

print(f"Alice's public key (p, q): ({p}, {q})")

print(f"Bob's public key (p1, q1): ({p1}, {q1})")

Мій код ранився нескінченно довго і ламався, тож, признаюсь, я закинула його в аішку для дебагінгу та оптимізації, звідки отримала більш оптимізований варіант перевірки теста міллера-рабіна, бо проблема була саме у ньому, ще й працюємо ми з дуже великими числами. Тому тепер код виглядає так:

import random

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

def trial\_division(n):

    if n < 2:

        return False

    for i in range(2, 100):

        if n % i == 0:

            return False

    return True

def miller\_rabin\_test(p, k=10):

    if p < 2:

        return False

    if p != 2 and p % 2 == 0:

        return False

    factorization\_base = p - 1

    while factorization\_base % 2 == 0:

        factorization\_base //= 2

    for \_ in range(k):

        witness = random.randint(1, p - 1)

        if gcd(witness, p) > 1:

            continue

        exponent = factorization\_base

        residue = pow(witness, exponent, p)

        while exponent != p - 1 and residue != 1 and residue != p - 1:

            residue = (residue \* residue) % p

            exponent \*= 2

        if residue != p - 1 and exponent % 2 == 0:

            return False

    return True

''' Тестування

p = 2659  # Приклад простого числа

result = miller\_rabin\_test(p)

if result:

    print(f"{p} сильно псевдопросте (може бути простим).")

else:

    print(f"{p} складене число.") '''

def generate\_random\_prime(start, end, k=10):

    while True:

        p = random.randint(start, end)

        if trial\_division(p) and miller\_rabin\_test(p, k):

            return p

#print (generate\_random\_prime(80,580))

def generate\_key\_pairs():

    bit\_length = 256

    primes = [generate\_random\_prime(2\*\*(bit\_length-1), 2\*\*bit\_length - 1) for \_ in range(4)]

    primes.sort()

    p, q = primes[:2]

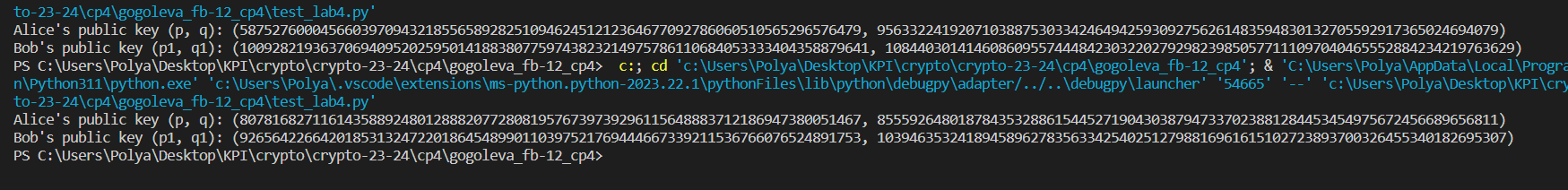
    p1, q1 = primes[2:]

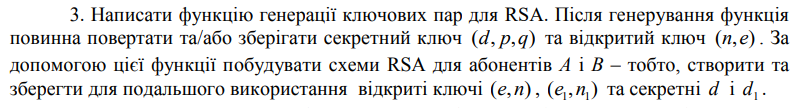
    return p, q, p1, q1

p, q, p1, q1 = generate\_key\_pairs()

print(f"Alice's public key (p, q): ({p}, {q})")

print(f"Bob's public key (p1, q1): ({p1}, {q1})")





def generate\_rsa\_keys(p, q, p1, q1):

    n = p \* q

    phi\_n = (p - 1) \* (q - 1)

    while True:

        public\_exponent = random.randint(2, phi\_n - 1)

        gcd\_result, private\_exponent = gcd(phi\_n, public\_exponent)

        if gcd\_result == 1:

            break

    n1 = p1 \* q1

    phi\_n1 = (p1 - 1) \* (q1 - 1)

    while True:

        public\_exponent\_1 = random.randint(2, phi\_n1 - 1)

        gcd\_result\_1, private\_exponent\_1 = gcd(phi\_n1, public\_exponent\_1)

        if gcd\_result\_1 == 1:

            break

    public\_key = (n, public\_exponent)

    private\_key = (private\_exponent, p, q)

    public\_key\_1 = (n1, public\_exponent\_1)

    private\_key\_1 = (private\_exponent\_1, p1, q1)

    return public\_key, private\_key, public\_key\_1, private\_key\_1

# Generate RSA key pairs for Alice and Bob

p, q, p1, q1 = generate\_key\_pairs()

alice\_public\_key, alice\_private\_key, bob\_public\_key, bob\_private\_key = generate\_rsa\_keys(p, q, p1, q1)

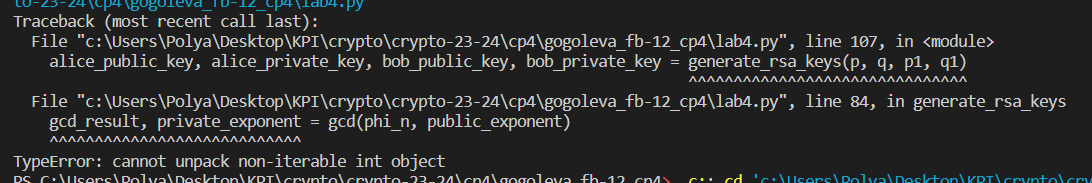
# Output the results

print("Alice's RSA Public Key:", alice\_public\_key)

print("Alice's RSA Private Key:", alice\_private\_key)

print("Bob's RSA Public Key:", bob\_public\_key)

print("Bob's RSA Private Key:", bob\_private\_key)

З цим кодом мали проблемку  
  
Довелось пофіксити gcd, замінила його на extended версію, бо функція НСД мала повертати три числа для правильного генерування РСА ключів.

Новий код виглядає так

import random

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

def extended\_gcd(a, b):

    if a == 0:

        return b, 0, 1

    else:

        g, x, y = extended\_gcd(b % a, a)

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

def trial\_division(n):

    if n < 2:

        return False

    for i in range(2, 100):

        if n % i == 0:

            return False

    return True

def miller\_rabin\_test(p, k=10):

    if p < 2:

        return False

    if p != 2 and p % 2 == 0:

        return False

    factorization\_base = p - 1

    while factorization\_base % 2 == 0:

        factorization\_base //= 2

    for \_ in range(k):

        witness = random.randint(1, p - 1)

        if gcd(witness, p) > 1:

            continue

        exponent = factorization\_base

        residue = pow(witness, exponent, p)

        while exponent != p - 1 and residue != 1 and residue != p - 1:

            residue = (residue \* residue) % p

            exponent \*= 2

        if residue != p - 1 and exponent % 2 == 0:

            return False

    return True

def generate\_random\_prime(start, end, k=10):

    while True:

        p = random.randint(start, end)

        if trial\_division(p) and miller\_rabin\_test(p, k):

            return p

def generate\_key\_pairs():

    bit\_length = 256

    primes = [generate\_random\_prime(2\*\*(bit\_length-1), 2\*\*bit\_length - 1) for \_ in range(4)]

    primes.sort()

    p, q = primes[:2]

    p1, q1 = primes[2:]

    return p, q, p1, q1

def generate\_rsa\_keys(p, q, p1, q1):

    n = p \* q

    phi\_n = (p - 1) \* (q - 1)

    while True:

        public\_exponent = random.randint(2, phi\_n - 1)

        gcd\_result, \_, private\_exponent = extended\_gcd(phi\_n, public\_exponent)

        if gcd\_result == 1:

            break

    n1 = p1 \* q1

    phi\_n1 = (p1 - 1) \* (q1 - 1)

    while True:

        public\_exponent\_1 = random.randint(2, phi\_n1 - 1)

        gcd\_result\_1, \_, private\_exponent\_1 = extended\_gcd(phi\_n1, public\_exponent\_1)

        if gcd\_result\_1 == 1:

            break

    public\_key = (n, public\_exponent)

    private\_key = (private\_exponent, p, q)

    public\_key\_1 = (n1, public\_exponent\_1)

    private\_key\_1 = (private\_exponent\_1, p1, q1)

    return public\_key, private\_key, public\_key\_1, private\_key\_1

# Generate RSA key pairs for Alice and Bob

p, q, p1, q1 = generate\_key\_pairs()

alice\_public\_key, alice\_private\_key, bob\_public\_key, bob\_private\_key = generate\_rsa\_keys(p, q, p1, q1)

# Output the results

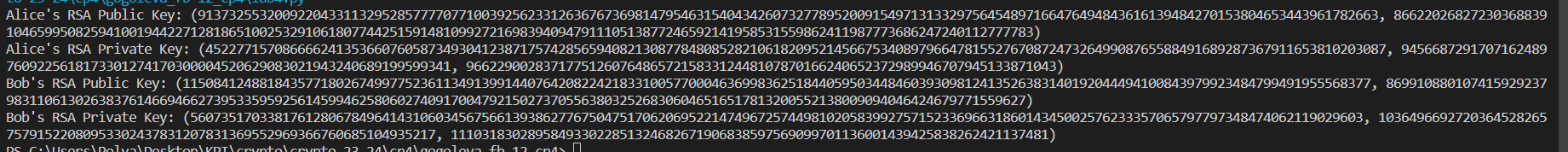
print("Alice's RSA Public Key:", alice\_public\_key)

print("Alice's RSA Private Key:", alice\_private\_key)

print("Bob's RSA Public Key:", bob\_public\_key)

print("Bob's RSA Private Key:", bob\_private\_key)

І виводить нам



Детальніше про те, що відбувається:

У функції створення ключів ми для кожної знайденої пари (p, q) та (p1, q1), обчислюємо n = p \* q та phi\_n = (p - 1) \* (q - 1). Аналогічно для (p1, q1), це частина алгоритму Ейлера.

n представляє собою модуль, який використовується у всіх операціях шифрування та розшифрування. Він обчислюється як добуток двох простих чисел, p та q, для кожної пари (p, q) та (p1, q1). Модуль n є великим непарним числом.

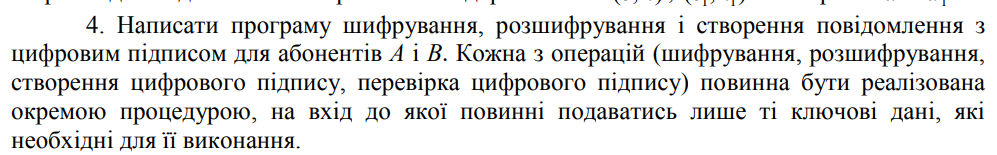
phi\_n (функція Ейлера для n) представляє собою кількість натуральних чисел, менших за n, які взаємно прості з n. Для простого числа p, phi(p) = p - 1. Таким чином, phi\_n обчислюється як (p - 1) \* (q - 1) для кожної пари (p, q) та (p1, q1).

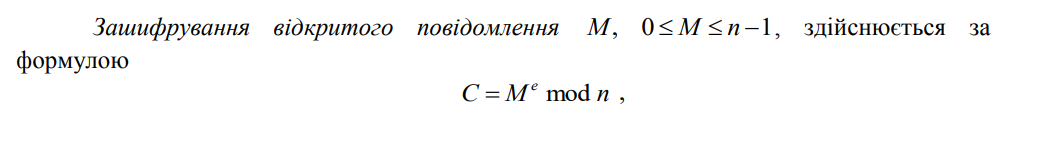
Генеруємо випадкові числа e та e1 так, щоб вони були взаємно простими з phi\_n та phi\_n1 відповідно. Це гарантує, що їхній GCD (найбільший спільний дільник) дорівнює 1.

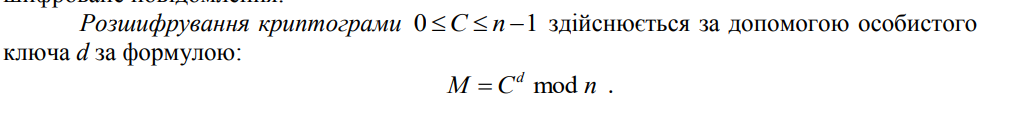
Для кожної пари (e, phi\_n) та (e1, phi\_n1), використовуючи розширений алгоритм Евкліда, знаходимо d та d1, такі що (e \* d) % phi\_n = 1 та (e1 \* d1) % phi\_n1 = 1.

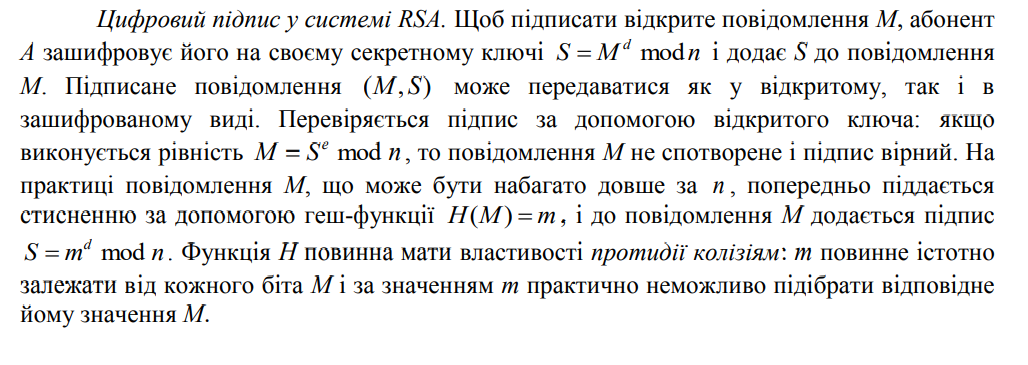
Публічний ключ представляється парою (n, e) для кожного абонента.

Приватний ключ представляється трійкою (d, p, q) для відповідного абонента.









Шифрування:

def encrypt(message, public\_key):

    n, e = public\_key

    cipher\_text = pow(message, e, n)

    return cipher\_text

Для шифрування використовується операція залишку від ділення (pow(message, e, n)), де e - публічний експонент, n – модуль.

Дешифрування:

def decrypt(cipher\_text, private\_key):

    d, p, q = private\_key

    n = p \* q

    plain\_text = pow(cipher\_text, d, n)

    return plain\_text

Для розшифрування використовується операція залишку від ділення (pow(cipher\_text, d, n)), де d - приватний експонент, n - модуль. Результат - відкрите повідомлення.

Створення та перевірка цифрового підпису:

def create\_digital\_signature(message, private\_key):

    d, p, q = private\_key

    n = p \* q

    hashed\_message = int.from\_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

    signature = pow(hashed\_message, d, n)

    return (message, signature)

def verify\_digital\_signature(signed\_message, public\_key):

    n, e = public\_key

    message, signature = signed\_message

    hashed\_message = int.from\_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

    decrypted\_signature = pow(signature, e, n)

    return decrypted\_signature == hashed\_message

message: Повідомлення, для якого ми хочемо створити підпис.

private\_key: Секретний ключ, який складається з d, p та q.

Спершу генерується хеш (hashed\_message) від відкритого повідомлення.

Далі генерується підпис за допомогою операції залишку від ділення (pow(hashed\_message, d, n)), де d - приватний експонент, n - модуль.

signed\_message: Підписане повідомлення у форматі (message, signature).

public\_key: Відкритий ключ, який складається з n та e.

Знову генерується хеш (hashed\_message) від відкритого повідомлення.

Перевірка підпису виконується порівнянням з допомогою операції залишку від ділення (pow(signature, e, n)) та порівнянням результату з гешем повідомлення. Якщо вони співпадають, це свідчить про вірність підпису.

Приклад використання:

p, q, p1, q1 = generate\_key\_pairs()

alice\_public\_key, alice\_private\_key, bob\_public\_key, bob\_private\_key = generate\_rsa\_keys(p, q, p1, q1)

message\_to\_alice = 42

cipher\_text\_for\_alice = encrypt(message\_to\_alice, alice\_public\_key)

decrypted\_message\_for\_alice = decrypt(cipher\_text\_for\_alice, alice\_private\_key)

message\_to\_bob = "Hello, Bob!"

signed\_message\_for\_bob = create\_digital\_signature(message\_to\_bob, bob\_private\_key)

signature\_verified = verify\_digital\_signature(signed\_message\_for\_bob, bob\_public\_key)

print("Message to Alice:", message\_to\_alice)

print("Encrypted Cipher Text for Alice:", cipher\_text\_for\_alice)

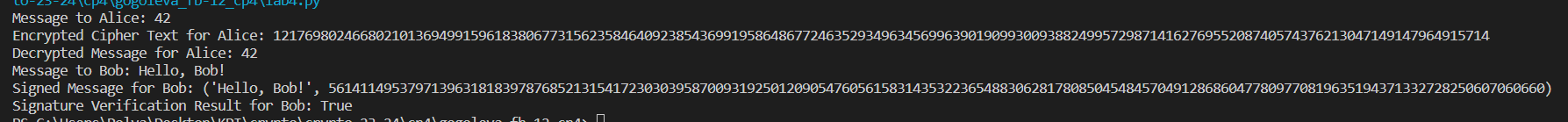
print("Decrypted Message for Alice:", decrypted\_message\_for\_alice)

print("Message to Bob:", message\_to\_bob)

print("Signed Message for Bob:", signed\_message\_for\_bob)

print("Signature Verification Result for Bob:", signature\_verified)

Вивід:



Message to Alice: 42

Encrypted Cipher Text for Alice: 1217698024668021013694991596183806773156235846409238543699195864867724635293496345699639019099300938824995729871416276955208740574376213047149147964915714

Decrypted Message for Alice: 42

Message to Bob: Hello, Bob!

Signed Message for Bob: ('Hello, Bob!', 5614114953797139631818397876852131541723030395870093192501209054760561583143532236548830628178085045484570491286860477809770819635194371332728250607060660)

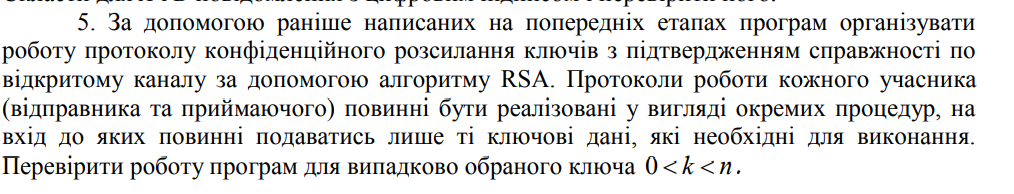
Signature Verification Result for Bob: True

**Підсумовуючи, давайте глянем на основні етапи коду:**

Функція generate\_key\_pairs генерує дві пари ключів для Alice і Bob, використовуючи алгоритм RSA. Публічні ключі (public\_key) використовуються для шифрування, приватні (private\_key) - для розшифрування та цифрового підпису.

Alice вибирає повідомлення (message\_to\_alice), шифрує його використовуючи свій публічний ключ (alice\_public\_key), отримуючи cipher\_text\_for\_alice. Потім розшифровує його, використовуючи свій приватний ключ (alice\_private\_key), отримуючи decrypted\_message\_for\_alice.

Аналогічно, Bob вибирає повідомлення (message\_to\_bob), створює цифровий підпис з використанням свого приватного ключа (bob\_private\_key), отримуючи signed\_message\_for\_bob. Потім він перевіряє цифровий підпис, використовуючи публічний ключ Alice (bob\_public\_key), отримуючи результат signature\_verified.



Аліса передає свій відкритий ключ (e\_A, n\_A) Бобу, і на навпаки.

Аліса обирає випадкове секретне значення k (0 < k < n), яке буде використано для створення спільного ключа.

Аліса зашифровує секретне значення k за допомогою відкритого ключа Боба (e\_B, n\_B), отримуючи k\_1.

Аліса обчислює S\_1, яке є підписом секретного значення k за допомогою свого секретного ключа (d\_A, n\_A).

Аліса відправляє Бобу пару значень (k\_1, S\_1).

Боб отримує (k\_1, S\_1) від Аліси.

Боб розшифровує k\_1 за допомогою свого секретного ключа (d\_B, n\_B), отримуючи секретне значення k.

Боб перевіряє підпис S\_1, використовуючи відкритий ключ Аліси (e\_A, n\_A), для перевірки автентичності.

def send\_key(public\_key\_B, private\_key\_A, public\_key\_A):

    k = random.randint(0, public\_key\_A[0] - 1)

    print("k value:", k)

    s = pow(k, private\_key\_A[0], public\_key\_A[0])

    k\_1 = pow(k, public\_key\_B[1], public\_key\_B[0])

    S\_1 = pow(s, public\_key\_B[1], public\_key\_B[0])

    return k\_1, S\_1

def recv\_key(k\_1, S\_1, public\_key\_B, private\_key\_B, public\_key\_A):

    k = pow(k\_1, private\_key\_B[0], public\_key\_B[0])

    S = pow(S\_1, private\_key\_B[0], public\_key\_B[0])

    return k == pow(S, public\_key\_A[1], public\_key\_A[0])

k\_1, S\_1 = send\_key(bob\_public\_key, alice\_private\_key, alice\_public\_key)

key\_received = recv\_key(k\_1, S\_1, bob\_public\_key, bob\_private\_key, alice\_public\_key)

if key\_received:

    print("Key exchange successful!")

else:

    print("Key exchange failed!")

k value: 206676004617717102138659510415740635300291262768313998448491281901630099226646504287642179586949578086451357280205489731856628689093567176096582659895014

Message to Alice: 42

Encrypted Cipher Text for Alice: 2065591371856434057191640459213532027609136371819832393709442489504964995372551360406677001899497208310076621814269058620859733363799756503469395319150791

Decrypted Message for Alice: 42

Message to Bob: Hello, Bob!

Signed Message for Bob: ('Hello, Bob!', 5388231223831640905551943255220231194891006550390090650668375828632897465547429825772200670227681130133559742300694925540585782293162397987005149274561001)

Signature Verification Result for Bob: True

Key exchange successful!

**Висновки**

Через два з половиною роки навчання я нарешті зрозуміла як працює RSA і тепер маю власний код, що допоможе мені розв’язувати схожі задачі на CTF, котрі раніше здавались мені складними. Все ще відчуваю себе трохи заплутано у цій кількості ключів, але до мене дійшло навіщо ми всі ці роки вчили виш мат і де він знадобиться у нашій сфері, я у це не вірила, чесне слово.