

**Experiment No. 4**

**Title:** RSA Cipher

**Batch: B-2 Roll No.: 16010422234 Experiment No.: 4**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Results:** (Program printout as per the format)

**from math import gcd**

**import random**

**def is\_prime(num):**

**if num <= 1:**

**return False**

**if num <= 3:**

**return True**

**if num % 2 == 0 or num % 3 == 0:**

**return False**

**i = 5**

**while i \* i <= num:**

**if num % i == 0 or num % (i + 2) == 0:**

**return False**

**i += 6**

**return True**

**def mod\_inverse(a, m):**

**m0, x0, x1 = m, 0, 1**

**if m == 1:**

**return 0**

**while a > 1:**

**q = a // m**

**m, a = a % m, m**

**x0, x1 = x1 - q \* x0, x0**

**if x1 < 0:**

**x1 += m0**

**return x1**

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

**n = p \* q**

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

**e = random.randrange(2, phi)**

**while gcd(e, phi) != 1:**

**e = random.randrange(2, phi)**

**d = mod\_inverse(e, phi)**

**public\_key = (e, n)**

**private\_key = (d, n)**

**return public\_key, private\_key**

**def string\_to\_int\_list(text):**

**return [ord(char) for char in text]**

**def int\_list\_to\_string(int\_list):**

**return ''.join(chr(num) for num in int\_list)**

**def encrypt(int\_list, public\_key):**

**e, n = public\_key**

**encrypted\_message = [pow(char, e, n) for char in int\_list]**

**return encrypted\_message**

**def decrypt(encrypted\_message, private\_key):**

**d, n = private\_key**

**decrypted\_message = [pow(char, d, n) for char in encrypted\_message]**

**return decrypted\_message**

**def print\_menu():**

**print("\nMenu:")**

**print("1. Encrypt a message")**

**print("2. Decrypt a message")**

**print("3. Exit")**

**def main():**

**public\_key = None**

**private\_key = None**

**while True:**

**print\_menu()**

**choice = input("Enter your choice (1/2/3): ")**

**if choice == '3':**

**print("Exiting.")**

**break**

**if choice in ['1', '2']:**

**if public\_key is None or private\_key is None:**

**p = int(input("Enter prime number p: "))**

**q = int(input("Enter prime number q: "))**

**if not (is\_prime(p) and is\_prime(q)):**

**print("Both p and q must be prime numbers.")**

**continue**

**public\_key, private\_key = generate\_keys(p, q)**

**print("Public Key:", public\_key)**

**print("Private Key:", private\_key)**

**if choice == '1':**

**plaintext = input("Enter the plaintext to encrypt: ")**

**int\_list = string\_to\_int\_list(plaintext)**

**print("Integer Representation of Plaintext:", int\_list)**

**encrypted\_message = encrypt(int\_list, public\_key)**

**print("Encrypted Message:", encrypted\_message)**

**elif choice == '2':**

**encrypted\_message = list(map(int, input("Enter the encrypted message as space-separated integers: ").split()))**

**decrypted\_int\_list = decrypt(encrypted\_message, private\_key)**

**decrypted\_text = int\_list\_to\_string(decrypted\_int\_list)**

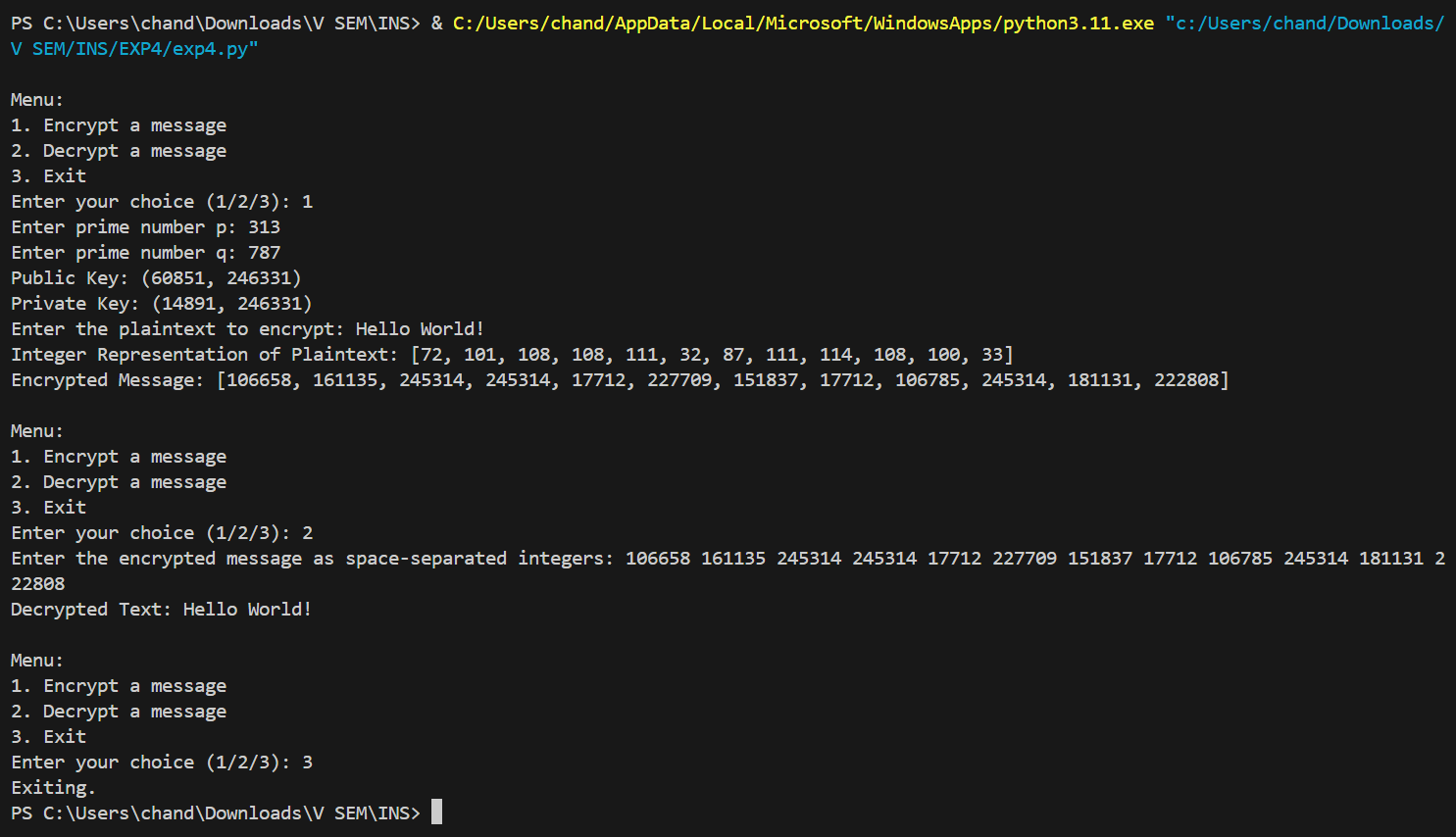
**print("Decrypted Text:", decrypted\_text)**

**else:**

**print("Invalid choice. Please enter 1, 2, or 3.")**

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

**main()**

****

**import random**

**# Function to check if a number is prime**

**def is\_prime(n):**

**if n <= 1:**

**return False**

**if n <= 3:**

**return True**

**if n % 2 == 0 or n % 3 == 0:**

**return False**

**i = 5**

**while i \* i <= n:**

**if n % i == 0 or n % (i + 2) == 0:**

**return False**

**i += 6**

**return True**

**# Function to generate a large prime number**

**def generate\_large\_prime(key\_size):**

**while True:**

**num = random.getrandbits(key\_size)**

**if is\_prime(num):**

**return num**

**# Function to calculate the greatest common divisor (GCD)**

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

**while b != 0:**

**a, b = b, a % b**

**return a**

**# Function to find the modular inverse of e mod phi using the Extended Euclidean Algorithm**

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

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

**if a == 0:**

**return b, 0, 1**

**g, x1, y1 = egcd(b % a, a)**

**x = y1 - (b // a) \* x1**

**y = x1**

**return g, x, y**

**g, x, y = egcd(e, phi)**

**if g != 1:**

**raise Exception('Modular inverse does not exist')**

**else:**

**return x % phi**

**# Function to generate public and private keys**

**def generate\_keypair(key\_size):**

**p = generate\_large\_prime(key\_size // 2) # Typically, use key\_size // 2 for p and q**

**q = generate\_large\_prime(key\_size // 2)**

**n = p \* q**

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

**# Choose e such that 1 < e < phi and e is coprime to phi**

**e = random.randrange(2, phi)**

**while gcd(e, phi) != 1:**

**e = random.randrange(2, phi)**

**d = mod\_inverse(e, phi)**

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

**# Function to sign a message (Non-Repudiation)**

**def sign(private\_key, message):**

**d, n = private\_key**

**signature = [pow(ord(char), d, n) for char in message]**

**return signature**

**# Function to verify a signature**

**def verify(public\_key, message, signature):**

**e, n = public\_key**

**decrypted\_signature = ''.join([chr(pow(char, e, n)) for char in signature])**

**return decrypted\_signature == message**

**def main():**

**# Take key size as input from the user**

**key\_size = int(input("Enter the key size in bits: "))**

**# Generate key pairs for Alice (sender) and Bob (receiver)**

**print("Generating key pairs.")**

**alice\_public\_key, alice\_private\_key = generate\_keypair(key\_size)**

**bob\_public\_key, bob\_private\_key = generate\_keypair(key\_size)**

**print(f"Alice's Public Key: {alice\_public\_key}")**

**print(f"Alice's Private Key: {alice\_private\_key}")**

**print(f"Bob's Public Key: {bob\_public\_key}")**

**print(f"Bob's Private Key: {bob\_private\_key}")**

**# Input the message**

**message = input("Enter the message to send: ")**

**# Alice signs the message using her private key (non-repudiation)**

**signature = sign(alice\_private\_key, message)**

**print("Message signed successfully by Alice.")**

**# Verify the signature using Alice's public key**

**if verify(alice\_public\_key, message, signature):**

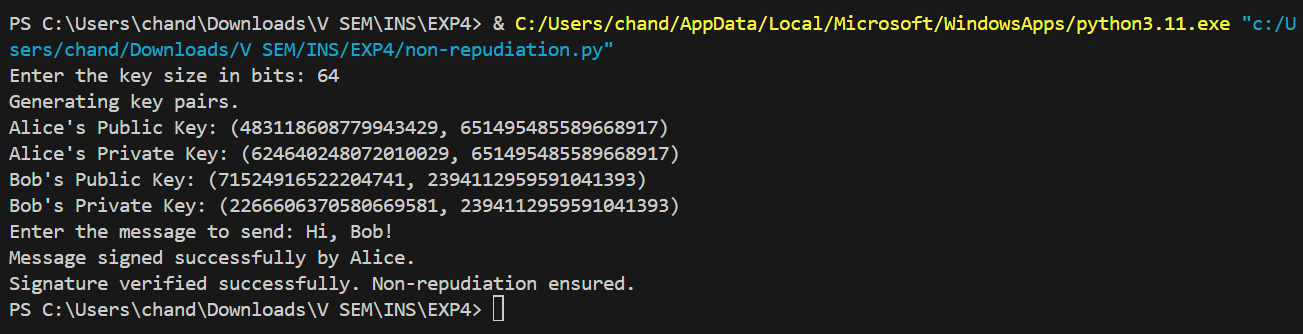
**print("Signature verified successfully. Non-repudiation ensured.")**

**else:**

**print("Signature verification failed. Non-repudiation could not be ensured.")**

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

**main()**

****

**import random**

**# Optimized function to check if a number is prime**

**def is\_prime(n):**

**if n <= 1:**

**return False**

**if n <= 3:**

**return True**

**if n % 2 == 0 or n % 3 == 0:**

**return False**

**# Use 6k +/- 1 optimization for larger primes**

**i = 5**

**while i \* i <= n:**

**if n % i == 0 or n % (i + 2) == 0:**

**return False**

**i += 6**

**return True**

**# Optimized function to generate a large prime number**

**def generate\_large\_prime(key\_size):**

**while True:**

**# Generate a random odd number of the specified size**

**num = random.getrandbits(key\_size) | 1 # Ensures the number is odd**

**if is\_prime(num):**

**return num**

**# Function to calculate the greatest common divisor (GCD)**

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

**while b != 0:**

**a, b = b, a % b**

**return a**

**# Optimized function to find the modular inverse of e mod phi using the Extended Euclidean Algorithm**

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

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

**if a == 0:**

**return b, 0, 1**

**g, x1, y1 = egcd(b % a, a)**

**return g, y1 - (b // a) \* x1, x1**

**g, x, \_ = egcd(e, phi)**

**if g != 1:**

**raise Exception('Modular inverse does not exist')**

**else:**

**return x % phi**

**# Function to generate public and private keys**

**def generate\_keypair(key\_size):**

**p = generate\_large\_prime(key\_size // 2) # Typically, use key\_size // 2 for p and q**

**q = generate\_large\_prime(key\_size // 2)**

**n = p \* q**

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

**# Choose e such that 1 < e < phi and e is coprime to phi**

**e = random.randrange(2, phi)**

**while gcd(e, phi) != 1:**

**e = random.randrange(2, phi)**

**d = mod\_inverse(e, phi)**

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

**# Function to encrypt the message**

**def encrypt(public\_key, plaintext):**

**e, n = public\_key**

**ciphertext = [pow(ord(char), e, n) for char in plaintext]**

**return ciphertext**

**# Function to decrypt the message**

**def decrypt(private\_key, ciphertext):**

**d, n = private\_key**

**plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])**

**return plaintext**

**# Function to sign a message (Non-Repudiation)**

**def sign(private\_key, message):**

**d, n = private\_key**

**signature = [pow(ord(char), d, n) for char in message]**

**return signature**

**# Function to verify a signature**

**def verify(public\_key, message, signature):**

**e, n = public\_key**

**decrypted\_signature = ''.join([chr(pow(char, e, n)) for char in signature])**

**return decrypted\_signature == message**

**def main():**

**# Take key size as input from the user**

**key\_size = int(input("Enter the key size in bits: "))**

**# Generate key pairs for Alice (sender) and Bob (receiver)**

**print("Generating key pairs.")**

**alice\_public\_key, alice\_private\_key = generate\_keypair(key\_size)**

**bob\_public\_key, bob\_private\_key = generate\_keypair(key\_size)**

**print(f"Alice's Public Key: {alice\_public\_key}")**

**print(f"Alice's Private Key: {alice\_private\_key}")**

**print(f"Bob's Public Key: {bob\_public\_key}")**

**print(f"Bob's Private Key: {bob\_private\_key}")**

**# Input the message**

**message = input("Enter the message to send: ")**

**# Encrypt the message using Bob's public key**

**encrypted\_message = encrypt(bob\_public\_key, message)**

**print("Encrypted message:", encrypted\_message)**

**# Convert the encrypted message integers to strings (to avoid OverflowError)**

**encrypted\_message\_str = ','.join(map(str, encrypted\_message))**

**# Sign the encrypted message using Alice's private key (non-repudiation)**

**signature = sign(alice\_private\_key, encrypted\_message\_str)**

**print("Encrypted message signed successfully by Alice.")**

**# Verify the signature using Alice's public key**

**if verify(alice\_public\_key, encrypted\_message\_str, signature):**

**print("Signature verified successfully. Non-repudiation ensured.")**

**# Decrypt the message using Bob's private key**

**decrypted\_message = decrypt(bob\_private\_key, encrypted\_message)**

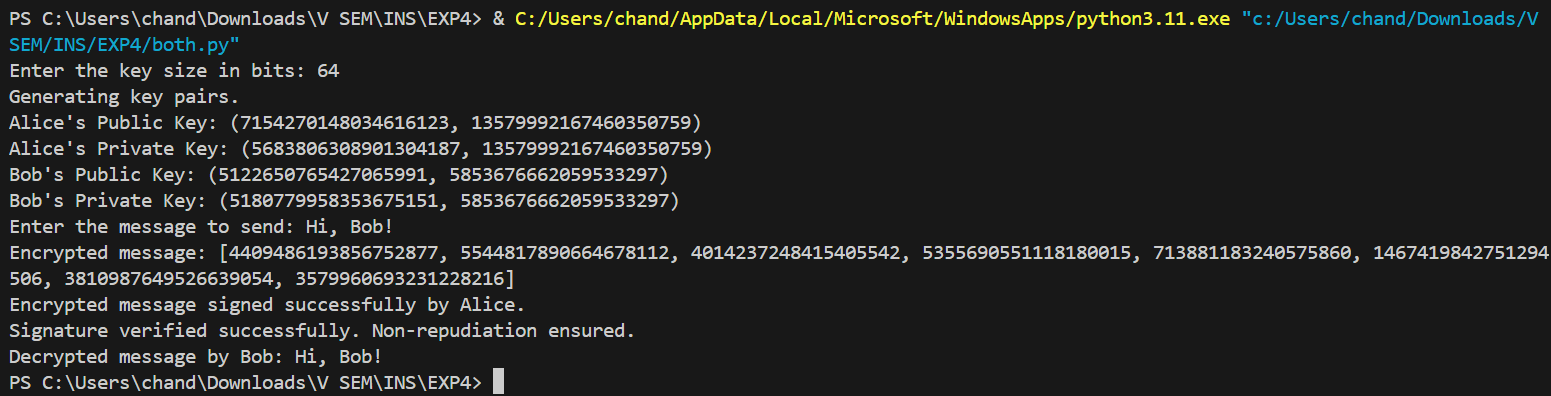
**print("Decrypted message by Bob:", decrypted\_message)**

**else:**

**print("Signature verification failed. Non-repudiation could not be ensured.")**

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

**main()**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Questions:**

**1. In RSA cryptosystem each plaintext character is presented by the number between 00(A) and 25(Z). The number 26 represents the blank character. Bob wants to send Alice the message “Hello World”. So the plaintext is as below,**

**07 04 11 11 14 26 22 14 17 11 03 . Suppose p=11, q=3. Generate receiver’s key pair and show encryption and decryption of the message using RSA cipher.**

Given:

𝑝 = 11

𝑞 = 3

Step 1: Calculate 𝑛

𝑛 = 𝑝 × 𝑞 = 11 × 3 = 33

Step 2: Calculate 𝜙 (𝑛)

𝜙 (𝑛) = (𝑝 − 1) × (𝑞 − 1) = (11 − 1) × (3 − 1) = 10 × 2 = 20

Step 3: Choose 𝑒 such that 1 < 𝑒 < 𝜙 (𝑛) and gcd⁡ (𝑒, 𝜙 (𝑛)) = 1

Let's choose 𝑒 = 3 (commonly used small prime)

Step 4: Calculate 𝑑 such that 𝑒 × 𝑑 ≡ 1 (mod 𝜙(𝑛))

3 × 𝑑 ≡ 1 (mod 20)

𝑑 = 7(since 3 × 7 = 21 ≡ 1 (mod 20))

So, the public key is (𝑒, 𝑛) = (3, 33) and the private key is (𝑑, 𝑛) = (7, 33)

Encryption:

Plaintext: "Hello World" => 07 04 11 11 14 26 22 14 17 11 03

Using the public key (3, 33), each number 𝑀 is encrypted to 𝐶 = 𝑀e (mod 𝑛).

𝐶1 = 073 (mod 33) = 343 (mod 33) =13

𝐶2 = 043 (mod 33) = 64 (mod 33) = 31

𝐶3 = 113 (mod 33) = 1331 (mod 33) = 7

𝐶4 = 113 (mod 33) = 1331 (mod 33) = 7

𝐶5 = 143 (mod 33) = 2744 (mod 33) = 2

𝐶6 = 263 (mod 33) = 17576 (mod 33) = 10

𝐶7 = 223 (mod 33) = 10648 (mod 33) = 14

𝐶8 = 143 (mod 33) = 2744 (mod 33) = 2

𝐶9 = 173 (mod 33) = 4913 (mod 33) = 23

𝐶10 = 113 (mod 33) = 1331 (mod 33) = 7

𝐶11 = 033 (mod 33) = 27 (mod 33) = 27

So, the ciphertext is: 13 31 7 7 2 10 14 2 23 7 27

Decryption:

Using the private key (7, 33), each number 𝐶 is decrypted to 𝑀 = 𝐶𝑑 (mod 𝑛).

𝑀1 = 137 (mod 33) =7

𝑀2 = 317 (mod 33) = 4

𝑀3 = 77 (mod 33) = 11

𝑀4 = 77 (mod 33) = 11

𝑀5 = 27 (mod 33) = 14

𝑀6 = 107 (mod 33) = 26

𝑀7 = 147 (mod 33) = 22

𝑀8 = 27 (mod 33) = 14

𝑀9 = 237 (mod 33) = 17

𝑀10 = 77 (mod 33) = 11

𝑀11 = 277 (mod 33) = 3

So, the decrypted plaintext is: 07 04 11 11 14 26 22 14 17 11 03 ("Hello World")

**2. List the attacks on RSA.**

* Brute Force Attack: Trying all possible private keys.
* Mathematical Attacks:
  + Factorization Attack: Breaking the RSA by factorizing 𝑛 into 𝑝 and 𝑞.
  + Timing Attack: Measuring the time taken for decryption to infer the private key.
  + Chosen Ciphertext Attack: The attacker chooses a ciphertext and gets it decrypted to obtain plaintext.
  + Key Size Attack: Using keys that are too small, making them vulnerable to factorization.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Outcomes: Illustrate different cryptographic algorithms for security.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Conclusion: (Conclusion to be based on the objectives and outcomes achieved)**

The RSA cipher is a foundational public key algorithm used extensively in secure communications. This experiment demonstrated the process of key generation, encryption, and decryption in the RSA algorithm. Through practical implementation, the strengths and computational requirements of RSA were observed, illustrating its suitability for encrypting small data blocks such as keys and passwords. Understanding RSA also highlights the importance of key size in maintaining security and the potential vulnerabilities to various attacks. This experiment underlines the critical role of cryptographic algorithms in ensuring data security in digital communications.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

