**NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY**



**KARAIKAL – 609 609**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Roll Number:** CS22B1009 **Name: CHINTHAMANI KIRANMAYI**

**Subject Code:** CS1702  **Subject Name:** Network Security

**Assignment**

Implementation of RSA algorithms

**Code:**

import random

import math

def is\_prime(n):

    """Check if a number is prime."""

    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

def gcd(a, b):

    """Compute the greatest common divisor of a and b."""

    while b != 0:

        a, b = b, a % b

    return a

def extended\_gcd(a, b):

    """Extended Euclidean Algorithm to find modular inverse."""

    if a == 0:

        return b, 0, 1

    else:

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

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

def modinv(e, phi):

    """Find the modular inverse of e modulo phi."""

    g, x, y = extended\_gcd(e, phi)

    if g != 1:

        raise Exception('Modular inverse does not exist')

    else:

        return x % phi

def generate\_keys(p, q):

    """Generate RSA public and private keys."""

    # Step 1: Ensure p and q are prime

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

        raise ValueError("Both p and q must be prime numbers.")

    # Step 2: Compute n = p \* q

    n = p \* q

    # Step 3: Compute Euler's totient function phi(n)

    phi = (p - 1) \* (q - 1)

    # Step 4: Choose an integer e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1

    e = random.randrange(1, phi)

    while gcd(e, phi) != 1:

        e = random.randrange(1, phi)

    # Step 5: Compute d, the modular inverse of e modulo phi(n)

    d = modinv(e, phi)

    # Public key is (e, n), private key is (d, n)

    return (e, n), (d, n)

def encrypt(public\_key, plaintext):

    """Encrypt a message using the public key."""

    e, n = public\_key

    # Convert each character to its ASCII value and encrypt

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

    return ciphertext

def decrypt(private\_key, ciphertext):

    """Decrypt a message using the private key."""

    d, n = private\_key

    # Decrypt each number to its ASCII value and convert to character

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

    return plaintext

print("Starting RSA...")

# Get prime numbers p and q from the user

while True:

    try:

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

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

        if is\_prime(p) and is\_prime(q):

            break

        else:

            print("Both numbers must be prime. Please try again.")

    except ValueError:

        print("Invalid input. Please enter integers.")

# Generate keys

print("Generating keys...")

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

print("Public Key (e, n):", public\_key)

print("Private Key (d, n):", private\_key)

# Message to encrypt

message = input("Enter a message to encrypt: ")

print("Original Message:", message)

# Encrypt the message

print("Encrypting message...")

encrypted\_message = encrypt(public\_key, message)

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

# Decrypt the message

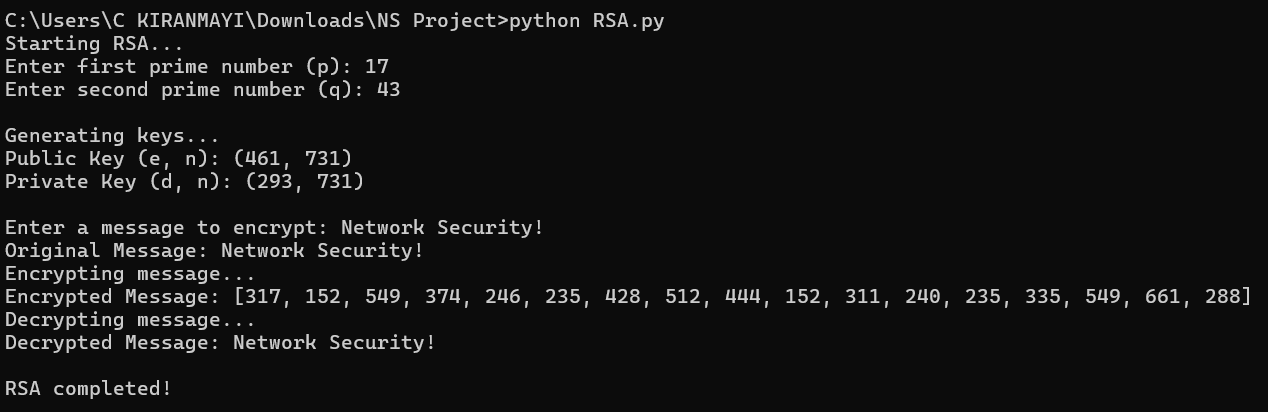
print("Decrypting message...")

decrypted\_message = decrypt(private\_key, encrypted\_message)

print("Decrypted Message:", decrypted\_message)

print("RSA completed!")

**Output:**

****

**Implementation:**

**1. Key Generation**

Choose two prime numbers p and q.

**Steps:**

1. Compute n:

n = p × q

1. Compute ϕ(n)*:*

ϕ(n) = (p−1) × (q−1)

* + This value is used to generate e and d.

1. Choose Public Exponent e:
   * Select e such that:

1<e<ϕ(n) and gcd(e,ϕ(n)) = 1

1. Compute Private Exponent d:
   * d is the modular inverse of e modulo ϕ(n)

d≡e−1(mod ϕ(n))

Solved using the Extended Euclidean Algorithm.

**2. Encryption**

* Plaintext: M
* Ciphertext:

C ≡M*e* (mod *n*)

**3. Decryption**

* Ciphertext: C
* Plaintext:

M ≡ Cd (mod n)