**RSA ALGORITHM**

**A PROJECT REPORT (CS1702)**

Submitted By

**Chagamreddy Ragavi (CS22B1007)**

To

**Dr. Narendran Rajagopalan**

***Associate Professor***

***Department of Computer Science and Engineering***

***National Institute of Technology Puducherry***

***Karaikal – 609609***

****

**DEPARTMENT OF**

**COMPUTER SCIENCE AND ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY**

**KARAIKAL – 609 609**

**APRIL 2025**

**Introduction:**

This report documents the implementation of the RSA algorithm (using Python), a widely used public-key cryptosystem for secure data transmission. It allows users to:

* Generate RSA key pairs (public and private keys)
* Encrypt messages using the public key
* Decrypt messages using the private key
* The tool supports both auto-generated primes and user-provided primes for key generation.

**Implementation:**

import random

import math

from secrets import randbelow

def is\_prime(n, k=5):

    """Miller-Rabin primality test with k rounds"""

    if n <= 1:

        return False

    elif n <= 3:

        return True

    elif n % 2 == 0:

        return False

    d = n - 1

    s = 0

    while d % 2 == 0:

        d //= 2

        s += 1

    for \_ in range(k):

        a = random.randint(2, n - 2)

        x = pow(a, d, n)

        if x == 1 or x == n - 1:

            continue

        for \_\_ in range(s - 1):

            x = pow(x, 2, n)

            if x == n - 1:

                break

        else:

            return False

    return True

def generate\_prime(bits=16):

    """Generate a random prime number with specified bits"""

    while True:

        num = randbelow(2\*\*bits - 2\*\*(bits-1)) + 2\*\*(bits-1)

        if num % 2 != 0 and is\_prime(num):

            return num

def gcd(a, b):

    """Compute greatest common divisor using Euclid's algorithm."""

    while b != 0:

        a, b = b, a % b

    return a

def modular\_inverse(e, phi):

    """Find modular inverse using extended Euclidean algorithm"""

    original\_phi = phi

    x1, x2 = 1, 0

    y1, y2 = 0, 1

    while phi != 0:

        quotient = e // phi

        e, phi = phi, e % phi

        x1, x2 = x2, x1 - quotient \* x2

        y1, y2 = y2, y1 - quotient \* y2

    if e != 1:

        raise ValueError("Modular inverse doesn't exist")

    return x1 % original\_phi

def generate\_keypair(p=None, q=None):

    """Generate RSA public and private key pair."""

    if p is None or q is None:

        p = generate\_prime()

        q = generate\_prime()

        while q == p:  # Ensure distinct primes

            q = generate\_prime()

    else:

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

            raise ValueError("Both numbers must be prime.")

        if p == q:

            raise ValueError("p and q cannot be equal")

    n = p \* q

    phi = (p - 1) \* (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1

    e = random.randrange(2, phi)

    while gcd(e, phi) != 1:

        e = random.randrange(2, phi)

    d = modular\_inverse(e, phi)

    return ((e, n), (d, n), (p, q))

def rsa\_encrypt(pk, plaintext):

    """Encrypt plaintext using RSA public key."""

    key, n = pk

    cipher = []

    for char in plaintext:

        m = ord(char)

        cipher.append(pow(m, key, n))

    return cipher

def rsa\_decrypt(pk, ciphertext):

    """Decrypt ciphertext using RSA private key."""

    key, n = pk

    plain = []

    for num in ciphertext:

        plain.append(chr(pow(num, key, n)))

    return ''.join(plain)

def get\_prime\_input(prompt):

    """Get and validate prime number input from user."""

    while True:

        try:

            num = int(input(prompt))

            if not is\_prime(num):

                print("Please enter a valid prime number.")

                continue

            return num

        except ValueError:

            print("Invalid input. Please enter an integer.")

def main():

    print("Enhanced RSA Encryption/Decryption Tool")

    print("--------------------------------------")

    # Choose prime generation method

    while True:

        choice = input("\nChoose an option:\n"

                      "1. Generate random primes\n"

                      "2. Enter my own primes\n"

                      "Enter choice (1/2): ").strip()

        if choice == '1':

            # Auto-generate primes

            public\_key, private\_key, primes = generate\_keypair()

            p, q = primes

            print(f"\nGenerated primes: p = {p}, q = {q}")

            break

        elif choice == '2':

            # User provides primes

            p = get\_prime\_input("Enter first prime number: ")

            q = get\_prime\_input("Enter second different prime number: ")

            while p == q:

                print("Numbers must be different.")

                q = get\_prime\_input("Enter second different prime number: ")

            try:

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

                break

            except ValueError as e:

                print(f"Error: {e}")

        else:

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

    print("\nGenerated Keys:")

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

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

    # Message encryption/decryption

    message = input("\nEnter a message to encrypt: ")

    # Encryption

    encrypted\_msg = rsa\_encrypt(public\_key, message)

    print("\nEncrypted Message:", ' '.join(map(str, encrypted\_msg)))

    # Decryption

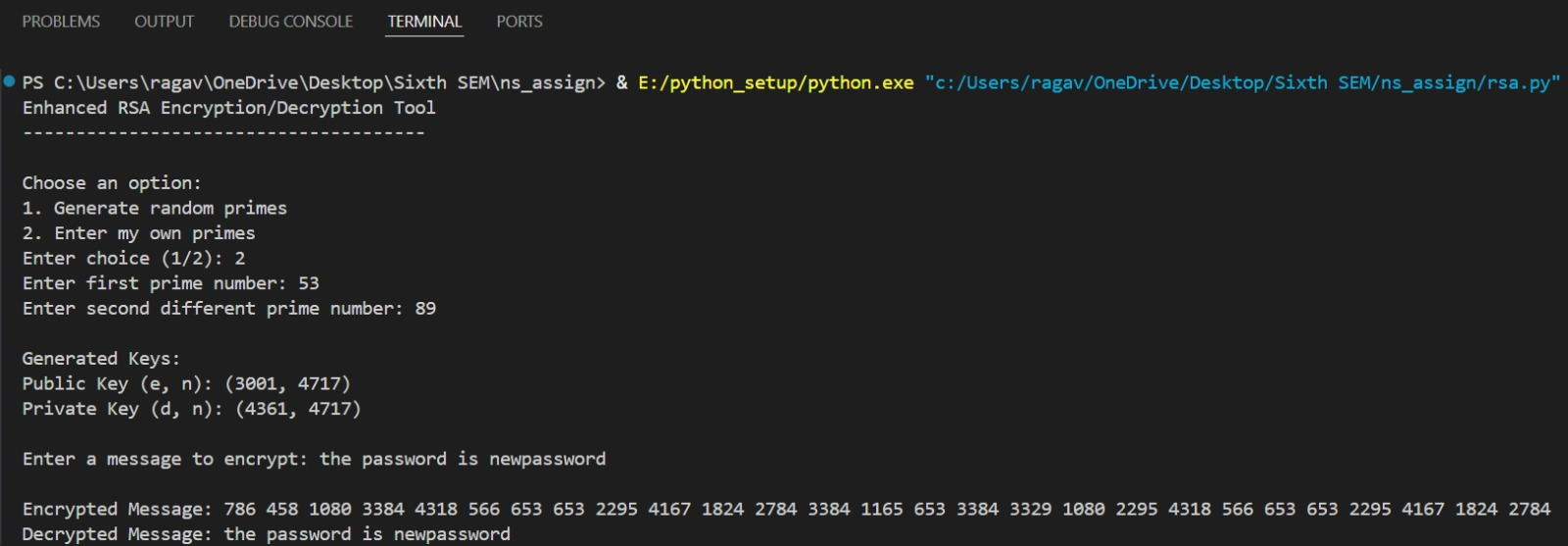
    decrypted\_msg = rsa\_decrypt(private\_key, encrypted\_msg)

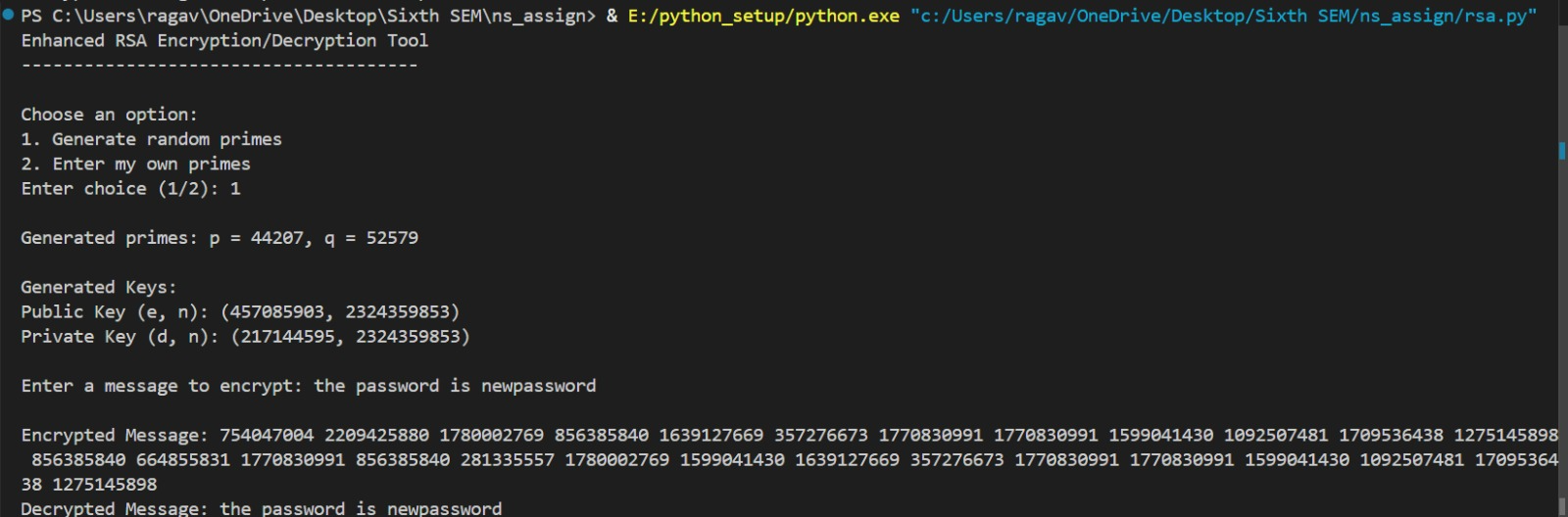
    print("Decrypted Message:", decrypted\_msg)

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

    main()

**Sample Output:**

****



**Function Explanations:**

1. **is\_prime**(n, k=5)

* Checks if a number n is prime using the Miller-Rabin primality test (more reliable than basic methods).
* Performs k rounds of testing for higher accuracy in detecting primes.

2. **generate\_prime**(bits=16)

* Generates a cryptographically secure random prime number of specified bit length (default: 16 bits).
* Uses secrets.randbelow for secure random number generation.

### 3. gcd(a, b)

* Computes the greatest common divisor (GCD) of two numbers using Euclid's algorithm.
* Essential for ensuring e and φ(n) are coprime during key generation

### 4. modular\_inverse(e, φ)

### Finds the modular inverse of e under modulus phi using the Extended Euclidean Algorithm.

### Required to compute the private exponent d in RSA.

### 5. generate\_keypair(p=None, q=None)

### Generates RSA public and private keys from two primes (p and q).

### If primes are not provided, auto-generates secure random primes.

### Compute modulus: n = p × q

### Compute ϕ(n) = (p-1)(q-1)

### Choose e such that 1 < e < ϕ(n) and gcd(e, ϕ(n)) = 1

### Compute d ≡ e⁻¹ (mod ϕ(n))

### 6. rsa\_encrypt(pk, plaintext)

* Encrypts a plaintext message using the public key (e, n).
* Each character is converted to ASCII and encrypted using pow(ord(char), e, n).
* For each character: c ≡ mᵉ (mod n) where m is the ASCII value.

**7. rsa\_decrypt**(pk, ciphertext)

## Decrypts ciphertext using the private key (d, n).

## Each number in ciphertext is decrypted using pow(num, d, n) and converted back to a character.

## For each ciphertext number: m ≡ cᵈ (mod n)

**Conclusion:**

This RSA implementation successfully demonstrates the fundamental principles of public-key cryptography, making it ideal for educational purposes. It intentionally simplifies certain aspects—such as using smaller primes and character-wise encryption—to prioritize clarity over performance. For real-world applications, these limitations can be addressed by implementing industry standards like larger key sizes and proper padding schemes.