RSA
Encryption and Decryption

Aim: To study and implement the RSA Encryption and Decryption

Theory:

RSA (Rivest-Shamir-Adleman) Algorithm:

RSA is a widely used asymmetric encryption algorithm that provides secure communication over untrusted networks. It is based on the mathematical problem of factoring large prime numbers, which is computationally difficult and forms the foundation of RSA's security.

Key Generation:

The RSA algorithm involves the generation of a public-private key pair. The key generation process consists of the following steps:

a) Select two distinct prime numbers, p and q.
b) Compute the modulus, N, by multiplying p and q: N = p * q.
c) Calculate Euler's function, $\phi(N)$, where $\phi(N) = (p - 1) * (q - 1)$.
d) Choose an integer, e, such that $1 < e < \phi(N)$ and e is coprime with $\phi(N)$. This means that e and $\phi(N)$ should have no common factors other than 1.
e) Find the modular multiplicative inverse of e modulo $\phi(N)$, denoted as d. In other words, d is an integer such that $(d * e) \% \phi(N) = 1$.
f) The public key consists of the modulus, N, and the public exponent, e. The private key consists of the modulus, N, and the private exponent, d.

Encryption Process:

To encrypt a message using RSA encryption, follow these steps:

a) Obtain the recipient's public key, which includes the modulus, N, and the public exponent, e.
b) Represent the plaintext message as an integer, M, where $0 \leq M < N$.
c) Compute the ciphertext, C, using the encryption formula: $C = M^e \bmod N$.

Decryption Process:

To decrypt a message encrypted with RSA encryption, the recipient uses their private key. Follow these steps:

a) Obtain the recipient's private key, which includes modulus, N, and the private exponent, d.
b) Receive the ciphertext, C.
c) Compute the plaintext, M, using the decryption formula: $M = C^d \bmod N$.

Security Considerations:

RSA encryption relies on the difficulty of factoring large prime numbers. The security of RSA is based on the assumption that factoring large numbers is computationally infeasible within a reasonable time frame. Breaking RSA encryption requires factoring the modulus, N, into its constituent prime factors, which becomes exponentially more difficult as N grows larger.

To ensure the security of RSA, it is essential to use sufficiently large prime numbers for key generation and to protect the private key from unauthorized access.

Code: Python code for implementing RSA Algorithm

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import binascii

keyPair = RSA.generate(1024)

pubKey = keyPair.publickey()
print(f"Public key:  (n={hex(pubKey.n)}, e={hex(pubKey.e)})")
pubKeyPEM = pubKey.exportKey()
print(pubKeyPEM.decode('ascii'))

print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")
privKeyPEM = keyPair.exportKey()
print(privKeyPEM.decode('ascii'))

#encryption
msg = 'SIES Nerul'
encryptor = PKCS1_OAEP.new(pubKey)
encrypted = encryptor.encrypt(msg)
print("Encrypted:", binascii.hexlify(encrypted))
```

```
# Python for RSA asymmetric cryptographic algorithm.
# For demonstration, values are
# relatively small compared to practical application
import math


def gcd(a, h):
    temp = 0
    while(1):
        temp = a % h
        if (temp == 0):
            return h
        a = h
        h = temp


p = 3
```

```
q = 7
n = p*q
e = 2
phi = (p-1)*(q-1)

while (e < phi):

    # e must be co-prime to phi and
    # smaller than phi.
    if(gcd(e, phi) == 1):
        break
    else:
        e = e+1

# Private key (d stands for decrypt)
# choosing d such that it satisfies
# d*e = 1 + k * totient

k = 2
d = (1 + (k*phi))/e

# Message to be encrypted
msg = 12.0

print("Message data = ", msg)

# Encryption c = (msg ^ e) % n
c = pow(msg, e)
c = math.fmod(c, n)
print("Encrypted data = ", c)

# Decryption m = (c ^ d) % n
m = pow(c, d)
m = math.fmod(m, n)
print("Original Message Sent = ", m)
```

Output

```
Message data = 12.000000

Encrypted data = 3.000000

Original Message Sent = 12.000000
```

Method 2

Encrypting and decrypting plain text messages containing alphabets and numbers using their ASCII value

```
import random
import math

# A set will be the collection of prime numbers,
# where we can select random primes p and q
prime = set()

public_key = None
private_key = None
n = None

# We will run the function only once to fill the set of
# prime numbers
def primefiller():
    # Method used to fill the primes set is Sieve of
    # Eratosthenes (a method to collect prime numbers)
    seive = [True] * 250
    seive[0] = False
    seive[1] = False
    for i in range(2, 250):
        for j in range(i * 2, 250, i):
            seive[j] = False

    # Filling the prime numbers
    for i in range(len(seive)):
        if seive[i]:
            prime.add(i)


# Picking a random prime number and erasing that prime
# number from list because p!=q
def pickrandomprime():
    global prime
    k = random.randint(0, len(prime) - 1)
    it = iter(prime)
    for _ in range(k):
        next(it)

    ret = next(it)
    prime.remove(ret)
    return ret
```

```
def setkeys():
    global public_key, private_key, n
    prime1 = pickrandomprime()  # First prime number
    prime2 = pickrandomprime()  # Second prime number

    n = prime1 * prime2
    fi = (prime1 - 1) * (prime2 - 1)

    e = 2
    while True:
        if math.gcd(e, fi) == 1:
            break
        e += 1

    # d = (k*Φ(n) + 1) / e for some integer k
    # d*e mod * Φ(n) = 1
    public_key = e

    d = 2
    while True:
        if (d * e) % fi == 1:
            break
        d += 1

    private_key = d


# To encrypt the given number
def encrypt(message):
    global public_key, n
    e = public_key
    encrypted_text = 1
    while e > 0:
        encrypted_text *= message
        encrypted_text %= n
        e -= 1
    return encrypted_text


# To decrypt the given number
def decrypt(encrypted_text):
    global private_key, n
    d = private_key
    decrypted = 1
    while d > 0:
        decrypted *= encrypted_text
        decrypted %= n
        d -= 1
    return decrypted


# First converting each character to its ASCII value and
# then encoding it then decoding the number to get the
# ASCII and converting it to character
def encoder(message):
    encoded = []
    # Calling the encrypting function in encoding function
    for letter in message:
        encoded.append(encrypt(ord(letter)))
    return encoded


def decoder(encoded):
    s = ''
    # Calling the decrypting function decoding function
    for num in encoded:
        s += chr(decrypt(num))
    return s


if __name__ == '__main__':
    primefiller()
    setkeys()
    message = "Test Message"
    # Uncomment below for manual input
    # message = input("Enter the message\n")
    # Calling the encoding function
    coded = encoder(message)

    print("Initial message:")
    print(message)
    print("\n\nThe encoded message(encrypted by public key)\n")
    print(''.join(str(p) for p in coded))
    print("\n\nThe decoded message(decrypted by public key)\n")
    print(''.join(str(p) for p in decoder(coded)))
```

Output

```
Initial message:

Test Message


The encoded message(encrypted by public key)

863312887135951593413927434912887135951359583051879012887
```

```
The decoded message(decrypted by private key)
Test Message
```