21CSE1003 Ashish Singh

Q1. Implement RSA Signature Algorithm.

1. **Input Data**:
   - Read two prime numbers, p and q.
   - Read and convert the message to uppercase.
2. **Key Generation**:
   - Compute n=p×qn = p \times q.
   - Compute φ=(p−1)×(q−1)\phi = (p - 1) \times (q - 1).
   - Find public exponent e such that gcd(e,φ)=1\text{gcd}(e, \phi) = 1.
   - Compute private exponent d as the modular inverse of e modulo φ\phi using the Extended Euclidean Algorithm.
3. **RSA Encryption**:
   - Convert the message to a number.
   - Encrypt the number using public key (e,n)(e, n).
4. **RSA Decryption**:
   - Decrypt the number using private key (d,n)(d, n).
   - Convert the decrypted number back to text.
5. **RSA Signature Generation**:
   - Convert the message to a number.
   - Sign the number using private key (d,n)(d, n).
6. **RSA Signature Verification**:
   - Convert the message to a number.
   - Verify the signature by comparing it with the number obtained by decrypting the signature using public key (e,n)(e, n).

LAB_11\RSA_signature.py

```python
# Implement RSA Signature Algorithm.

import math

# Function for modular exponentiation
def Modular_Expo(base, exp, mod):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result

# Function for calculating modular inverse using the Extended Euclidean Algorithm
def MI_EEA(a, m):
    m0, y, x = m, 0, 1
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a, m % a
        y, x = x - q * y, y
    return x + m0 if x < 0 else x

p = int(input('Enter prime-1: '))
q = int(input('Enter prime-2: '))
msg = input('Enter message: ')
msg = msg.upper()

def RSA_encrypt(msg, e, n):
    m = ''
    for c in msg:
        num = ord(c) - 64
        m += str(num).zfill(2)  # Zero pad to handle multi-digit numbers
    num = int(m)
    encrypt = Modular_Expo(num, e, n)
    return encrypt

def RSA_decrypt(num, d, n):
    text = Modular_Expo(num, d, n)
    decrypt = str(text)
    plain = ''
    for i in range(0, len(decrypt), 2):
        num = int(decrypt[i:i+2])
        num += 64
        plain += chr(num)
    return plain
```

```python

def RSA_sign(msg, d, n):
    m = ''
    for c in msg:
        num = ord(c) - 64
        m += str(num).zfill(2)
    num = int(m)
    signature = Modular_Expo(num, d, n)
    return signature

def RSA_verify(msg, signature, e, n):
    m = ''
    for c in msg:
        num = ord(c) - 64
        m += str(num).zfill(2)
    num = int(m)
    verify = Modular_Expo(signature, e, n)
    return verify == num

n = p * q
phi = (p - 1) * (q - 1)
e = 0
for i in range(2, phi):
    if math.gcd(i, phi) == 1:
        e = i
        break
d = MI_EEA(e, phi)

print('Public Key:', n, e)
print('Private Key:', p, q, d)

signature = RSA_sign(msg, d, n)
print('Signature:', signature)

is_valid = RSA_verify(msg, signature, e, n)
print('Is the signature valid?', is_valid)

num = RSA_encrypt(msg, e, n)
print('Encrypted message:', num)
plain = RSA_decrypt(num, d, n)
print('Decrypted message:', plain)

```

Q2. Implement Elgamal digital Signature Algorithm.

1. **Key Generation**:

- **Input**: Set p (a large prime), g (a primitive root modulo p), and x (private key).
- **Compute**: Public key component y as y=gxmod py = g^x \mod p.
- **Output**: Public key (p, g, y) and private key x.

2. **Signature Generation**:

- **Hash**: Convert message to hash value m_hash using SHA-256.
- **Generate Random** k: Choose random integer k such that gcd(k, p-1) = 1.
- **Compute** r: r=gkmod pr = g^k \mod p.
- **Compute Inverse of** k: kinv=kp−2mod (p−1)k_{\text{inv}} = k^{p-2} \mod (p-1).
- **Compute** s: s=(kinv×(mhash−x×r))mod (p−1)s = (k_{\text{inv}} \times (m_{\text{hash}} - x \times r)) \mod (p-1).
- **Output**: Signature (r, s).

3. **Signature Verification**:

- **Hash**: Convert message to hash value m_hash using SHA-256.
- **Compute** v1: v1=gmhashmod pv1 = g^{m_{\text{hash}}} \mod p.
- **Compute** v2: v2=(yr×rs)mod pv2 = (y^r \times r^s) \mod p.
- **Verify**: Check if v1 equals v2.

4. **Main Execution**:

- **Key Generation**: Generate public and private keys and display them.
- **Input**: Read the message to sign.
- **Sign**: Generate and display the signature.
- **Verify**: Verify the signature and display the result.

LAB_11\ElGamal_signature.py

```python
# 21CSE1003 Ashish Singh

# Implement Elgamal digital Signature Algorithm.

from math import gcd

def mod_exp(base, exp, mod):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result

# ElGamal Key Generation
def key_generation():
    p = 467  # Should be a large prime number
    g = 2
    x = 3
    y = mod_exp(g, x, p)  # Public key component y = g^x % p
    print(y)
    return p, g, y, x

# ElGamal Signature Generation
def sign(message, p, g, x):
    from hashlib import sha256
    import random
    k = random.randint(1, p-2)
    while gcd(k, p-1) != 1:
        k = random.randint(1, p-2)

    r = mod_exp(g, k, p)
    k_inv = mod_exp(k, p - 2, p - 1)
    m_hash = int(sha256(message.encode()).hexdigest(), 16)
    s = (k_inv * (m_hash - x * r)) % (p - 1)

    return r, s

# ElGamal Signature Verification
def verify(message, signature, p, g, y):
    from hashlib import sha256
    r, s = signature
    if not (0 < r < p and 0 < s < p - 1):
        return False

    m_hash = int(sha256(message.encode()).hexdigest(), 16)
    v1 = mod_exp(g, m_hash, p)
```

```python
    v2 = (mod_exp(y, r, p) * mod_exp(r, s, p)) % p

    return v1 == v2

if __name__ == "__main__":
    p, g, y, x = key_generation()
    print(f"Public Key (p, g, y): ({p}, {g}, {y})")
    print(f"Private Key (x): {x}")

    message = input("Enter the message to sign: ")

    signature = sign(message, p, g, x)
    print(f"Signature (r, s): {signature}")

    is_valid = verify(message, signature, p, g, y)
    print(f"Is the signature valid? {is_valid}")

```