21CSE1003 Ashish Singh

Q 1. Implement the Knapsack Algorithm.

 Modular Inverse Function:

- Define mod_inverse(w, M) to calculate the modular inverse of w with respect to M using the Extended Euclidean Algorithm.

 Input Data:

- Read the binary message pt.
- Read the number of groups in the knapsack grps.
- Generate the private key by taking grps inputs.
- Read M and w (both must be coprime).

 Generate Public Key:

- For each private key value pk, calculate the corresponding public key value as (pk * w) % M.

 Encryption:

- Divide the binary message into groups of size grps.
- For each group, compute the sum of the corresponding public key values where the group bits are '1'.
- Append this sum to the ciphertext.

 Decrypt Message:

- Calculate the modular inverse of w with respect to M as w_inverse.
- For each value in the ciphertext, multiply it by w_inverse and take modulo M to get the decrypted sum.
- For each decrypted sum, reconstruct the group bits by iterating through the private key in reverse and checking if the current value can be subtracted from the sum.

 Output:

- Print the encrypted message.
- Print the decrypted message.

LAB_10\knapsack.py

```python
def mod_inverse(w, M):
    m0, y, x = M, 0, 1
    while w > 1:
        q, w, M = w // M, M, w % M
        y, x = x - q * y, y
    return x + m0 if x < 0 else x

# Get inputs
pt = input("Enter the binary message: ")
grps = int(input("Enter the number of groups in knapsack: "))
private_key = [int(input(f"Enter private key value {i + 1}: ")) for i in range(grps)]
M, w = map(int, input("Enter M and w (coprime) separated by space: ").split())

# Generate public key
public_key = [(pk * w) % M for pk in private_key]

# Encrypt message
cipher_text = []
for i in range(0, len(pt), grps):
    group = pt[i:i + grps].ljust(grps, '0')  # Pad if the last group is shorter
    sum_encryption = sum(public_key[j] for j, bit in enumerate(group) if bit == '1')
    cipher_text.append(sum_encryption)
print("Encrypted Message:", cipher_text)

# Decrypt message
w_inverse = mod_inverse(w, M)
decrypted_message = ''
for c in cipher_text:
    sum_decryption = (c * w_inverse) % M
    group_bits = []
    for k in reversed(private_key):
        if sum_decryption >= k:
            sum_decryption -= k
            group_bits.append('1')
        else:
            group_bits.append('0')
    decrypted_message += ''.join(reversed(group_bits))

print("Decrypted Message:", decrypted_message)

```

Q 2. Implement the Elgamal Algorithm.

▯ **Modular Exponentiation**:

- Define mod_exp(base, exp, mod) to calculate baseexpmod mod\text{base}^{\text{exp}} \mod \text{mod} using iterative squaring.

▯ **Key Generation**:

- Define key_generation() to generate public and private keys:
  - Set prime p, base g, and private key x.
  - Calculate public key component y as y=gxmod py = g^x \mod p.
  - Return the values pp, gg, yy, and xx.

▯ **Encryption**:

- Define encrypt(m, p, g, y) to encrypt a message m using the public key:
  - Choose a random integer k.
  - Calculate ciphertext components c1=gkmod pc1 = g^k \mod p and c2=m×ykmod pc2 = m \times y^k \mod p.
  - Return the ciphertext as a tuple (c1,c2)(c1, c2).

▯ **Decryption**:

- Define decrypt(ciphertext, p, x) to decrypt a ciphertext using the private key:
  - Extract c1 and c2 from the ciphertext.
  - Calculate shared secret s as s=c1xmod ps = c1^x \mod p.
  - Calculate modular inverse of s as sinv=sp−2mod ps_{\text{inv}} = s^{p-2} \mod p.
  - Compute the plaintext m as m=(c2×sinv)mod pm = (c2 \times s_{\text{inv}}) \mod p.
  - Return the decrypted plaintext m.

▯ **Main Function**:

- Generate keys using key_generation().
- Display the public and private keys.
- Encrypt a user-provided message and display the ciphertext.
- Decrypt the ciphertext and display the decrypted message.

LAB_10\ElGamal.py

```python

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 = 9
    g = 2
    x = 3
    y = mod_exp(g, x, p)  # Public key component y = g^x % p
    return p, g, y, x

# ElGamal Encryption
def encrypt(m, p, g, y):
    k = 4
    c1 = mod_exp(g, k, p)
    c2 = (m * mod_exp(y, k, p))
    return c1, c2

# ElGamal Decryption
def decrypt(ciphertext, p, x):
    c1, c2 = ciphertext
    s = mod_exp(c1, x, p)
    s_inv = mod_exp(s, p - 2, p)
    m = (c2 * s_inv) % p
    return m

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 = int(input("Enter the message to encrypt (as an integer < p): "))

    ciphertext = encrypt(message, p, g, y)
    print(f"Encrypted Message (c1, c2): {ciphertext}")

    decrypted_message = decrypt(ciphertext, p, x)
    print(f"Decrypted Message: {decrypted_message}")
```