21CSE1003 Ashish Singh

Q1. Write a program to implement the DES algorithm.

1. **Initial Permutation:**
    - Apply the initial permutation to the 64-bit block using a predefined IP table.
2. **Key Generation:**
    - Apply Permuted Choice 1 (PC-1) to the 64-bit key to get a 56-bit key.
    - Split the 56-bit key into two 28-bit halves.
    - Perform left circular shifts on each half according to predefined shift values.
    - Apply Permuted Choice 2 (PC-2) to each combined 56-bit key to generate 16 round keys.
3. **Feistel Round Function:**
    - Expand the right half of the block to 48 bits using an expansion table.
    - XOR the expanded right half with the subkey.
    - Apply the S-box substitution to the result.
    - Apply a permutation to the S-box output.
    - XOR the result with the left half of the block to get the new right half.
    - Swap the left and right halves.
4. **Encryption:**
    - Apply 16 Feistel rounds to the 64-bit block using the 16 round keys.
    - Perform a final permutation using a predefined FP table.
5. **Decryption:**
    - Perform the same steps as encryption but use the round keys in reverse order.
6. **Main Function:**
    - Take plaintext and key as inputs.
    - Convert them to 64-bit binary strings.
    - Encrypt the plaintext using the generated keys.
    - Convert the encrypted binary string back to text and print it.
    - Decrypt the encrypted text and print the original plaintext.

LAB_9\DES.py

```
# 21CSE1003 Ashish Singh

# Q1. Write a program to implement the DES algorithm.

def xor(bits1, bits2):
    return ''.join(['0' if b1 == b2 else '1' for b1, b2 in zip(bits1, bits2)])

def initial_permutation(block):
    IP = [58, 50, 42, 34, 26, 18, 10, 2,
          60, 52, 44, 36, 28, 20, 12, 4,
          62, 54, 46, 38, 30, 22, 14, 6,
          64, 56, 48, 40, 32, 24, 16, 8,
          57, 49, 41, 33, 25, 17, 9, 1,
          59, 51, 43, 35, 27, 19, 11, 3,
          61, 53, 45, 37, 29, 21, 13, 5,
          63, 55, 47, 39, 31, 23, 15, 7]
    return ''.join(block[i - 1] for i in IP)

def permuted_choice_1(key):
    PC1 = [57, 49, 41, 33, 25, 17, 9,
           1, 58, 50, 42, 34, 26, 18,
           10, 2, 59, 51, 43, 35, 27, 19,
           11, 3, 60, 52, 44, 36, 63, 55,
           47, 39, 31, 23, 15, 7, 62, 54,
           46, 38, 30, 22, 14, 6, 61, 53,
           45, 37, 29, 21, 13, 5, 28, 20,
           12, 4]
    return ''.join(key[i - 1] for i in PC1)

def permuted_choice_2(key):
    PC2 = [14, 17, 11, 24, 1, 5, 3, 28,
           15, 6, 21, 10, 23, 19, 12, 4,
           26, 8, 16, 7, 27, 20, 13, 2,
           41, 52, 31, 37, 47, 55, 30, 40,
           51, 45, 33, 48, 44, 49, 39, 56,
           34, 53, 46, 42, 50, 36, 29, 32]
    return ''.join(key[i - 1] for i in PC2)

def left_circular_shift(bits, shifts):
    return bits[shifts:] + bits[:shifts]

def generate_keys(key):
    permuted_key = permuted_choice_1(key)
    left, right = permuted_key[:28], permuted_key[28:]
    shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
    keys = []
    for shift in shifts:
        left = left_circular_shift(left, shift)
```

```
        right = left_circular_shift(right, shift)
        combined_key = left + right
        round_key = permuted_choice_2(combined_key)
        keys.append(round_key)
    return keys

def expansion(right):
    E = [32, 1, 2, 3, 4, 5, 4, 5,
         6, 7, 8, 9, 8, 9, 10, 11,
         12, 13, 12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21, 20, 21,
         22, 23, 24, 25, 24, 25, 26, 27,
         28, 29, 28, 29, 30, 31, 32, 1]
    return ''.join(right[i - 1] for i in E)

S_BOXES = [
    [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],
    ],
    [
        [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],
    ],
    [
        [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
    ],
    [
        [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
    ],
    [
        [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],
    ],
    [
        [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
```

```python
    return final_permutation(combined)

def final_permutation(block):
    FP = [40, 8, 48, 16, 56, 24, 64, 32,
          39, 7, 47, 15, 55, 23, 63, 31,
          38, 6, 46, 14, 54, 22, 62, 30,
          37, 5, 45, 13, 53, 21, 61, 29,
          36, 4, 44, 12, 52, 20, 60, 28,
          35, 3, 43, 11, 51, 19, 59, 27,
          34, 2, 42, 10, 50, 18, 58, 26,
          33, 1, 41, 9, 49, 17, 57, 25]
    return ''.join(block[i - 1] for i in FP)

def get_64_bit_binary_string(input_str):
    return ''.join(format(ord(c), '08b') for c in input_str).ljust(64, '0')[:64]

def get_64_bit_key(key_str):
    return ''.join(format(ord(c), '08b') for c in key_str).ljust(64, '0')[:64]

def des_decrypt(block, keys):
    block = initial_permutation(block)
    left, right = block[:32], block[32:]
    for i in range(15, -1, -1):  # Decryption uses the keys in reverse order
        left, right = feistel_round(left, right, keys[i])
    combined = right + left  # Swap left and right before final permutation
    return final_permutation(combined)

def main():
    plaintext = input("Enter plaintext (max 8 characters): ")
    key = input("Enter key (max 8 characters): ")
    block = get_64_bit_binary_string(plaintext)
    keys = generate_keys(get_64_bit_key(key))

    # Encryption
    encrypted_block = des_encrypt(block, keys)
    encrypted_text = ''.join(chr(int(encrypted_block[i:i+8], 2)) for i in range(0,
len(encrypted_block), 8))
    print(f"Encrypted text: {encrypted_text}")

    # Decryption
    decrypted_block = des_decrypt(encrypted_block, keys)
    decrypted_text = ''.join(chr(int(decrypted_block[i:i+8], 2)) for i in range(0,
len(decrypted_block), 8))
    print(f"Decrypted text: {decrypted_text}")

if __name__ == "__main__":
    main()
```

```
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],
    ],
    [
        [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
        [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
        [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
        [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],
    ],
    [
        [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
        [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
        [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
        [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],
    ],
]

def s_box_substitution(block):
    segments = [block[i:i+6] for i in range(0, len(block), 6)]
    output = ''
    for i, segment in enumerate(segments):
        row = int(segment[0] + segment[5], 2)
        col = int(segment[1:5], 2)
        output += bin(S_BOXES[i][row][col])[2:].zfill(4)
    return output

def permutation(block):
    P = [16, 7, 20, 21,
         29, 12, 28, 17,
         1, 15, 23, 26,
         5, 18, 31, 10,
         2, 8, 24, 14,
         32, 27, 3, 9,
         19, 13, 30, 6,
         22, 11, 4, 25]
    return ''.join(block[i - 1] for i in P)

def feistel_round(left, right, subkey):
    expanded_right = expansion(right)
    xored = xor(expanded_right, subkey)
    substituted = s_box_substitution(xored)
    permuted = permutation(substituted)
    new_right = xor(left, permuted)
    return right, new_right

def des_encrypt(block, keys):
    block = initial_permutation(block)
    left, right = block[:32], block[32:]
    for i in range(16):
        left, right = feistel_round(left, right, keys[i])
    combined = right + left  # Swap left and right before final permutation
```

Q2. Implement the Diffie-Hellman Key Exchange mechanism.

1. **Input Parameters:**
    - Read prime number p.
    - Read primitive root g.
    - Read private key a for Alice.
    - Read private key b for Bob.
2. **Calculate Public Keys:**
    - Alice calculates her public key A as A=gamod p.
    - Bob calculates his public key B as B=gbmod p.
3. **Exchange Public Keys:**
    - Alice sends her public key A to Bob.
    - Bob sends his public key B to Alice.
4. **Compute Shared Secret Keys:**
    - Alice computes the shared secret key Ka as Ka=Bamod p.
    - Bob computes the shared secret key Kb as Kb=Abmod p.
5. **Output Shared Secret Keys:**
    - Print the shared secret keys Ka and Kb.

LAB_9\Diffie-Hellman_key_exchange.py

```python

# Q2. Implement the Diffie-Hellman Key Exchange mechanism.

p = int(input("Enter the value of p: "))
g = int(input("Enter the value of g: "))

a = int(input(f"Enter a value for Alice between 0 and {p - 1}: "))
b = int(input(f"Enter b value for Bob between 0 and {p - 1}: "))

A = g ** a % p
B = g ** b % p

Ka = B ** a % p
Kb = A ** b % p

print(f"The keys are: {Ka} & {Kb}")

```

Q3. Write a program to implement RSA Algorithm.

1. **Input Prime Numbers:**
    - Read two distinct large prime numbers p and q.
2. **Calculate** n**:**
    - Compute n as $n=p\times q$.
3. **Compute Euler's Totient Function:**
    - Calculate the totient function $\varphi(n)=(p-1)\times(q-1)$.
4. **Select Public Key Exponent** e**:**
    - Choose an integer e such that $1<e<\varphi(n)$ and e is coprime with $\varphi(n)$.
5. **Compute Private Key Exponent** d**:**
    - Calculate d such that $(d\times e) \bmod \varphi(n)=1$.
6. **Encryption:**
    - Public key is (e,n).
    - Read the message m as a number.
    - Compute the ciphertext c as $C=memod\ n$.
7. **Decryption:**
    - Private key is (d,n).
    - Compute the plaintext m as $M=cdmod\ n$.

LAB_9\RSA_algorithm.py

```python

# Q3. Write a program to implement RSA Algorithm.

from math import gcd

p, q = (map(int, input("Enter two distinct large prime numbers: ").split()))

print(f"n = p x q")
print(f"n = {p} x {q}")
n = p * q
print(f"n = {n}")

def euler_totient(p, q):
    """
    As p and q are prime numbers we can calculate Euler Totient using (p - 1) * (q - 1)
    """
    phi = (p - 1) * (q - 1)
    return phi

print(f"\nphi(n) = (p - 1) x (q - 1)")
print(f"phi({n}) = ({p} - 1) x ({q} - 1)")
phi = euler_totient(p, q)
print(f"phi({n}) = {phi}")

e = int(input(f"\nChoose an integer such that 1 < e < {phi} and e and {phi} are coprime: "))

print(f"\nd, such that (d * {e} mod {phi} = 1)")
d = 1
while (e * d) % phi != 1:
    d += 1
print(f"d = {d}")

print("\nEncryption: ")
print(f"Public key: <{e}, {n}>")
m = int(input("Messaage represented as a number: "))
print(f"Ciphertext: C = M ^ e mod n")
print(f"Ciphertext: C = {m} ^ {e} mod {n}")
c = m ** e % n
print(f"Ciphertext: C = {c}")

print("\nDecryption: ")
print(f"Private key: <{d}, {n}>")
print(f"Ciphertext represented as a number: {c}")
print(f"Plaintext: M = C ^ d mod n")
print(f"Plaintext: M = {c} ^ {d} mod {n}")
m = c ** d % n
print(f"Plaintext: M = {m}")
```