CIS Practicals

PRACTICAL NO. 1

Aim: Write a program to implement following:

A) Chinese Reminder Theorem

B) Fermat's Little Theorem

A) Chinese Reminder Theorem

INPUT:-

```
def findMinX(num, rem, k):
   x = 1;
   while(True):
      j = 0;
      while(j < k):
         if (x % num[j] != rem[j]):
            break;
         j += 1;
      if (j == k):
         return x;
      x += 1;

# Driver Code
num = [3, 4, 5];
rem = [2, 3, 1];
k = len(num);
print("x is", findMinX(num, rem, k));
```

OUTPUT:-

```
>>>
==== RESTART: C:\Users\Administrator\Desktop\python_program\CIS_Pract1(A).py
x is 11
>>> |
```

B) Fermat's Little Theorem

INPUT:-

```
def __gcd(a, b):
   if(b == 0):
      return a
   else:
      return __gcd(b, a % b)
def power(x, y, m):
   if (y == 0):
      return 1
   p = power(x, y // 2, m) % m
   p = (p * p) % m
   return p if(y % 2 == 0) else (x * p) % m
def modInverse(a, m):
   if (__gcd(a, m) != 1):
      print("Inverse doesn't exist")
   else:
```

```
    print("Modular multiplicative inverse is ",
        power(a, m - 2, m))

# Driver code
a = 3
m = 11
modInverse(a, m)
```

OUTPUT:-

```
>>>
==== RESTART: C:\Users\Administrator\Desktop\python_program\CIS_Pract1(B).py ===
Modular multiplicative inverse is  4
```

PRACTICAL NO. 6

Aim: Write a program to implement the Diffie-Hellman Key Agreement algorithm to generate symmetric keys

INPUT:-

```
from random import randint
if __name__ == '__main__':
    P = 23
    G = 9
    print('The Value of P is :%d'%(P))
    print('The Value of G is :%d'%(G))
    a = 4
    print('Secret Number for Alice is :%d'%(a))
    x = int(pow(G,a,P))
    b = 6
    print('Secret Number for Bob is :%d'%(b))
    y = int(pow(G,b,P))
    ka = int(pow(y,a,P))
    kb = int(pow(x,b,P))
    print('Secret key for the Alice is : %d'%(ka))
    print('Secret Key for the Bob is : %d'%(kb))
```

OUTPUT:-

```
>>>
===== RESTART: C:\Users\Administrator\Desktop\python_program\CIS_Pract6.py =====
The Value of P is :23
The Value of G is :9
Secret Number for Alice is :4
Secret Number for Bob is :6
Secret key for the Alice is : 12
Secret Key for the Bob is : 12
>>>
```

PRACTICAL NO. 2

Aim: Write a program to implement the

(i) Affine Cipher **(ii) Rail Fence Technique**

(iii) Simple Columnar Technique **(iv) Vermin Cipher**

(v) Hill Cipher to perform encryption and decryption.

(i) <u>Affine Cipher</u>

INPUT:-

```
def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y
def modinv(a, m):
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None  # modular inverse does not exist
    else:
        return x % m
def affine_encrypt(text, key):
    #C = (a*P + b) % 26
    return ''.join([ chr((( key[0]*(ord(t) - ord('A')) + key[1] ) % 26)
                 + ord('A')) for t in text.upper().replace(' ', '') ])
def affine_decrypt(cipher, key):
    P = (a^-1 * (C - b)) % 26
    return ''.join([ chr((( modinv(key[0], 26)*(ord(c) - ord('A') - key[1]))
                   % 26) + ord('A')) for c in cipher ])
def main():
    text = 'AFFINE CIPHER'
    key = [17, 20]
    affine_encrypted_text = affine_encrypt(text, key)
    print('Encrypted Text: {}'.format( affine_encrypted_text ))
```

```
    print('Decrypted Text: {}'.format
    ( affine_decrypt(affine_encrypted_text, key) ))
if __name__ == '__main__':
    main()
```

OUTPUT:-

```
>>>
==== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract2(i).py ===
Encrypted Text: UBBAHKCAPJKX
Decrypted Text: AFFINECIPHER
```

(ii) <u>Rail Fence Technique</u>

INPUT:-

```
def encryptRailFence(text, key):
    print("Encryted text:")
    rail = [['\n' for i in range(len(text))]
                for j in range(key)]
    dir_down = False
    row, col = 0, 0
    for i in range(len(text)):
        if (row == 0) or (row == key - 1):
            dir_down = not dir_down
        rail[row][col] = text[i]
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
    return("" . join(result))
def decryptRailFence(cipher, key):
    print("Dencryted text:")
    rail = [['\n' for i in range(len(cipher))]
                for j in range(key)]
    dir_down = None
    row, col = 0, 0
    for i in range(len(cipher)):
```

```
        if row == 0:
            dir_down = True
        if row == key - 1:
            dir_down = False
        rail[row][col] = '*'
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((rail[i][j] == '*') and
            (index < len(cipher))):
                rail[i][j] = cipher[index]
                index += 1
    result = []
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key-1:
            dir_down = False
        if (rail[row][col] != '*'):
            result.append(rail[row][col])
            col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    return("".join(result))
if __name__ == "__main__":
    print(encryptRailFence("attack at once", 2))
    print(encryptRailFence("defend the east wall", 3))
    print(decryptRailFence("atc toctaka ne", 2))
    print(decryptRailFence("dnhaweedtees alf  tl", 3))
```

OUTPUT:-

```
>>>
=== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract2(ii).py ===
Encryted text:
atc toctaka ne
Encryted text:
dnhaweedtees alf  tl
Dencryted text:
attack at once
Dencryted text:
defend the east wall
```

(iii) <u>Simple Columnar Technique</u>

INPUT:-

```
import math
key = "HACK"
def encryptMessage(msg):
    cipher = ""
    k_indx = 0
    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))
    col = len(key)
    row = int(math.ceil(msg_len / col))
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)
    matrix = [msg_lst[i: i + col]
              for i in range(0, len(msg_lst), col)]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        cipher += ''.join([row[curr_idx]
                           for row in matrix])
        k_indx += 1
    return cipher
def decryptMessage(cipher):
    msg = ""
        k_indx = 0
      msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)
      col = len(key)
      row = int(math.ceil(msg_len / col))
    key_lst = sorted(list(key))
    dec_cipher = []
    for _ in range(row):
        dec_cipher += [[None] * col]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        for j in range(row):
```

```
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
            msg_indx += 1
        k_indx += 1
    try:
        msg = ''.join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot",
                        "handle repeating words.")
    null_count = msg.count('_')
    if null_count > 0:
        return msg[: -null_count]
    return msg
# Driver Code
msg = "MSc Computer Science"
cipher = encryptMessage(msg)
print("Encrypted Message: {}".
      format(cipher))
print("Decryped Message: {}".
      format(decryptMessage(cipher)))
```

OUTPUT:-

```
>>>
=== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract2(iii).py ==
Encrypted Message: SotSncmeccMCu e prie
Decryped Message: MSc Computer Science
```

(iv)<u>Vernam Cipher</u>

INPUT:-

```
import random
import base64
def generate_key(plaintext_length):
    key = ''.join(random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
              for _ in range(plaintext_length))
    return key
def encrypt(plaintext, key):
    ciphertext_bytes = bytes([ord(p) ^ ord(k)
                              for p, k in zip(plaintext, key)])
    ciphertext = base64.b64encode(ciphertext_bytes).decode('utf-8')
    return ciphertext
def decrypt(ciphertext, key):
    ciphertext_bytes = base64.b64decode(ciphertext)
    decrypted_text = ''.join(chr(c ^ ord(k))
                             for c, k in zip(ciphertext_bytes, key))
    return decrypted_text
if __name__ == "__main__":
    plaintext = "HELLO"
```

```
    key = generate_key(len(plaintext))
    print("Plaintext:", plaintext)
    print("Key:", key)
    ciphertext = encrypt(plaintext, key)
    print("Ciphertext:", ciphertext)
    decrypted_text = decrypt(ciphertext, key)
    print("Decrypted Text:", decrypted_text)
```

OUTPUT:-

```
>>>
    === RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract2(iv).py ===
    Plaintext: HELLO
    Key: KZJAE
    Ciphertext: Ax8GDQo=
    Decrypted Text: HELLO
>>> |
```

(v) <u>Hill Cipher to perform encryption and decryption</u>.

INPUT:-

```
keyMatrix = [[0] * 3 for i in range(3)]
messageVector = [[0] for i in range(3)]
cipherMatrix = [[0] for i in range(3)]
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                       messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26
def HillCipher(message, key):
    getKeyMatrix(key)
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65
    encrypt(messageVector)
    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))
```

```
    print("Ciphertext: ", "".join(CipherText))
def main():
    message = "COMPUTER"
    key = "GYBNQKURP"
    HillCipher(message, key)

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

OUTPUT:-

```
>>>
==== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract2(v).py ===
Ciphertext:  WGQ
```

PRACTICAL NO. 3

Aim: Write a program to implement the
(i) RSA Algorithm to perform encryption and decryption.

INPUT:-

```
def power(base, expo, m):
    res = 1
    base = base % m
    while expo > 0:
        if expo & 1:
            res = (res * base) % m
        base = (base * base) % m
        expo = expo // 2
    return res
def modInverse(e, phi):
    for d in range(2, phi):
        if (e * d) % phi == 1:
            return d
    return -1
def generateKeys():
    p = 7919
    q = 1009
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 0
    for e in range(2, phi):
        if gcd(e, phi) == 1:
            break
        d = modInverse(e, phi)
    return e, d, n
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
```

```
    return a
def encrypt(m, e, n):
    return power(m, e, n)
def decrypt(c, d, n):
    return power(c, d, n)
if __name__ == "__main__":
    e, d, n = generateKeys()
    print(f"Public Key (e, n): ({e}, {n})")
    print(f"Private Key (d, n): ({d}, {n})")
    M = 123
    print(f"Original Message: {M}")
    C = encrypt(M, e, n)
    print(f"Encrypted Message: {C}")
    decrypted = decrypt(C, d, n)
    print(f"Decrypted Message: {decrypted}")
```

OUTPUT:-

```
>>>
===== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract3.py =====
Public Key (e, n): (5, 7990271)
Private Key (d, n): (1596269, 7990271)
Original Message: 123
Encrypted Message: 3332110
Decrypted Message: 123
```

PRACTICAL NO 5

Aim: Write a program to implement the ElGamal Cryptosystem to generate keys and perform encryption and decryption

INPUT:

```
import random
from math import gcd
def generate_key(q):
    """Generate a random key that is coprime with q"""
    key = random.randint(1, q - 1)
    while gcd(q, key) != 1:
        key = random.randint(1, q - 1)
    return key
def modular_exponentiation(base, exp, mod):
    """Efficiently compute (base^exp) mod mod"""
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        exp = exp // 2
        base = (base * base) % mod
```

```
    return result
def encrypt(message, q, h, g):
    """Encrypt the message using ElGamal"""
    k = generate_key(q)  # Ephemeral key
    s = modular_exponentiation(h, k, q)  # Shared secret (g^ak)
    p = modular_exponentiation(g, k, q)   # g^k
    encrypted_chars = [s * ord(char) for char in message]
    print("\nEncryption Process:")
    print(f" - g used: {g}")
    print(f" - g^a used (h): {h}")
    print(f" - g^k used (p): {p}")
    print(f" - g^ak used (s): {s}")
    return encrypted_chars, p
def decrypt(encrypted_msg, p, private_key, q):
    """Decrypt the message using ElGamal"""
    h = modular_exponentiation(p, private_key, q)  # Regenerate shared secret
    decrypted_chars = [chr(value // h) for value in encrypted_msg]
    return decrypted_chars
def main():
    print("ELGAMAL ENCRYPTION DEMO")
    print("======================")
    # Original message
    message = "encryption"
    print(f"\nOriginal Message: '{message}'")
    q = random.randint(1000, 10000)  # Prime modulus (in practice should be large prime)
    g = random.randint(2, q - 1)     # Generator
    private_key = generate_key(q)    # Private key (a)
    h = modular_exponentiation(g, private_key, q)  # Public key (g^a)
    encrypted_msg, p = encrypt(message, q, h, g)
    print(f"\nEncrypted Text: {encrypted_msg}")
    decrypted_chars = decrypt(encrypted_msg, p, private_key, q)
    decrypted_msg = ''.join(decrypted_chars)
    print(f"\nDecrypted Message: '{decrypted_msg}'")
if __name__ == '__main__':
    main()
```

OUTPUT:

```
>>>
===== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract5.py =====
ELGAMAL ENCRYPTION DEMO
=======================

Original Message: 'encryption'

Encryption Process:
 - g used: 92
 - g^a used (h): 5624
 - g^k used (p): 680
 - g^ak used (s): 434

Encrypted Text: [43834, 47740, 42966, 49476, 52514, 48608, 50344, 45570, 48174, 47740]

Decrypted Message: 'encryption'
>>>
```

PRACTICAL NO. 7

Aim: Write a program to implement the MD5 algorithm compute the message digest.

INPUT:-

```
import hashlib
def compute_md5(message):
    """Compute the MD5 hash of a given message."""
    md5_hash = hashlib.md5()
    md5_hash.update(message.encode('utf-8'))
    return md5_hash.hexdigest()
def main():
    print("MD5 Hash Computation")
    print("=====================")
    message = input("Enter the message to hash: ")
    md5_digest = compute_md5(message)
    print(f"MD5 Digest: {md5_digest}")
if __name__ == '__main__':
    main()
```

OUTPUT:-

```
>>>
===== RESTART: C:/Users/Administrator/Desktop/python_program/CIS_Pract7.py =====
MD5 Hash Computation
=====================
Enter the message to hash: MSc Computer science
MD5 Digest: 6b61bb4f3548cefa09e8877d1d632ad2
>>>
```

PRACTICAL NO. 8

Aim: Write a program to implement different processes of DES algorithm like
(i) Initial Permutation process of DES algorithm,
(ii) Generate Keys for DES algorithm,
(iii) S-Box substitution for DES algorithm

INPUT:-

```
import random
# Utility functions
def hex2bin(s):
    """Convert hexadecimal to binary"""
    mp = {'0': "0000", '1': "0001", '2': "0010", '3': "0011",
          '4': "0100", '5': "0101", '6': "0110", '7': "0111",
          '8': "1000", '9': "1001", 'A': "1010", 'B': "1011",
          'C': "1100", 'D': "1101", 'E': "1110", 'F': "1111"}
    binary = ""
    for ch in s:
        binary += mp[ch]
    return binary
def bin2hex(s):
    """Convert binary to hexadecimal"""
    mp = {"0000": '0', "0001": '1', "0010": '2', "0011": '3',
          "0100": '4', "0101": '5', "0110": '6', "0111": '7',
          "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
          "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'}
    hex_str = ""
    for i in range(0, len(s), 4):
        chunk = s[i:i+4]
        hex_str += mp[chunk]
    return hex_str
def permute(k, arr, n):
    """Permute the input bits according to the permutation table"""
    permutation = ""
    for i in range(n):
        permutation += k[arr[i] - 1]
    return permutation
def shift_left(k, nth_shifts):
    """Left shift the bits"""
    s = ""
    for _ in range(nth_shifts):
        s = k[1:] + k[0]
        k = s
    return k
# (i) Initial Permutation Process
def initial_permutation(plain_text):
```

```
    """Perform initial permutation on the plaintext"""
    # Initial Permutation Table (64 bits)
    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]

    # Convert plaintext to binary
    binary_pt = hex2bin(plain_text)
    # Perform permutation
    permuted = permute(binary_pt, IP, 64)
    return bin2hex(permuted)
```

 # (ii) Key Generation Process

```
def generate_keys(key):
    """Generate 16 round keys for DES"""
    # Parity bit drop table (64 bits to 56 bits)
    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]

    # Key compression table (56 bits to 48 bits)
    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]

    # Number of left shifts for each round
    SHIFT_TABLE = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
    # Convert key to binary
    binary_key = hex2bin(key)
    # Apply PC1 permutation
    key_56 = permute(binary_key, PC1, 56)
```

```
    # Split into left and right halves
    left = key_56[:28]
    right = key_56[28:]
    round_keys = []
    # Generate 16 round keys
    for i in range(16):
        # Left shift both halves
        left = shift_left(left, SHIFT_TABLE[i])
        right = shift_left(right, SHIFT_TABLE[i])
        # Combine and apply PC2 permutation
        combined = left + right
        round_key = permute(combined, PC2, 48)
        round_keys.append(bin2hex(round_key))
    return round_keys
```

 # (iii) S-Box Substitution Process

```
def s_box_substitution(input_48bit):
    """Perform S-Box substitution on 48-bit input"""
    # S-Box tables (8 boxes, each 4x16)
    SBOX = [
        # S1
        [[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]],
        # S2
        [[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]],
        # S3
        [[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]],
        # S4
        [[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]],
        # S5
        [[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]],
    # S6
    [[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],
     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],
    # S7
    [[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]],
    # S8
    [[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]]
  ]

  binary_input = hex2bin(input_48bit)
  output_32bit = ""

  # Process each 6-bit chunk with corresponding S-Box
  for i in range(8):
    chunk = binary_input[i*6:(i+1)*6]
    # Calculate row and column
    row = int(chunk[0] + chunk[5], 2)
    col = int(chunk[1:5], 2)
    # Get value from S-Box
    val = SBOX[i][row][col]
    # Convert to 4-bit binary
    output_32bit += format(val, '04b')
  return bin2hex(output_32bit)

# Main function to demonstrate all processes
def main():
  print("DES Algorithm Components Demonstration")
  print("====================================")

  # Sample plaintext and key (64-bit hexadecimal)
  plaintext = "0123456789ABCDEF"
  key = "133457799BBCDFF1"

  # (i) Initial Permutation
  print("\n1. Initial Permutation Process:")
  print(f"Original Plaintext: {plaintext}")
  permuted = initial_permutation(plaintext)
```

```
    print(f"After Initial Permutation: {permuted}")
    # (ii) Key Generation
    print("\n2. Key Generation Process:")
    print(f"Original Key: {key}")
    round_keys = generate_keys(key)
    print("Generated Round Keys:")
    for i, rk in enumerate(round_keys):
        print(f"Round {i+1:2d}: {rk}")
    # (iii) S-Box Substitution
    print("\n3. S-Box Substitution Process:")
    sample_input = "6D5A8C1A2B3E"  # 48-bit input
    print(f"Input (48-bit): {sample_input}")
    sbox_output = s_box_substitution(sample_input)
    print(f"Output (32-bit): {sbox_output}")

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

OUTPUT:-

```
>>>
===== RESTART: C:\Users\Administrator\Desktop\python_program\CIS_Pract8.py =====
DES Algorithm Components Demonstration
=====================================

1. Initial Permutation Process:
Original Plaintext: 0123456789ABCDEF
After Initial Permutation: CC00CCFFF0AAF0AA

2. Key Generation Process:
Original Key: 133457799BBCDFF1
Generated Round Keys:
Round  1: 1B02EFFC7072
Round  2: 79AED9DBC9E5
Round  3: 55FC8A42CF99
Round  4: 72ADD6DB351D
Round  5: 7CEC07EB53A8
Round  6: 63A53E507B2F
Round  7: EC84B7F618BC
Round  8: F78A3AC13BFB
Round  9: E0DBEBEDE781
Round 10: B1F347BA464F
Round 11: 215FD3DED386
Round 12: 7571F59467E9
Round 13: 97C5D1FABA41
Round 14: 5F43B7F2E73A
Round 15: BF918D3D3F0A
Round 16: CB3D8B0E17F5

3. S-Box Substitution Process:
Input (48-bit): 6D5A8C1A2B3E
Output (32-bit): 51F91E78
```