**1.** **Write a program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used**

**in reverse order. Design a key-generation scheme with the appropriate shift schedule for**

**the decryption process.**

**CODE:**

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

def pad\_message(message, block\_size):

"""Pads the message using '1' followed by '0's up to the next block size."""

padding\_length = block\_size - (len(message) % block\_size)

if padding\_length == 0:

padding\_length = block\_size

padding = b'\x80' + b'\x00' \* (padding\_length - 1)

return message + padding

def des\_encrypt(plaintext, key):

"""Encrypts using DES in ECB mode (for simplicity)."""

cipher = DES.new(key, DES.MODE\_ECB)

padded\_text = pad\_message(plaintext, DES.block\_size)

return cipher.encrypt(padded\_text)

def des\_decrypt(ciphertext, key):

"""Decrypts using DES with keys applied in reverse order."""

cipher = DES.new(key, DES.MODE\_ECB)

decrypted\_text = cipher.decrypt(ciphertext)

return decrypted\_text.rstrip(b'\x80\x00')

key = get\_random\_bytes(8)

plaintext = b"SecretMsg"

ciphertext = des\_encrypt(plaintext, key)

print("Encrypted:", ciphertext.hex())

decrypted\_text = des\_decrypt(ciphertext, key)

print("Decrypted:", decrypted\_text.decode())

**OUTPUT:**



**2. Write a program for encryption in the cipher block chaining (CBC) mode using an**

**algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the**

**definition of CBC. Which of the two would you choose:**

1. **For security? b. For performance?**

**CODE:**

from Crypto.Cipher import DES3

from Crypto.Random import get\_random\_bytes

def pad\_message(message, block\_size):

"""Pads the message using '1' followed by '0's up to the next block size."""

padding\_length = block\_size - (len(message) % block\_size)

if padding\_length == 0:

padding\_length = block\_size

padding = b'\x80' + b'\x00' \* (padding\_length - 1)

return message + padding

def cbc\_encrypt\_3des(plaintext, key, iv):

cipher = DES3.new(key, DES3.MODE\_CBC, iv)

padded\_text = pad\_message(plaintext, DES3.block\_size)

return cipher.encrypt(padded\_text)

key = DES3.adjust\_key\_parity(get\_random\_bytes(24))

iv = get\_random\_bytes(8)

plaintext = b"SensitiveData1234"

ciphertext = cbc\_encrypt\_3des(plaintext, key, iv)

print("3DES CBC Encrypted:", ciphertext**.**hex())

**OUTPUT:**



**3. Write a program for ECB, CBC, and CFB modes, the plaintext must be a sequence of**

**one or more complete data blocks (or, for CFB mode, data segments). In other words, for**

**these three modes, the total number of bits in the plaintext must be a positive multiple of**

**the block (or segment) size. One common method of padding, if needed, consists of a 1**

**bit followed by as few zero bits, possibly none, as are necessary to complete the final**

**block. It is considered good practice for the sender to pad every message, including**

**messages in which the final message block is already complete. What is the motivation**

**for including a padding block when padding is not needed?**

**CODE:**

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

def pad\_message(message, block\_size):

"""Pads the message using '1' followed by '0's up to the next block size."""

padding\_length = block\_size - (len(message) % block\_size)

if padding\_length == 0:

padding\_length = block\_size

padding = b'\x80' + b'\x00' \* (padding\_length - 1)

return message + padding

def ecb\_encrypt(plaintext, key):

cipher = AES.new(key, AES.MODE\_ECB)

padded\_text = pad\_message(plaintext, AES.block\_size)

return cipher.encrypt(padded\_text)

def cbc\_encrypt(plaintext, key, iv):

cipher = AES.new(key, AES.MODE\_CBC, iv)

padded\_text = pad\_message(plaintext, AES.block\_size)

return cipher.encrypt(padded\_text)

def cfb\_encrypt(plaintext, key, iv, segment\_size=8):

cipher = AES.new(key, AES.MODE\_CFB, iv, segment\_size=segment\_size)

return cipher.encrypt(plaintext)

key = get\_random\_bytes(16)

iv = get\_random\_bytes(16)

plaintext = b"ConfidentialData123"

ciphertext\_ecb = ecb\_encrypt(plaintext, key)

ciphertext\_cbc = cbc\_encrypt(plaintext, key, iv)

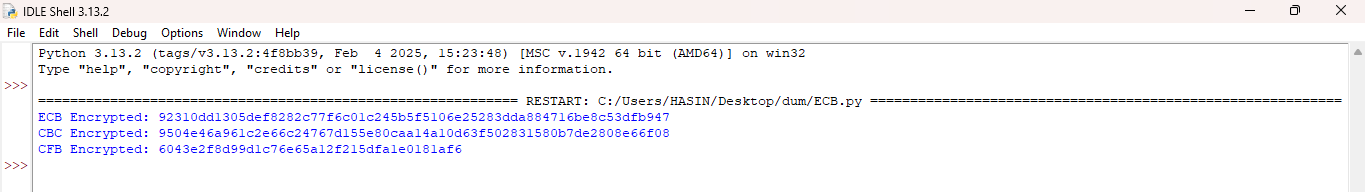
ciphertext\_cfb = cfb\_encrypt(plaintext, key, iv)

print("ECB Encrypted:", ciphertext\_ecb.hex())

print("CBC Encrypted:", ciphertext\_cbc.hex())

print("CFB Encrypted:", ciphertext\_cfb.hex())

**OUTPUT:**

****

**4.Write a program for Encrypt and decrypt in cipher block chaining mode using one of the**

**following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-**

**DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001**

**0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of**

**1111 0100 0000 1011. Decryption should work correspondingly.**

**CODE:**

def binary\_to\_decimal(binary\_str):

return int(binary\_str, 2)

def decimal\_to\_binary(decimal, length=8):

return format(decimal, f'0{length}b')

def xor(bin\_str1, bin\_str2):

return ''.join('0' if b1 == b2 else '1' for b1, b2 in zip(bin\_str1, bin\_str2))

def sdes\_encrypt\_block(plain\_block, key):

return xor(plain\_block, key[:len(plain\_block)])

def sdes\_decrypt\_block(cipher\_block, key):

return xor(cipher\_block, key[:len(cipher\_block)])

def cbc\_encrypt(plaintext\_bin, key\_bin, iv\_bin):

block\_size = len(iv\_bin)

ciphertext\_bin = ""

prev\_block = iv\_bin

for i in range(0, len(plaintext\_bin), block\_size):

block = plaintext\_bin[i:i + block\_size].ljust(block\_size, '0')

xored\_block = xor(block, prev\_block)

encrypted\_block = sdes\_encrypt\_block(xored\_block, key\_bin)

ciphertext\_bin += encrypted\_block

prev\_block = encrypted\_block

return ciphertext\_bin

def cbc\_decrypt(ciphertext\_bin, key\_bin, iv\_bin):

block\_size = len(iv\_bin)

plaintext\_bin = ""

prev\_block = iv\_bin

for i in range(0, len(ciphertext\_bin), block\_size):

block = ciphertext\_bin[i:i + block\_size]

decrypted\_block = sdes\_decrypt\_block(block, key\_bin)

xored\_block = xor(decrypted\_block, prev\_block)

plaintext\_bin += xored\_block

prev\_block = block

return plaintext\_bin

plaintext\_bin = "0000000100100011"

key\_bin = "0111111101".ljust(16, '0')

iv\_bin = "10101010".ljust(16, '0')

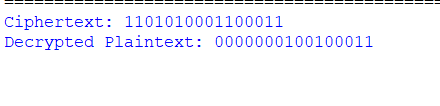
ciphertext\_bin = cbc\_encrypt(plaintext\_bin, key\_bin, iv\_bin)

decrypted\_bin = cbc\_decrypt(ciphertext\_bin, key\_bin, iv\_bin)

print (f"Ciphertext: {ciphertext\_bin}")

print(f"Decrypted Plaintext: {decrypted\_bin}")

**OUTPUT:**



**5. Write a program for RSA system, the public key of a given user is e = 31, n = 3599. What**

**is the private key of this user? Hint: First use trial-and-error to determine p and q; then**

**use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).**

**CODE:**

import math

def factorize\_n(n):

for p in range(2, int(math.sqrt(n)) + 1):

if n % p == 0:

q = n // p

return p, q

return None, None

def extended\_gcd(a, b):

if a == 0:

return b, 0, 1

gcd, x1, y1 = extended\_gcd(b % a, a)

x = y1 - (b // a) \* x1

y = x1

return gcd, x, y

def modular\_inverse(e, phi):

gcd, x, y = extended\_gcd(e, phi)

if gcd != 1:

return None # No modular inverse exists

return x % phi

def rsa\_private\_key(e, n):

p, q = factorize\_n(n)

if not p or not q:

return "Factorization failed"

phi\_n = (p - 1) \* (q - 1)

d = modular\_inverse(e, phi\_n)

return d

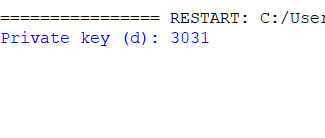
e = 31

n = 3599

d = rsa\_private\_key(e, n)

print("Private key (d):", d)

**OUTPUT:**

****

**6. Write a program for Diffie-Hellman protocol, each participant selects a secret number x**

**and sends the other participant ax mod q for some public number a. What would happen**

**if the participants sent each other xa for some public number a instead? Give at least one**

**method Alice and Bob could use to agree on a key. Can Eve break your system without**

**finding the secret numbers? Can Eve find the secret numbers?**

**CODE:**

import random

def diffie\_hellman(p, g):

a = random.randint(2, p-2)

b = random.randint(2, p-2)

A = pow(g, a, p)

B = pow(g, b, p)

shared\_secret\_alice = pow(B, a, p)

shared\_secret\_bob = pow(A, b, p)

return a, b, A, B, shared\_secret\_alice, shared\_secret\_bob

p = 23 # A prime number (public)

g = 5 # A primitive root modulo p (public)

a, b, A, B, shared\_secret\_alice, shared\_secret\_bob = diffie\_hellman(p, g)

print(f"Alice's Private Key: {a}")

print(f"Bob's Private Key: {b}")

print(f"Alice's Public Key: {A}")

print(f"Bob's Public Key: {B}")

print(f"Shared Secret Computed by Alice: {shared\_secret\_alice}")

print(f"Shared Secret Computed by Bob: {shared\_secret\_bob}")

**OUTPUT**

