1. **Aim :-** The aim of your program is to demonstrate the working of the Hill Cipher — a classical encryption technique based on linear algebra over modulo arithmetic

**Program :**

import numpy as np

def mod\_inverse(a, m):

"""Find modular inverse of a under modulo m."""

for i in range(1, m):

if (a \* i) % m == 1:

return i

return None

def matrix\_mod\_inverse(matrix, mod):

"""Calculate the modular inverse of a 2x2 matrix under mod."""

det = int(np.round(np.linalg.det(matrix))) % mod

det\_inv = mod\_inverse(det, mod)

if det\_inv is None:

raise ValueError("Matrix is not invertible under mod 26.")

# Adjugate of 2x2 matrix

adj = np.array([[matrix[1][1], -matrix[0][1]],

[-matrix[1][0], matrix[0][0]]])

inv = (det\_inv \* adj) % mod

return inv

def text\_to\_numbers(text):

"""Convert text to numbers (A=0, B=1, ..., Z=25)."""

return [ord(c) - ord('A') for c in text.upper() if c.isalpha()]

def numbers\_to\_text(numbers):

"""Convert numbers back to text (0=A, 1=B, ..., 25=Z)."""

return ''.join([chr(n % 26 + ord('A')) for n in numbers])

def hill\_encrypt(plaintext, key\_matrix):

"""Encrypt plaintext using Hill cipher and key matrix."""

nums = text\_to\_numbers(plaintext)

if len(nums) % 2 != 0:

nums.append(ord('X') - ord('A')) # Padding if needed

ciphertext = []

for i in range(0, len(nums), 2):

pair = np.array([[nums[i]], [nums[i + 1]]])

result = np.dot(key\_matrix, pair) % 26

ciphertext.extend(result.flatten())

return numbers\_to\_text(ciphertext)

def hill\_decrypt(ciphertext, key\_matrix):

"""Decrypt ciphertext using Hill cipher and key matrix."""

nums = text\_to\_numbers(ciphertext)

inverse\_key = matrix\_mod\_inverse(key\_matrix, 26)

decrypted = []

for i in range(0, len(nums), 2):

pair = np.array([[nums[i]], [nums[i + 1]]])

result = np.dot(inverse\_key, pair) % 26

decrypted.extend(result.flatten())

return numbers\_to\_text(decrypted)

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

plaintext = "meet me at the usual place at ten rather than eight oclock"

key\_matrix = np.array([[9, 4], [5, 7]])

encrypted = hill\_encrypt(plaintext, key\_matrix)

print("Encrypted text:", encrypted)

decrypted = hill\_decrypt(encrypted, key\_matrix)

print("Decrypted text:", decrypted)

**Sample input :-**

plaintext = "MEET ME AT THE USUAL PLACE AT TEN RATHER THAN EIGHT OCLOCK" key\_matrix = np.array([[9, 4], [5, 7]])

**Sample output :-**

Encryptedtext: HZICPWYGMRXIBKFXAUNWGRXBOBEWUXYQHPHEULRLHENMEWMFDI

Decrypted text: MEETMEATTHEUSUALPLACEATTENRATHERTHANEIGHTOCLOCKX

1. **Aim :-** To implement encryption using the Cipher Block Chaining (CBC) mode with the 3DES (Triple DES) algorithm in Python, and compare it with DES for both security and performance considerations.

**Program :**

from Crypto.Cipher import DES3

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

def encrypt\_3des\_cbc(plaintext, key, iv):

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

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

ciphertext = cipher.encrypt(padded\_text)

return ciphertext

def decrypt\_3des\_cbc(ciphertext, key, iv):

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

decrypted\_padded = cipher.decrypt(ciphertext)

decrypted = unpad(decrypted\_padded, DES3.block\_size)

return decrypted.decode()

# Main

if \_\_name\_\_ == "\_\_main\_\_":

# Sample input

plaintext = "Meet me at 10 PM behind the station."

# Generate a secure 24-byte key (3DES requires 16 or 24 bytes)

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

# Generate a random 8-byte IV (3DES block size is 8 bytes)

iv = get\_random\_bytes(8)

print("Plaintext:", plaintext)

print("Key:", key.hex())

print("IV:", iv.hex())

# Encrypt

ciphertext = encrypt\_3des\_cbc(plaintext, key, iv)

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

# Decrypt

decrypted\_text = decrypt\_3des\_cbc(ciphertext, key, iv)

print("Decrypted:", decrypted\_text)

**Sample input :-**

plaintext = "Meet me at 10 PM behind the station."

**Sample output :-**

Plaintext: Meet me at 10 PM behind the station.

Key: a7ef1027e9aa983c7f5c7129dd13fdc5a72e4c88cf5e6a1a

IV: 9d1a3e5caae5bb84

Encrypted (hex): d4a1c6fe27d4294ff23e16a8dbb5d8eec22a5dbec9b3834f6c2e6b4c835bf618

Decrypted: Meet me at 10 PM behind the station.

1. **Aim :-**

To implement encryption using ECB, CBC, and CFB block cipher modes. The plaintext must be a multiple of the block or segment size, and should be padded even if the final block is complete.

**Program :-**

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

BLOCK\_SIZE = 8 # DES block size in bytes

def pad\_message(msg):

""" Pad with 0x80 followed by 0x00 to next block size """

pad\_len = BLOCK\_SIZE - (len(msg) % BLOCK\_SIZE)

return msg + b'\x80' + b'\x00' \* (pad\_len - 1)

def unpad\_message(padded):

""" Remove padding: find 0x80 from end """

if b'\x80' not in padded:

return padded

return padded[:padded.rfind(b'\x80')]

def encrypt\_ecb(plaintext, key):

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

return cipher.encrypt(pad\_message(plaintext))

def decrypt\_ecb(ciphertext, key):

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

return unpad\_message(cipher.decrypt(ciphertext))

def encrypt\_cbc(plaintext, key, iv):

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

return cipher.encrypt(pad\_message(plaintext))

def decrypt\_cbc(ciphertext, key, iv):

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

return unpad\_message(cipher.decrypt(ciphertext))

def encrypt\_cfb(plaintext, key, iv):

cipher = DES.new(key, DES.MODE\_CFB, iv, segment\_size=8\*BLOCK\_SIZE)

return cipher.encrypt(pad\_message(plaintext))

def decrypt\_cfb(ciphertext, key, iv):

cipher = DES.new(key, DES.MODE\_CFB, iv, segment\_size=8\*BLOCK\_SIZE)

return unpad\_message(cipher.decrypt(ciphertext))

if \_\_name\_\_ == "\_\_main\_\_":

plaintext = b"HELLO123" # 8 bytes (already a full block, but still padded)

key = get\_random\_bytes(8) # DES key = 8 bytes

iv = get\_random\_bytes(8) # IV also 8 bytes

print("Original:", plaintext)

# ECB

ecb\_ct = encrypt\_ecb(plaintext, key)

print("ECB Encrypted:", ecb\_ct.hex())

print("ECB Decrypted:", decrypt\_ecb(ecb\_ct, key))

# CBC

cbc\_ct = encrypt\_cbc(plaintext, key, iv)

print("CBC Encrypted:", cbc\_ct.hex())

print("CBC Decrypted:", decrypt\_cbc(cbc\_ct, key, iv))

# CFB

cfb\_ct = encrypt\_cfb(plaintext, key, iv)

print("CFB Encrypted:", cfb\_ct.hex())

print("CFB Decrypted:", decrypt\_cfb(cfb\_ct, key, iv))

**Sample input :-**

plaintext = b"HELLO123"

user\_input = input("Enter the plaintext (minimum 1 character): ")

plaintext = user\_input.encode('utf-8')

**Sample output :-**

Original: b'HELLO123'

ECB Encrypted: 5b6f54b45b8e92f1b89e912cd372d469

ECB Decrypted: b'HELLO123'

CBC Encrypted: cdd2c40f8f29647cb4b5298b84597818

CBC Decrypted: b'HELLO123'

CFB Encrypted: 9bd7d86bfb77cfaa83bbef2b54618b48

CFB Decrypted: b'HELLO123'

1. **Aim :-**

To implement Cipher Block Chaining (CBC) mode using a block cipher such as Simplified DES (S-DES) for both encryption and decryption.

**Program :-**

# S-DES implementation with CBC mode

P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]

P8 = [6, 3, 7, 4, 8, 5, 10, 9]

P4 = [2, 4, 3, 1]

IP = [2, 6, 3, 1, 4, 8, 5, 7]

IP\_INV = [4, 1, 3, 5, 7, 2, 8, 6]

EP = [4, 1, 2, 3, 2, 3, 4, 1]

S0 = [[1, 0, 3, 2],

[3, 2, 1, 0],

[0, 2, 1, 3],

[3, 1, 3, 2]]

S1 = [[0, 1, 2, 3],

[2, 0, 1, 3],

[3, 0, 1, 0],

[2, 1, 0, 3]]

def permute(bits, table):

return [bits[i - 1] for i in table]

def left\_shift(bits, n):

return bits[n:] + bits[:n]

def xor(bits1, bits2):

return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]

def sbox(input\_bits, sbox):

row = (input\_bits[0] << 1) + input\_bits[3]

col = (input\_bits[1] << 1) + input\_bits[2]

value = sbox[row][col]

return [int(x) for x in format(value, '02b')]

def fk(bits, key):

left = bits[:4]

right = bits[4:]

ep = permute(right, EP)

temp = xor(ep, key)

s0\_out = sbox(temp[:4], S0)

s1\_out = sbox(temp[4:], S1)

combined = permute(s0\_out + s1\_out, P4)

return xor(left, combined) + right

def switch(bits):

return bits[4:] + bits[:4]

def sdes\_encrypt(plaintext\_bits, key):

key = [int(k) for k in key]

k1, k2 = generate\_keys(key)

bits = permute(plaintext\_bits, IP)

bits = fk(bits, k1)

bits = switch(bits)

bits = fk(bits, k2)

return permute(bits, IP\_INV)

def sdes\_decrypt(ciphertext\_bits, key):

key = [int(k) for k in key]

k1, k2 = generate\_keys(key)

bits = permute(ciphertext\_bits, IP)

bits = fk(bits, k2)

bits = switch(bits)

bits = fk(bits, k1)

return permute(bits, IP\_INV)

def generate\_keys(key\_bits):

p10 = permute(key\_bits, P10)

left = p10[:5]

right = p10[5:]

left1 = left\_shift(left, 1)

right1 = left\_shift(right, 1)

k1 = permute(left1 + right1, P8)

left2 = left\_shift(left1, 2)

right2 = left\_shift(right1, 2)

k2 = permute(left2 + right2, P8)

return k1, k2

def binary\_str\_to\_list(s):

return [int(b) for b in s]

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

ciphertext\_blocks = []

prev = iv

for block in plaintext\_blocks:

block = xor(block, prev)

encrypted = sdes\_encrypt(block, key)

ciphertext\_blocks.append(encrypted)

prev = encrypted

return ciphertext\_blocks

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

plaintext\_blocks = []

prev = iv

for block in ciphertext\_blocks:

decrypted = sdes\_decrypt(block, key)

plaintext\_blocks.append(xor(decrypted, prev))

prev = block

return plaintext\_blocks

# Helper: print binary lists as strings

def blocks\_to\_str(blocks):

return ' '.join(''.join(str(b) for b in blk) for blk in blocks)

# ---- TEST DATA ----

if \_\_name\_\_ == "\_\_main\_\_":

key = "0111111101"

iv = binary\_str\_to\_list("10101010")

plaintext\_binary = "00000001 00100011"

plaintext\_blocks = [binary\_str\_to\_list(b) for b in plaintext\_binary.split()]

print("Plaintext:", blocks\_to\_str(plaintext\_blocks))

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

print("Ciphertext:", blocks\_to\_str(ciphertext\_blocks))

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

print("Decrypted:", blocks\_to\_str(decrypted\_blocks))

**Sample input :-** Plaintext: 00000001 00100011

Key: 0111111101

IV: 10101010

**Sample output** :- Plaintext: 00000001 00100011

Ciphertext: 11110100 00001011

Decrypted: 00000001 00100011

1. **Aim :-** To find the private key for a given public key in the RSA encryption system. Given the public key �=31*e*=31 and �=3599*n*=3599**.**

**Program :-**

# Function to compute gcd using Euclidean algorithm

def gcd(a, b):

while b:

a, b = b, a % b

return a

# Function to find the modular inverse of a number using Extended Euclidean Algorithm

def mod\_inverse(a, m):

m0, x0, x1 = m, 0, 1

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

if x1 < 0:

x1 += m0

return x1

# Function to find prime factors of n using trial and error

def find\_prime\_factors(n):

for i in range(2, int(n \*\* 0.5) + 1):

if n % i == 0:

return i, n // i

return None, None

# Function to compute RSA private key d

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

# Step 1: Find p and q

p, q = find\_prime\_factors(n)

if p is None or q is None:

return "Cannot find prime factors"

# Step 2: Compute Euler's Totient Function f(n)

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

# Step 3: Find the multiplicative inverse of e modulo f(n)

d = mod\_inverse(e, f\_n)

return d, p, q, f\_n

# ---- Test Data ----

if \_\_name\_\_ == "\_\_main\_\_":

# Public key values

e = 31

n = 3599

# Find the private key

private\_key, p, q, f\_n = rsa\_private\_key(e, n)

# Output the private key and factors

if isinstance(private\_key, tuple):

print(f"Private Key (d): {private\_key[0]}")

print(f"Prime Factors (p, q): {p}, {q}")

print(f"Euler's Totient Function (f(n)): {f\_n}")

else:

print(private\_key)

**Sample input :-**

Public Key:

* + *e*=31
  + *n*=3599

**Sample output :-**

Private Key (d): 1791

Prime Factors (p, q): 59, 61

Euler's Totient Function (f(n)): 3480

1. **Aim** :- implementing the Diffie-Hellman protocol, which allows two participants (Alice and Bob) to securely agree on a shared secret key over an insecure channel.

**Program :-**

# Function to compute power mod q (efficient computation of a^b mod q)

def power\_mod(base, exp, mod):

result = 1

while exp > 0:

if exp % 2 == 1:

result = (result \* base) % mod

base = (base \* base) % mod

exp //= 2

return result

# Diffie-Hellman Key Exchange

def diffie\_hellman\_protocol(a, q, x\_alice, x\_bob):

# Alice computes A = a^x\_Alice % q

A = power\_mod(a, x\_alice, q)

# Bob computes B = a^x\_Bob % q

B = power\_mod(a, x\_bob, q)

# Alice computes the shared key using B

shared\_key\_alice = power\_mod(B, x\_alice, q)

# Bob computes the shared key using A

shared\_key\_bob = power\_mod(A, x\_bob, q)

return A, B, shared\_key\_alice, shared\_key\_bob

# ---- Test Data ----

if \_\_name\_\_ == "\_\_main\_\_":

# Public values

a = 2 # Base, public number

q = 23 # Prime modulus

# Secret values chosen by Alice and Bob

x\_alice = 6 # Alice's private key

x\_bob = 15 # Bob's private key

# Run Diffie-Hellman Protocol

A, B, shared\_key\_alice, shared\_key\_bob = diffie\_hellman\_protocol(a, q, x\_alice, x\_bob)

# Output results

print(f"Alice sends A = {A}")

print(f"Bob sends B = {B}")

print(f"Alice computes the shared key: {shared\_key\_alice}")

print(f"Bob computes the shared key: {shared\_key\_bob}")

# Verify both keys are the same

if shared\_key\_alice == shared\_key\_bob:

print("Key exchange successful, shared key:", shared\_key\_alice)

else:

print("Key exchange failed")

**Sample output** :- Public values:

* + *a*=2 (Base)
  + *q*=23 (Prime modulus)
* Alice's private key: *xAlice*​=6
* Bob's private key: *xBob*​=15

**Sample output** :- Alice sends A = 13

Bob sends B = 6

Alice computes the shared key: 18

Bob computes the shared key: 18

Key exchange successful, shared key: 18