1. **Aim :-** To implement the Caesar Cipher encryption technique in Python, which involves shifting each letter in a message by a fixed number of positions (k) down the alphabet (where 1 ≤ k ≤ 25).

**Program :**

def caesar\_cipher\_encrypt(text, key):

result = ""

for char in text:

if char.isalpha():

# Shift uppercase characters

if char.isupper():

shifted = (ord(char) - ord('A') + key) % 26 + ord('A')

# Shift lowercase characters

else:

shifted = (ord(char) - ord('a') + key) % 26 + ord('a')

result += chr(shifted)

else:

# Non-alphabetic characters remain unchanged

result += char

return result

# Main program

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

plaintext = input("Enter the plaintext: ")

key = int(input("Enter the key (1-25): "))

if 1 <= key <= 25:

encrypted = caesar\_cipher\_encrypt(plaintext, key)

print("Encrypted text:", encrypted)

else:

print("Invalid key. Please enter a number between 1 and 25.")

**Sample input :-**

Enter the plaintext: Hello World!

Enter the key (1-25): 3

**Sample output :-**

Encrypted text: Khoor Zruog!

1. **Aim :-** To write a C program for a monoalphabetic substitution cipher, where each letter in the plaintext is replaced with a unique corresponding letter from a substitution key (ciphertext alphabet).

**Program :**

def monoalphabetic\_encrypt(plaintext, key\_map):

ciphertext = ""

for char in plaintext:

if char.isalpha():

if char.isupper():

ciphertext += key\_map[char]

else:

ciphertext += key\_map[char.upper()].lower()

else:

ciphertext += char # Keep non-alphabetic characters unchanged

return ciphertext

# Main program

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

# Example substitution key: must be a permutation of all 26 unique uppercase letters

key = "QWERTYUIOPASDFGHJKLZXCVBNM"

if len(set(key)) != 26 or not key.isalpha() or not key.isupper():

print("Invalid key. Please use a permutation of all 26 uppercase letters.")

else:

# Create key mapping from A-Z to substitution key

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

key\_map = dict(zip(alphabet, key))

plaintext = input("Enter the plaintext: ")

encrypted = monoalphabetic\_encrypt(plaintext, key\_map)

print("Encrypted text:", encrypted)

**Sample input :-**

Enter the plaintext: Hello World!

**Sample output :-**

Encrypted text: Itssg Vgksr!

1. **Aim :-** To write a program for Monoalphabetic Substitution Cipher that encrypts a message by replacing each letter in the plaintext with a corresponding unique letter from a substitution alphabet (key).

**Program :**

def monoalphabetic\_encrypt(plaintext, key):

"""

Encrypts the plaintext using monoalphabetic substitution cipher.

"""

ciphertext = ""

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

key\_map = dict(zip(alphabet, key)) # Map A-Z to key letters

for char in plaintext:

if char.isalpha():

if char.isupper():

ciphertext += key\_map[char]

else:

ciphertext += key\_map[char.upper()].lower()

else:

ciphertext += char # Keep punctuation, digits, spaces unchanged

return ciphertext

# Main function

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

# 26 unique uppercase letters (cipher alphabet)

key = "QWERTYUIOPASDFGHJKLZXCVBNM"

# Validate key

if len(key) != 26 or not key.isalpha() or len(set(key)) != 26:

print("Invalid key. Must be 26 unique alphabetic characters.")

else:

plaintext = input("Enter the plaintext: ")

encrypted\_text = monoalphabetic\_encrypt(plaintext, key)

print("Encrypted text:", encrypted\_text)

**Sample input :-**

Enter the plaintext: Hello World!

**Sample output :-**

Encrypted text: Itssg Vgksf!

1. **Aim** :- To implement the Playfair Cipher encryption technique using a 5x5 matrix generated from a given keyword, where the plaintext is encrypted in pairs of letters (digraphs**).**

**Program :**

def generate\_key\_matrix(keyword):

keyword = keyword.upper().replace('J', 'I')

seen = set()

matrix = []

for char in keyword:

if char not in seen and char.isalpha():

seen.add(char)

matrix.append(char)

for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':

if char not in seen:

matrix.append(char)

return [matrix[i\*5:(i+1)\*5] for i in range(5)]

def preprocess\_text(text):

text = text.upper().replace('J', 'I').replace(" ", "")

processed = ""

i = 0

while i < len(text):

a = text[i]

b = text[i+1] if i+1 < len(text) else 'X'

if a == b:

processed += a + 'X'

i += 1

else:

processed += a + b

i += 2

if len(processed) % 2 != 0:

processed += 'X'

return [processed[i:i+2] for i in range(0, len(processed), 2)]

def find\_position(matrix, letter):

for row in range(5):

for col in range(5):

if matrix[row][col] == letter:

return row, col

return None

def encrypt\_pair(pair, matrix):

r1, c1 = find\_position(matrix, pair[0])

r2, c2 = find\_position(matrix, pair[1])

if r1 == r2:

return matrix[r1][(c1+1)%5] + matrix[r2][(c2+1)%5]

elif c1 == c2:

return matrix[(r1+1)%5][c1] + matrix[(r2+1)%5][c2]

else:

return matrix[r1][c2] + matrix[r2][c1]

def playfair\_encrypt(plaintext, keyword):

matrix = generate\_key\_matrix(keyword)

digraphs = preprocess\_text(plaintext)

ciphertext = ""

for pair in digraphs:

ciphertext += encrypt\_pair(pair, matrix)

return ciphertext

# Main

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

keyword = input("Enter the keyword: ")

plaintext = input("Enter the plaintext: ")

ciphertext = playfair\_encrypt(plaintext, keyword)

print("Encrypted text:", ciphertext)

**Sample input :-**

Enter the keyword: MONARCHY

Enter the plaintext: HELLO WORLD

**Sample output :-**

Encrypted text: KCTUPMVOMTUZ

1. **Aim** :- To implement the Hill Cipher for encryption and decryption of a message using a 2×2 matrix key. The Hill cipher uses linear algebra (matrix multiplication) to transform plaintext into ciphertext and vice versa.

**Program :**

import numpy as np

def mod\_inverse(a, m):

for i in range(1, m):

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

return i

return None

def matrix\_mod\_inverse(matrix, 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.")

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):

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

def numbers\_to\_text(numbers):

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

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

nums = text\_to\_numbers(plaintext)

if len(nums) % 2 != 0:

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

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):

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

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 :-**

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

**Sample output :-**

Encryptedtext: HZICPWYGMRXIBKFXAUNWGRXBOBEWUXYQHPHEULRLHENMEWMFDI

Decrypted text: MEETMEATTHEUSUALPLACEATTENRATHERTHANEIGHTOCLOCKX