Practical 1

AIM: Implement Caesar cipher encryption-decryption.

Caesar cipher is one of the simplest and oldest method of encrypting message.

It was developed by Julius Caesar to protect military communication.

This technique involves shifting the letter of alphabet by fix number. which is known as “Shift/Key”.

It’s simplest type of substitution Cipher. In which each letter of given text is replaced by a shift or key position alphabet.

CODE:

```
def caesar_cipher_encrypt(msg, shift):
    ciphertext = ""
    for char in msg:
        if char.isalpha():
            if char.isupper(): #Checks if the character is alphabetic
                shifted_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))
            else:
                shifted_char = chr((ord(char) - ord('a') + shift) % 26 + ord('a'))
            ciphertext += shifted_char
        else:
            ciphertext += char
    return ciphertext


def caesar_cipher_decrypt(ciphertext, shift):
    msg = ""
    for char in ciphertext:
```

```
        if char.isalpha():
            if char.isupper():
                shifted_char = chr((ord(char) - ord('A') - shift) % 26 + ord('A'))
            else:
                shifted_char = chr((ord(char) - ord('a') - shift) % 26 + ord('a'))
            msg += shifted_char
        else:
            msg += char
    return msg

msg = input("Enter the message: ")
shift = int(input("Enter the shift value: "))

encrypted_text = caesar_cipher_encrypt(msg, shift)
print("Encrypted:", encrypted_text)

decrypted_text = caesar_cipher_decrypt(encrypted_text, shift)
print("Decrypted:", decrypted_text)
```

OUTPUT:

```
Enter the message: Karan
Enter the shift value: 6
Encrypted: Qgxgt
Decrypted: Karan
```

Practical 2

AIM: Implement Monoalphabetic cipher encryption-decryption.

Monoalphabetic cipher is substitution technique in which a single alphabet is used for message.

It provides protection from brute force attack.

In Monoalphabetic cipher the mapping is done randomly not in uniform format.

CODE:

```
import numpy as np

import random

import string


def generate_monoalphabetic_key():

    """Generate a random Monoalphabetic cipher key."""

    letters = list(string.ascii_uppercase)

    key = {}

    for char in string.ascii_uppercase:

        random_char = random.choice(letters)

        key[char] = random_char

        letters.remove(random_char)  # Remove selected character to ensure unique
mapping

    return key


def encrypt_monoalphabetic(message, key):

    """Encrypt a message using a Monoalphabetic cipher."""

    encrypted_message = []
```

```
    capitalization_info = []

    for char in message:
        if char.upper() in key:
            encrypted_char = key[char.upper()]
            encrypted_message.append(encrypted_char)
            capitalization_info.append(char.isupper())
        else:
            encrypted_message.append(char)  # if character is not in the key, add it as-
is
            capitalization_info.append(False)  # mark as non-alphabetic or lowercase

    return ''.join(encrypted_message), capitalization_info

def decrypt_monoalphabetic(encrypted_message, capitalization_info, key):
    """Decrypt a message encrypted with a Monoalphabetic cipher."""
    decrypted_message = []
    reverse_key = {v: k for k, v in key.items()}  # create reverse key for decryption

    for i, char in enumerate(encrypted_message):
        if char.upper() in reverse_key:
            decrypted_char = reverse_key[char.upper()]
            if capitalization_info[i]:
                decrypted_char = decrypted_char.upper()
            else:
```

```
            decrypted_char = decrypted_char.lower()
        decrypted_message.append(decrypted_char)
    else:
        decrypted_message.append(char)  # if character is not in the key, add it as-
is

    return ''.join(decrypted_message)

def analyze_frequency(message):
    """Analyze the frequency of characters in a message."""
    frequency = np.zeros((26,), dtype=int)
    for char in message.upper():
        if char.isalpha():
            frequency[ord(char) - ord('A')] += 1
    return frequency

# Generate a random Monoalphabetic key
key = generate_monoalphabetic_key()
print("Generated Monoalphabetic Key:")
print(key)

# Encrypt a message
message = input("Enter the message: ")
encrypted_message, capitalization_info = encrypt_monoalphabetic(message, key)
print("Original Message:", message)
```

```
print("Encrypted Message:", encrypted_message)


# Decrypt the message
decrypted_message = decrypt_monoalphabetic(encrypted_message, capitalization_info, key)
print("Decrypted Message:", decrypted_message)


# Analyze the frequency of characters in the original and encrypted messages
original_frequency = analyze_frequency(message)
encrypted_frequency = analyze_frequency(encrypted_message)
```

OUTPUT:

```
Generated Monoalphabetic Key:
{'A': 'Z', 'B': 'F', 'C': 'Q', 'D': 'A', 'E': 'K', 'F': 'V', 'G': 'W', 'H': 'L', 'I': 'I', 'J': 'S', 'K': 'X', 'L': 'P', 'M': 'M', 'N': 'U', 'O': 'E', 'P': 'R', 'Q': 'C', 'R': 'O', 'S': 'J', 'T': 'Y', 'U': 'D', 'V': 'N', 'W': 'B', 'X': 'H', 'Y': 'G', 'Z': 'T'}
Enter the message: Karan
Original Message: Karan
Encrypted Message: XZOZU
Decrypted Message: KARAN
```

Practical 3

AIM: Implement Playfair cipher encryption-decryption.

Playfair cipher was invented by Charles Wheatstone. But later in 90’s lord playfair make it more useful and popular, so the name “Playfair cipher”.

It’s also substitution technique. Unlike a single alphabet substitution in encryption it replaces pair of alphabet.

Encryption:

Steps:

1) Generate a key square of 5x5 for encryption the plain text. In this table we have to omit any single character and consider as “J”.

2) Keep the alphabet in the key square unit. That no alphabet should be repeated. Place the key first in the key square and then remaining alphabet in order.

3) Encrypt the plain text. The plain text is split into pair of two letter called “Diagraph”.

- No alphabet remain single. It makes the plain text of even. Suppose any plain text has odd number then add any dummy letter.
- If any letter appears more than one time, then side by side then place any dummy letter to make it unique.
- If both the letter are in the same column take the letter below each one. If it’s bottom, then take it to top.
- If both the letter are in the same row take the letter to the immediate right of each one. If it’s at last position, then take it back to the first.
- If neither of the above rule is true form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

Decryption:

It is same as encryption but the steps are applied in reverse order.

Steps:

1) Split the plain text into diagraph.

2) Construct the 5x5 key matrix.

3) It will traverse the key matrix step by step and find the corresponding encipher for the pair.

CODE:

```
def generate_key_matrix(key):
    key = key.upper().replace('J', 'I')
    matrix = []
    used = set()

    # Add unique letters from the key
    for char in key:
        if char not in used and char.isalpha():
            used.add(char)
            matrix.append(char)

    # Add remaining letters of the alphabet
    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':
        if char not in used:
            used.add(char)
            matrix.append(char)

    # Convert to 5x5 matrix
    return [matrix[i:i+5] for i in range(0, 25, 5)]
```

```
def print_matrix(matrix):
    for row in matrix:
        print(' '.join(row))
    print()  # For better readability


def preprocess_text(text):
    text = text.upper().replace('J', 'I')
    text = ''.join(filter(str.isalpha, text))
    digraphs = []

    i = 0
    while i < len(text):
        if i + 1 < len(text):
            if text[i] == text[i + 1]:
                digraphs.append(text[i] + 'X')
                i += 1
            else:
                digraphs.append(text[i] + text[i + 1])
                i += 2
        else:
            digraphs.append(text[i] + 'X')
            i += 1

    return digraphs
```

```
def find_position(matrix, char):
    for r, row in enumerate(matrix):
        if char in row:
            return (r, row.index(char))
    return None

def encrypt_digraph(matrix, digraph):
    pos1 = find_position(matrix, digraph[0])
    pos2 = find_position(matrix, digraph[1])

    if pos1[0] == pos2[0]:
        return matrix[pos1[0]][(pos1[1] + 1) % 5] + matrix[pos2[0]][(pos2[1] + 1) % 5]
    elif pos1[1] == pos2[1]:
        return matrix[(pos1[0] + 1) % 5][pos1[1]] + matrix[(pos2[0] + 1) % 5][pos2[1]]
    else:
        return matrix[pos1[0]][pos2[1]] + matrix[pos2[0]][pos1[1]]

def decrypt_digraph(matrix, digraph):
    pos1 = find_position(matrix, digraph[0])
    pos2 = find_position(matrix, digraph[1])

    if pos1[0] == pos2[0]:
        return matrix[pos1[0]][(pos1[1] - 1) % 5] + matrix[pos2[0]][(pos2[1] - 1) % 5]
```

```
    elif pos1[1] == pos2[1]:
        return matrix[(pos1[0] - 1) % 5][pos1[1]] + matrix[(pos2[0] - 1) % 5][pos2[1]]
    else:
        return matrix[pos1[0]][pos2[1]] + matrix[pos2[0]][pos1[1]]

def playfair_cipher(key, text, mode='encrypt'):
    matrix = generate_key_matrix(key)

    digraphs = preprocess_text(text)

    if mode == 'encrypt':
        print("Key Matrix:")
        print_matrix(matrix)
        process_digraph = encrypt_digraph
    elif mode == 'decrypt':
        process_digraph = decrypt_digraph
    else:
        raise ValueError("Mode must be 'encrypt' or 'decrypt'")

    processed_text = ''.join(process_digraph(matrix, digraph) for digraph in
digraphs)
    return processed_text

key = input("Enter the key: ")
plaintext = input("Enter the message: ")
```

```
ciphertext = playfair_cipher(key, plaintext, mode='encrypt')
print("Encrypted:", ciphertext)


decrypted_text = playfair_cipher(key, ciphertext, mode='decrypt')
print("Decrypted:", decrypted_text)
```

OUTPUT:

```
Enter the key: hello
Enter the message: cyan
Key Matrix:
H E L O A
B C D F G
I K M N P
Q R S T U
V W X Y Z

Encrypted: FWOP
Decrypted: CYAN
```

Practical 4

AIM: Implement hill cipher encryption-decryption.

Hill cipher is polygraphic substitution cipher, based on linear algebra. This algorithm use matrix multiplication, and factorization.

Encryption:

Step:

1) Create the matrix of a key and convert that, into a numerical value.

2) Convert plain text into vector form and do the matrix multiplication.

3) Multiply the key matrix with each plain text vector and take the modulo of result, then concate the result to get the cipher text.

CODE:

```
import numpy as np


# Helper function to convert text to numbers and vice versa
def text_to_numbers(text):
    return [ord(char) - ord('A') for char in text.upper()]


def numbers_to_text(numbers):
    return ''.join(chr(num + ord('A')) for num in numbers)


# Encrypt function
def hill_encrypt(plaintext, key_matrix):
    plaintext_numbers = text_to_numbers(plaintext)
    plaintext_vector = np.array(plaintext_numbers).reshape(-1, 5)
    ciphertext_vector = np.dot(plaintext_vector, key_matrix) % 26
```

```
    ciphertext_numbers = ciphertext_vector.flatten()
    return numbers_to_text(ciphertext_numbers)

# Decrypt function
def hill_decrypt(ciphertext, key_matrix):
    ciphertext_numbers = text_to_numbers(ciphertext)
    ciphertext_vector = np.array(ciphertext_numbers).reshape(-1, 5)

    # Calculate inverse of the key matrix modulo 26
    determinant = int(round(np.linalg.det(key_matrix)))
    determinant_inv = pow(determinant, -1, 26)
    key_matrix_inv = determinant_inv * np.round(determinant *
np.linalg.inv(key_matrix)).astype(int) % 26

    plaintext_vector = np.dot(ciphertext_vector, key_matrix_inv) % 26
    plaintext_numbers = plaintext_vector.flatten()
    return numbers_to_text(plaintext_numbers)

# Function to input the 5x5 key matrix from user
def input_key_matrix():
    print("Enter the 5x5 key matrix (each row separated by a space):")
    matrix = []
    for i in range(5):
        row = list(map(int, input(f"Row {i+1}: ").strip().split()))
        if len(row) != 5:
```

```
        raise ValueError("Each row must have exactly 5 integers.")
      matrix.append(row)
    return np.array(matrix)

# Function to input the plaintext from user
def input_plaintext():
    plaintext = input("Enter the plaintext: ").upper().replace(" ", "")
    if len(plaintext) % 5 != 0:
        padding_length = 5 - (len(plaintext) % 5)
        plaintext += 'X' * padding_length
    return plaintext

# Main function to execute the encryption and decryption
def main():
    key_matrix = input_key_matrix()
    plaintext = input_plaintext()
    ciphertext = hill_encrypt(plaintext, key_matrix)
    decrypted_text = hill_decrypt(ciphertext, key_matrix)
    print(f"\nPlaintext: {plaintext}")
    print(f"Ciphertext: {ciphertext}")
    print(f"Decrypted text: {decrypted_text}")

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

OUTPUT:

```
Enter the 5x5 key matrix (each row separated by a space):
Row 1: 3 3 2 6 2
Row 2: 4 2 4 1 7
Row 3: 2 1 2 5 8
Row 4: 9 3 1 1 3
Row 5: 7 5 6 4 1
Enter the plaintext: hello world

Plaintext: HELLOWORLD
Ciphertext: WNRMVQDHUC
Decrypted text: HELLOWORLD
```