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.

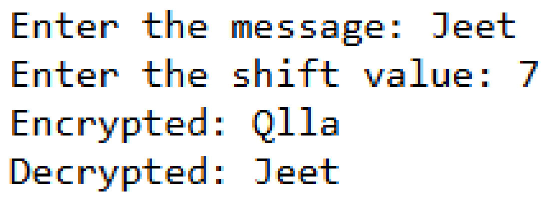
This technique involves shifting the letter of alphabet by fix number. which is known as “Shift/Key”.

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: Jeet
Enter the shift value: 7
Encrypted: Qlla
Decrypted: Jeet
```

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.

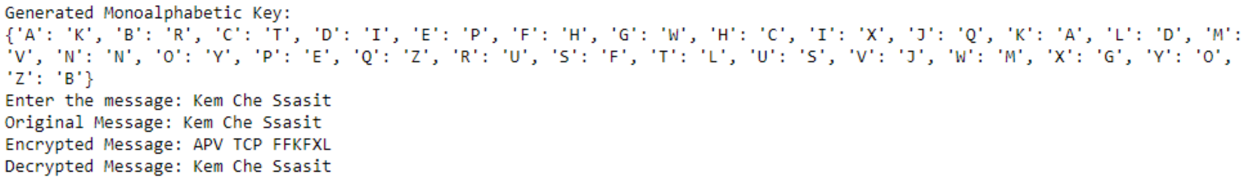
In Monoalphabetic cipher the mapping is done randomly not in uniform format.

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': 'K', 'B': 'R', 'C': 'T', 'D': 'I', 'E': 'P', 'F': 'H', 'G': 'W', 'H': 'C', 'I': 'X', 'J': 'Q', 'K': 'A', 'L': 'D', 'M':
'V', 'N': 'N', 'O': 'Y', 'P': 'E', 'Q': 'Z', 'R': 'U', 'S': 'F', 'T': 'L', 'U': 'S', 'V': 'J', 'W': 'M', 'X': 'G', 'Y': 'O',
'Z': 'B'}
Enter the message: Kem Che Ssasit
Original Message: Kem Che Ssasit
Encrypted Message: APV TCP FFKFXL
Decrypted Message: Kem Che Ssasit
```

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.

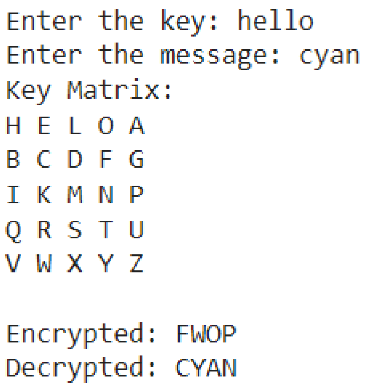
CODE:

```
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.

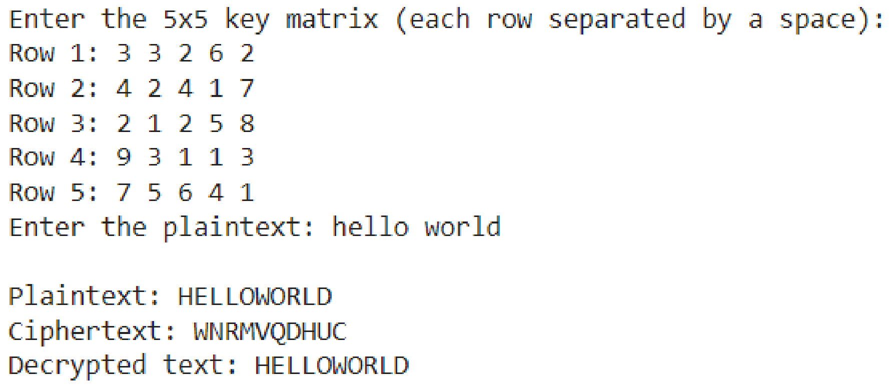
2) Convert plain text into vector form and do the matrix multiplication.

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
```

Practical 5

AIM: Implement polyalphabetic Cipher.

Polyalphabetic cipher is a substitution alphabetic technique using multiple substitution alphabet. We can use more than one substitution for the same alphabet.

Encryption= Ei =(Pi+ki) mod 26

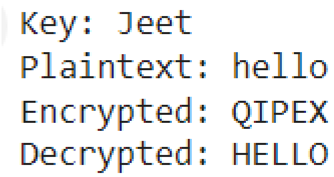
Decryption= Di = Ei-Ki

There are various technique to implement poly alphabetic. One of them is Vigenere . Which is Simplest and most popular method. Whenever a vigenere table is not given then it will be performed by formula given.

CODE:

```
def generate_vigenere_table():

    """Generate a Vigenère cipher table."""

    table = []

    for i in range(26):

        row = [(chr((j + i) % 26 + ord('A'))) for j in range(26)]

        table.append(row)

    return table


def vigenere_encrypt(plaintext, keyword):

    """Encrypt plaintext using the Vigenère cipher with the given keyword."""

    table = generate_vigenere_table()

    plaintext = plaintext.upper()

    keyword = keyword.upper()

    encrypted_text = []
    keyword_length = len(keyword)

    keyword_index = 0
    for char in plaintext:
        if char.isalpha():
            row = ord(keyword[keyword_index]) - ord('A')
            col = ord(char) - ord('A')
            encrypted_char = table[row][col]
            encrypted_text.append(encrypted_char)
            keyword_index = (keyword_index + 1) % keyword_length
        else:
            encrypted_text.append(char)

    return ''.join(encrypted_text)

def vigenere_decrypt(ciphertext, keyword):
    """Decrypt ciphertext using the Vigenère cipher with the given keyword."""
    table = generate_vigenere_table()
    ciphertext = ciphertext.upper()
    keyword = keyword.upper()
    decrypted_text = []
    keyword_length = len(keyword)

    keyword_index = 0
    for char in ciphertext:
        if char.isalpha():
            row = ord(keyword[keyword_index]) - ord('A')
            col = table[row].index(char)
            decrypted_char = chr(col + ord('A'))
            decrypted_text.append(decrypted_char)
            keyword_index = (keyword_index + 1) % keyword_length
        else:
            decrypted_text.append(char)

    return ''.join(decrypted_text)

# Example usage
key = input("Key: ")
plaintext = input("Plaintext: ")
encrypted = vigenere_encrypt(plaintext, key)
print("Encrypted:", encrypted)
decrypted = vigenere_decrypt(encrypted, key)
print("Decrypted:", decrypted)
```

OUTPUT:



```
Key: Jeet
Plaintext: hello
Encrypted: QIPEX
Decrypted: HELLO
```

Practical 6

AIM: Implement rail fence cipher encryption decryption.

It is also known as zigzag cipher. It is form of substitution cipher.

Encryption:

steps:

1) In rail fence cipher plaintext is written in downwards and diagonally on succesive raise of imaginary fence.

2) After reaching at bottom rail move upward diagonally. After reaching top move towards bottom diagonally. That create the alphabetical zigzag.

3) After each alphabet has been written the individual raws are combine to obtain the cipher text.

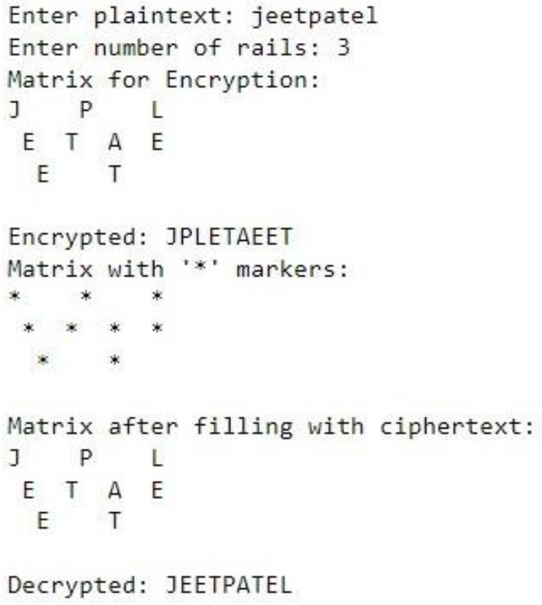
CODE:

```
def print_matrix(matrix):
    for row in matrix:
        print(' '.join(char if char != '' else ' ' for char in row))
    print()
def rail_fence_encrypt(message, num_rails):
    if num_rails == 1:
        return message
    rails = [['' for _ in range(len(message))] for _ in range(num_rails)]
    direction = 1
    rail = 0
    for i in range(len(message)):
        rails[rail][i] = '*'
        rail += direction
        if rail == 0 or rail == num_rails - 1:
            direction *= -1
    print_matrix(rails)
    rails = [['' for _ in range(len(message))] for _ in range(num_rails)]
    direction = 1
    rail = 0
    index = 0
    for i in range(len(message)):
        rails[rail][i] = message[index]
        index += 1
        rail += direction
        if rail == 0 or rail == num_rails - 1:
            direction *= -1
    print("Filled Encryption Matrix:")
    print_matrix(rails)
    encrypted_message = ''.join(''.join(row).strip() for row in rails)
    return encrypted_message
def rail_fence_decrypt(encrypted_message, num_rails):
    if num_rails == 1:
        return encrypted_message
    length = len(encrypted_message)
    rails = [['' for _ in range(length)] for _ in range(num_rails)]
    direction = 1
    rail = 0
```

```
for i in range(length):
    rails[rail][i] = '*'
    rail += direction
    if rail == 0 or rail == num_rails - 1:
        direction *= -1
print("Rail Fence Decryption Matrix (marked positions):")
print_matrix(rails)
index = 0
for r in range(num_rails):
    for c in range(length):
        if rails[r][c] == '*':
            rails[r][c] = encrypted_message[index]
            index += 1
print("Filled Decryption Matrix:")
print_matrix(rails)
decrypted_message = []
rail = 0
direction = 1
for i in range(length):
    decrypted_message.append(rails[rail][i])
    rail += direction
    if rail == 0 or rail == num_rails - 1:
        direction *= -1
return ''.join(decrypted_message)
```

```
def main():
    message = input("Enter the message to be encrypted: ").replace(' ', '').upper()
    num_rails = int(input("Enter the number of rails: "))
    if num_rails <= 0:
        print("Error: Number of rails must be greater than 0.")
        return
    encrypted_message = rail_fence_encrypt(message, num_rails)
    print("Encrypted message:", encrypted_message)
    decrypted_message = rail_fence_decrypt(encrypted_message, num_rails)
    print("Decrypted message:", decrypted_message)
if __name__ == "__main__":
    main()
```

OUTPUT:



```
Enter plaintext: jeetpatel
Enter number of rails: 3
Matrix for Encryption:
J     P     L
  E  T  A  E
    E     T

Encrypted: JPLETAEET
Matrix with '*' markers:
*     *     *
  *  *  *  *
    *     *

Matrix after filling with ciphertext:
J     P     L
  E  T  A  E
    E     T

Decrypted: JEETPATEL
```

Practical 7

AIM: Implement columnar transposition cipher encryption decryption.

Columnar transposition is a simple encryption method where plaintext is written into a grid and then read column by column based on a keyword. Here's a brief description:

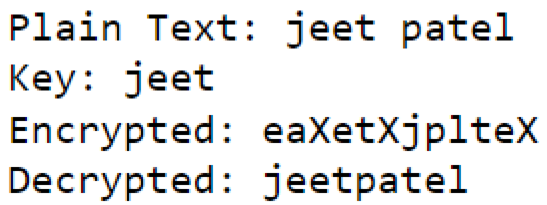
Steps:

1. Write the Plaintext: Arrange the plaintext into a grid, where the number of columns equals the length of a chosen keyword.
2. Label Columns: Assign a number to each column based on the alphabetical order of the letters in the keyword.
3. Read Columns in Order: Read the columns according to their assigned numbers to create the cipher text.

CODE:

```
def encrypt(plaintext, keyword):
    # Remove spaces from the plaintext
    plaintext = plaintext.replace(' ', '')

    # Fill in the grid with the plaintext and padding
    num_cols = len(keyword)
    num_rows = -(-len(plaintext) // num_cols)  # Ceiling division
    padded_plaintext = plaintext.ljust(num_cols * num_rows, 'X')

    # Create a matrix
    matrix = [padded_plaintext[i:i + num_cols] for i in range(0, len(padded_plaintext), num_cols)]
    # Create a dictionary to map column order
    keyword_order = sorted(range(len(keyword)), key=lambda x: keyword[x])
    keyword_order_dict = {char: idx for idx, char in enumerate(sorted(keyword))}

    # Read columns in the order determined by the sorted keyword
    ciphertext = ''
    for col_index in keyword_order:
        for row in matrix:
            ciphertext += row[col_index]

    return ciphertext

def decrypt(ciphertext, keyword):
    # Calculate number of columns and rows
    num_cols = len(keyword)
    num_rows = len(ciphertext) // num_cols

    # Create a matrix with empty values
    matrix = [['' for _ in range(num_cols)] for _ in range(num_rows)]

    # Create a dictionary to map column order
    keyword_order = sorted(range(len(keyword)), key=lambda x: keyword[x])

    # Fill the matrix column by column in the order determined by the keyword
    index = 0
    for col_index in keyword_order:
        for row in range(num_rows):
            matrix[row][col_index] = ciphertext[index]
            index += 1

    # Read rows to get the plaintext
    plaintext = ''.join([''.join(row) for row in matrix])

    # Remove padding characters
    return plaintext.rstrip('X')

# Example usage
plaintext = input("Plain Text: ")
keyword = input("Key: ")

encrypted = encrypt(plaintext, keyword)
print(f"Encrypted: {encrypted}")

decrypted = decrypt(encrypted, keyword)
print(f"Decrypted: {decrypted}")
```

OUTPUT:



```
Plain Text: jeet patel
Key: jeet
Encrypted: eaXetXjplteX
Decrypted: jeetpatel
```

Practical 8

AIM: Implement Diffie-Hellman cipher encryption decryption.

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

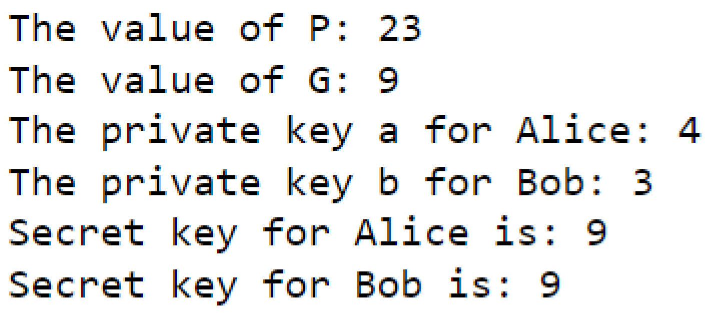
- For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.
- P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

CODE:

```
# Power function to return value of a^b mod P

def power(a, b, p):

   if b == 1:

      return a

   else:

      return pow(a, b) % p

# Main function

def main():

   # Both persons agree upon the public keys G and P

   # A prime number P is taken

   P = 23

   print("The value of P:", P)
```

```
    # A primitive root for P, G is taken
    G = 9
    print("The value of G:", G)
    # Alice chooses the private key a
    # a is the chosen private key
    a = 4
    print("The private key a for Alice:", a)

    # Gets the generated key
    x = power(G, a, P)
    # Bob chooses the private key b
    # b is the chosen private key
    b = 3
    print("The private key b for Bob:", b)
    # Gets the generated key
    y = power(G, b, P)
    # Generating the secret key after the exchange of keys
    ka = power(y, a, P)  # Secret key for Alice
    kb = power(x, b, P)  # Secret key for Bob
    print("Secret key for Alice is:", ka)
    print("Secret key for Bob is:", kb)

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

OUTPUT:



```
The value of P: 23
The value of G: 9
The private key a for Alice: 4
The private key b for Bob: 3
Secret key for Alice is: 9
Secret key for Bob is: 9
```

Practical 9

AIM: Implement RSA encryption-decryption algorithm.

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

An example of asymmetric cryptography:

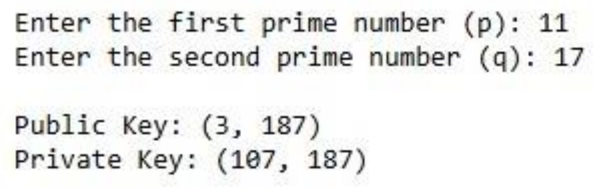
1. A client (for example browser) sends its public key to the server and requests some data.
2. The server encrypts the data using the client’s public key and sends the encrypted data.
3. The client receives this data and decrypts it.

CODE:

```
import math from sympy import
mod_inverse, isprime def gcd(a, b):

   """Compute the greatest common divisor using Euclidean
algorithm."""     while b != 0:

       a, b = b, a % b
return a def
generate_keys(p, q):

   """Generate RSA public and private
keys."""     n = p * q

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

   # Choose e such that 1 < e < phi and gcd(e,
phi) = 1     e = 2     while e < phi:

       if gcd(e, phi) == 1:

           break
e += 1
```

```
   # Calculate d, the modular multiplicative inverse of e
mod phi     d = mod_inverse(e, phi)     return (e, n), (d, n)
# Public key and private key


def encrypt(message, public_key):

   """Encrypt message using the public key."""

   e, n = public_key     return
pow(message, e, n) def
decrypt(ciphertext, private_key):

   """Decrypt ciphertext using the private
key."""     d, n = private_key     return
pow(ciphertext, d, n) def main():

   # Get user input for primes p and q     p =
int(input("Enter the first prime number (p): "))
q = int(input("Enter the second prime number (q):
"))     # Validate if p and q are prime     if not
(isprime(p) and isprime(q)):

      print("Both numbers must be prime.")
return

   # Generate keys

   public_key, private_key =
generate_keys(p, q)     print(f"\nPublic Key:
{public_key}")     print(f"Private Key:
{private_key}") if __name__ == "__main__":
main()
```

OUTPUT :



```
Enter the first prime number (p): 11
Enter the second prime number (q): 17

Public Key: (3, 187)
Private Key: (107, 187)
```

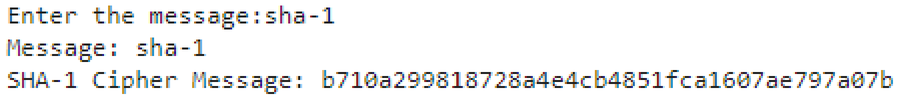
Practical 10

Aim: Write a program to generate SHA-1 Hash.

CODE:

```
import hashlib

message = input("Enter the message:")
cipherMessage = hashlib.sha1(message.encode()).hexdigest()

print("Message: " + message)
print("SHA-1 Cipher Message: " + cipherMessage)
```

OUTPUT:



```
Enter the message:sha-1
Message: sha-1
SHA-1 Cipher Message: b710a299818728a4e4cb4851fca1607ae797a07b
```

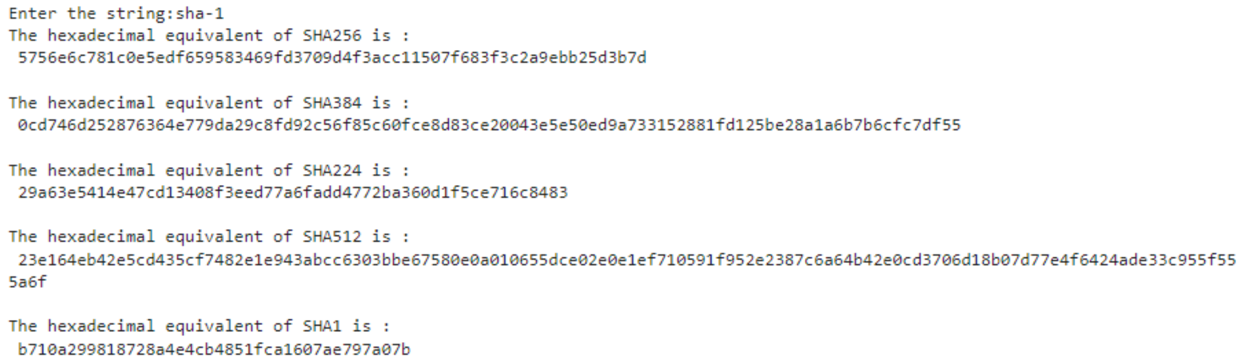
OR

CODE:

```
import hashlib
def print_sha_hashes(input_string):
    """
    Prints the SHA hash values of the input string.
    Args:
        input_string (str): The string to be hashed.
    """
    hash_functions = [hashlib.sha256, hashlib.sha384, hashlib.sha224,
hashlib.sha512, hashlib.sha1]
    hash_names = ["SHA256", "SHA384", "SHA224", "SHA512", "SHA1"]

    for hash_function, hash_name in zip(hash_functions, hash_names):
        print(f"The hexadecimal equivalent of {hash_name} is : \n",
hash_function(input_string.encode()).hexdigest())
        print("\r")
```

```
# Example usage
input_string = input("Enter the string:")
print_sha_hashes(input_string)
```

OUTPUT:



```
Enter the string:sha-1
The hexadecimal equivalent of SHA256 is :
 5756e6c781c0e5edf659583469fd3709d4f3acc11507f683f3c2a9ebb25d3b7d

The hexadecimal equivalent of SHA384 is :
 0cd746d252876364e779da29c8fd92c56f85c60fce8d83ce20043e5e50ed9a733152881fd125be28a1a6b7b6cfc7df55

The hexadecimal equivalent of SHA224 is :
 29a63e5414e47cd13408f3eed77a6fadd4772ba360d1f5ce716c8483

The hexadecimal equivalent of SHA512 is :
 23e164eb42e5cd435cf7482e1e943abcc6303bbe67580e0a010655dce02e0e1ef710591f952e2387c6a64b42e0cd3706d18b07d77e4f6424ade33c955f55
5a6f

The hexadecimal equivalent of SHA1 is :
 b710a299818728a4e4cb4851fca1607ae797a07b
```

Practical 11

Aim: Implement Digital Signature Algorithm.

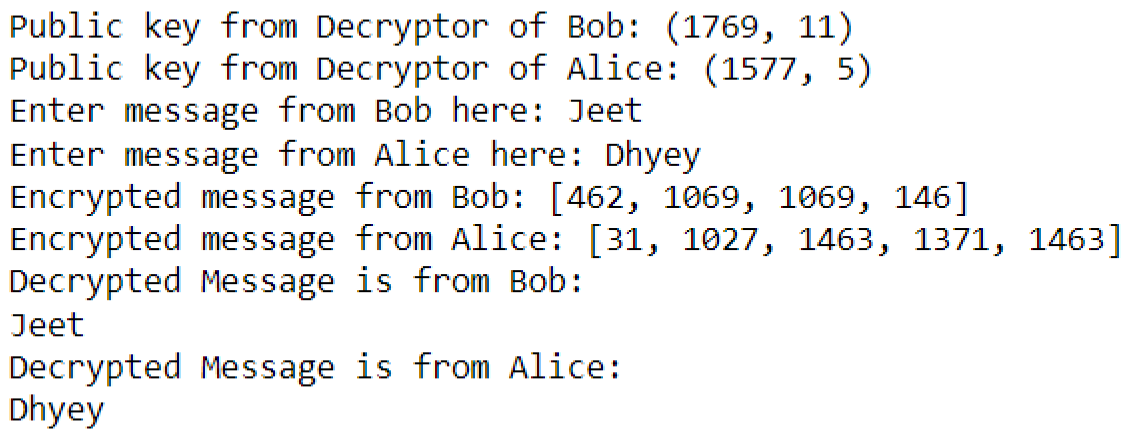
CODE:

```
import hashlib
import random
def get_odd_number(a, b):
    """Generate a random odd number between a and b."""
    rand = int(random.randrange(a, b))
    if rand % 2 == 0:
        rand += 1
    return rand
def is_composite(n, a, r, s):
    """Check if a number n is composite using a base a."""
    res = int(pow(a, r, n))
    if res == 1 or res == n - 1:
        return False
    for _ in range(int(s)):
        res = int(pow(res, 2, n))
        if res == n - 1:
            return False
    return True
def is_prime(n):
    """Check if a number n is prime."""
    if n <= 1:
        return False
    r = int(n) - 1
    s = 0
    while r % 2 == 0:
        r //= 2
        s += 1
    limit = (n - 1) // 4
    for _ in range(int(limit)):
        a = random.randint(2, n - 2)
        if is_composite(n, a, r, s):
            return False
    return True
def generate_prime_number(a, b):
    """Generate a prime number between a and b."""
    while True:
        number = get_odd_number(a, b)
        if is_prime(number):
            return number
def gcd(a, b):
    """Compute the greatest common divisor of a and b."""
    while b != 0:
        a, b = b, a % b
    return a
```

```
def multiplicative_inverse(e, phi):
    """Compute the modular multiplicative inverse of e modulo phi."""
    temp_phi = phi
    y = 0
    x = 1
    while e > 1:
        quotient = e // temp_phi
        t = temp_phi
        temp_phi = e % temp_phi
        e = t
        t = y
        y = x - (quotient * y)
        x = t
    if x < 0:
        x += phi
    return x
class Decryptor:
    def __init__(self):
        """Initialize Decryptor with generated prime numbers and keys."""
        p = generate_prime_number(2, 100)
        q = generate_prime_number(2, 100)
        self.n = p * q
        self.phi = (p - 1) * (q - 1)
        self.e = self.generate_public_key()
        self.d = multiplicative_inverse(self.e, self.phi)
    def generate_public_key(self):
        """Generate a public key."""
        public_key = 2
        while public_key < self.phi:
            if gcd(public_key, self.phi) == 1:
                return public_key
            public_key += 1
    def get_public_key(self):
        """Return the public key (n, e)."""
        return self.n, self.e
    def decrypt(self, data):
        """Decrypt data using the private key."""
        return "".join([chr(pow(char, self.d, self.n)) for char in data])
    def sign_message(self, message):
        """Sign a message using the private key."""
        message_hash = hashlib.sha256(message.encode("UTF-8")).hexdigest()
        signature = [(pow(ord(char), self.d, self.n)) for char in message_hash]
        return signature
class Encryptor:
    def __init__(self, n, e):
        """Initialize Encryptor with public key (n, e)."""
        self.n = n
        self.e = e
    def encrypt(self, data):
        """Encrypt data using the public key."""
        return [pow(ord(char), self.e, self.n) for char in data]
    def verify_signature(self, signature, message):
        """Verify the signature of a message."""
        try:
            message_hash = hashlib.sha256(message.encode("UTF-8")).hexdigest()
            signature_hash_value = "".join([chr(pow(char, self.e, self.n)) for char in
signature])
            return signature_hash_value == message_hash
        except Exception as e:
            print(f"Error: Signature verification process failed - {e}")
            return False
# Example usage
bob = Decryptor()
print("Public key from Decryptor of Bob:", bob.get_public_key())
alice = Decryptor()
print("Public key from Decryptor of Alice:", alice.get_public_key())
alice_encryptor = Encryptor(*alice.get_public_key())
bob_encryptor = Encryptor(*bob.get_public_key())
bob_message = input("Enter message from Bob here: ")
alice_message = input("Enter message from Alice here: ")
# Encrypt messages
encrypted_bob = alice_encryptor.encrypt(bob_message)
encrypted_alice = bob_encryptor.encrypt(alice_message)
```

```
# Print encrypted messages
print("Encrypted message from Bob:", encrypted_bob)
print("Encrypted message from Alice:", encrypted_alice)
# Sign messages
bob_signature = bob.sign_message(bob_message)
# Decrypt messages
decrypted_bob = alice.decrypt(encrypted_bob)
if bob_encryptor.verify_signature(bob_signature, decrypted_bob):
    print("Decrypted Message is from Bob:")
    print(decrypted_bob)
alice_signature = alice.sign_message(alice_message)
decrypted_alice = bob.decrypt(encrypted_alice)
if alice_encryptor.verify_signature(alice_signature, decrypted_alice):
    print("Decrypted Message is from Alice:")
    print(decrypted_alice)
```

OUTPUT:



```
Public key from Decryptor of Bob: (1769, 11)
Public key from Decryptor of Alice: (1577, 5)
Enter message from Bob here: Jeet
Enter message from Alice here: Dhyey
Encrypted message from Bob: [462, 1069, 1069, 146]
Encrypted message from Alice: [31, 1027, 1463, 1371, 1463]
Decrypted Message is from Bob:
Jeet
Decrypted Message is from Alice:
Dhyey
```