**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: "))

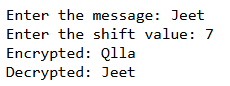
encrypted\_text = caesar\_cipher\_encrypt(msg, shift)

print("Encrypted:", encrypted\_text)

decrypted\_text = caesar\_cipher\_decrypt(encrypted\_text, shift)

print("Decrypted:", decrypted\_text)

**OUTPUT:**

****

**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)

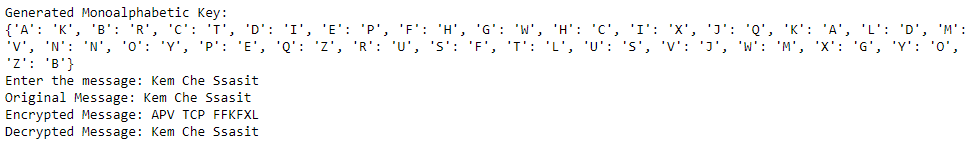
print("Decrypted Message:", decrypted\_message)

# Analyze the frequency of characters in the original and encrypted messages

original\_frequency = analyze\_frequency(message)

encrypted\_frequency = analyze\_frequency(encrypted\_message)

**OUTPUT:**



**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: ")

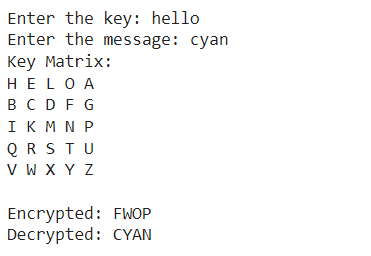
ciphertext = playfair\_cipher(key, plaintext, mode='encrypt')

print("Encrypted:", ciphertext)

decrypted\_text = playfair\_cipher(key, ciphertext, mode='decrypt')

print("Decrypted:", decrypted\_text)

**OUTPUT:**



**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}")

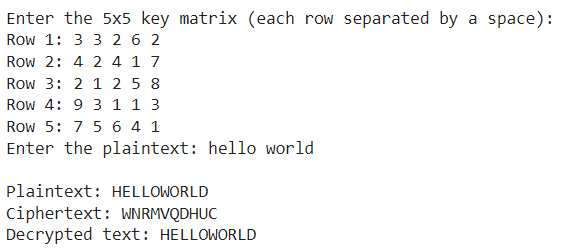
print(f"Ciphertext: {ciphertext}")

print(f"Decrypted text: {decrypted\_text}")

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

main()

**OUTPUT:**

****

**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: ")

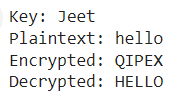
encrypted = vigenere\_encrypt(plaintext, key)

print("Encrypted:", encrypted)

decrypted = vigenere\_decrypt(encrypted, key)

print("Decrypted:", decrypted)

**OUTPUT:**

****

**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)

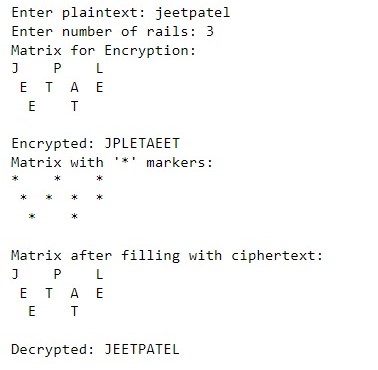
decrypted\_message = rail\_fence\_decrypt(encrypted\_message, num\_rails)

print("Decrypted message:", decrypted\_message)

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

main()

**OUTPUT:**

****

**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: ")

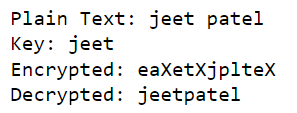
encrypted = encrypt(plaintext, keyword)

print(f"Encrypted: {encrypted}")

decrypted = decrypt(encrypted, keyword)

print(f"Decrypted: {decrypted}")

**OUTPUT:**

****

**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

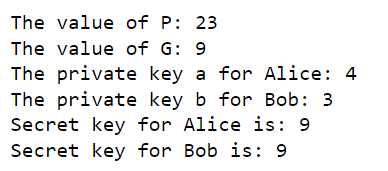
print("Secret key for Alice is:", ka)

print("Secret key for Bob is:", kb)

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

main()

**OUTPUT:**



**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():

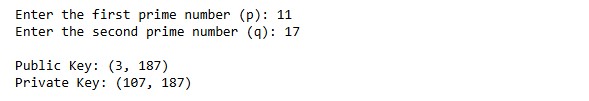
# Get user input for primes p and q p = int(input("Enter the first prime number (p): ")) q = int(input("Enter the second prime number (q): ")) # Validate if p and q are prime if not (isprime(p) and isprime(q)):

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 :**



**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:**

****

**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())

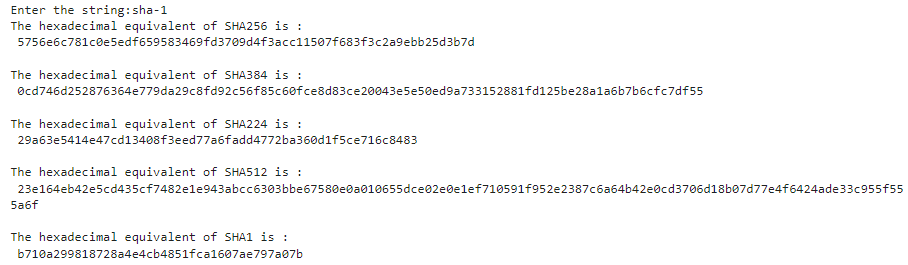
print("\r")

*# Example usage*

input\_string **=** input("Enter the string:")

print\_sha\_hashes(input\_string)

**OUTPUT:**



**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:**

