**PRACTICAL : 1**

**Aim: Implement Caesar cipher.**

**Code:**

def encrypt\_text(plaintext, n):

    ans = ""

    for i in range(len(plaintext)):

        ch = plaintext[i]

        if ch == " ":

            ans += " "

        elif (ch.isupper()):

            ans += chr((ord(ch) + n-65) % 26 + 65)

        else:

            ans += chr((ord(ch) + n-97) % 26 + 97)

    return ans

def decrypt():

    encrypted\_message = input("Enter the message i.e to be decrypted: ").strip()

    letters = "abcdefghijklmnopqrstuvwxyz"

    k = int(input("Enter the key to decrypt: "))

    decrypted\_message = ""

    for ch in encrypted\_message:

        if ch in letters:

            position = letters.find(ch)

            new\_pos = (position - k) % 26

            new\_char = letters[new\_pos]

            decrypted\_message += new\_char

        else:

            decrypted\_message += ch

    return decrypted\_message

plaintext = str(input("Enter Plain Text:-"))

n = int(input("Enter n:-"))

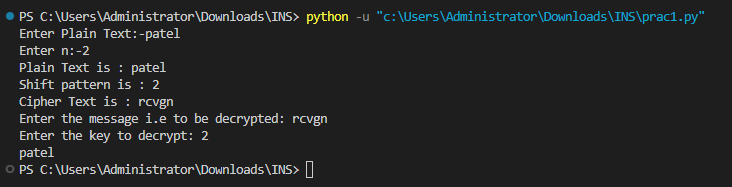
print("Plain Text is : " + plaintext)

print("Shift pattern is : " + str(n))

print("Cipher Text is : " + encrypt\_text(plaintext, n))

print(decrypt())

**Output:**

****

**PRACTICAL : 2**

**Aim: Implement Transposition cipher.**

**Code:**

import math

key = input("Enter keyword text (Contains unique letters only): ").lower().replace(" ", "")

plain\_text = input("Enter plain text (Letters only): ").lower().replace(" ", "")

len\_key = len(key)

len\_plain = len(plain\_text)

row = int(math.ceil(len\_plain / len\_key))

matrix = [['X']\*len\_key for i in range(row)]

t = 0

for r in range(row):

    for c, ch in enumerate(plain\_text[t: t + len\_key]):

        matrix[r][c] = ch

    t += len\_key

sort\_order = sorted([(ch, i) for i, ch in enumerate(key)])

cipher\_text = ''

for ch, c in sort\_order:

    for r in range(row):

        cipher\_text += matrix[r][c]

print("Encryption")

print("Plain text is :", plain\_text)

print("Cipher text is:", cipher\_text)

matrix\_new = [ ['X']\*len\_key for i in range(row) ]

key\_order = [ key.index(ch) for ch in sorted(list(key))]

t = 0

for c in key\_order:

  for r,ch in enumerate(cipher\_text[t : t+ row]):

    matrix\_new[r][c] = ch

  t += row

p\_text = ''

for r in range(row):

  for c in range(len\_key):

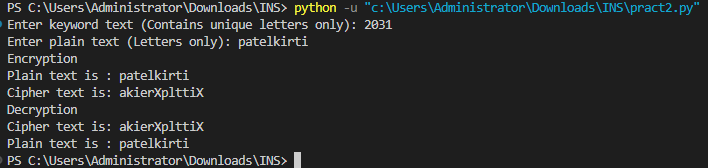
    p\_text += matrix\_new[r][c] if matrix\_new[r][c] != 'X' else ''

print("Decryption")

print("Cipher text is:",cipher\_text)

print("Plain text is :",p\_text)

**Output:**

****

**PRACTICAL : 3**

**Aim: Implement Play-fair cipher.**

**Code:**

from collections import OrderedDict

def generate\_pairs():

i = 0

while i != len(plain\_text\_list):

if i == (len(plain\_text\_list) - (len(plain\_text\_list) % 2)) and len(plain\_text\_list) % 2 != 0:

plain\_text\_list.append('x')

break

if plain\_text\_list[i] == plain\_text\_list[i + 1]:

plain\_text\_list.insert(i + 1, 'x')

i += 2

create\_key\_matrix()

def create\_key\_matrix():

key\_list\_tmp.extend(key\_duplicates)

var = 0

while len(key\_list\_tmp) != 25:

value = chr(97 + var)

if value not in key\_list\_tmp:

if value != 'j':

key\_list\_tmp.append(value)

var += 1

for i in range(0, len(key\_list\_tmp), 5):

matrix\_pf.append(key\_list\_tmp[i:i + 5])

print("\nMatrix:")

for i in matrix\_pf:

print(i, end="\n")

playfair\_cipher\_algorithm()

def fetch\_index(value\_fn):

for index\_one\_fe, i in enumerate(matrix\_pf):

for index\_two\_fe, j in enumerate(i):

if j == value\_fn:

return index\_one\_fe, index\_two\_fe

def playfair\_cipher\_algorithm():

for i in range(0, len(plain\_text\_list) - 1, 2):

index\_one\_pf, index\_two\_pf = fetch\_index(plain\_text\_list[i])

index\_three\_pf, index\_four\_pf = fetch\_index(plain\_text\_list[i + 1])

if index\_one\_pf == index\_three\_pf:

index\_two\_pf = (index\_two\_pf + 1) % 5

index\_four\_pf = (index\_four\_pf + 1) % 5

cipher\_text.extend(matrix\_pf[index\_one\_pf][index\_two\_pf])

cipher\_text.extend(matrix\_pf[index\_three\_pf][index\_four\_pf])

elif index\_two\_pf == index\_four\_pf:

index\_one\_pf = (index\_one\_pf + 1) % 5

index\_three\_pf = (index\_three\_pf + 1) % 5

cipher\_text.extend(matrix\_pf[index\_one\_pf][index\_two\_pf])

cipher\_text.extend(matrix\_pf[index\_three\_pf][index\_four\_pf])

else:

cipher\_text.extend(matrix\_pf[index\_one\_pf][index\_four\_pf])

cipher\_text.extend(matrix\_pf[index\_three\_pf][index\_two\_pf])

print("\nCipher Text:", "".join(cipher\_text))

plain\_text = input("\nEnter Plain Text: ")

key = input("Enter Key: ")

plain\_text = plain\_text.replace(" ", "").lower()

plain\_text = plain\_text.replace("j", "i")

key = key.replace(" ", "").lower()

key = key.replace("j", "i")

key\_duplicates = "".join(OrderedDict.fromkeys(key))

plain\_text\_list = list(plain\_text)

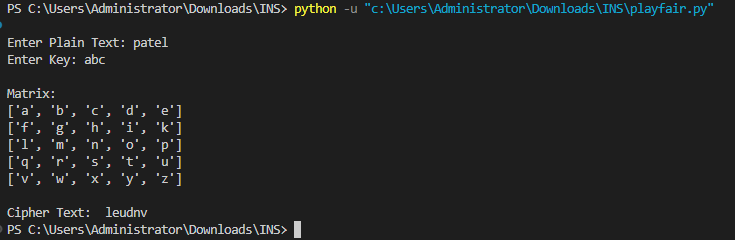
matrix\_pf = []

key\_list\_tmp = []

cipher\_text = []

generate\_pairs()

**Output:**

****

**PRACTICAL : 4**

**Aim: Implement substitution cipher.**

**Code:**

import random

import string

def generate\_substitution\_key():

    alphabet = string.ascii\_uppercase

    shuffled\_alphabet = list(alphabet)

    random.shuffle(shuffled\_alphabet)

    substitution\_key = {}

    for i in range(len(alphabet)):

        substitution\_key[alphabet[i]] = shuffled\_alphabet[i]

    return substitution\_key

def encrypt\_substitution(plaintext, substitution\_key):

    ciphertext = ""

    for char in plaintext:

        if char.isalpha() and char.isupper():

            ciphertext += substitution\_key[char]

        else:

            ciphertext += char

    return ciphertext

def decrypt\_substitution(ciphertext, substitution\_key):

    decryption\_key = {v: k for k, v in substitution\_key.items()}

    plaintext = ""

    for char in ciphertext:

        if char.isalpha() and char.isupper():

            plaintext += decryption\_key[char]

        else:

            plaintext += char

    return plaintext

substitution\_key = generate\_substitution\_key()

plaintext = "PATELKIRTI"

ciphertext = encrypt\_substitution(plaintext, substitution\_key)

decrypted\_text = decrypt\_substitution(ciphertext, substitution\_key)

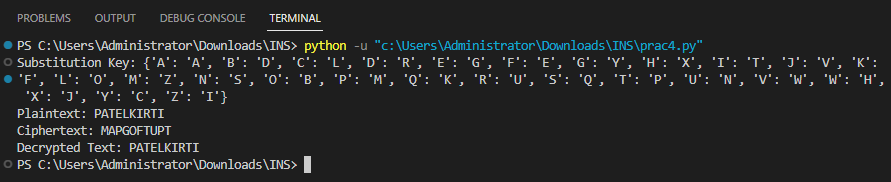
print("Substitution Key:", substitution\_key)

print("Plaintext:", plaintext)

print("Ciphertext:", ciphertext)

print("Decrypted Text:",decrypted\_text)

**Output:**

****

**PRACTICAL : 5**

**Aim: Implement rail-fence cipher.**

**Code:**

def encrypt\_rail\_fence(plaintext, num\_rails):

    rail\_fence = [['' for \_ in range(len(plaintext))] for \_ in range(num\_rails)]

    rail = 0

    direction = 1

    for char in plaintext:

        rail\_fence[rail].append(char)

        rail += direction

        if rail == num\_rails - 1 or rail == 0:

            direction = -direction

    ciphertext = ''.join([char for rail in rail\_fence for char in rail if char != ''])

    return ciphertext

def decrypt\_rail\_fence(ciphertext, num\_rails):

    rail\_fence = [['' for \_ in range(len(ciphertext))] for \_ in range(num\_rails)]

    rail = 0

    direction = 1

    pattern = []

    for \_ in range(len(ciphertext)):

        pattern.append(rail)

        rail += direction

        if rail == num\_rails - 1 or rail == 0:

            direction = -direction

    index = 0

    for char in ciphertext:

        rail\_fence[pattern[index]].append(char)

        index += 1

    plaintext = ''.join([char for rail in rail\_fence for char in rail])

    return plaintext

plaintext = "PATELKIRTI"

num\_rails = 3

ciphertext = encrypt\_rail\_fence(plaintext, num\_rails)

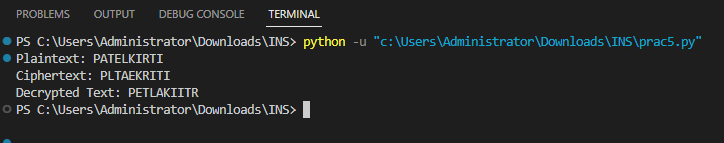
decrypted\_text = decrypt\_rail\_fence(ciphertext, num\_rails)

print("Plaintext:", plaintext)

print("Ciphertext:", ciphertext)

print("Decrypted Text:",decrypted\_text)

**Output:**

****

**PRACTICAL : 6**

**Aim: Implement hill-cipher.**

**Code:**

import numpy as np

import string

from sympy import Matrix

def hc\_encrypt(msg, key):

    dimension = 3

    msg = msg.replace(" ", "")

    alphabets = string.ascii\_lowercase

    encrypted\_message = ""

    for index, i in enumerate(msg):

        values = []

        if index % dimension == 0:

            for j in range(0, dimension):

                if index + j < len(msg):

                    values.append([alphabets.index(msg[index + j])])

                else:

                    values.append([25])

            vector = np.matrix(values)

            vector = key \* vector

            vector = vector % 26

            for j in range(0, dimension):

                encrypted\_message = encrypted\_message + alphabets[vector.item(j)]

    return encrypted\_message

def hc\_decrypt(msg, key):

    dimension = 3

    alphabet = string.ascii\_lowercase

    decrypted\_message = ""

    key = Matrix(key)

    key = key.inv\_mod(26)

    key = key.tolist()

    for index, i in enumerate(msg):

        values = []

        if index % dimension == 0:

            for j in range(0, dimension):

                values.append([alphabet.index(msg[index + j])])

            vector = np.matrix(values)

            vector = key \* vector

            vector = vector % 26

            for j in range(0, dimension):

                decrypted\_message = decrypted\_message + alphabet[vector.item(j)]

    return decrypted\_message

message = input("Enter String: ").lower()

print("Original Message: ", message)

# key = hillciphr

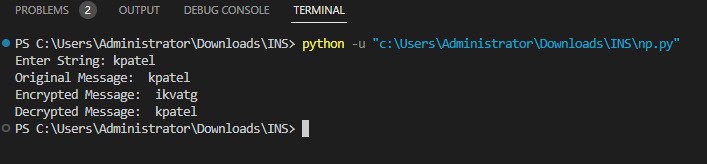
key\_matrix = np.matrix([[7, 8, 11], [11, 2, 8], [15, 7, 17]])

enc = hc\_encrypt(message, key\_matrix)

print("Encrypted Message: ", enc)

print("Decrypted Message: ", hc\_decrypt(enc, key\_matrix))

**Output:**

****

**PRACTICAL : 7**

**Aim: Implement mono-alphabet cipher.**

**Code:**

def monoalphabetic\_encrypt(plaintext, key):

    ciphertext = ""

    for char in plaintext:

        if char.isalpha():

            index = ord(char.upper()) - 65

            ciphertext += key[index]

        else:

            ciphertext += char

    return ciphertext

def monoalphabetic\_decrypt(ciphertext, key):

    plaintext = ""

    for char in ciphertext:

        if char.isalpha():

            index = key.index(char.upper())

            plaintext += chr(index + 65)

        else:

            plaintext += char

    return plaintext

plaintext = "KIRTIPATEL"

key = "KRTYUIOPASDFGHEJLZXCVBNMWQ"

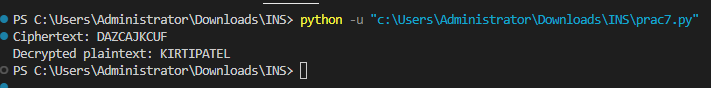
ciphertext = monoalphabetic\_encrypt(plaintext, key)

print("Ciphertext:", ciphertext)

decrypted = monoalphabetic\_decrypt(ciphertext, key)

print("Decrypted plaintext:", decrypted)

**Output:**

****

**PRACTICAL : 8**

**Aim: Implement Polyalphabetic cipher encryption-decryption.**

**Code:**

def vigenere\_encrypt(plain\_text, key):

    encrypted\_text = ""

    key\_index = 0

    for char in plain\_text:

        if char.isalpha():

            shift = ord(key[key\_index % len(key)].lower()) - ord('a')

            encrypted\_char = chr((ord(char.lower()) - ord('a') + shift) % 26 + ord('a'))

            encrypted\_text += encrypted\_char

            key\_index += 1

        else:

            encrypted\_text += char

    return encrypted\_text

def vigenere\_decrypt(encrypted\_text, key):

    decrypted\_text = ""

    key\_index = 0

    for char in encrypted\_text:

        if char.isalpha():

            shift = ord(key[key\_index % len(key)].lower()) - ord('a')

            decrypted\_char = chr((ord(char.lower()) - ord('a') - shift) % 26 + ord('a'))

            decrypted\_text += decrypted\_char

            key\_index += 1

        else:

            decrypted\_text += char

    return decrypted\_text

plain\_text = "KirtiPatel"

key = "secret"

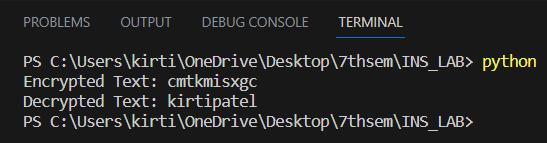
encrypted\_text = vigenere\_encrypt(plain\_text, key)

decrypted\_text = vigenere\_decrypt(encrypted\_text, key)

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

print("Decrypted Text:", decrypted\_text)

**Output:**



**PRACTICAL : 9**

**Aim: Implement one time pad encryption-decryption.**

**Code:**

import string

import random

def encrypt(code\_book):

    cipherText = ''

    plainText = input("Enter Plain Text: ").upper()

    plainText = plainText.replace(" ","")

    for i in plainText:

        cipherText += code\_book[i]

    print("\nCipher Text :", cipherText)

def decrypt(code\_book):

    code\_book = { v:k for (k,v) in code\_book.items()}

    plainText = ''

    cipherText = input("\nEnter Cipher Text: ").upper()

    plainText = plainText.replace(" ","")

    for i in cipherText:

        plainText += code\_book[i]

    print("\nPlanin Text :", plainText)

def get\_code\_book():

    keys = [i for i in string.ascii\_uppercase]

    values = keys.copy()

    random.shuffle(values)

    res = {}

    for key in keys:

        for value in values:

            res[key] = value

            values.remove(value)

            break

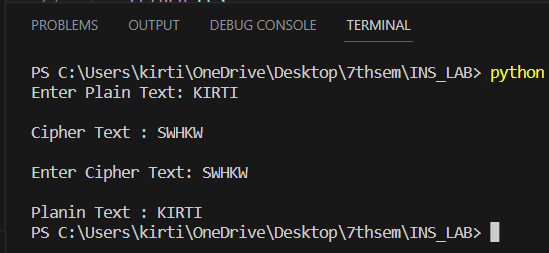
    return res

code\_book = get\_code\_book()

encrypt(code\_book)

decrypt(code\_book)

**Output:**

****

**PRACTICAL : 10**

**Aim: Implement DES encryption-decryption.**

**Code:**

from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

def des\_encrypt(plaintext, key):

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

    padded\_plaintext = pad(plaintext, DES.block\_size)

    ciphertext = cipher.encrypt(padded\_plaintext)

    return ciphertext

def des\_decrypt(ciphertext, key):

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

    decrypted\_data = cipher.decrypt(ciphertext)

    unpadded\_data = unpad(decrypted\_data, DES.block\_size)

    return unpadded\_data

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

    predefined\_key = bytes.fromhex("AABBCCDDEEFF0011")

    plaintext = input("Enter the plaintext to encrypt: ").encode('utf-8')

    try:

        encrypted\_data = des\_encrypt(plaintext, predefined\_key)

        decrypted\_data = des\_decrypt(encrypted\_data, predefined\_key)

        print("Key:", predefined\_key)

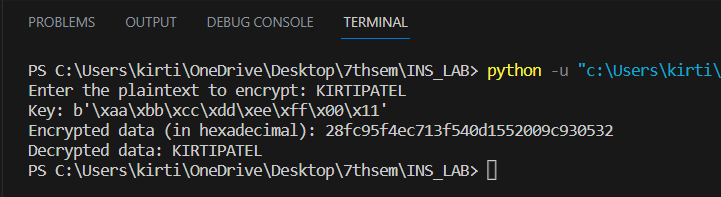
        print("Encrypted data (in hexadecimal):", encrypted\_data.hex())

        print("Decrypted data:", decrypted\_data.decode('utf-8'))

    except Exception as e:

        print("Error:", e)

**Output:**

****

**PRACTICAL : 11**

**Aim: Implement AES encryption-decryption.**

**Code:**

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives import padding as sym\_padding

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

import os

def derive\_key(password, salt):

    kdf = PBKDF2HMAC(

        algorithm=hashes.SHA256(),

        iterations=100000,

        salt=salt,

        length=32,

        backend=default\_backend()

    )

    return kdf.derive(password.encode())

def encrypt\_data(key, data):

    iv = os.urandom(16)  # Initialization vector

    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())

    encryptor = cipher.encryptor()

    padder = sym\_padding.PKCS7(128).padder()

    padded\_data = padder.update(data) + padder.finalize()

    encrypted\_data = encryptor.update(padded\_data) + encryptor.finalize()

    return iv + encrypted\_data

def decrypt\_data(key, encrypted\_data):

    iv = encrypted\_data[:16]

    data = encrypted\_data[16:]

    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())

    decryptor = cipher.decryptor()

    unpadder = sym\_padding.PKCS7(128).unpadder()

    decrypted\_data = decryptor.update(data) + decryptor.finalize()

    unpadded\_data = unpadder.update(decrypted\_data) + unpadder.finalize()

    return unpadded\_data

def main():

    password = "secrettt"

    salt = os.urandom(16)  # Random salt

    key = derive\_key(password, salt)

    data\_to\_encrypt = b"KIRTIPATEL"

    encrypted\_data = encrypt\_data(key, data\_to\_encrypt)

    decrypted\_data = decrypt\_data(key, encrypted\_data)

    print("Original Data:", data\_to\_encrypt)

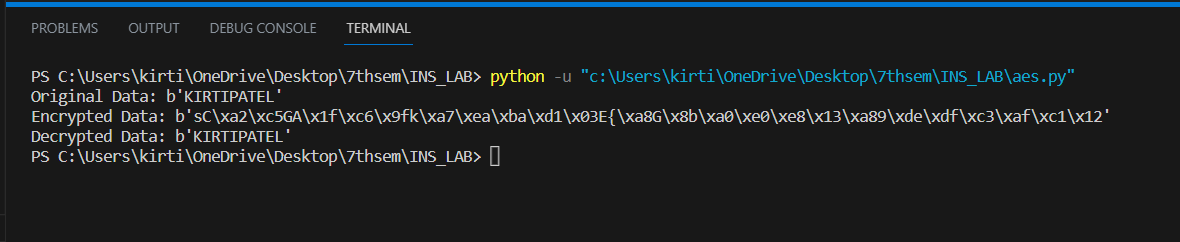
    print("Encrypted Data:", encrypted\_data)

    print("Decrypted Data:", decrypted\_data)

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

    main()

**Output:**

****

**PRACTICAL : 12**

**Aim: Implement RSA Algorithm.**

**Code:**

import random

def is\_prime(num):

    if num <= 1:

        return False

    if num <= 3:

        return True

    if num % 2 == 0 or num % 3 == 0:

        return False

    i = 5

    while i \* i <= num:

        if num % i == 0 or num % (i + 2) == 0:

            return False

        i += 6

    return True

def generate\_prime(bits):

    while True:

        num = random.getrandbits(bits)

        if is\_prime(num):

            return num

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

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

    return x1 + m0 if x1 < 0 else x1

bits = 16

p = generate\_prime(bits)

q = generate\_prime(bits)

n = p \* q

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

while True:

    e = random.randrange(2, phi\_n)

    if gcd(e, phi\_n) == 1:

        break

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

def encrypt(message, n, e):

    return pow(message, e, n)

def decrypt(ciphertext, n, d):

    return pow(ciphertext, d, n)

message = int(input("Enter a number to encrypt: "))

if message >= n:

    print("Message is too large for this RSA setup. Please choose a smaller number.")

else:

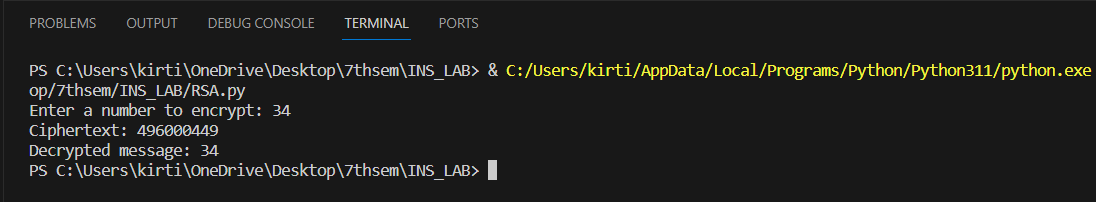
    ciphertext = encrypt(message, n, e)

    print(f"Ciphertext: {ciphertext}")

    decrypted\_message = decrypt(ciphertext, n, d)

    print(f"Decrypted message: {decrypted\_message}")

**Output:**

****

**PRACTICAL : 13**

**Aim: Implement Diffie–Hellman Algorithm.**

**Code:**

import random

def fast\_modulo(base, exp, mod):

    result = 1

    base = base % mod

    while exp > 0:

        if exp % 2 == 1:

            result = (result \* base) % mod

        exp = exp // 2

        base = (base \* base) % mod

    return result

def generate\_prime(bits=256):

    while True:

        num = random.getrandbits(bits)

        if num > 1 and is\_prime(num):

            return num

def is\_prime(n, k=5):

    if n <= 1:

        return False

    if n <= 3:

        return True

    if n % 2 == 0:

        return False

       for \_ in range(k):

        a = random.randint(2, n - 2)

        x = fast\_modulo(a, n - 1, n)

        if x != 1:

            return False

    return True

def diffie\_hellman():

    prime = generate\_prime()

    primitive\_root = random.randint(2, prime - 2)

    alice\_private\_key = random.randint(2, prime - 2)

    alice\_public\_key = fast\_modulo(primitive\_root, alice\_private\_key, prime)

    bob\_private\_key = random.randint(2, prime - 2)

    bob\_public\_key = fast\_modulo(primitive\_root, bob\_private\_key, prime)

    alice\_shared\_secret = fast\_modulo(bob\_public\_key, alice\_private\_key, prime)

    bob\_shared\_secret = fast\_modulo(alice\_public\_key, bob\_private\_key, prime)

    return prime, primitive\_root, alice\_private\_key, alice\_public\_key, bob\_private\_key, bob\_public\_key, alice\_shared\_secret, bob\_shared\_secret

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

    prime, primitive\_root, alice\_private\_key, alice\_public\_key, bob\_private\_key, bob\_public\_key, alice\_shared\_secret, bob\_shared\_secret = diffie\_hellman()

    print(f"Prime: {prime}")

    print(f"Primitive Root: {primitive\_root}")

    print(f"Alice's Private Key: {alice\_private\_key}")

    print(f"Alice's Public Key: {alice\_public\_key}")

    print(f"Bob's Private Key: {bob\_private\_key}")

    print(f"Bob's Public Key: {bob\_public\_key}")

    print(f"Alice's Shared Secret: {alice\_shared\_secret}")

    print(f"Bob's Shared Secret: {bob\_shared\_secret}")

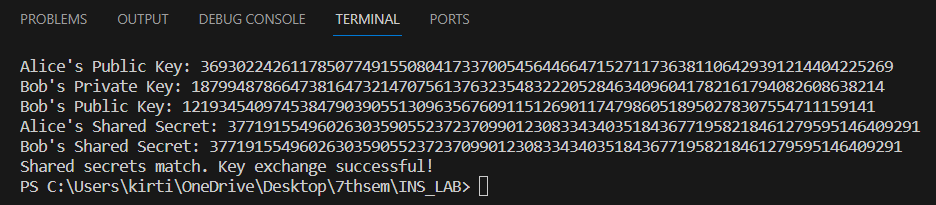
    if alice\_shared\_secret == bob\_shared\_secret:

        print("Shared secrets match. Key exchange successful!")

    else:

        print("Shared secrets do not match. Key exchange failed!")

**Output:**

****

**PRACTICAL : 14**

**Aim: Demonstrate working of Digital Signature using Cryptool.**

**Code:**

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives import serialization

private\_key = rsa.generate\_private\_key(

    public\_exponent=65537,

    key\_size=2048,

    backend=default\_backend()

)

private\_pem = private\_key.private\_bytes(

    encoding=serialization.Encoding.PEM,

    format=serialization.PrivateFormat.PKCS8,

    encryption\_algorithm=serialization.NoEncryption()

)

public\_key = private\_key.public\_key()

public\_pem = public\_key.public\_bytes(

    encoding=serialization.Encoding.PEM,

    format=serialization.PublicFormat.SubjectPublicKeyInfo

)

data\_to\_sign = b"Patel Kirti"

print(data\_to\_sign)

signature = private\_key.sign(

    data\_to\_sign,

    padding.PSS(

        mgf=padding.MGF1(hashes.SHA256()),

        salt\_length=padding.PSS.MAX\_LENGTH

    ),

    hashes.SHA256()

)

try:

    public\_key.verify(

        signature,

        data\_to\_sign,

        padding.PSS(

            mgf=padding.MGF1(hashes.SHA256()),

            salt\_length=padding.PSS.MAX\_LENGTH

        ),

        hashes.SHA256()

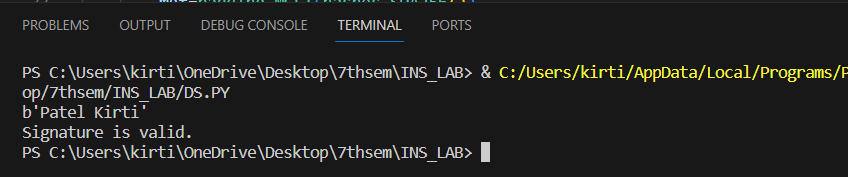
    )

    print("Signature is valid.")

except Exception as e:

    print("Signature is not valid:", str(e))

**Output:**

****