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:

```
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\prac1.py"
Enter Plain Text:-patel
Enter n:-2
Plain Text is : patel
Shift pattern is : 2
Cipher Text is : rcvgn
Enter the message i.e to be decrypted: rcvgn
Enter the key to decrypt: 2
patel
PS C:\Users\Administrator\Downloads\INS>
```

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:

```
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\pract2.py"
Enter keyword text (Contains unique letters only): 2031
Enter plain text (Letters only): patelkirti
Encryption
Plain text is : patelkirti
Cipher text is: akierXplttiX
Decryption
Cipher text is: akierXplttiX
Plain text is : patelkirti
PS C:\Users\Administrator\Downloads\INS>
```

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:

```
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\playfair.py"

Enter Plain Text: patel
Enter Key: abc

Matrix:
['a', 'b', 'c', 'd', 'e']
['f', 'g', 'h', 'i', 'k']
['l', 'm', 'n', 'o', 'p']
['q', 'r', 's', 't', 'u']
['v', 'w', 'x', 'y', 'z']

Cipher Text:  leudnv
PS C:\Users\Administrator\Downloads\INS>
```

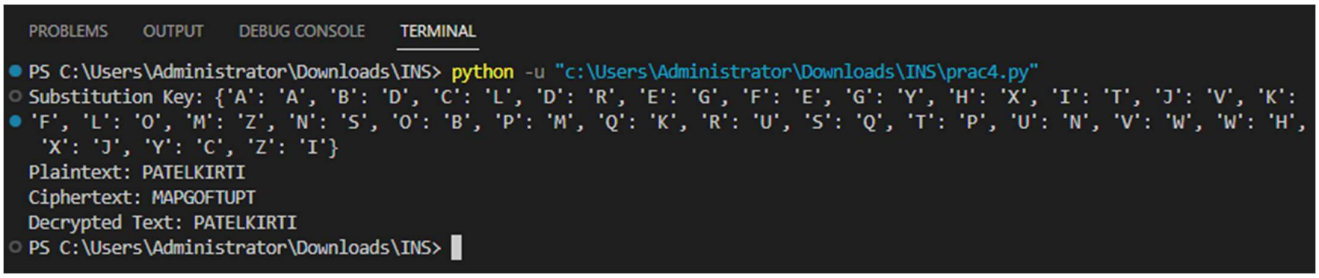
PRACTICAL : 4

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:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\prac4.py"
Substitution Key: {'A': 'A', 'B': 'D', 'C': 'L', 'D': 'R', 'E': 'G', 'F': 'E', 'G': 'Y', 'H': 'X', 'I': 'T', 'J': 'V', 'K': 'F', 'L': 'O', 'M': 'Z', 'N': 'S', 'O': 'B', 'P': 'M', 'Q': 'K', 'R': 'U', 'S': 'Q', 'T': 'P', 'U': 'N', 'V': 'W', 'W': 'H', 'X': 'J', 'Y': 'C', 'Z': 'I'}
Plaintext: PATELKIRTI
Ciphertext: MAPGOFTUPT
Decrypted Text: PATELKIRTI
PS C:\Users\Administrator\Downloads\INS>

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:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\prac5.py"
Plaintext: PATELKIRTI
Ciphertext: PLTAEKRITI
Decrypted Text: PETLAKIITR
PS C:\Users\Administrator\Downloads\INS>

PRACTICAL : 6

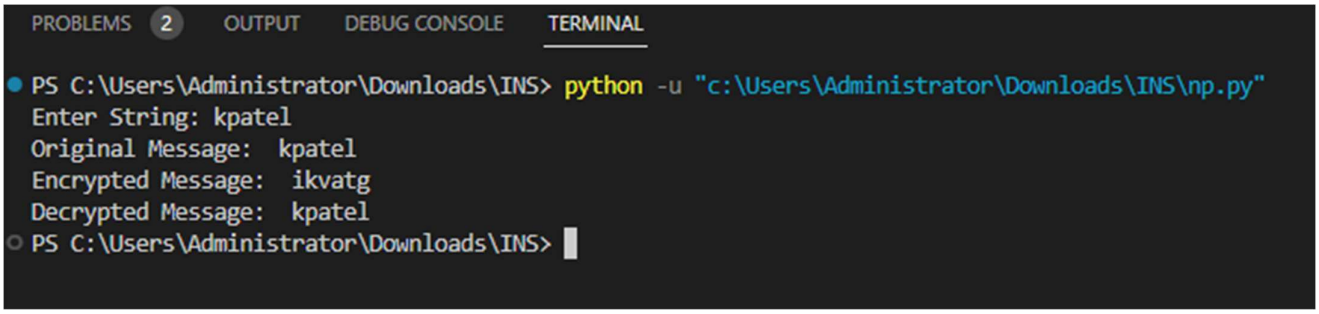
Aim: Implement hill-cipher.

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:

```
PS C:\Users\Administrator\Downloads\INS> python -u "c:\Users\Administrator\Downloads\INS\prac7.py"
Ciphertext: DAZCAJKCUF
Decrypted plaintext: KIRTIPATEL
PS C:\Users\Administrator\Downloads\INS>
```

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:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\kirti\OneDrive\Desktop\7thsem\INS_LAB> python
Encrypted Text: cmtkmisxgc
Decrypted Text: kirtipatel
PS C:\Users\kirti\OneDrive\Desktop\7thsem\INS_LAB>
```

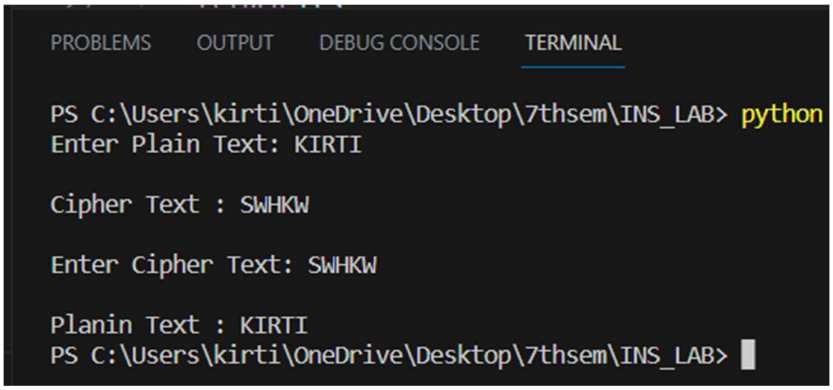
PRACTICAL : 9

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:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS C:\Users\kirti\OneDrive\Desktop\7thsem\INS_LAB> python
Enter Plain Text: KIRTI
Cipher Text : SWHKW
Enter Cipher Text: SWHKW
Planin Text : KIRTI
PS C:\Users\kirti\OneDrive\Desktop\7thsem\INS_LAB>

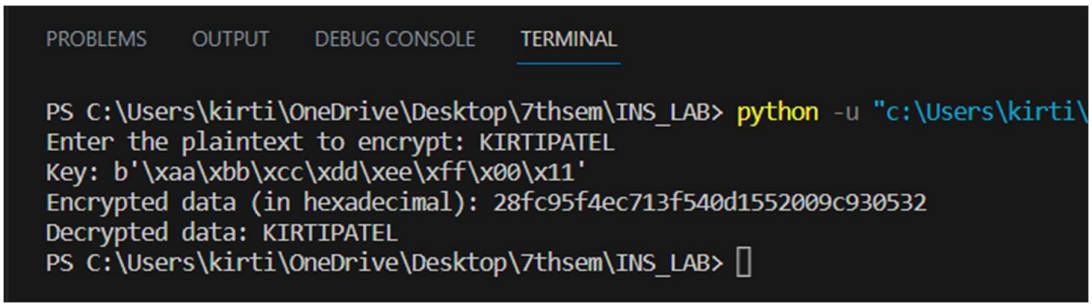
PRACTICAL : 10

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

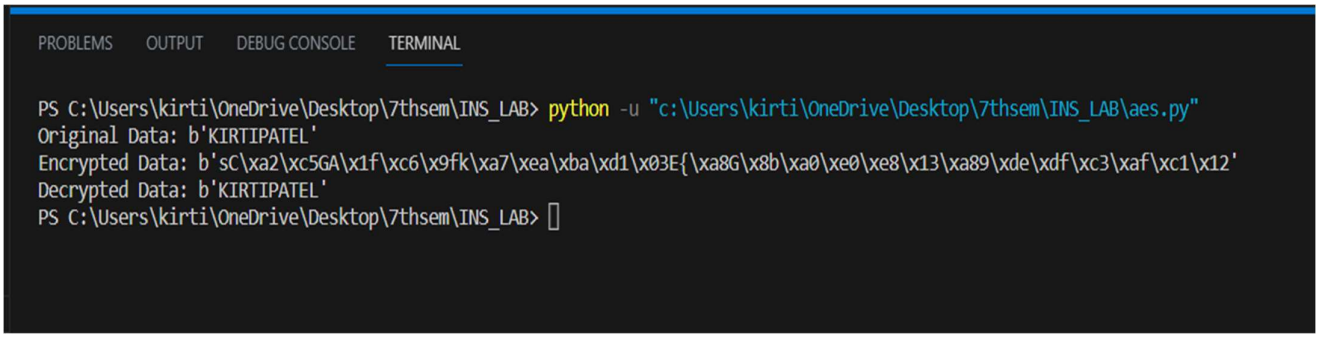
Aim: Implement AES encryption-decryption.

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

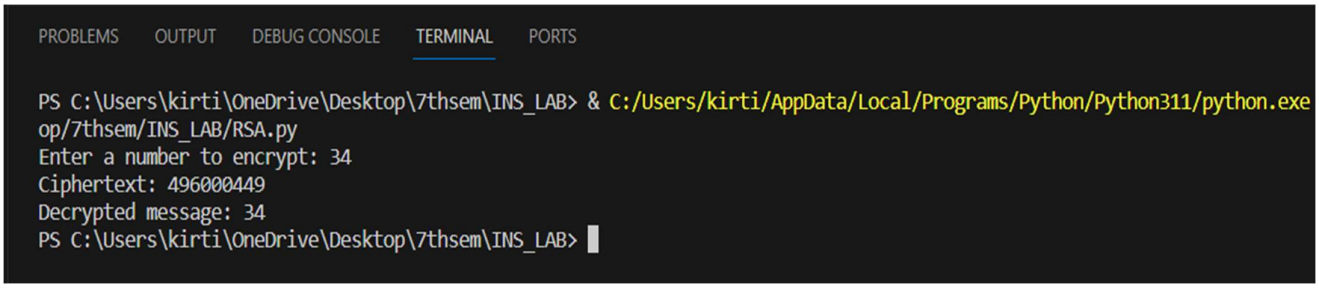
Aim: Implement RSA Algorithm.

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:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Alice's Public Key: 369302242611785077491550804173370054564466471527117363811064293912144042252 69
Bob's Private Key: 187994878664738164732147075613763235483222052846340960417821617940826086382 14
Bob's Public Key: 121934540974538479039055130963567609115126901174798605189502783075547111591 41
Alice's Shared Secret: 377191554960263035905523723709901230833434035184367719582184612795951464092 91
Bob's Shared Secret: 377191554960263035905523723709901230833434035184367719582184612795951464092 91
Shared secrets match. Key exchange successful!
PS C:\Users\kirti\OneDrive\Desktop\7thsem\INS_LAB>
```

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: