1. Write a program to implement Shift Cipher. (Encryption/Decryption/ Input (key/plaintext for encryption/cipher text for decryption) should be taken from user).
Solution:

<u>SOURCE CODE:</u> (in python)

```
def shift_encrypt(plain_text, key):
    cipher_text = ""
    for char in plain_text:
        if char.isupper():
            cipher_text += chr((ord(char) + key - 65) % 26 + 65)
        elif char.islower():
            cipher_text += chr((ord(char) + key - 97) % 26 + 97)
        else:
            cipher_text += char
    return cipher_text
def shift_decrypt(cipher_text, key):
    plain_text = ""
    for char in cipher_text:
        if char.isupper():
            plain_text += chr((ord(char) - key - 65) % 26 + 65)
        elif char.islower():
            plain_text += chr((ord(char) - key - 97) % 26 + 97)
        else:
            plain_text += char
    return plain_text
mode = input("Enter mode (encrypt or decrypt): ")
key = int(input("Enter key (number of positions to shift): "))
text = input("Enter text: ")
if mode == "encrypt":
    result = shift_encrypt(text, key)
    print("Encrypted text:", result)
elif mode == "decrypt":
    result = shift_decrypt(text, key)
    print("Decrypted text:", result)
else:
    print("Invalid mode.")
```

<u>OUTPUT:</u>

```
Enter mode (encrypt or decrypt): encrypt
Enter key (number of positions to shift): 3
Enter text: Hello
Encrypted text: Khoor

Enter mode (encrypt or decrypt): decrypt
Enter key (number of positions to shift): 3
Enter text: Khoor
Decrypted text: Hello
```

2. Write a program to implement Playfair Cipher. (Encryption/Decryption/ Input should be taken from user, Display the key matrix as well).
Solution:

SOURCE CODE:

```
def toLowerCase(text):
    return text.lower()
def removeSpaces(text):
    newText = ""
    for i in text:
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText
def Diagraph(text):
    Diagraph = []
    group = 0
    for i in range(2, len(text), 2):
        Diagraph.append(text[group:i])
        group = i
    Diagraph.append(text[group:])
    return Diagraph
def FillerLetter(text):
    k = len(text)
    if k % 2 == 0:
        for i in range(0, k, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    return new_word
list1 = ['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']
def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
            key_letters.append(i)
    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)
    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]
    return matrix
def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j
def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]
    char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
    else:
        char2 = matr[e2r][e2c+1]
    return char1, char2
def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1r == 4:
        char1 = matr[0][e1c]
    else:
        char1 = matr[e1r+1][e1c]
    char2 = ''
    if e2r == 4:
        char2 = matr[0][e2c]
    else:
        char2 = matr[e2r+1][e2c]
    return char1, char2
def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    char1 = matr[e1r][e2c]
    char2 = ''
    char2 = matr[e2r][e1c]
    return char1, char2
def encryptByPlayfairCipher(Matrix, plainList):
    CipherText = []
    for i in range(0, len(plainList)):
        c1 = 0
        c2 = 0
        ele1_x, ele1_y = search(Matrix, plainList[i][0])
        ele2_x, ele2_y = search(Matrix, plainList[i][1])
        if ele1_x == ele2_x:
            c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
        elif ele1_y == ele2_y:
            c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
        else:
            c1, c2 = encrypt_RectangleRule(
                Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
        cipher = c1 + c2
        CipherText.append(cipher)
    return CipherText
text_Plain =input("Enter Plain Text:")
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'
key =input("Enter Key:")
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)
print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)
CipherText = ""
for i in CipherList:
    CipherText += i
```

```
print("CipherText:", CipherText)
```

OUTPUT:

```
Enter Plain Text:Playfair
Enter Key:Cipher
Key text: Cipher
Plain Text: playfair
CipherText: bsdwrbca
```

3. Write a program to implement Rail Fence Cipher. (Encryption/Decryption/ Input should be taken from user).
Solution:

SOURCE CODE:

```
def encryptRailFence(text, key):
    rail = [['\n' for i in range(len(text))]
                for j in range(key)]
    dir_down = False
    row, col = 0, 0
    for i in range(len(text)):
        if (row == 0) or (row == key - 1):
            dir_down = not dir_down
        rail[row][col] = text[i]
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
    return("" . join(result))
def decryptRailFence(cipher, key):
    rail = [['\n' for i in range(len(cipher))]
                for j in range(key)]
    dir_down = None
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key - 1:
            dir_down = False
        rail[row][col] = '*'
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((rail[i][j] == '*') and
            (index < len(cipher))):
                rail[i][j] = cipher[index]
                index += 1
    result = []
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key-1:
            dir_down = False
        if (rail[row][col] != '*'):
            result.append(rail[row][col])
            col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    return("".join(result))
if __name__ == "__main__":
    mode = input("Enter mode (encrypt or decrypt): ")
    key = int(input("Enter number of rail:"))
    if mode == "encrypt":
      plain_text=input("Enter Plain Text:")
      result = encryptRailFence(plain_text, key)
      print("Encrypted text:", result)
    elif mode == "decrypt":
      cipher_text=input("Enter Cipher text:")
      result = decryptRailFence(cipher_text, key)
      print("Decrypted text:", result)
    else:
      print("Invalid mode.")
```

OUTPUT:

```
Enter mode (encrypt or decrypt): encrypt
Enter number of rail:3
Enter Plain Text:Rail Cipher
Encrypted text: R halCpeiir

Enter mode (encrypt or decrypt): decrypt
Enter number of rail:3
Descrypted text: Rail Cipher
```

4. Write a program to implement Vigenere Cipher. (Encryption/Decryption/ Input should be taken from user).
Solution:

SOURCE CODE:

```
def vigenere_cipher(text, key, encrypt=True):
    key = key.upper()
    result = ''
    key_index = 0
    for char in text:
        if char.isalpha():
            char = char.upper()
            shift = ord(key[key_index]) - ord('A')
            if encrypt:
                char = chr((ord(char) + shift - 2*ord('A')) % 26 + ord('A'))
            else:
                char = chr((ord(char) - shift - 2*ord('A')) % 26 + ord('A'))
            key_index = (key_index + 1) % len(key)
        result += char
    return result
text = input("Enter the text to be encrypted/decrypted: ")
key = input("Enter the key: ")
mode = input("Enter 'e' to encrypt or 'd' to decrypt: ")
if mode == 'e':
    result = vigenere_cipher(text, key)
    print("Encrypted text:", result)
elif mode == 'd':
    result = vigenere_cipher(text, key, False)
    print("Decrypted text:", result)
else:
    print("Invalid mode. Please enter 'e' or 'd'.")
```

OUTPUT:

```
Enter the text to be encrypted/decrypted: Vigenere
Enter the key: Cipher
Enter 'e' to encrypt or 'd' to decrypt: e
Encrypted text: KDIYEIGZ

Enter the text to be encrypted/decrypted: KDIYEIGZ
Enter the key: Cipher
Enter 'e' to encrypt or 'd' to decrypt: d
Decrypted text: VIGENERE
```

5. WAP to implement Euclidean Algorithm to find GCD of given numbers.
Solution:

SOURCE CODE:

```
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
print(f"Gcd of {a} and {b} is {gcd(a,b)}")
```

OUTPUT:

```
Enter first number:4
Enter second number:6
Gcd of 4 and 6 is 2
```

6. Write a program that computes additive inverse in given modulo n.
Solution:

SOURCE CODE:

```
def additive_inverse(num, n):
    for i in range(n):
        if (num + i) % n == 0:
            return i
    return None
num = int(input("Enter the number: "))
n = int(input("Enter the modulo: "))
result = additive_inverse(num, n)
if result is None:
    print("Additive inverse does not exist in modulo", n)
else:
    print(f"Additive inverse of {num} in modulo {n} is", result)
```

OUTPUT:

```
Enter the number: 4
Enter the modulo: 8
Additive inverse of 4 in modulo 8 is 4
```

7. Write a program which takes two numbers and display whether they are relatively prime or not.
Solution:

SOURCE CODE:

```
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
if (gcd(a,b)==1):
    print(f"{a} and {b} are relatively prime numbers.")
else:
    print(f"{a} and {b} are not relatively prime number.")
```

OUTPUT:

```
Enter first number:2
Enter second number:3
2 and 3 are relatively prime numbers.
```

8. Write a program to implement Extended Euclidean Algorithm. (Display the results of iterations in tabular format) .
Solution:

SOURCE CODE:

```
def extended_euclidean_algorithm(a, b):
    r1,r2 = a,b
    x1,x2,y1,y2=1,0,0,1
    q,x,y='-','-','-'
    print("{:2s}  {:2s}  {:2s}  {:2s}  {:2s}  {:2s}  {:2s}  {:2s}  {:2s}  {:2s} ".format("q","r1","r2",
"r","x1","x2","x","y1","y2","y"))
    print("-" * 38)
    while r2 != 0:
        q = r1// r2
        r=r1-q*r2
        x=x1-q*x2
        y=y1-q*y2
        print("{:2d}  {:2d}  {:2d}  {:2d}  {:2d}  {:2d}  {:2d}  {:2d}  {:2d}  {:2d}
".format(q,r1,r2,r,x1,x2,x,y1,y2,y))
        r1=r2
        r2=r
        x1=x2
        x2=x
        y1=y2
        y2=y
    print("{:2s}  {:2d}  {:2d}  {:2s}  {:2d}  {:2d}  {:2s}  {:2d}  {:2d}  {:2s} ".format( ' ',r1,r2,' ',x1,x2,'
',y1,y2,' '))
    print("-" * 38)
    print("gcd({:d}, {:d}) = {:d} = {:d}*{:d} + {:d}*{:d}".format(a, b, r1, a, x1, b, y1))
    return r1,x1,y1
a,b=input("Enter two numbers:").split()
extended_euclidean_algorithm(int(a),int(b))
```

OUTPUT:

```
Enter two numbers:4 8
q   r1  r2  r   x1  x2  x   y1  y2  y
--------------------------------------
 0   4   8   4   1   0   1   0   1   0
 2   8   4   0   0   1  -2   1   0   1
     4   0       1  -2       0   1
--------------------------------------
gcd(4, 8) = 4 = 4*1 + 8*0
```

9. WAP to compute multiplicative inverse in given modulo n using Extended Euclidean Algorithm.
Solution:

SOURCE CODE:

```
def extended_euclidean(a, b):
    x1, x2 = 1, 0
    y1, y2 = 0, 1
    while b != 0:
        q, r = divmod(a, b)
        a, b = b, r
        x1, x2 = x2, x1 - q * x2
        y1, y2 = y2, y1 - q * y2
    return (a,x1,y1)
```

```
def multiplicative_inverse(a, n):
    gcd, x, y = extended_euclidean(a, n)
    if gcd != 1:
        return("a is not invertible in modulo n")
    else:
        return (f"Multiplicative inverse of {a} modulo {n} is {x%n}")
a=input("Enter any integer to find mul inverse:")
n=input("Enter modulo:")
print(multiplicative_inverse(int(a),int(n)))
```

<u>OUTPUT:</u>

```
Enter any integer to find mul inverse:7
Enter modulo:9
Multiplicative inverse of 7 modulo 9 is 4
```

10. Write a program to implement Hill Cipher (Key matrix of size 2*2/ Encryption/ Decryption/ Input should be taken from user).
Solution:

<u>SOURCE CODE:</u>

```
import numpy as np
import math

# function to generate inverse of a 2x2 matrix
def inverse(mat):
    det = mat[0][0]*mat[1][1] - mat[0][1]*mat[1][0]
    inv = np.array([[mat[1][1], -mat[0][1]], [-mat[1][0], mat[0][0]]])
    inv = (1/det) * inv
    return inv

# function to pad message if its length is odd
def pad_message(msg):
    if len(msg) % 2 == 1:
        msg += 'x'
    return msg

# function to convert message to matrix
def msg_to_matrix(msg):
    msg = pad_message(msg)
    n = len(msg) // 2
    mat = np.zeros((n, 2))
    for i in range(n):
        mat[i][0] = ord(msg[2*i]) - 97
        mat[i][1] = ord(msg[2*i+1]) - 97
    return mat

# function to convert matrix to message
def matrix_to_msg(mat):
    n = mat.shape[0]
    msg = ''
    for i in range(n):
        msg += chr(math.floor(mat[i][0] + 97))
        msg += chr(math.floor(mat[i][1] + 97))
    return msg

# function to encrypt message using Hill Cipher
def hill_encrypt(msg, key):
    mat = msg_to_matrix(msg)
    enc_mat = np.dot(mat, key) % 26
    enc_msg = matrix_to_msg(enc_mat)
    return enc_msg

# function to decrypt message using Hill Cipher
def hill_decrypt(msg, key):
    mat = msg_to_matrix(msg)
    inv_key = inverse(key)
    dec_mat = np.dot(mat, inv_key) % 26
    dec_msg = matrix_to_msg(dec_mat)
    return dec_msg

# example usage
msg = input("Enter message to encrypt: ")
key = np.array([[3, 5], [2, 7]])  # 2x2 key matrix
enc_msg = hill_encrypt(msg, key)
print("Encrypted message:", enc_msg)
dec_msg = hill_decrypt(enc_msg, key)
print("Decrypted message:", dec_msg)
```

OUTPUT:

```
Enter message to encrypt: test
Encrypted message: ntop
Decrypted message: ezgx
```

11. WAP to demonstrate how output of S-Box (S1) is generated in DES.
Solution:

SOURCE CODE:

```
# Define the S-Box (S1) table
S1 = [
    [14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7],
    [ 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8],
    [ 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0],
    [15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13]
]

input_str = input("Enter a 6-bit input to S1 (e.g. 011011): ")

row = int(input_str[0] + input_str[-1], 2)
column = int(input_str[1:5], 2)

output = S1[row][column]

# Convert the output to binary and pad with leading zeros
output_str = format(output, '04b')

print("Output of S1:", output_str)
```

OUTPUT:

```
Enter a 6-bit input to S1 (e.g. 011011): 110011
Output of S1: 1011
```

12. Write a program to implement Robin Miller algorithm for primality test.
Solution:

SOURCE CODE:

```
import random

def power_mod(base, exponent, modulus):
    result = 1
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exponent = exponent // 2
    return result

def rabin_miller(n,):
    if n == 2 or n == 3:
        return True
    if n == 1 or n % 2 == 0:
        return False

    # Find r and d such that n-1 = 2^r * d
    d = n - 1
    r = 0
    while d % 2 == 0:
        d //= 2
        r += 1

    for i in range(r):
        a = random.randint(2, n-2)
        x = power_mod(a, d, n)
        if x == 1 or x == n-1:
            continue
        for j in range(r-1):
            x = power_mod(x, 2, n)
            if x == n-1:
                break
        else:
            return False
    return True
n=int(input("Enter any integer to check:"))
print(rabin_miller(n))
```

OUTPUT:

```
Enter any integer to check:6
False

Enter any integer to check:7
True
```

13. Write a program that takes any positive number and display the result after computing Totient value.
Solution:

SOURCE CODE:

```
def gcd(a, b):

    if (a == 0):
        return b
    return gcd(b % a, a)
```

```
def phi(n):

    result = 1
    for i in range(2, n):
        if (gcd(i, n) == 1):
            result+=1
    return result
n=int(input("Enter any positive integer:"))
print(f"Totient of {n} is :",phi(n))
```

OUTPUT:

```
Enter any positive integer:6
Totient of 6 is : 2
```

14. Write a program to compute primitive roots of given number.
Solution:

SOURCE CODE:

```
from math import gcd

def is_primitive_root(a, p):
    """
    Checks if a is a primitive root modulo p.
    """
    if gcd(a, p) != 1:
        return False
    phi = p - 1 # Euler's totient function
    for d in range(2, phi+1):
        if pow(a, d, p) == 1: # a^d mod p = 1
            if d == phi:
                return True # a is a primitive root
            else:
                return False # a is not a primitive root
    return False # should never reach this line

def primitive_roots(p):
    """
    Computes the primitive roots of p.
    """
    roots = []
    for a in range(2, p):
        if is_primitive_root(a, p):
            roots.append(a)
    return roots

# Example usage:
p = int(input("Enter any prime number:"))
print(f"Primitive roots of {p}: {primitive_roots(p)}")
```

<u>OUTPUT:</u>

```
Enter any prime number:7
Primitive roots of 7: [3, 5]
```

15. WAP to compute discrete log of given number (provided the modulo and primitive root).
Solution:

<u>SOURCE CODE:</u>

```
def discrete_log(g, h, p):
    m = int(p**0.5) + 1
    table = {pow(g, i, p): i for i in range(m)}
    gm = pow(g, m*(p-2), p)
    y = h
    for i in range(m):
        if y in table:
            return i*m + table[y]
        y = (y * gm) % p
    return None

# example usage
g = 3
h = 7
p = 17
x = discrete_log(g, h, p)
print("x =", x)
```

<u>OUTPUT:</u>

```
x = 11
```

16. WAP to implement Diffie-Helman Key Exchange Algorithm.
Solution:

<u>SOURCE CODE:</u>

```
from random import randint

def diffie_hellman():
    P = int(input("Enter any prime p(large prime) number:"))
    G = int(input(f"Enter value of g (primitive root of {P}):"))

    a =int(input("Enter value of a (i.e a<p):"))
    x = int(pow(G,a,P))
    b =int(input("Enter value of b (i.e b<p):"))
    y = int(pow(G,b,P))
    ka = int(pow(y,a,P))
    kb = int(pow(x,b,P))
    print('The Private Key a for Alice is :%d'%(a))
    print('The Private Key b for Bob is :%d'%(b))
    print('Secret key for the Alice is : %d'%(ka))
    print('Secret Key for the Bob is : %d'%(kb))
diffie_hellman()
```

<u>OUTPUT:</u>

```
Enter any prime p(large prime) number:7
Enter value of g (primitive root of 7):3
Enter value of a (i.e a<p):4
Enter value of b (i.e b<p):2
The Private Key a for Alice is :4
The Private Key b for Bob is :2
Secret key for the Alice is : 2
Secret Key for the Bob is : 2
```

17. WAP to implement RSA Algorithm (Encryption/Decryption/ Input Should be taken from user)
Solution:

<u>SOURCE CODE:</u>

```
import math

# find gcd
def gcd(a, b):
    while b != 0:
        t = a % b
        a = b
        b = t
    return a

# 2 random prime numbers
p = int(input("Enter first prime:"))
q = int(input("Enter second prime:"))
```

```
n = p * q # calculate n
phi = (p-1) * (q-1) # calculate phi

# public key
# e stands for encrypt
e = 2
# for checking that 1 < e < phi(n) and gcd(e, phi(n)) = 1; i.e., e and phi(n) are coprime.
while e < phi:
    track = gcd(e, phi)
    if track == 1:
        break
    else:
        e += 1

# private key
# d stands for decrypt
# choosing d such that it satisfies d * e = 1 mod phi
d1 = 1 / e
d = math.fmod(d1, phi)

message =int(input("Enter message in integer:"))
c = pow(message, e) # encrypt the message
m = pow(c, d)
c = math.fmod(c, n)
m = math.fmod(m, n)

print("Original Message = ", message)
print("p = ", p)
print("q = ", q)
print("n = pq = ", n)
print("phi = ", phi)
print("e = ", e)
print("d = ", d)
print("Encrypted message = ", c)
print("Decrypted message = ", m)
```

OUTPUT:

```
Enter first prime:5
Enter second prime:7
Enter message in integer:445
Original Message =  445
p =  5
q =  7
n = pq =  35
```

```
phi =  24
e =  5
d =  0.2
Encrypted message =  30.0
Decrypted message =  25.00000000000017
```

18. WAP to implement Elgamal Cryptographic System.
Solution:

SOURCE CODE:

```
import random

# function to generate a large prime number
def generate_prime():
    while True:
        p = random.randint(100000, 1000000)
        if is_prime(p):
            return p

# function to check if a number is prime
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

# function to calculate the modular inverse of a number
def mod_inverse(a, m):
    for i in range(1, m):
        if (a*i) % m == 1:
            return i
    return None

# function to generate the public and private keys for the ElGamal system
def generate_keys():
    # generate a large prime number
    p = generate_prime()
    # choose a random number g between 2 and p-2
    g = random.randint(2, p-2)
    # choose a random number x between 1 and p-2
    x = random.randint(1, p-2)
    # calculate y = g^x mod p
    y = pow(g, x, p)
    # return the public and private keys
    public_key = (p, g, y)
    private_key = x
    return public_key, private_key

# function to encrypt a message using the ElGamal system
def encrypt(message, public_key):
    # unpack the public key
    p, g, y = public_key
    # choose a random number k between 1 and p-2
    k = random.randint(1, p-2)
    # calculate a = g^k mod p and b = (y^k * message) mod p
    a = pow(g, k, p)
    b = (pow(y, k, p) * message) % p
    # return the ciphertext as a tuple (a, b)
    return (a, b)

# function to decrypt a ciphertext using the ElGamal system
def decrypt(ciphertext, public_key, private_key):
    # unpack the public key
    p, g, y = public_key
    # unpack the ciphertext as a tuple (a, b)
    a, b = ciphertext
    # calculate a^(p-1-x) mod p
    ax = pow(a, p-1-private_key, p)
    # calculate the plaintext message as (b * ax) mod p
    message = (b * ax) % p
    # return the plaintext message
    return message

# example usage
public_key, private_key = generate_keys()
message = 123456789
ciphertext = encrypt(message, public_key)
plaintext = decrypt(ciphertext, public_key, private_key)
print("Public key:", public_key)
print("Private key:", private_key)
print("Message:", message)
print("Ciphertext:", ciphertext)
print("Plaintext:", plaintext)
```

OUTPUT:

Public key: (482243, 120982, 425361)
Private key: 440447
Message: 123456789
Ciphertext: (132565, 461734)
Plaintext: 2581

19. Write a malicious logic code (Trojan Horse/Virus) program that performs some malicious works.
Solution:

SOURCE CODE:
A locker that creates a full-screen window and prevents the user from closing it:

```
# This is the locker

from tkinter import *

def locker():
    # Create an instance of tkinter frame
    win = Tk()

    # Set window size
    win.geometry("312x324")

    # Disable resizing
    win.resizable(0, 0)

    # Give title
    win.title("Locker")

    # Create a label with a message
    label = Label(win, text="You are locked!", font=("Arial", 20))
    label.pack(pady=20)

    # Override the close button
    win.protocol("WM_DELETE_WINDOW", lambda: None)

# Execute the function
locker()
```