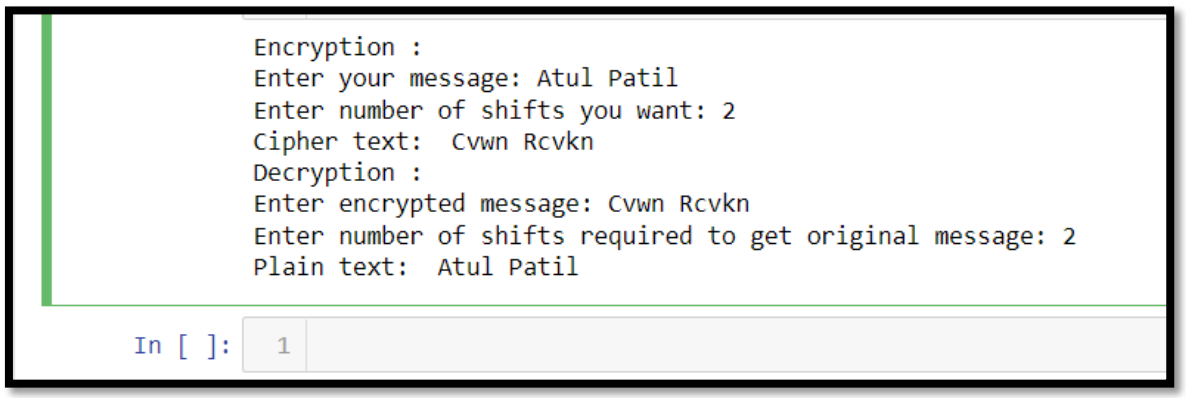
Practical-1

AIM: Implement Caesar Cipher Encryption-Decryption.

```
import math
def encrypt(message, move):
   str = ""
   for i in range(len(message)):
      ch = message[i]
      if ch.isupper():
         str += chr((ord(ch) + move - 65) % 26 + 65)
      elif ch.islower():
         str += chr((ord(ch) + move - 97) % 26 + 97)
      else:
         str += ch
   return str
def decrypt(message, move):
   destr = ""
   for i in range(len(message)):
      ch = message[i]
      if ch.isupper():
         destr += chr(int(math.fmod((ord(ch) - move - 90)%26 )) +90)
      elif ch.islower():
         destr += chr(int(math.fmod((ord(ch) - move - 122)%26)) +122)
       else:
         destr += ch
   return destr
print("Encryption : ")
simple_msg = input("Enter your message: ")
move = int(input("Enter number of shifts you want: "))
ct = encrypt(simple_msg, move)
print("Cipher text: " , ct)
print("Decryption : ")
```

```
enc_msg = input("Enter encrypted message: ")
move = int(input("Enter number of shifts required to get original message: "))
pt = decrypt(enc_msg, move)
print("Plain text: ", pt)
```

- **OUTPUT:**



Practical-2

AIM: Implement Monoalphabetic Cipher Encryption-Decryption.

```
import string
print(“ENCRYPTION”)
plaintext = input("Enter the plain text for encryption : \n")
key1 = input("Enter the key in lowercase for encryption : \n")
key2 = key1.upper()
ciphertext = ""
for c in plaintext:
   if c in string.ascii_lowercase:
      index = ord(c) - ord("a")
      ciphertext += key1[index]
   elif c in string.ascii_uppercase:
      index = ord(c) - ord("A")
      ciphertext += key2[index]
   else:
      ciphertext += c
print("\nPlaintext: " + plaintext) print("Ciphertext: " + ciphertext + "\n")
print("DECRYPTION")
#Decryption
ciphertext = input("Enter the ciphertext for decryption \n")
key1 = input("Enter the key in lowercase for decryption \n")
key2 = key1.upper()
plaintext = ""
for c in ciphertext:
   if c in string.ascii_lowercase:
      index = key1.find(c)
      plaintext += chr(index + ord("a"))
   elif c in string.ascii_uppercase:
      index = key2.find(c)
      plaintext += chr(index + ord("A"))
```

```
    else:
    plaintext += c print("\nCiphertext:+ ciphertext)
print("Plaintext: " + plaintext + "\n")
```

- **OUTPUT:**

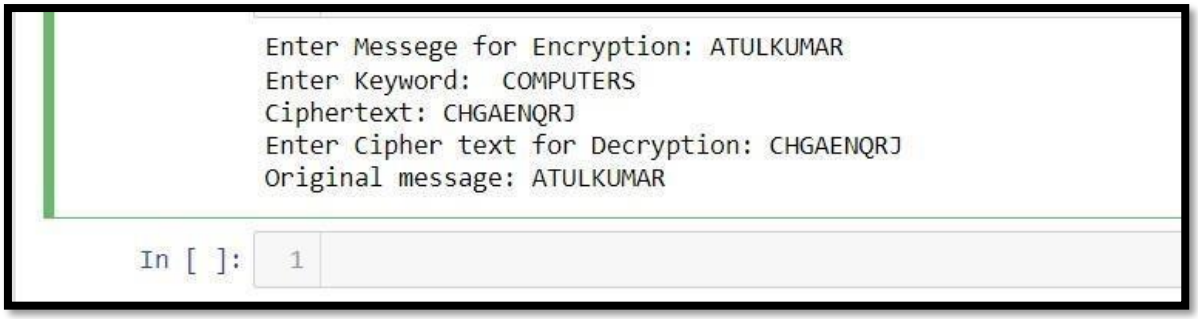
```
ENCRYPTION
Enter the plain text for encryption :
Atulkumar Patil
Enter the key in lowercase for encryption :
qazwsxedcrfvtgbyhnujmikolp

Plaintext: Atulkumar Patil
Ciphertext: Qjmvfmtqn Yqjcv

DECRYPTION
Enter the ciphertext for decryption
Qjmvfmtqn Yqjcv
Enter the key in lowercase for decryption
qazwsxedcrfvtgbyhnujmikolp

Ciphertext: Qjmvfmtqn Yqjcv
Plaintext: Atulkumar Patil

In [ ]: 1
```

Practical-3

AIM: Implement Polyalphabetic Cipher Encryption-Decryption.

```
def generateKey(string, key):
    key = list(key)
    if len(string) == len(key):
        return(key)
    else:
        for i in range(len(string) - len(key)):
        key.append(key[i % len(key)])
    return("" . join(key))
def encryption(string, key):
    cipher_text = []
    for i in range(len(string)):
        x = (ord(string[i]) +ord(key[i])) % 26
        x += ord('A') cipher_text.append(chr(x))
    return("" . join(cipher_text))
def decryption(cipher_text, key):
    orig_text = []
    for i in range(len(cipher_text)):
        x = (ord(cipher_text[i]) - ord(key[i]) + 26) % 26x += ord('A')
        orig_text.append(chr(x))
    return("" . join(orig_text))
# Driver code
if_name__== "__main__":
    string = input("Enter Messege for Encryption: ")
    keyword = input("Enter Keyword:  ")
    key = generateKey(string, keyword)
    print("Ciphertext:", encryption(string, key))
    cipher_text = input("Enter Cipher text for Decryption: ")
    print("Original message:", decryption(cipher_text, key))
```

- **OUTPUT:**


Enter Messege for Encryption: ATULKUMAR
Enter Keyword: COMPUTERS
Ciphertext: CHGAENQRJ
Enter Cipher text for Decryption: CHGAENQRJ
Original message: ATULKUMAR
In [ ]:
1

Practical-4

AIM: Implement Playfair Cipher Encryption-Decryption.

```
key=input("Enter key: ")
key=key.replace(" ", "")
key=key.upper()
def matrix(x,y,initial):
  return [[initial for i in range(x)] for j in range(y)]
result=list()
for c in key: #storing key
  if c not in result:
    if c=='J':
      result.append('I')
    else:
      result.append(c)
flag=0
for i in range(65,91): #storing other character
 if chr(i) not in result:
    if i==73 and chr(74) not in result:
      result.append("I")
      flag=1
    elif flag==0 and i==73 or i==74:
      pass
    else:
      result.append(chr(i))
k=0
my_matrix=matrix(5,5,0) #initialize matrix
for i in range(0,5): #creating matrix
  for j in range(0,5):
    my_matrix[i][j]=result[k]
    k+=1
```

```
def locindex(c): #get location of each character
    loc=list()
    if c=='J':
        c='I'
    for i ,j in enumerate(my_matrix):
        for k,l in enumerate(j):
            if c==l:
                loc.append(i)
                loc.append(k)return loc
#Encryption
def encrypt():
    msg=str(input("Enter message: "))
    msg=msg.upper()
    msg=msg.replace(" ", "")
    i=0
    for s in range(0,len(msg)+1,2):if
        s<len(msg)-1:
            if msg[s]==msg[s+1]:
                msg=msg[:s+1]+'X'+msg[s+1:]
    if len(msg)%2!=0:
        msg=msg[:]+'X'
    print("Cipher text:",end=' ')
    while i<len(msg):
        loc=list()
        loc=locindex(msg[i])
        loc1=list()
        loc1=locindex(msg[i+1])
        if loc[1]==loc1[1]:
print("{}{}".format(my_matrix[(loc[0]+1)%5][loc[1]],my_matrix[(loc1[0]+1)%5][loc1[1]]),
end='')
elif loc[0]==loc1[0]:
```

```
 print("{}{}".format(my_matrix[loc[0]][(loc[1]+1)%5],my_matrix[loc1[0]][(loc1[1]+1)%5]), end='')
else:
        print("{}{}".format(my_matrix[loc[0]][loc1[1]],my_matrix[loc1[0]][loc[1]]),end='')
        i=i+2
 #decryption
 def decrypt():
    msg=str(input("Enter Cipher text:"))msg=msg.upper() msg=msg.replace(" ", "")
    print("Plain text:",end=' ')
    i=0
    while i<len(msg): loc=list()
       loc=locindex(msg[i])
       loc1=list()
       loc1=locindex(msg[i+1])
       if loc[1]==loc1[1]:
          print("{}{}".format(my_matrix[(loc[0]-1)%5][loc[1]],my_matrix[(loc1[0]-
          1)%5][loc1[1]]),end='')
       elif loc[0]==loc1[0]:
          print("{}{}".format(my_matrix[loc[0]][(loc[1]-1)%5],my_matrix[loc1[0]][(loc1[1]-
          1)%5]),end='')
       else:
          print("{}{}".format(my_matrix[loc[0]][loc1[1]],my_matrix[loc1[0]][loc[1]]),end='')
          i=i+2
 while(1):
    choice = input("\n\n 1.Encryption \n 2.Decryption: \n 3.Exit \n")
    if choice=='1':
       encrypt()
    elif choice=='2':
       decrypt()
    elif choice=='3':
       break
    else:
       print("Invalid Input!!")
```

- **OUTPUT:**

```
Enter key: ATULKUMAR


 1.Encryption
 2.Decryption:
 3.Exit
1
Enter message: TECHNOLOGY
Cipher text: AFHQOPTQHX

 1.Encryption
 2.Decryption:
 3.Exit
2
Enter Cipher text:AFHQOPTQHX
Plain text: TECHNOLOGY

 1.Encryption
 2.Decryption:
 3.Exit
3
```

```
In [ ]: 1
```

Practical-5

AIM: Implement columnar transposition Cipher Encryption Decryption.

```
import math
key = input("Enter Key: ")
# Encryption
def encryptMessage(msg):
   cipher = ""
   # track key indices
   k_indx = 0
   msg_len = float(len(msg))
   msg_lst = list(msg)
   key_lst = sorted(list(key))
   # calculate column of the matrix
   col = len(key)
   # calculate maximum row of the matrix
   row = int(math.ceil(msg_len / col))
   # add the padding character '_' in empty#
   the empty cell of the matix
   fill_null = int((row * col) - msg_len)
   msg_lst.extend("_" * fill_null)
   # create Matrix and insert message and#
   padding characters row-wise
   matrix = [msg_lst[i : i + col] for i in range(0, len(msg_lst), col)]
   # read matrix column-wise using key
   for _ in range(col):
      curr_idx = key.index(key_lst[k_indx])
      cipher += "".join([row[curr_idx] for row in matrix])
      k_indx += 1
   return cipher
## Decryption
```

```
def decryptMessage(cipher):
    msg = ""
    k_indx = 0
    # track msg indices
    msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)
    col = len(key)
    row = int(math.ceil(msg_len / col))
    key_lst = sorted(list(key))
    dec_cipher = []
    for _  in range(row):
        dec_cipher += [[None] * col]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        for j in range(row):
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
            msg_indx += 1
        k_indx += 1
    try:
        msg = "".join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot", "handle repeating words.")
        null_count = msg.count("_")
    if null_count > 0:
        return msg[:-null_count]
    return msg
msg = input("Enter message for encryption: ")
cipher = encryptMessage(msg) print("Encrypted Message: {}".format(cipher))
cipher = input("Enter cipher text for decryption: ")
print("Decryped Message: {}".format(decryptMessage(cipher)))
```

➢ **OUTPUT**

```
Enter Key: 23451
Enter message for encryption: ATUL SATISHBHAI PATIL
Encrypted Message:  SII_ASH LTABP_UTHA_LIAT_
Decryped Message: ATUL SATISHBHAI PATIL

In [ ]:  1
```

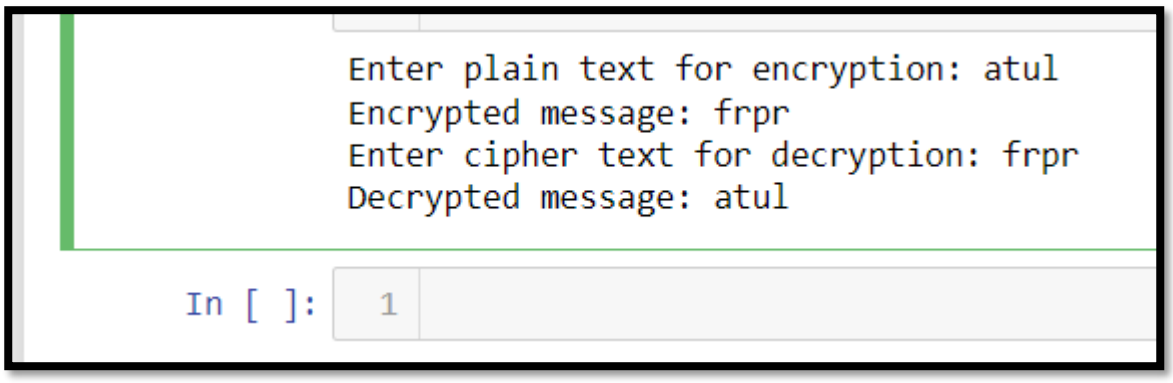
Practical-6

AIM: Implement Hill Cipher Encryption-Decryption.

```
import numpy as np
from egcd import egcd
alphabet = "abcdefghijklmnopqrstuvwxyz"
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))
def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix))) # Step 1)
    det_inv = egcd(det, modulus)[1] % modulus  # Step 2)
    matrix_modulus_inv = (
        det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus) # Step 3)
    return matrix_modulus_inv
    def encrypt(message, K):
        encrypted = "" message_in_numbers = []for letter in message:
        message_in_numbers.append(letter_to_index[letter])
        split_P = [message_in_numbers[i : i + int(K.shape[0])]
        for i in range(0, len(message_in_numbers), int(K.shape[0]))]
        for P in split_P:
        P = np.transpose(np.asarray(P))[:, np.newaxis]
        while P.shape[0] != K.shape[0]:
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]
            numbers = np.dot(K, P) % len(alphabet)
        n = numbers.shape[0] # length of encrypted message (in numbers)for idx
        in range(n):
            number = int(numbers[idx, 0])
            encrypted += index_to_letter[number]
    return encrypted
    def decrypt(cipher, Kinv):
    decrypted = ""
```

```
    cipher_in_numbers = []
    for letter in cipher:
        cipher_in_numbers.append(letter_to_index[letter]) split_C = [
        cipher_in_numbers[i : i + int(Kinv.shape[0])]
        for i in range(0, len(cipher_in_numbers), int(Kinv.shape[0]))
    ]
    for C in split_C:
        C = np.transpose(np.asarray(C))[:, np.newaxis]
        numbers = np.dot(Kinv, C) % len(alphabet)
        n = numbers.shape[0]
        for idx in range(n):
            number = int(numbers[idx, 0])
            decrypted += index_to_letter[number]
    return decrypted
def main():
    message = input("Enter plain text for encryption: ")
    K = np.matrix([[3, 3], [2, 5]])
    Kinv = matrix_mod_inv(K, len(alphabet))
    print("Encrypted message: " + encrypt(message, K))
    encrypted_message = input("Enter cipher text for decryption: ")
    print("Decrypted message: " + decrypt(encrypted_message, Kinv))
main()
```

- **OUTPUT:**



Practical-7

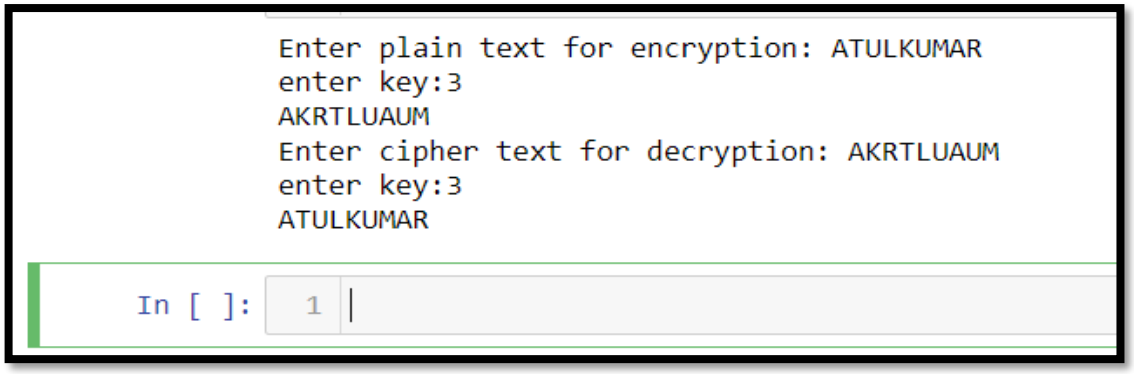
AIM: Implement Rail Fence Cipher Encryption-Decryption.

```
def encryption(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:
            result = []
            row -= 1
    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 decryption(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
            # place the marker
            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_":
        print(encryption(input("Enter plain text for encryption: "), int(input("enterkey: "))))
```

```
print(decryption(input("Enter cipher text for decryption: "), int(input("enterkey:"))))
```

- **OUTPUT**



Practical-8

AIM: Write a Program For an Encryption Algorithm.

```
def encrypt(text,s):
result = ""
for i in range(len(text)):
char = text[i]
if (char.isupper()):
result += chr((ord(char) + s-65) % 26 + 65)
else:
result += chr((ord(char) + s - 97) % 26 + 97)
return result
text = input("Enter a String: ")
s = int(input("Enter key:"))
print ("Text : " + text)
print ("Shift : " + str(s))
print ("Cipher Text is: " + encrypt(text,s))
```

- **OUTPUT:**

```
Enter a String: hello
Enter key:4
Text : hello
Shift : 4
Cipher Text is: lipps


...Program finished with exit code 0
Press ENTER to exit console.
```

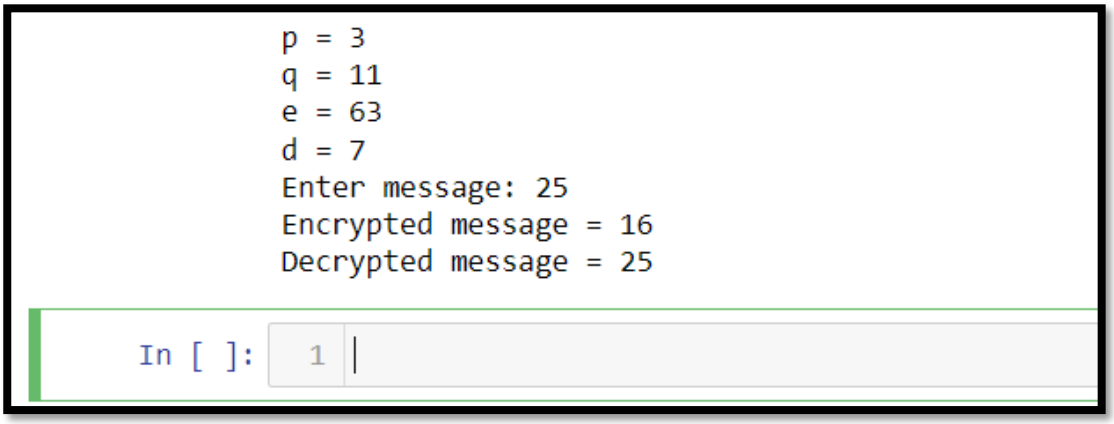
Practical-9

AIM: Implement RSA Encryption-Decryption Algorithm.

```
from random import randint
import math
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        return None
    else:
        return x % m
if __name__ == "__main__":
    p = 3
    q = 11
    print("p = %d" % p)
    print("q = %d" % q)
    n = p * q
    n1 = (p - 1) * (q - 1)
    r = randint(2, 100)
    while True:
        if gcd(r, n1) == 1:
        break
```

```
        else:
        r += 1
e = r
print("e = %d" % e)
d = modinv(e, n1)
print("d = %d" % d)
m = input("Enter message: ")
c = (int(m) ** e) % n
print("Encrypted message = %d" % c)
m1 = (c**d) % n
print("Decrypted message = %d" % m1)
```

- **OUTPUT:**



Practical-10

AIM: Write a Program to generate SHA-1 Hash.

```
from random import randint
if __name__ == '__main__':
P = int(input("Enter P:"))
G = int(input("Enter G:"))
a = 4
print('The Private Key a for Sender is :%d'%(a))
x = int(pow(G,a,P))
b = 3
print('The Private Key b for Receiver is :%d'%(b))
y = int(pow(G,b,P))
ka = int(pow(y,a,P))
kb = int(pow(x,b,P))
print('Secret key for the Sender is : %d'%(ka))
print('Secret Key for the Receiver is : %d'%(kb))
```

- **OUTPUT:**