**NAME: THAKKAR MALAY TUSHARBHAI**

**ENROLLMENT NO.:20012011169**

**DEPARTMENT: B.Tech. (CE)**

**Batch: CEIT-B\_6AB4**

**SUBJECT: 2CEIT6PE3: CRYPTOGRAPHY AND NETWORK SECURITY**

**COLLEGE: U. V. PATEL COLLEGE OF ENGINEERING**

**Practical-1**

1. **Write a program to perform encryption and decryption using Caesar cipher algorithm. Encryption procedure: C=E(P)=(P+K) mod 26**

**Decryption Procedure: P=D(C)=(C-K) mod 26**

* **CODE:**

#include <iostream>;

using namespace std;

string cipher\_encryption(string text,int key)

{

    string result = "";

    for (int i = 0; i < text.length(); i++) {

        if (isupper(text[i]))

            result += char(int(text[i] + key - 65) % 26 + 65);

        else

            result += char(int(text[i] + key - 97) % 26 + 97);

    }

    return result;

}

string cipher\_decryption(string text,int key)

{

   char ch;

      for(int i = 0; text[i] != '\0'; ++i) {

         ch = text[i];

         //decrypt for lowercase letter

         if(ch >= 'a' && ch <= 'z') {

            ch = ch - key;

            if(ch < 'a'){

               ch = ch + 'z' - 'a' + 1;

            }

            text[i] = ch;

         }

         //decrypt for uppercase letter

         else if(ch >= 'A' && ch <= 'Z') {

            ch = ch - key;

            if(ch < 'A') {

               ch = ch + 'Z' - 'A' + 1;

            }

            text[i] = ch;

         }

      }

      return text;

}

int main(){

    string simpletext="";

    string result\_simpletext="";

    string ciphertext="";

    string result\_ciphertext="";

    int key=0;

    cout<<"Enter simple text: ";

    cin>>simpletext;

     cout<<"Enter cipher text: ";

    cin>>ciphertext;

    cout<<"Enter key: ";

    cin>>key;

    result\_ciphertext=cipher\_encryption(simpletext,key);

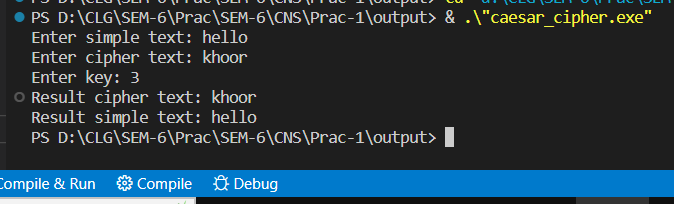
    result\_simpletext=cipher\_decryption(ciphertext,key);

    cout<<"Result cipher text: "<<result\_ciphertext<<endl;

    cout<<"Result simple text: "<<result\_simpletext<<endl;

}

* **OUTPUT:**



**2. Write a program to perform encryption and decryption using Modified Caesar cipher algorithm.**

* **CODE:**

def encryption(planeText):

    encryptData = ""

    for i in range(len(planeText)):

        char = planeText[i]

        if char.isupper():

            if i % 2 == 0:

              encryptData += chr((ord(char) + 1 - 65) % 26 + 65)

            else:

                encryptData += chr((ord(char) - 1 - 65) % 26 + 65)

        elif char.islower():

            if i % 2 == 0:

                encryptData += chr((ord(char) + 1 - 97) % 26 + 97)

            else:

                encryptData += chr((ord(char) - 1 - 97) % 26 + 97)

        else:

            encryptData += char

    return encryptData

def decryption(cipherText):

    decryptData = ""

    for i in range(len(cipherText)):

        char = cipherText[i]

        if char.isupper():

            if i % 2 == 0:

                decryptData += chr((ord(char) - 1 - 65) % 26 + 65)

            else:

                decryptData += chr((ord(char) + 1 - 65) % 26 + 65)

        elif char.islower():

            if i % 2 == 0:

                decryptData += chr((ord(char) - 1 - 97) % 26 + 97)

            else:

                decryptData += chr((ord(char) + 1 - 97) % 26 + 97)

        else:

            decryptData += char

    return decryptData

planeText = input("Plain text: ")

ct = encryption(planeText)

print("Cipher text:", ct)

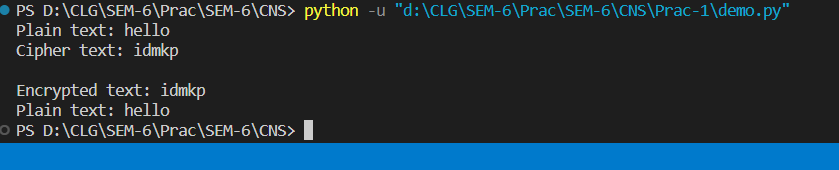
print()

cipherText = input("Encrypted text: ")

dt = decryption(cipherText)

print("Plain text:", dt)

**OUTPUT:**



**Practical-2**

**AIM: Write a program to find plain text messages and key information**

**corresponds to following cipher text messages using brute-force technique on Caesar cipher.**

* **Code:**

#finding plain text messages and key information of

#cipher text messages using brute-force technique on Caesar cipher

def decryption(text, key):

    decrypted\_text = ""

    for char in text:

        if char.isupper():

            decrypted\_text += chr((ord(char) - key - 65) % 26 + 65)

        else:

            decrypted\_text += chr((ord(char) - key - 97) % 26 + 97)

    return decrypted\_text

encrypted\_text = input("Enter encrypted text : ")

for key in range(26):

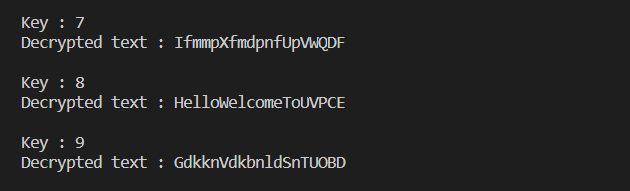
    decrypted\_text = decryption(encrypted\_text, key)

    print("Key : " + str(key))

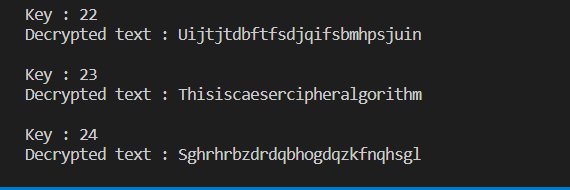
    print("Decrypted text : " + decrypted\_text + "\n")

* **Output:**

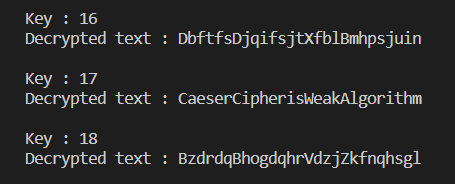
**1.** PmttwEmtkwumBwCDXKM



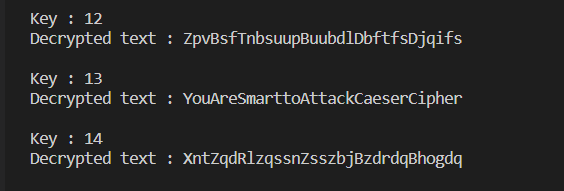
**2.** Qefpfpzxbpbozfmeboxidlofqej



**3.** TrvjviTzgyvizjNvrbRcxfizkyd



**4.** LbhNerFzneggbNggnpxPnrfrePvcure



**Practical-3**

**AIM: -** Write a program to perform encryption and decryption using Mono-alphabetic Cipher Technique.

* **CODE:**

#encryption and decryption using Mono-alphabetic Cipher Technique

mono\_alpha = {

    'A':'Z','B':'Y','C':'X','D':'W','E':'V','F':'U','G':'T','H':'S','I':'R','J':'Q','K':'P','L':'O','M':'N','N':'M','O':'L','P':'K','Q':'J','R':'I','S':'H','T':'G','U':'F','V':'E','W':'D','X':'C','Y':'B','Z':'A'

}

def encryption(text):

    encrypted\_text = ""

    for char in text:

        encrypted\_text += mono\_alpha[char]

    return encrypted\_text

def decryption(text):

    decrypted\_text = ""

    for char in text:

        decrypted\_text += list(mono\_alpha.keys())[list(mono\_alpha.values()).index(char)]

    return decrypted\_text

text = input("Enter the text : ")

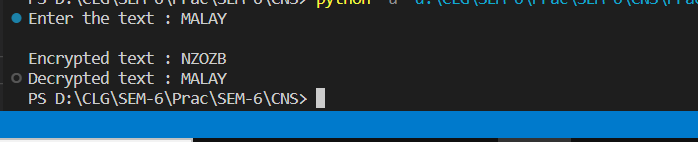
encrypted\_text = encryption(text)

print("\nEncrypted text : " + encrypted\_text)

decrypted\_text = decryption(encrypted\_text)

print("Decrypted text : " + decrypted\_text)

* **OUTPUT:**



**Practical-4**

**AIM: - Write a program to perform encryption and decryption using Polyalphabetic Cipher (Vigenere Cipher) Technique.**

* **CODE:**

def encryption(plain\_text, key\_word):

    j = 0

    encrypted\_text = ""

    for p in plain\_text:

        cipher\_value = (ord(p)-65 + ord(key\_word[j])-65) % 26

        cipher\_text = chr(cipher\_value + 65)

        encrypted\_text += cipher\_text

        if(j == len(key\_word)-1):

            j = 0

        else:

            j += 1

    return encrypted\_text

def decryption(encrypted\_text, key\_word):

    j = 0

    decrypted\_text = ""

    for p in encrypted\_text:

        plain\_text\_value = (ord(p)-65 - ord(key\_word[j])-65) % 26

        plain\_text = chr(plain\_text\_value + 65)

        decrypted\_text += plain\_text

        if(j == len(key\_word)-1):

            j = 0

        else:

            j += 1

    return decrypted\_text

plain\_text =input("Enter Plaintext: ")

key\_word = input("Enter Keyword: ")

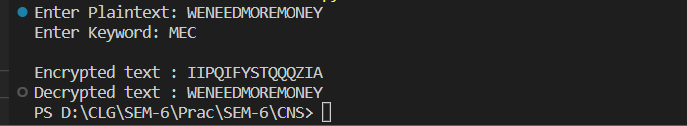
encrypted\_text = encryption(plain\_text, key\_word)

print("\nEncrypted text : " + encrypted\_text)

decrypted\_text = decryption(encrypted\_text, key\_word)

print("Decrypted text : " + decrypted\_text)

* **OUTPUT:**

****

**Practical-5**

**AIM: - Write a program to perform encryption and decryption using Play-**

**fair Cipher Technique.**

**CODE:**

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 i in key:  # storing key

    if i not in result:

        if i == "J":

            result.append("I")

        else:

            result.append(i)

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):  # making matrix

    for j in range(0, 5):

        my\_matrix[i][j] = result[k]

        k += 1

print("\nPLAY-FAIR KEYWORD MATRIX:")

for i in range(0, 5):

    for j in range(0, 5):

        print(my\_matrix[i][j], end=" ")

    print()

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

def encrypt():  # Encryption

    msg = str(input("Enter the plaintext: "))

    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("\nCipher 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

    print()

def decrypt():  # decryption

    msg = str(input("Enter Cipher text: "))

    msg = msg.upper()

    msg = msg.replace(" ", "")

    decrypted\_msg = ""

    i = 0

    while i < len(msg):

        loc = list()

        loc = locindex(msg[i])

        loc1 = list()

        loc1 = locindex(msg[i + 1])

        if loc[1] == loc1[1]:

            decrypted\_msg += "{}{}".format(my\_matrix[(loc[0] - 1) %

                                                     5][loc[1]], my\_matrix[(loc1[0] - 1) % 5][loc1[1]]) + " "

        elif loc[0] == loc1[0]:

            decrypted\_msg += "{}{}".format(my\_matrix[loc[0]][(loc[1] - 1) %

                                                             5], my\_matrix[loc1[0]][(loc1[1] - 1) % 5]) + " "

        else:

            decrypted\_msg += "{}{}".format(my\_matrix[loc[0]]

                                           [loc1[1]], my\_matrix[loc1[0]][loc[1]]) + " "

            i = i + 2

        print("\nPlain text:", decrypted\_msg)

        decrypted\_msg = decrypted\_msg.replace('X', '').replace(" ", "")

        print("\nPlain text after romoving all X and spces:", decrypted\_msg)

|  |  |
| --- | --- |
| while 1:      choice = int(input("\nChoose one: \n1. Encryption \n2. Decryption \n3. Exit\n\n"))      if choice == 1:           encrypt()      elif choice == 2:          decrypt()      elif choice == 3:          break      else:          print("Please, choose correct choice.") | **OUTPUT:** |

**Practical-6**

**AIM:** Write a program to perform encryption and decryption using Rail-Fence Cipher

Technique.

* **CODE:**

import re

def cipher\_encryption():

    msg = input("Enter message: ")

    rails = int(input("Enter number of rails: "))

    msg = msg.replace(" ", "")

    railMatrix = []

    for i in range(rails):

        railMatrix.append([])

    for row in range(rails):

        for column in range(len(msg)):

            railMatrix[row].append('.')

    row = 0

    check = 0

    for i in range(len(msg)):

        if check == 0:

            railMatrix[row][i] = msg[i]

            row += 1

            if row == rails:

                check = 1

                row -= 1

        elif check == 1:

            row -= 1

            railMatrix[row][i] = msg[i]

            if row == 0:

                check = 0

                row = 1

    encryp\_text = ""

    for i in range(rails):

        for j in range(len(msg)):

            encryp\_text += railMatrix[i][j]

    encryp\_text = re.sub(r"\.", " ", encryp\_text)

    print("Encrypted Text: {}".format(encryp\_text))

    print()

def cipher\_decryption():

    msg = input("Enter message: ")

    rails = int(input("Enter number of rails: "))

    msg = msg.replace(" ", "")

    railMatrix = []

    for i in range(rails):

        railMatrix.append([])

    for row in range(rails):

        for column in range(len(msg)):

            railMatrix[row].append('.')

    row = 0

    check = 0

    for i in range(len(msg)):

        if check == 0:

            railMatrix[row][i] = msg[i]

            row += 1

            if row == rails:

                check = 1

                row -= 1

        elif check == 1:

            row -= 1

            railMatrix[row][i] = msg[i]

            if row == 0:

                check = 0

                row = 1

    ordr = 0

    for i in range(rails):

        for j in range(len(msg)):

            temp = railMatrix[i][j]

            if re.search("\\.", temp):

                continue

            else:

                railMatrix[i][j] = msg[ordr]

                ordr += 1

    for i in railMatrix:

        for column in i:

            print(column, end="")

        print("\n")

    check = 0

    row = 0

    decryp\_text = ""

    for i in range(len(msg)):

        if check == 0:

            decryp\_text += railMatrix[row][i]

            row += 1

            if row == rails:

                check = 1

                row -= 1

        elif check == 1:

            row -= 1

            decryp\_text += railMatrix[row][i]

            if row == 0:

                check = 0

                row = 1

    decryp\_text = re.sub(r"\.", " ", decryp\_text)

    print("Decrypted Text: {}".format(decryp\_text))

    print()

def main():

    while 1:

        choice = int(

            input("1. Encryption\n2. Decryption]\n3. Exit\nChoose(1,2,3): "))

        if choice == 1:

            print("\n---Encryption---\n")

            cipher\_encryption()

        elif choice == 2:

            print("\n---Decryption---\n")

            cipher\_decryption()

        elif choice == 3:

            break

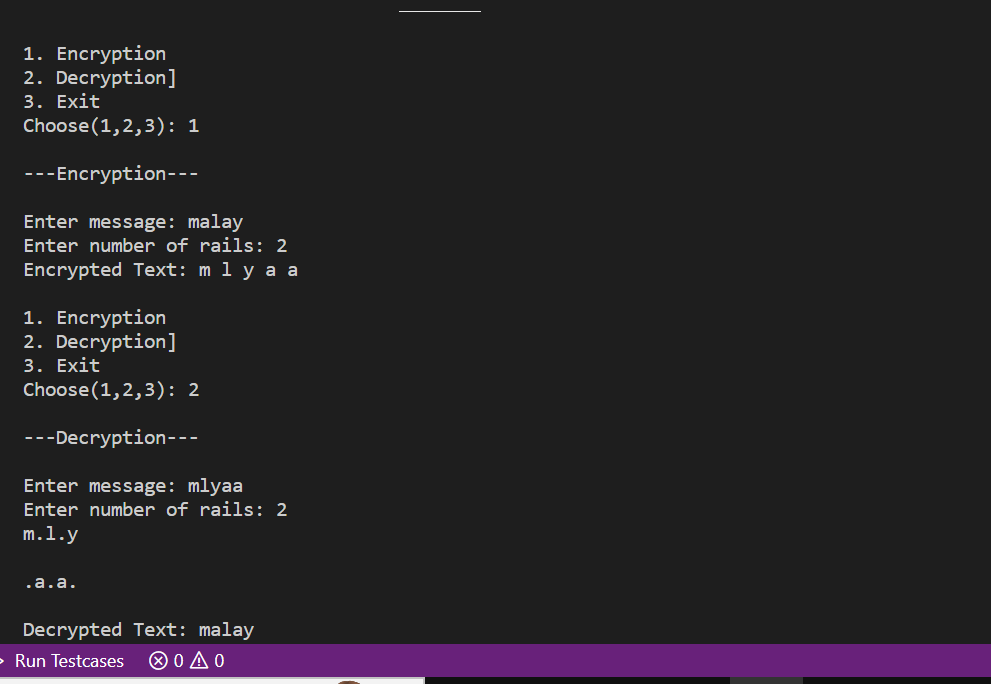
        else:

            print("\nInvalid Choice.\n")

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

    main()

* **OUTPUT:**

****

**Practical-7**

**AIM: Write a program to perform encryption and decryption using Hill**

**Cipher algorithm**

* **Encryption (CODE)**

import numpy as np

import string

import random

# Define variables

dimension = 3 # Your N

key = np.matrix([[6, 24, 1], [13, 16, 10], [20, 17, 15]]) # Your key

message = 'HILLCRYPT' # Your message

# Generate the alphabet

alphabet = string.ascii\_uppercase

# Encrypted message

encryptedMessage = ""

# Group message in vectors and generate crypted message

for index, i in enumerate(message):

    values = []

    # Make bloc of N values

    if index % dimension == 0:

        for j in range(0, dimension):

            if(index + j < len(message)):

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

            else:

                values.append([random.randint(0,25)])

        # Generate vectors and work with them

        vector = np.matrix(values)

        vector = key \* vector

        vector %= 26

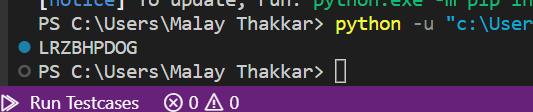
        for j in range(0, dimension):

            encryptedMessage += alphabet[vector.item(j)]

# Show the result

print(encryptedMessage)

* **OUTPUT:**

****

* **Encryption (CODE):**

# Decryption:

import numpy as np

from sympy import Matrix

import string

# Define variables

dimension = 3 # Your N

key = np.matrix([[6, 24, 1], [13, 16, 10], [20, 17, 15]]) # Your key

message = 'LRZBHPDOG' # You message

alphabet = string.ascii\_uppercase # Generate the alphabet

# Encrypted message

decryptedMessage = ""

# Get the decrypt key

key = Matrix(key)

key = key.inv\_mod(26)

key = key.tolist()

# Group message in vectors and generate crypted message

for index, i in enumerate(message):

    values = []

    # Create the N blocs

    if index % dimension == 0:

        for j in range(0, dimension):

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

        # Create the vectors and work with them

        vector = np.matrix(values)

        vector = key \* vector

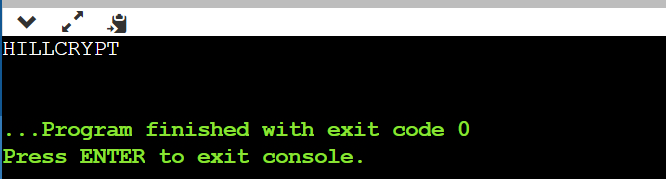
        vector %= 26

        for j in range(0, dimension):

            decryptedMessage += alphabet[vector.item(j)]

print(decryptedMessage) # Show the result

* **OUTPUT:**

****

**Practical-8**

* **AIM:** **Implement columnar transposition cipher encryption and**

**decryption**

* **CODE:**
* **Encryption:**

import math

def row(s,key):

    # to remove repeated alphabets in key

    temp=[]

    for i in key:

        if i not in temp:

            temp.append(i)

    k=""

    for i in temp:

        k+=i

    print("The key used for encryption is: ",k)

    # ceil is used to adjust the count of

    # rows according to length of message

    b=math.ceil(len(s)/len(k))

    # if b is less than length of key, then it will not form square matrix when

    # length of meessage not equal to rowsize\*columnsize of square matrix

    if(b<len(k)):

        b=b+(len(k)-b)

    # if b is greater than length of key, then it will not from a

    # square matrix, but if less then length of key, we have to add padding

    arr=[['\_' for i in range(len(k))]

         for j in range(b)]

    i=0

    j=0

    # arranging the message into matrix

    for h in range(len(s)):

        arr[i][j]=s[h]

        j+=1

        if(j>len(k)-1):

            j=0

            i+=1

    print("The message matrix is: ")

    for i in arr:

        print(i)

    cipher\_text=""

    # To get indices as the key numbers instead of alphabets in the key, according

    # to algorithm, for appending the elementsof matrix formed earlier, column wise.

    kk=sorted(k)

    for i in kk:

        # gives the column index

        h=k.index(i)

        for j in range(len(arr)):

            cipher\_text+=arr[j][h]

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

msg=input("Enter the message: ")

key=input("Enter the key in alphabets: ")

row(msg,key)

* **Decryption:**

import math

def row(s,key):

    # to remove repeated alphabets in key

    temp=[]

    for i in key:

        if i not in temp:

            temp.append(i)

    k=""

    for i in temp:

        k+=i

    print("The key used for encryption is: ",k)

    arr=[['' for i in range(len(k))]

         for j in range(int(len(s)/len(k)))]

    # To get indices as the key numbers instead of alphabets in the key, according

    # to algorithm, for appending the elementsof matrix formed earlier, column wise.

    kk=sorted(k)

    d=0

    # arranging the cipher message into matrix

    # to get the same matrix as in encryption

    for i in kk:

        h=k.index(i)

        for j in range(len(k)):

            arr[j][h]=s[d]

            d+=1

    print("The message matrix is: ")

    for i in arr:

        print(i)

    # the plain text

    plain\_text=""

    for i in arr:

        for j in i:

            plain\_text+=j

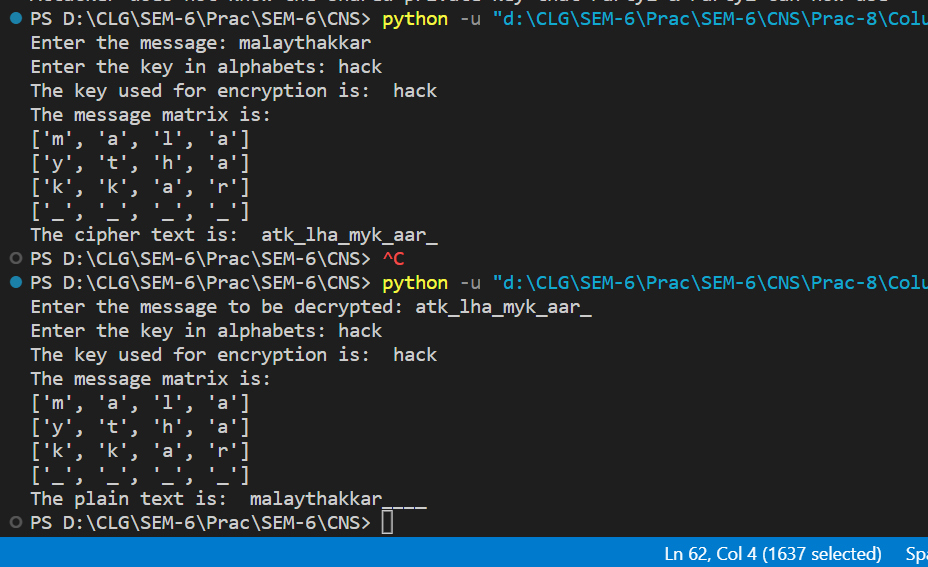
    print("The plain text is: ",plain\_text)

msg=input("Enter the message to be decrypted: ")

key=input("Enter the key in alphabets: ")

row(msg,key)

* **OUTPUT:**

****

**Practical-9**

**AIM: Implementation of diffie-hellman ford.**

* **CODE:**

print ("Both parties agree to a single prime")

prime=int(input("Enter the prime number to be considered: "))

# Primitive root to be used use

print ("Both must agree with single primitive root to use")

root=int(input("Enter the primitive root: "))

# Party1 chooses a secret number

alicesecret=int(input("Enter a secret number for Party1: "))

# Party2 chooses a secret number (bs)

bobsecret=int(input("Enter a secret number for Party2: "))

print("\n")

# Party1 public key A=(root^alicesecret)\*mod(prime)

print ("Party1's  public key -> A = root^alicesecre\*mod(prime))")

alicepublic=(root\*\*alicesecret)%prime

print ("Party1 public key is: ",alicepublic, "\n")

# Party2 public key B=(root^bobsecret)\*mod(prime)

print ("Party2's public key -> B = root^bobsecret\*)mod(prime))")

bobpublic=(root\*\*bobsecret)%prime

print ("Party2 public key is", bobpublic, "\n")

# Party1 and Party2 exchange their public keys

# Eve(attacker) nows both parties public keys

# Party1 now calculates the shared key K:

# K = B^(alicesecret)\*mod(prime)

print ("Party1 calculates the shared key as K=B^alicesecret\*(mod(prime))")

alicekey=(bobpublic\*\*alicesecret)%prime

print ("Party1 calculates the shared key and results: ",alicekey, "\n")

# Party2 calculates the shared key K:

# K = A^(bobsecret)\*mod(prime)

print ("Party2 calculates the shared key as K = A^bobsecret\*(mod(prime))")

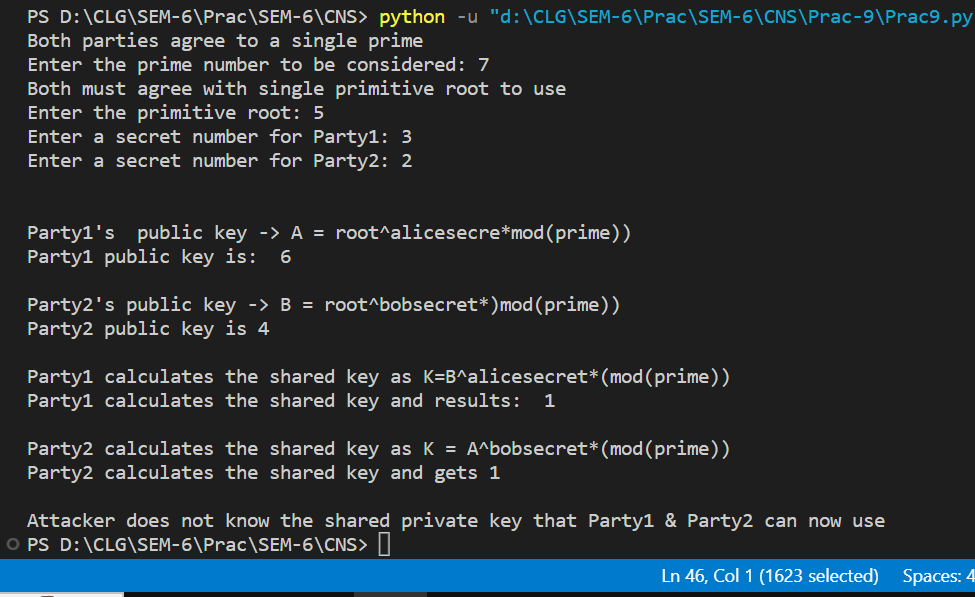
bobkey =(alicepublic\*\*bobsecret)%prime

print ("Party2 calculates the shared key and gets", bobkey, "\n")

#Both Alice and Bob now share a key which Eve cannot calculate

print ("Attacker does not know the shared private key that Party1 & Party2 can now use")

* **OUTPUT:**

****

**Practical-10**

**AIM: Implications of AES Algorithms.**

* **CODE:**

import math

import string

main=string.ascii\_lowercase

# Pseudo code for GCD is

'''

function gcd(a, b)

    if b = 0

        return a

    else

        return gcd(b, a mod b)

'''

# Naive method finding the multiplicative inverse of two numbers

def multiplicative\_inverse(a, m):

    a=a%m;

    for x in range(1,m) :

        if((a\*x)%m==1) :

            return x

    return 1

# Fast modular exponentiation function (can be used when

# something is very large for calculation modular)

'''

def get\_mod\_expo(base,exponent,modulus):

    result=1

    while exponent:

        d=exponent%2

        exponent=exponent//2

        if d:

            result=result\*base%modulus

        base=base\*base%modulus

    return result

'''

# Function to generate a public and private key pair

def generate\_keypair(p, q):

    n=p\*q

    print("Value of n: ",n)

    # Phi is the Euler's totient of n

    phi = (p-1)\*(q-1)

    print("Value of phi(n): ", phi)

    # Choose an integer e such that e and phi(n) are co-prime

    # e = random.randrange(1, phi) for random pick

    print("Enter e such that is co-prime to ", phi,": ")

    e=int(input())

    # Using Euclid's Algorithm to verify that e and phi(n) are co-prime

    # THe built in function gcd helps with the same

    g=math.gcd(e,phi)

    while(g!=1):

        print("The number you entered is not co-prime")

        e=int(input())

        g=math.gcd(e,phi)

    print("Value of exponent(e) entered is: ", e)

    # To generate the private key

    d = multiplicative\_inverse(e, phi)

    # We can use Extended Euclidean Algorithm because

    # we know that e and phi are coprimes

    # Public key is (e, n) and private key is (d, n)

    return (e,n),(d,n)

# Function to Encrypt the message

def encrypt(public\_key, to\_encrypt):

    key, n = public\_key

    # we can also use fast modular exponentiation here

    cipher=pow(to\_encrypt,key)%n

    return cipher

# Function to Decrypt the message

def decrypt(private\_key, to\_decrypt):

    key, n = private\_key

    # we can also use fast modular exponentiation here

    decrypted=pow(to\_decrypt,key)%n

    return decrypted

# Main Program

# primes of 8 bits in length in binary

p=int(input("Enter prime p: "))

q=int(input("Enter prime q (!=p): "))

# to make sure that p not equal to q while generating randomly

while(p==q):

    p=int(input("Enter prime p: "))

    q=int(input("Enter prime q (!=p): "))

print("Prime number p: ",p)

print("Prime number q: ",q)

print("Generating Public/Private key-pairs!")

public, private = generate\_keypair(p, q)

print("Your public key is (e,n) ", public)

print("Your private key is (d,n) ", private)

message = input("Enter the message: ")

# converting into lower case and removing spaces

message=message.replace(" ","")

message=message.lower()

arr=[]

cipher\_text=[]

for i in message:

    if i in main:

        arr.append(main.index(i))

for i in arr:

    cipher\_text.append(encrypt(public,i))

print("Encrypted message (Cipher Text): ",cipher\_text)

plain=[]

for i in cipher\_text:

    plain.append(decrypt(private,i))

plain\_text=''

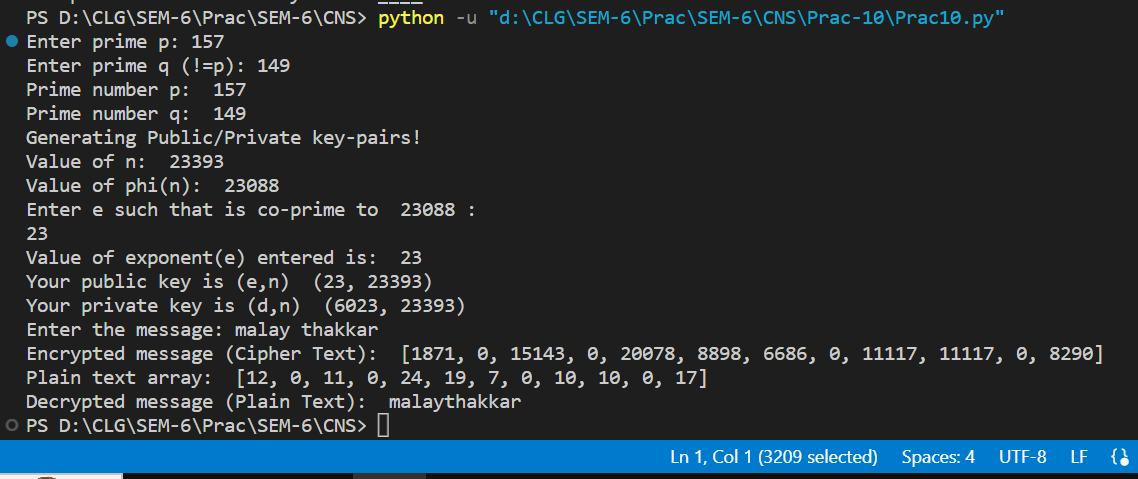
for i in plain:

    plain\_text=plain\_text+main[i]

print("Plain text array: ",plain)

print("Decrypted message (Plain Text): ", plain\_text)

* **OUTPUT:**

****