**1) write a program to implement a Substitution cipher.**

**Aim:** to implement a substitution cipher using a python program.

**Program:**

import string

letters = string.ascii\_letters[:26]

print(letters)

key =int(input("enter the key"))

plain\_text = input("enter the plain\_text")

encrypted\_text = ''

decrypted\_text = ''

for i in plain\_text:

if i in letters:

encrypted\_text += letters[(letters.index(i)+key)%26]

else:

encrypted\_text += i

print(encrypted\_text)

for i in encrypted\_text:

if i in letters:

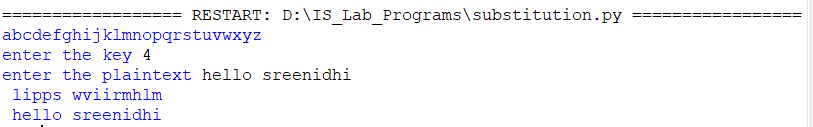
decrypted\_text += letters[(letters.index(i)-key)%26]

else:

decrypted\_text += i

print(decrypted\_text)

**Output:**

****

**2) write a program to implement a Transposition cipher.**

**Aim:** to implement a Transposition cipher using a python program.

**Program:**

plain\_text =input("enter plain text")

key = int(input("enter key"))

key\_len = len(str(key))

encrypted\_text = ''

decrypted\_text = ''

ans = []

for i in range(0,len(plain\_text),key\_len):

ans.append(plain\_text[i:i+key\_len])

if len(ans[-1]) != key\_len:

ans[-1] = ans[-1]+ (key\_len - len(ans[-1]))\*'|'

print(ans)

matrix = []

for i in ans:

a = []

for j in i:

a.append(j)

matrix.append(a)

for i in matrix:

print(i)

for i in str(key):

row = 0

col = int(i)-1

temp = ''

for i in range(len(matrix)):

#print('row = {} , col = {}'.format(row,col))

temp += matrix[row][col]

row += 1

encrypted\_text += temp

original\_encrypted\_text = encrypted\_text

encrypted\_text = encrypted\_text.replace("|","")

print()

print(encrypted\_text)

#matrix reordering

temp\_matrix=[]

for i in range(len(matrix)):

a = []

temp\_matrix.append(a)

# print(temp\_matrix)

for i in str(key):

row = 0

col = int(i)-1

temp = []

for i in range(len(matrix)):

#print('row = {} , col = {}'.format(row,col))

temp\_matrix[row].append(matrix[row][col])

row += 1

print()

for i in temp\_matrix:

print(i)

temp\_matrix\_2 = []

for i in range(len(matrix)):

a = []

temp\_matrix\_2.append(a)

for i in sorted(str(key)):

row = 0

col = int(i)-1

temp = []

for i in range(len(matrix)):

#print('row = {} , col = {}'.format(row,col))

temp\_matrix\_2[row].append(matrix[row][col])

row += 1

print()

for i in temp\_matrix\_2:

print(i)

for i in range(len(temp\_matrix\_2)):

for j in range(len(temp\_matrix\_2[0])):

if temp\_matrix\_2[i][j] == '|':

pass

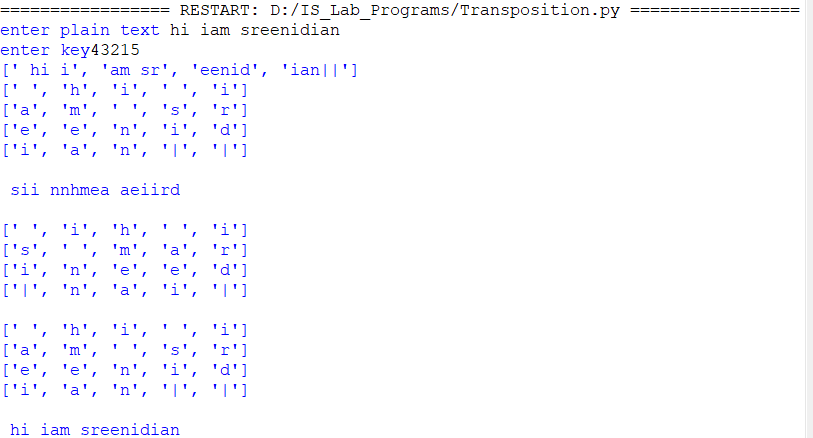
else:

decrypted\_text += temp\_matrix\_2[i][j]

print()

print(decrypted\_text)

**Output:**

****

**3)write a program to implement a DES.**

**Aim:** to implement a DES algorithm using a java program.

Program:

import javax.crypto.\*;

import java.util.Scanner;

public class Main

{

public static void main(String[] args) {

try{

KeyGenerator kg=KeyGenerator.getInstance("DES");

SecretKey myDESkey=kg.generateKey();

Cipher cipher=Cipher.getInstance("DES/ECB/PKCS5Padding");

cipher.init(Cipher.ENCRYPT\_MODE,myDESkey);

Scanner s=new Scanner(System.in);

System.out.print("enter message you want to sent: ");

String st=s.nextLine();

System.out.println("you entered text is ::"+st);

byte[] text=st.getBytes();

System.out.println("Text in Bytes: "+text);

System.out.println("Text "+new String(text));

byte[] textenc=cipher.doFinal(text);

System.out.println("Text in Bytes"+textenc);

System.out.println("Text Encrypted: " +new String(textenc));

cipher.init(Cipher.DECRYPT\_MODE,myDESkey);

byte[] textdec=cipher.doFinal(textenc);

System.out.println("Text Decrypted: " +new String(textdec));

}

catch(Exception e)

{

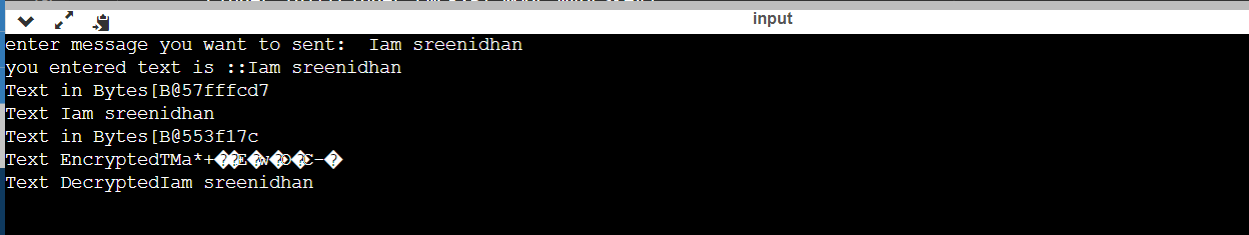
System.out.println(e);

}

}

}

**Output:**



**4) write a program to implement a RSA.**

**Aim:** to implement a RSA algorithm using a python program.

Program:

import math

p , q =map(int,input("enter two prime numbers").split())

print('p == {} , q == {}'.format(p,q))

n = p\*q

print( 'n == {}'.format(n))

def phi(p,q):

return (p-1)\*(q-1)

print( 'phi == {}'.format(phi(p,q)))

e = 2

while e < phi(p,q) :

if math.gcd(e,phi(p,q)) == 1:

break

else:

e += 1

print( 'e == {}'.format(e))

def compute\_d(e,phi):

d = 1

while True:

if (d\*e)%phi == 1:

return d

else:

d += 1

print( 'd == {}'.format(compute\_d(e,phi(p,q))))

d = compute\_d(e,phi(p,q))

plain\_text =int(input("enter plain\_text"))

cipher\_text = (plain\_text\*\*e) % n

print( 'plain\_text == {}'.format(plain\_text))

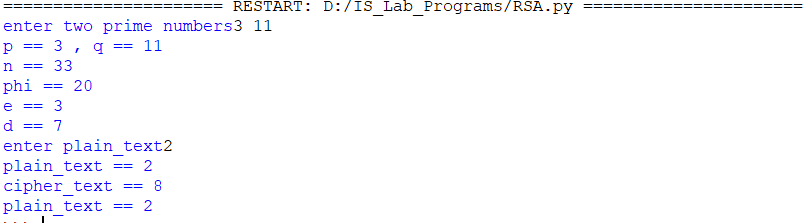
print( 'cipher\_text == {}'.format(cipher\_text))

#decrypt

plain\_text = (cipher\_text \*\* d) % n

print( 'plain\_text == {}'.format(plain\_text))

**Output:**



**5) write a program to implement a Diffie Hellman Algorithm.**

**Aim:** to implement a Diffie hellman algorithm using a python program.

**Program:**

from random import getrandbits

from random import randint

import sys

def is\_prime\_calc(num):

return all(num % i for i in range(2, num))

def is\_prime(num):

return is\_prime\_calc(num)

def get\_random\_prime():

while True:

n = getrandbits(12) + 3;

if is\_prime(n):

return n

def gcd(a,b):

while a != b:

if a > b:

a = a - b

else:

b = b - a

return a

def primitive\_root(modulo):

required\_set = set(num for num in range (1, modulo) if gcd(num, modulo) == 1)

for g in range(1, modulo):

actual\_set = set(pow(g, powers) % modulo for powers in range (1, modulo))

if required\_set == actual\_set:

return g

alice\_private = int(input("enter alice private key"))

print ('Alice private key is {}'.format(alice\_private))

bob\_private = int(input("enter bob private key"))

print ('Bob private key is {}'.format(bob\_private))

p = int(input("enter prime number"))

print ('primitive root is {}'.format(primitive\_root(p)))

g = primitive\_root(p)

print ('p parameter is {}, g parameter is {}'.format(p, g))

alice\_public = pow(g, alice\_private) % p

bob\_public = pow(g, bob\_private) % p

print ('Alice public key is {}'.format(alice\_public))

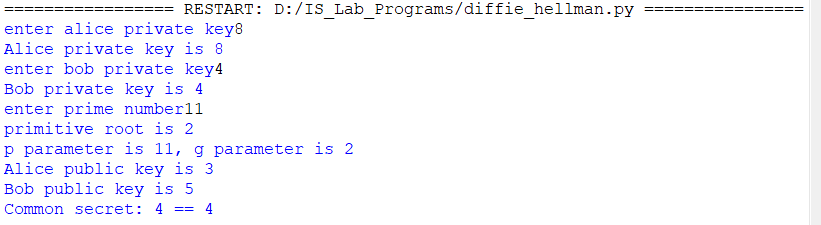
print ('Bob public key is {}'.format(bob\_public))

alice\_key = (pow(bob\_public, alice\_private)) % p

bob\_key = (pow(alice\_public, bob\_private)) % p

print ('Common secret: {} == {}'.format(alice\_key, bob\_key))

**output:**



**6.** **write a program to implement a RSA Digital Signature**

**Aim:** to implement a RSA Digital Signature using a python program.

**Program:**

import random

def euclid(m, n):

if n == 0:

return m

else:

r = m % n

return euclid(n, r)

def exteuclid(a, b):

r1 = a

r2 = b

s1 = int(1)

s2 = int(0)

t1 = int(0)

t2 = int(1)

while r2 > 0:

q = r1//r2

r = r1-q \* r2

r1 = r2

r2 = r

s = s1-q \* s2

s1 = s2

s2 = s

t = t1-q \* t2

t1 = t2

t2 = t

if t1 < 0:

t1 = t1 % a

return (r1, t1)

p = int(input("enter first prime number"))

q = int(input("enter second prime number"))

n = p \* q

Pn = (p-1)\*(q-1)

key = []

for i in range(2, Pn):

gcd = euclid(Pn, i)

if gcd == 1:

key.append(i)

e = int(random.choice(key))

r, d = exteuclid(Pn, e)

if r == 1:

d = int(d)

print("decryption key is: ", d)

else:

print("Multiplicative inverse for the given encryption key does not exist. Choose a different encryption key ")

M = int(input("enter the message to be sent"))

S = (M\*\*d) % n

M1 = (S\*\*e) % n

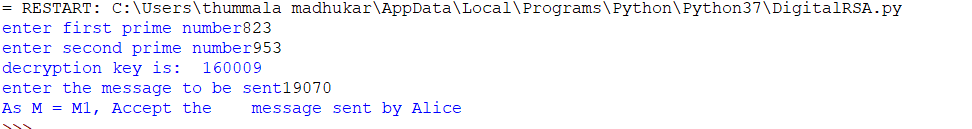
if M == M1:

print("As M = M1, Accept the message sent by Alice")

else:

print("As M not equal to M1, Do not accept the message sent by Alice ")

**Output:**

****

**7) write a program to implement a DSA Digital Signature.**

**Aim:** to implement a DSA Digital signature using a python program.

**Program:**

from random import randrange

from hashlib import sha1

from gmpy2 import xmpz, to\_binary, invert, powmod, is\_prime

def generate\_p\_q(L, N):

g = N # g >= 160

n = (L - 1) // g

b = (L - 1) % g

while True:

# generate q

while True:

s = xmpz(randrange(1, 2 \*\* (g)))

a = sha1(to\_binary(s)).hexdigest()

zz = xmpz((s + 1) % (2 \*\* g))

z = sha1(to\_binary(zz)).hexdigest()

U = int(a, 16) ^ int(z, 16)

mask = 2 \*\* (N - 1) + 1

q = U | mask

if is\_prime(q, 20):

break

# generate p

i = 0 # counter

j = 2 # offset

while i < 4096:

V = []

for k in range(n + 1):

arg = xmpz((s + j + k) % (2 \*\* g))

zzv = sha1(to\_binary(arg)).hexdigest()

V.append(int(zzv, 16))

W = 0

for qq in range(0, n):

W += V[qq] \* 2 \*\* (160 \* qq)

W += (V[n] % 2 \*\* b) \* 2 \*\* (160 \* n)

X = W + 2 \*\* (L - 1)

c = X % (2 \* q)

p = X - c + 1 # p = X - (c - 1)

if p >= 2 \*\* (L - 1):

if is\_prime(p, 10):

return p, q

i += 1

j += n + 1

def generate\_g(p, q):

while True:

h = randrange(2, p - 1)

exp = xmpz((p - 1) // q)

g = powmod(h, exp, p)

if g > 1:

break

return g

def generate\_keys(g, p, q):

x = randrange(2, q) # x < q

y = powmod(g, x, p)

return x, y

def generate\_params(L, N):

p, q = generate\_p\_q(L, N)

g = generate\_g(p, q)

return p, q, g

def sign(M, p, q, g, x):

if not validate\_params(p, q, g):

raise Exception("Invalid params")

while True:

k = randrange(2, q) # k < q

r = powmod(g, k, p) % q

m = int(sha1(M).hexdigest(), 16)

try:

s = (invert(k, q) \* (m + x \* r)) % q

return r, s

except ZeroDivisionError:

pass

def verify(M, r, s, p, q, g, y):

if not validate\_params(p, q, g):

raise Exception("Invalid params")

if not validate\_sign(r, s, q):

return False

try:

w = invert(s, q)

except ZeroDivisionError:

return False

m = int(sha1(M).hexdigest(), 16)

u1 = (m \* w) % q

u2 = (r \* w) % q

# v = ((g \*\* u1 \* y \*\* u2) % p) % q

v = (powmod(g, u1, p) \* powmod(y, u2, p)) % p % q

if v == r:

return True

return False

def validate\_params(p, q, g):

if is\_prime(p) and is\_prime(q):

return True

if powmod(g, q, p) == 1 and g > 1 and (p - 1) % q:

return True

return False

def validate\_sign(r, s, q):

if r < 0 and r > q:

return False

if s < 0 and s > q:

return False

return True

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

N = int(input()) #N=160

L = int(input()) #L=1024

p, q, g = generate\_params(L, N)

x, y = generate\_keys(g, p, q)

text = input()

M = str.encode(text, "ascii")

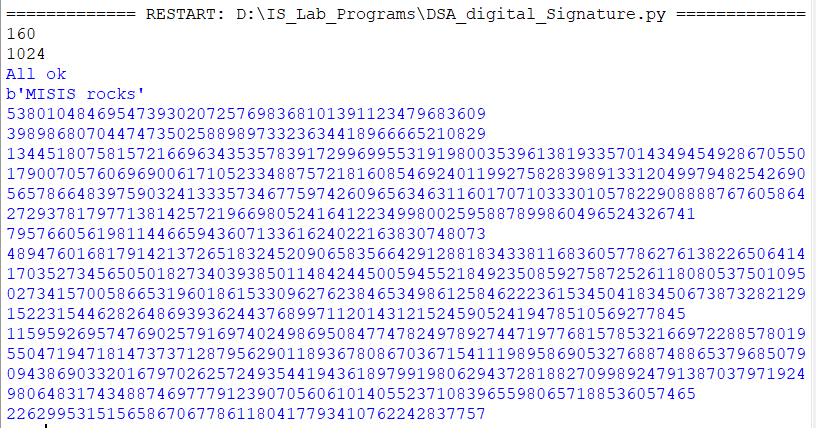
r, s = sign(M, p, q, g, x)

if verify(M, r, s, p, q, g, y):

print('All ok')

print(M, r, s, p, q, g, y, x, sep='\n')

**Output:**

****

**8)write a program to implement a Digital Envelope**

**Aim:** to implement a Digital Envelope using a python program.

**Program:**

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

import binascii

from cryptography.fernet import Fernet

message = input("enter plain text:")

key = Fernet.generate\_key()

print("key is:",end=" ")

print(key)

fernet = Fernet(key)

encMessage = fernet.encrypt(message.encode())

print("original string: ", message)

print("CipherText: ", encMessage)

keyPair = RSA.generate(1024)

pubKeyB = keyPair.publickey()

pubKeyPEMB = pubKeyB.exportKey()

print()

print(pubKeyPEMB.decode('ascii'))

privKeyPEMB= keyPair.exportKey()

print()

print(privKeyPEMB.decode('ascii'))

msg = bytes(str(key), 'utf-8')

encryptor = PKCS1\_OAEP.new(pubKeyB)

encrypted = encryptor.encrypt(msg)

EncryptedKey=binascii.hexlify(encrypted)

print("\n Encrypted key with public key B:", binascii.hexlify(encrypted))

digitalenvelope=encMessage+EncryptedKey

print(digitalenvelope)

decryptor = PKCS1\_OAEP.new(keyPair)

decrypted = decryptor.decrypt(encrypted)

print('Decrypted key is:', decrypted.decode('utf-8'))

decryptedkey=decrypted.decode('utf-8')

if(str(key) == str(decryptedkey)):

print("both keys matched")

else:

print("both keys not matched")

decMessage = fernet.decrypt(encMessage).decode()

print("decrypted string: ", decMessage)

**Output:**

****