CEASAR

def encrypt(text,s):

result = ""

# traverse text

for i in range(len(text)):

char = text[i]

# Encrypt uppercase characters

if (char.isupper()):

result += chr((ord(char) + s-65) % 26 + 65)

# Encrypt lowercase characters

else:

result += chr((ord(char) + s - 97) % 26 + 97)

return result

text = "ATTACKATONCE"

s = 4

print ("Text : " + text)

print ("Shift : " + str(s))

print ("Cipher: " + encrypt(text,s))

PLAYFAIR

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)

# Get 2 letter cipherText

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 = 'instruments'

text\_Plain = removeSpaces(toLowerCase(text\_Plain))

PlainTextList = Diagraph(FillerLetter(text\_Plain))

if len(PlainTextList[-1]) != 2:

PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"

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)

VIGENERE

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 cipherText(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 originalText(cipher\_text, key):

orig\_text = []

for i in range(len(cipher\_text)):

x = (ord(cipher\_text[i]) -

ord(key[i]) + 26) % 26

x += ord('A')

orig\_text.append(chr(x))

return("" . join(orig\_text))

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

string = "GEEKSFORGEEKS"

keyword = "AYUSH"

key = generateKey(string, keyword)

cipher\_text = cipherText(string,key)

print("Ciphertext :", cipher\_text)

print("Original/Decrypted Text :",

originalText(cipher\_text, key))

AFFINE

# Extended Euclidean Algorithm for finding modular inverse

# eg: modinv(7, 26) = 15

def egcd(a, b):

x,y, u,v = 0,1, 1,0

while a != 0:

q, r = b//a, b%a

m, n = x-u\*q, y-v\*q

b,a, x,y, u,v = a,r, u,v, m,n

gcd = b

return gcd, x, y

def modinv(a, m):

gcd, x, y = egcd(a, m)

if gcd != 1:

return None # modular inverse does not exist

else:

return x % m

# affine cipher encryption function

# returns the cipher text

def affine\_encrypt(text, key):

'''

C = (a\*P + b) % 26

'''

return ''.join([ chr((( key[0]\*(ord(t) - ord('A')) + key[1] ) % 26)

+ ord('A')) for t in text.upper().replace(' ', '') ])

# affine cipher decryption function

# returns original text

def affine\_decrypt(cipher, key):

'''

P = (a^-1 \* (C - b)) % 26

'''

return ''.join([ chr((( modinv(key[0], 26)\*(ord(c) - ord('A') - key[1]))

% 26) + ord('A')) for c in cipher ])

# Driver Code to test the above functions

def main():

# declaring text and key

text = 'AFFINE CIPHER'

key = [17, 20]

# calling encryption function

affine\_encrypted\_text = affine\_encrypt(text, key)

print('Encrypted Text: {}'.format( affine\_encrypted\_text ))

# calling decryption function

print('Decrypted Text: {}'.format

( affine\_decrypt(affine\_encrypted\_text, key) ))

if \_\_name\_\_ == '\_\_main\_\_':

main()

Transposition

# Python3 implementation of

# Columnar Transposition

import math

key = "HACK"

# 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 = ""

# track key indices

k\_indx = 0

# track msg indices

msg\_indx = 0

msg\_len = float(len(cipher))

msg\_lst = list(cipher)

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# convert key into list and sort

# alphabetically so we can access

# each character by its alphabetical position.

key\_lst = sorted(list(key))

# create an empty matrix to

# store deciphered message

dec\_cipher = []

for \_ in range(row):

dec\_cipher += [[None] \* col]

# Arrange the matrix column wise according

# to permutation order by adding into new matrix

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

# convert decrypted msg matrix into a string

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

# Driver Code

msg = "Geeks for Geeks"

cipher = encryptMessage(msg)

print("Encrypted Message: {}".

format(cipher))

print("Decryped Message: {}".

format(decryptMessage(cipher)))

RC4

# Python3 program for the above approach

# of RC4 algorithm

# Function for encryption

def encryption():

global key, plain\_text, n

# Given text and key

plain\_text = "001010010010"

key = "101001000001"

# n is the no: of bits to

# be considered at a time

n = 3

print("Plain text : ", plain\_text)

print("Key : ", key)

print("n : ", n)

print(" ")

# The initial state vector array

S = [i for i in range(0, 2\*\*n)]

print("S : ", S)

key\_list = [key[i:i + n] for i in range(0, len(key), n)]

# Convert to key\_stream to decimal

for i in range(len(key\_list)):

key\_list[i] = int(key\_list[i], 2)

# Convert to plain\_text to decimal

global pt

pt = [plain\_text[i:i + n] for i in range(0, len(plain\_text), n)]

for i in range(len(pt)):

pt[i] = int(pt[i], 2)

print("Plain text ( in array form ): ", pt)

# Making key\_stream equal

# to length of state vector

diff = int(len(S)-len(key\_list))

if diff != 0:

for i in range(0, diff):

key\_list.append(key\_list[i])

print("Key list : ", key\_list)

print(" ")

# Perform the KSA algorithm

def KSA():

j = 0

N = len(S)

# Iterate over the range [0, N]

for i in range(0, N):

# Find the key

j = (j + S[i]+key\_list[i]) % N

# Update S[i] and S[j]

S[i], S[j] = S[j], S[i]

print(i, " ", end ="")

# Print S

print(S)

initial\_permutation\_array = S

print(" ")

print("The initial permutation array is : ",

initial\_permutation\_array)

print("KSA iterations : ")

print(" ")

KSA()

print(" ")

# Perform PGRA algorithm

def PGRA():

N = len(S)

i = j = 0

global key\_stream

key\_stream = []

# Iterate over [0, length of pt]

for k in range(0, len(pt)):

i = (i + 1) % N

j = (j + S[i]) % N

# Update S[i] and S[j]

S[i], S[j] = S[j], S[i]

print(k, " ", end ="")

print(S)

t = (S[i]+S[j]) % N

key\_stream.append(S[t])

# Print the key stream

print("Key stream : ", key\_stream)

print(" ")

print("PGRA iterations : ")

print(" ")

PGRA()

# Performing XOR between generated

# key stream and plain text

def XOR():

global cipher\_text

cipher\_text = []

for i in range(len(pt)):

c = key\_stream[i] ^ pt[i]

cipher\_text.append(c)

XOR()

# Convert the encrypted text to

# bits form

encrypted\_to\_bits = ""

for i in cipher\_text:

encrypted\_to\_bits += '0'\*(n-len(bin(i)[2:]))+bin(i)[2:]

print(" ")

print("Cipher text : ", encrypted\_to\_bits)

encryption()

print("---------------------------------------------------------")

# Function for decryption of data

def decryption():

# The initial state vector array

S = [i for i in range(0, 2\*\*n)]

key\_list = [key[i:i + n] for i in range(0, len(key), n)]

# Convert to key\_stream to decimal

for i in range(len(key\_list)):

key\_list[i] = int(key\_list[i], 2)

# Convert to plain\_text to decimal

global pt

pt = [plain\_text[i:i + n] for i in range(0, len(plain\_text), n)]

for i in range(len(pt)):

pt[i] = int(pt[i], 2)

# making key\_stream equal

# to length of state vector

diff = int(len(S)-len(key\_list))

if diff != 0:

for i in range(0, diff):

key\_list.append(key\_list[i])

print(" ")

# KSA algorithm

def KSA():

j = 0

N = len(S)

# Iterate over the range [0, N]

for i in range(0, N):

j = (j + S[i]+key\_list[i]) % N

# Update S[i] and S[j]

S[i], S[j] = S[j], S[i]

print(i, " ", end ="")

print(S)

initial\_permutation\_array = S

print(" ")

print("The initial permutation array is : ",

initial\_permutation\_array)

print("KSA iterations : ")

print(" ")

KSA()

print(" ")

# Perform PRGA algorithm

def do\_PGRA():

N = len(S)

i = j = 0

global key\_stream

key\_stream = []

# Iterate over the range

for k in range(0, len(pt)):

i = (i + 1) % N

j = (j + S[i]) % N

# Update S[i] and S[j]

S[i], S[j] = S[j], S[i]

print(k, " ", end ="")

print(S)

t = (S[i]+S[j]) % N

key\_stream.append(S[t])

print("Key stream : ", key\_stream)

print(" ")

print("PGRA iterations : ")

print(" ")

do\_PGRA()

# Perform XOR between generated

# key stream and cipher text

def do\_XOR():

global original\_text

original\_text = []

for i in range(len(cipher\_text)):

p = key\_stream[i] ^ cipher\_text[i]

original\_text.append(p)

do\_XOR()

# convert the decrypted text to

# the bits form

decrypted\_to\_bits = ""

for i in original\_text:

decrypted\_to\_bits += '0'\*(n-len(bin(i)[2:]))+bin(i)[2:]

print(" ")

print("Decrypted text : ",

decrypted\_to\_bits)

# Driver Code

decryption()

DES

# Python3 code for the above approach

# Hexadecimal to binary conversion

def hex2bin(s):

mp = {'0': "0000",

'1': "0001",

'2': "0010",

'3': "0011",

'4': "0100",

'5': "0101",

'6': "0110",

'7': "0111",

'8': "1000",

'9': "1001",

'A': "1010",

'B': "1011",

'C': "1100",

'D': "1101",

'E': "1110",

'F': "1111"}

bin = ""

for i in range(len(s)):

bin = bin + mp[s[i]]

return bin

# Binary to hexadecimal conversion

def bin2hex(s):

mp = {"0000": '0',

"0001": '1',

"0010": '2',

"0011": '3',

"0100": '4',

"0101": '5',

"0110": '6',

"0111": '7',

"1000": '8',

"1001": '9',

"1010": 'A',

"1011": 'B',

"1100": 'C',

"1101": 'D',

"1110": 'E',

"1111": 'F'}

hex = ""

for i in range(0, len(s), 4):

ch = ""

ch = ch + s[i]

ch = ch + s[i + 1]

ch = ch + s[i + 2]

ch = ch + s[i + 3]

hex = hex + mp[ch]

return hex

# Binary to decimal conversion

def bin2dec(binary):

binary1 = binary

decimal, i, n = 0, 0, 0

while(binary != 0):

dec = binary % 10

decimal = decimal + dec \* pow(2, i)

binary = binary//10

i += 1

return decimal

# Decimal to binary conversion

def dec2bin(num):

res = bin(num).replace("0b", "")

if(len(res) % 4 != 0):

div = len(res) / 4

div = int(div)

counter = (4 \* (div + 1)) - len(res)

for i in range(0, counter):

res = '0' + res

return res

# Permute function to rearrange the bits

def permute(k, arr, n):

permutation = ""

for i in range(0, n):

permutation = permutation + k[arr[i] - 1]

return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

s = ""

for i in range(nth\_shifts):

for j in range(1, len(k)):

s = s + k[j]

s = s + k[0]

k = s

s = ""

return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

ans = ""

for i in range(len(a)):

if a[i] == b[i]:

ans = ans + "0"

else:

ans = ans + "1"

return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table

initial\_perm = [58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table

exp\_d = [32, 1, 2, 3, 4, 5, 4, 5,

6, 7, 8, 9, 8, 9, 10, 11,

12, 13, 12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21, 20, 21,

22, 23, 24, 25, 24, 25, 26, 27,

28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table

per = [16, 7, 20, 21,

29, 12, 28, 17,

1, 15, 23, 26,

5, 18, 31, 10,

2, 8, 24, 14,

32, 27, 3, 9,

19, 13, 30, 6,

22, 11, 4, 25]

# S-box Table

sbox = [[[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]],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table

final\_perm = [40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25]

def encrypt(pt, rkb, rk):

pt = hex2bin(pt)

# Initial Permutation

pt = permute(pt, initial\_perm, 64)

print("After initial permutation", bin2hex(pt))

# Splitting

left = pt[0:32]

right = pt[32:64]

for i in range(0, 16):

# Expansion D-box: Expanding the 32 bits data into 48 bits

right\_expanded = permute(right, exp\_d, 48)

# XOR RoundKey[i] and right\_expanded

xor\_x = xor(right\_expanded, rkb[i])

# S-boxex: substituting the value from s-box table by calculating row and column

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(

int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

# Straight D-box: After substituting rearranging the bits

sbox\_str = permute(sbox\_str, per, 32)

# XOR left and sbox\_str

result = xor(left, sbox\_str)

left = result

# Swapper

if(i != 15):

left, right = right, left

print("Round ", i + 1, " ", bin2hex(left),

" ", bin2hex(right), " ", rk[i])

# Combination

combine = left + right

# Final permutation: final rearranging of bits to get cipher text

cipher\_text = permute(combine, final\_perm, 64)

return cipher\_text

pt = "123456ABCD132536"

key = "AABB09182736CCDD"

# Key generation

# --hex to binary

key = hex2bin(key)

# --parity bit drop table

keyp = [57, 49, 41, 33, 25, 17, 9,

1, 58, 50, 42, 34, 26, 18,

10, 2, 59, 51, 43, 35, 27,

19, 11, 3, 60, 52, 44, 36,

63, 55, 47, 39, 31, 23, 15,

7, 62, 54, 46, 38, 30, 22,

14, 6, 61, 53, 45, 37, 29,

21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

# Number of bit shifts

shift\_table = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits

key\_comp = [14, 17, 11, 24, 1, 5,

3, 28, 15, 6, 21, 10,

23, 19, 12, 4, 26, 8,

16, 7, 27, 20, 13, 2,

41, 52, 31, 37, 47, 55,

30, 40, 51, 45, 33, 48,

44, 49, 39, 56, 34, 53,

46, 42, 50, 36, 29, 32]

# Splitting

left = key[0:28] # rkb for RoundKeys in binary

right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []

rk = []

for i in range(0, 16):

# Shifting the bits by nth shifts by checking from shift table

left = shift\_left(left, shift\_table[i])

right = shift\_left(right, shift\_table[i])

# Combination of left and right string

combine\_str = left + right

# Compression of key from 56 to 48 bits

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

print("Encryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

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

print("Decryption")

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

print("Plain Text : ", text)

DIffe

from random import randint

if \_\_name\_\_ == '\_\_main\_\_':

# Both the persons will be agreed upon the

# public keys G and P

# A prime number P is taken

P = 23

# A primitive root for P, G is taken

G = 9

print('The Value of P is :%d'%(P))

print('The Value of G is :%d'%(G))

# Alice will choose the private key a

a = 4

print('The Private Key a for Alice is :%d'%(a))

# gets the generated key

x = int(pow(G,a,P))

# Bob will choose the private key b

b = 3

print('The Private Key b for Bob is :%d'%(b))

# gets the generated key

y = int(pow(G,b,P))

# Secret key for Alice

ka = int(pow(y,a,P))

# Secret key for Bob

kb = int(pow(x,b,P))

print('Secret key for the Alice is : %d'%(ka))

print('Secret Key for the Bob is : %d'%(kb))

ELgamal

# Python program to illustrate ElGamal encryption

import random

from math import pow

a = random.randint(2, 10)

def gcd(a, b):

if a < b:

return gcd(b, a)

elif a % b == 0:

return b;

else:

return gcd(b, a % b)

# Generating large random numbers

def gen\_key(q):

key = random.randint(pow(10, 20), q)

while gcd(q, key) != 1:

key = random.randint(pow(10, 20), q)

return key

# Modular exponentiation

def power(a, b, c):

x = 1

y = a

while b > 0:

if b % 2 != 0:

x = (x \* y) % c;

y = (y \* y) % c

b = int(b / 2)

return x % c

# Asymmetric encryption

def encrypt(msg, q, h, g):

en\_msg = []

k = gen\_key(q)# Private key for sender

s = power(h, k, q)

p = power(g, k, q)

for i in range(0, len(msg)):

en\_msg.append(msg[i])

print("g^k used : ", p)

print("g^ak used : ", s)

for i in range(0, len(en\_msg)):

en\_msg[i] = s \* ord(en\_msg[i])

return en\_msg, p

def decrypt(en\_msg, p, key, q):

dr\_msg = []

h = power(p, key, q)

for i in range(0, len(en\_msg)):

dr\_msg.append(chr(int(en\_msg[i]/h)))

return dr\_msg

# Driver code

def main():

msg = 'encryption'

print("Original Message :", msg)

q = random.randint(pow(10, 20), pow(10, 50))

g = random.randint(2, q)

key = gen\_key(q)# Private key for receiver

h = power(g, key, q)

print("g used : ", g)

print("g^a used : ", h)

en\_msg, p = encrypt(msg, q, h, g)

dr\_msg = decrypt(en\_msg, p, key, q)

dmsg = ''.join(dr\_msg)

print("Decrypted Message :", dmsg);

if \_\_name\_\_ == '\_\_main\_\_':

main()