Caesar Cipher

```
uppercase_alphabets = [chr(i) for i in range(65, 91)]
def encryption(plain_text, shift_key):
    """ encrypts the plain text with the shift key and returns an encrypted msg """
    encrypted= ''
    for char in plain_text:
        if char.upper() in uppercase_alphabets:
            pos = uppercase_alphabets.index(char.upper())
            encrypted += uppercase_alphabets[(pos + shift_key) % len(uppercase_alphabets)]
        else:
            encrypted += char
    return encrypted

def decryption(encrypted_text, shift_key):
    """decrypts encrypted msg with shift key and returns plain text"""
    decrypted = ''
    for char in encrypted_text:
        if char.upper() in uppercase_alphabets:
            pos = uppercase_alphabets.index(char.upper())
            decrypted += uppercase_alphabets[(pos - shift_key) % len(uppercase_alphabets)]
        else:
            decrypted += char
    return decrypted

# shift_key= 3
# plain_text= 'decode me xyz'
print('-------Caesar Cipher-------')
plain_txt= input('Enter your plain-text: ')
key= int(input('Enter encryption key: '))
encrypted_msg= encryption(plain_txt, key)
decrypted_msg= decryption(encrypted_msg, key)

print(f'Pain text: {plain_txt}')
print(f'Encrypted msg: {encrypted_msg}')
print(f'Decrypted msg: {decrypted_msg}')
```

```
-------Caesar Cipher-------
Enter your plain-text: Symmetric Cipher
Enter encryption key: 5
Pain text: Symmetric Cipher
Encrypted msg: XDRRJYWNH HNUMJW
Decrypted msg: SYMMETRIC CIPHER
```

Playfair Cipher

```
uppercase_alphabets = [chr(i) for i in range(65, 91)]
lowercase_alphabets= [chr(i) for i in range (97, 123)]

import numpy as np
# playfair_size= 5
mat= np.resize(None, new_shape=(5,5))

def build_playfair_mat(keyword):
    """build a 5x5 playfair matrix with keyword """
    key = 0 # for index in keyword
    ind = 0 # for index in alphabets
    for row in range(5):
        col= 0  # start form 0 th index
        while col < 5:
            if key < len(keyword):  # appending keyword in matrix(till end)
                if keyword[key] not in mat[row]:   # ensure unique elements over mat(no duplicate)
                    mat[row][col]= keyword[key]
                    col+= 1 # ready to append on next
                key +=1 # always incremented whether it is appended on mat or not(loop for keyword)
            else:
                # completing playfair mat using remaining letters
                if ind < 26:    # upto 25->z
                    char= lowercase_alphabets[ind]
                    if char != 'j':  # j is ignored as it will be replaced by i later
                        if not any(char in i for i in mat): # append letters only if it doesn't exist
                            mat[row][col] = lowercase_alphabets[ind]
                            col += 1    # incremented after inserting on playfair mat
                    ind += 1  # always incremented whether it is appended on mat or not(loop for alphabets)
                else:
                    break   # playfair mat build successfully
    return mat

def get_diagraph(plain_text):
    """ returns a list of pair or chars """
    for char in plain_text:
        if char not in lowercase_alphabets:
            plain_text = plain_text.replace(char, '')   # all non-alphabetic chars are skipped
    plain_text= plain_text.replace('j','i')  # all j are replaced by i
    diagraph = []
    i=0
    while i < len(plain_text): # tills end or plain-text
       a = plain_text[i]    # first char of pair is always appended
       if i+1 == len(plain_text):   # check if last pair has only one char
```

```
        # b='z'
        b = 'x'
    elif plain_text[i] == plain_text[i + 1]: # if both char are same
        b = 'x'
        i -=1    # decremented cuz it is increased by 2 units later ensure no chars are skipped
    else:
        b = plain_text[i + 1]
    diagraph.append(a + b)
    i += 2
  return diagraph


def encrypt_pair(new_mat, pair):
  """ encrypt a diagram pair based on playfair mat"""
  mat_len= len(new_mat)   # usually 5
  index_a = [(row,col) for row in range(len(new_mat)) for col in range(len(new_mat[row])) if
new_mat[row][col] == pair[0]]
  index_b = [(row,col) for row in range(len(new_mat)) for col in range(len(new_mat[row])) if
new_mat[row][col] == pair[1]]

  # get actual index form [[row,col]] as [row,col]
  index_a= index_a[0]
  index_b= index_b[0]
  row_a= index_a[0]; col_a= index_a[1]; row_b= index_b[0]; col_b= index_b[1]

  # if same row next char are taken on same row in cycle
  if row_a == row_b:
     substr= str(new_mat[row_a][(col_a + 1) % mat_len])
     substr += str(new_mat[row_b][(col_b + 1) % mat_len])
     return substr
  elif col_a == col_b: # if same col next char are taken on same col in cycle
     substr = str(new_mat[(row_a + 1) % mat_len][col_a])
     substr += str(new_mat[(row_b + 1) % mat_len][col_b])
     return substr
  else:   # elements are on different rows and columns
     substr = str(new_mat[row_a][col_b]) # columns are interchanged
     substr += str(new_mat[row_b][col_a])
     return substr

def encryption(diagraph):
  encrypted= [encrypt_pair(new_mat=playfair_mat, pair=pair) for pair in diagraph]
  return str(''.join(encrypted))
```

```
def decrypt_pair(new_mat, pair):
    """ encrypt a diagram pair based on playfair mat"""
    mat_len= len(new_mat)   # usually 5
    index_a = [(row,col) for row in range(len(new_mat)) for col in range(len(new_mat[row])) if
new_mat[row][col] == pair[0]]
    index_b = [(row,col) for row in range(len(new_mat)) for col in range(len(new_mat[row])) if
new_mat[row][col] == pair[1]]

    # get actual index form [[row,col]] as [row,col]
    index_a= index_a[0]
    index_b= index_b[0]
    row_a= index_a[0]; col_a= index_a[1]; row_b= index_b[0]; col_b= index_b[1]

    # if same row previous char are taken on same row in cycle(decryption)
    if row_a == row_b:
        substr= str(new_mat[row_a][(col_a - 1) % mat_len])
        substr += str(new_mat[row_b][(col_b - 1) % mat_len])
        return substr
    elif col_a == col_b: # if same col previous char are taken on same col in cycle
        substr = str(new_mat[(row_a - 1) % mat_len][col_a])
        substr += str(new_mat[(row_b - 1) % mat_len][col_b])
        return substr
    else:   # no change on diagonal
        substr = str(new_mat[row_a][col_b]) # columns are interchanged
        substr += str(new_mat[row_b][col_a])
        return substr


def decryption(diagraph):
    decrypted= [decrypt_pair(new_mat=playfair_mat, pair=pair) for pair in diagraph]
    return str(''.join(decrypted))

def play_fair(playfair_keyword, plain_txt):
    # create a diagraph
    playfair_diagraph= get_diagraph(plain_txt)

    # build a 5x5 playfair matrix
    global playfair_mat
    playfair_mat= build_playfair_mat(playfair_keyword)

    # encrypt plain text
    encrypted_msg= encryption(playfair_diagraph)
```

```
    # decryption
    enc_diagraph= get_diagraph(encrypted_msg)
    decrypted_msg= decryption(enc_diagraph)

    print('Plain text: ',plain_txt)
    print('Plain Diagram: ',enc_diagraph)
    print('Encrypted msg: ',encrypted_msg)

    print('Enc Diagram: ',enc_diagraph)
    print('Decrypted msg: ',decrypted_msg)


print('-------Play-fair Cipher-------')

playfair_keyword= 'monarchy'
# plain_txt= 'destroy the letter'
plain_txt= "the imitation game"
play_fair(playfair_keyword, plain_txt)


"""
during decryption
all process is reversed except the diagonal one
"""
```

```
-------Play-fair Cipher-------
Plain text:  the imitation game
Plain Diagram:  ['pd', 'fk', 'ae', 'sr', 'sk', 'na', 'in', 'cl']
Encrypted msg:  pdfkaesrsknaincl
Enc Diagram:  ['pd', 'fk', 'ae', 'sr', 'sk', 'na', 'in', 'cl']
Decrypted msg:  theimitationgame
```

Rail Fence Cipher

```
def encryption(msg, rail_depth):
    pattern = [['_' for _ in range(len(msg))] for _ in range(rail_depth)]
    # print(pattern)

    dir_down = False
    row, col = 0, 0

    for i in range(len(msg)):
        if (row == 0) | (row == rail_depth - 1):
            dir_down = not dir_down

        # fill the rail fence with corresponding alphabet
        pattern[row][col] = msg[i]
        col += 1

        if dir_down:  # find next row using dir_down flag
            row += 1
        else:
            row -= 1

    # construct cipher
    cipher = []
    for i in range(rail_depth):
        for j in range(len(msg)):
            if pattern[i][j] != '_':
                cipher.append(pattern[i][j])

    return ''.join(cipher)

def decryption(cipher, rail_depth):
    pattern= [['_' for _ in range(len(cipher))] for _ in range(rail_depth)]

    dir_down= False
    row,col= 0,0

    # mark places with *
    for _ in range(len(cipher)):
        if (row == 0) | (row == rail_depth - 1):
            dir_down= not dir_down

        pattern[row][col] = '*'
        col += 1
```

```
        if dir_down:
            row += 1
        else:
            row -= 1

    # fill the fence with the cipher text
    index=0 # to count on cipher chars
    for i in range(rail_depth):
        for j in range(len(cipher)):
            if pattern[i][j] == '*' and index < len(cipher):
                pattern[i][j] = cipher[index]
                index += 1
    # now read zig-zag manner to get message
    plain= []
    dir_down= None
    row,col= 0,0
    for _ in range(len(cipher)):
        if (row == 0) | (row == rail_depth - 1):
            dir_down= not dir_down

        # read the alphabets
        if pattern[row][col] != '*':
            plain.append(pattern[row][col])
            col += 1

        if dir_down:
            row += 1
        else:
            row -= 1

    return ''.join(plain)

print("--------Rail Fence-------")
msg= input("Enter plain text: ")
rail_depth= int(input("Enter rail depth: "))

msg = msg.replace(' ', '')
cipher= encryption(msg, rail_depth)
plain= decryption(cipher,rail_depth)

print(f"Message: {msg}\n"
      f"Cipher: {cipher}\n"
      f"Plain: {plain}")
```

```
--------Rail Fence-------
Enter plain text: Democracy a joke
Enter rail depth: 4
Message: Democracyajoke
Cipher: Dakercoemcyjoa
Plain: Democracyajoke
```

Modular Arithmetic Operation

```
print("***** Modular Arithmetic Operation *****")
m= int(input("Enter modulo(+ve): "))
a= int(input('Enter (+ve, -ve) dividend: '))
print("------a mod m ------")

# a= -5
# m=11
if a<0:
   i=1
   while i * m + a < 0:  # as mul_inv is -ve
      i += 1
   r = i * m + a
   """ alternate way;  r= (abs(a)//m +1)*m + a """
else:
   r= a- a//m*m

print(f"{a} mod {m} = {r}")
```

```
***** Modular Arithmetic Operation *****
Enter modulo(+ve): 47
Enter (+ve, -ve) dividend: -1397
------a mod m ------
-1397 mod 47 = 13
```

Vegenere and Vernam

```
import random
import numpy as np

uppercase_alphabets= [chr(i) for i in range(65,91)]
# print(uppercase_alphabets)

def Vegenere_cipher():
    keyword = "deceptive"
    msg = "we are discovered save yourself"

    print('-----Encryption-----')
    print("Message: ", msg)
    print("keyword: ", keyword)

    msg = msg.replace(' ', '')

    # to make keyword equal to message lenth
    keyword = (keyword * (len(msg) // len(keyword) + 1))[:len(msg)]  # repeat exactly factor + 1 times and
slice down

    msg_value = [uppercase_alphabets.index(char.upper()) for char in msg]
    key_value = [uppercase_alphabets.index(char.upper()) for char in keyword]

    cipher_value = [(key + msg) % 26 for key, msg in zip(key_value, msg_value)]

    cipher = [uppercase_alphabets[i] for i in cipher_value]
    # cipher = ''.join(cipher)

    print("Message: ", msg)
    print("keyword: ", keyword)
    print("Cipher text:", ''.join(cipher))

    dec_plain= [(c - k + 26) % 26 for c,k in zip(cipher_value, key_value )]
    dec_plain= [uppercase_alphabets[i] for i in dec_plain]
    print('-----Decryption-----')
    print("Decrypted msg: ", ''.join(dec_plain))


def Vernam_cipher():
    # msg= 'Each letter of the plaintext is XORed with key'
    msg = 'Random key generation'
    msg= msg.replace(' ','')    # remove all whitespaces
```

```
    random_key_value = random.choices(np.arange(0,26).tolist(), k=len(msg))     # take a random sample from 0-26 of length equal to msg(number can repeat)
    random_key= [uppercase_alphabets[i] for i in random_key_value]

    msg_value= [uppercase_alphabets.index(_) for _ in msg.upper()]
    cipher_value= [(k + p) % 26 for k,p in zip(random_key_value, msg_value) ]
    cipher= [uppercase_alphabets[_] for _ in cipher_value]

    print("-----Encryption-----")
    print(f"Msg: {msg} \nRandom key: {''.join(random_key)} \nCipher: {''.join(cipher)}")

    print("-----Decryption-----")
    plain_value= [(c - k + 26) % 26 for c,k in zip(cipher_value, random_key_value)]
    plain_text= [uppercase_alphabets[_] for _ in plain_value]

    print("Plain: ", ''.join(plain_text))

print("*******Vegenere Cipher*******")
Vegenere_cipher()
print("\n*******Vernam Cipher*******")
Vernam_cipher()
```

```
*******Vegenere Cipher*******
-----Encryption-----
Message:  we are discovered save yourself
keyword:  deceptive
Message:  wearediscoveredsaveyourself
keyword:  deceptivedeceptivedeceptive
Cipher text: ZICVTWQNGRZGVTWAVZHCQYGLMGJ
-----Decryption-----
Decrypted msg:  WEAREDISCOVEREDSAVEYOURSELF

*******Vernam Cipher*******
-----Encryption-----
Msg: Randomkeygeneration
Random key: BSDIYIICVGKINGJXCRD
Cipher: SSQLMUSGTMOVRXJQKFQ
-----Decryption-----
Plain:  RANDOMKEYGENERATION
```

Hill cipher

```
import numpy as np
from mul_inv import multiplicative_inv

uppercase_alphabets = [chr(i) for i in range(65, 91)]

# matrix multiplication 3*3 into 3x1= 3x1
def encrypt():
    cipher_mat= [0]*n   # null matrix initialization
    for i in range(n):
        for j in range(n):
            cipher_mat[i] += key_mat[i][j] * plain_mat[j]
        cipher_mat[i] = cipher_mat[i] % 26     # modular operation 26 to fit alphabets
    return cipher_mat

def minor(mat, i, j):
    # remove row i and column j
    return np.delete(np.delete(mat, i, axis=0), j, axis=1)

def cofactor_matrix(mat):
    cofactors= [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            minor_ij= minor(mat, i, j)
            sign= (-1) ** (i+j)
            cofactors[i][j] = sign * round(np.linalg.det(minor_ij))
            # print(("is cofactors float: ", cofactors[i][j]))
    return cofactors

def adjoint_matrix(mat):
    cofactors= cofactor_matrix(mat)
    return np.transpose(cofactors)

def inverse_matrix(mat):
    adj_A = adjoint_matrix(key_mat)
    # print(f"Adjoint matrix:\n {adj_A}")
    det_inv= multiplicative_inv(np.linalg.det(key_mat), 26)
    if det_inv == None:
        print("The matrix is not invertible MOD 26")
        exit()
    # print(f"mul_inv of {np.linalg.det(key_mat)} is {det_inv}")
    inv_A= det_inv * adj_A
    #  modular operation on inverse matrix
    return inv_A % 26
```

```
def decrypt():
    plain_mat= [0] * n  # null matrix initialization
    inv_key_mat= inverse_matrix(key_mat)
    # print("inverse key matrix with multiplicative inv:\n", inv_key_mat)
    for i in range(n):
        for j in range(n):
            plain_mat[i] += inv_key_mat[i][j] * cipher_mat[j]
        plain_mat[i]= plain_mat[i] % 26
    print("Message matrix", np.round(plain_mat))
    print("Original msg: ",[uppercase_alphabets[round(u)] for u in plain_mat])

# given n= 3
key = 'GYBNQKURP'
plaintext= 'ACT'
n=3

key_mat= [[0] * n for _ in range(n)]
for i in range(n):
    for j in range(n):
        key_mat[i][j]= uppercase_alphabets.index(key[i*n +j])  # (1*3+2= 5)

print("-------Hill Cipher-------")
print(f"keyword: {key} \n"
      f"key matrix: {key_mat}")

plain_mat= [uppercase_alphabets.index(i) for i in plaintext]
print(f"plain text: {plaintext}\n"
      f"plain matrix: {plain_mat}")

cipher_mat= encrypt()
print(f"Cipher matrix: {cipher_mat}\n"
      f"Cipher text:{[uppercase_alphabets[i] for i in cipher_mat]}")

decrypt()
```

```
-------Hill Cipher-------
keyword: GYBNQKURP
key matrix: [[6, 24, 1], [13, 16, 10], [20, 17, 15]]
plain text: ACT
plain matrix: [0, 2, 19]
Cipher matrix: [15, 14, 7]
Cipher text:['P', 'O', 'H']
Message matrix [ 0.  2. 19.]
Original msg:  ['A', 'C', 'T']
```

Lucas Lehmar test

```
# The Lucas–Lehmer Test is a primality test specifically designed for Mersenne numbers, which are numbers of the form:
# Mp= 2^p - 1 where p = prime and to check if Mp is prime:
#  s0= 4 to find sn from 1 to p-2 ; sn= sn-1^2 - 2
def is_prime(a):
    if a<2:
        return False
    for i in range(2, a//2 + 1):
        if a%i == 0:
            return False
    return True
```

```
-----Lucas Lehmar Test (Mersenne Prime Number)----
Enter any positive integer: 2
Not a Mersenne prime number.
Enter any positive integer: 3
Given number is Mersenne Prime.
Enter any positive integer: 31
Given number is Mersenne prime number.
Enter any positive integer:
```

```
def lucas_lehmar(Mp):
    # Mp= 31
    p= 2    # smallest prime only applied for p>= 3 not apply for p = 2
    s= [4]
    while Mp > (2**p -1):
        p += 1

    if Mp == 2**p -1 :
        if is_prime(p):
            for i in range(1, p-1):      # upto p-2
                s.append((s[i-1] ** 2 - 2) % Mp)
            if s[p-2] == 0 :
                print("Given number is Mersenne prime number.")
            elif p == 2:
                print("Given number is Mersenne Prime.")  # lucas test not applicable for p==2
            else:
                print("Composite number.")
        else:
            print('Not a mersenne prime number')
    else:
        # print("p: ", p)
        print("Not a Mersenne prime number.")


print("-----Lucas Lehmar Test (Mersenne Prime Number)----")
while True:
    n= int(input("Enter any positive integer: "))
    lucas_lehmar(n)
```

GCD using Euclidean Algorithm

```
def euclidean_gcd(a,b):
    while (b!=0):
        r= a - a//b * b     # a- int(a/b)*b
        a= b
        b=r
        # print(f"a:{a}, b={b}")
    return a

print("-----GCD using euclidean algorithm-----")
x= int(input("Enter first number: "))
y= int(input("Enter second number: "))

print(f"GCD({x},{y}) = {euclidean_gcd(x,y)}")
```

```
-----GCD using euclidean algorithm-----
Enter first number: 768
Enter second number: 1248
GCD(768,1248) = 96
```

Coprime

```
def gcd(a,b):
    while (b!=0):
        r= a - a//b * b     # a- int(a/b)*b
        a= b
        b=r
    return a

def coprime_check(m,n):
    if gcd(m,n) == 1:
        print(f"{m} and {n} are coprime numbers.")
    else:
        print("Given numbers are not coprime.")

print("------Coprime check-------")
n1= int(input("Enter first number: "))
n2= int(input("Enter second number: "))

coprime_check(n1,n2)
```

```
------Coprime check-------
Enter first number: 34
Enter second number: 13
34 and 13 are coprime numbers.
```

Euler Totient

```
# Illustrate the Euler totient function.
def gcd(a,b):
   while (b!=0):
      r= a - a//b * b     # a- int(a/b)*b
      a= b
      b=r
   return a

def euler_totient(n):
   count =0
   for a in range(1,n):
      if gcd(n,a) == 1:
         count += 1
   return count

while True:
   n= int(input("Enter a number: "))
   print("Euler's totient: ",euler_totient(n))
```

```
Enter a number: 457
Euler's totient:  456
Enter a number: 223
Euler's totient:  222
Enter a number: 645
Euler's totient:  336
Enter a number: 090
Euler's totient:  24
```

Primitive root

```
import random
from RSA import random_prime

def is_primptive_root(a,p):
   ans=[]
   for i in range(1,p):
      ans.append(pow(a,i) % p)
   print("Remainders: ", ans)
   return sorted(ans) == [i for i in range(1,p)]
# a=2; p=11
print("------Primitive root check------")
p= random_prime()
a= random.randint(2,p)
if is_primptive_root(a,p):
   print(f"Yes {a} is primitive root of {p} (modulo)")
else:
   print(f"No {a} is not primitive root of {p} (modulo)")
```

```
------Primitive root check------
Remainders:  [32, 19, 5, 26, 28, 25, 63, 6, 58, 47, 30, 22, 34, 16, 43, 36, 13, 14,
Yes 32 is primitive root of 67 (modulo)
```

RSA

```
import random
def gcd(a,b):
    while b!=0:
        r= a - a//b * b     # a- int(a/b)*b
        a= b
        b=r
    return a

def random_prime():
    p= random.randint(0,100)
    while not is_prime(p):
        p = random.randint(0, 100)
    # print("prime: ",p)
    return p

def is_prime(a):
    if a<2:
        return False
    for i in range(2, a//2 + 1):
        if a%i == 0:
            return False
    return True

global e,n
def decryption():
    # private key generation
    while True:
        p, q = random_prime(), random_prime()   # random prime numbers p != q
        if p != q:
            break
    print(f"private key: \np={p} \tq={q}")
    global e,n  # public key
    n= p*q
    totient_n = (p-1) *(q-1) # as both prime (totient_n = euler_totient(n))
    # choose e as (1 < e < totient_n)
    # e= random.randint(1,totient_n)    # can be computationally heavy instead use loop so start fron 2
    e,d = 2,2
    while True:
        # print("d: ",d)
        if d >= totient_n:
            e+=1
            d=2 # start from beginning
            # print("e: ", e)
```

```
        if d*e - d*e//totient_n * totient_n == 1:
            # print("d: ", d)
            break
        d += 1
    # print("d: ",d)

    C= encryption()
    M= pow(C,d)
    M= M- M//n *n   # modular operation: M^d mod n
    # print("M: ", M)
    print(f"Decryption: d={d} \n\tMsg: {M} \n\tCipher: {C}")

def encryption():
    M= random.randint(0,n) # message < n
    C= pow(M,e)
    C= C - int(C/n) *n
    print(f"Encryption: public key: e={e}, n={n} \n\tMsg: {M} \n\tCipher: {C}")
    return C

if __name__ == '__main__':
    print("----------RSA Algorithm----------")
    decryption()
```

```
----------RSA Algorithm----------
private key:
p=11    q=61
Encryption: public key: e=7, n=671
    Msg: 592
    Cipher: -1096
Decryption: d=343
    Msg: 592
    Cipher: -1096
```

Diffe Hellman Key Exchange

```
import random
from RSA import random_prime, is_prime

def primitive_root(p):
    totient_p= p-1 # prime no
    pr = 0  # primitive root
    # prime factor of (p-1)
    for i in range(2,totient_p//2+1):
        found = 0
        if totient_p%i ==0: # if factor
            if is_prime(i): # factor and is prime
                pow_root= totient_p/i
                # let's test for primitive root g form 2 to p-1
                for g in range(2,p):
                    if pow(g,pow_root) % p != 1:
                        # print("primitive root: ", g)
                        # print("pow_root root: ", pow_root)
                        pr=g
                        found=1
                        break
        if found:
            break
    return pr

if __name__ == "__main__":
    q= random_prime()
    a= primitive_root(q)   # primitive root of q

    X_a= random.randint(2,q)    # less than q-1
    Y_a= pow(a, X_a) % q     # public a, X_a, q

    X_b= random.randint(2,q)
    Y_b= pow(a, X_b) % q

    K_a= pow(Y_b, X_a) % q
    K_b= pow(Y_a, X_b) % q

    print(f"-----Deffie Hellman Algorithm----- \n"
          f"q: {q} \t primitive root(alpha): {a} \n"
          f"For User A: X_a= {X_a}, Y_a= {Y_a} K_a= {K_a} \n"
          f"For User B: X_a= {X_b}, Y_a= {Y_b} K_b= {K_b}")
```

```
-----Deffie Hellman Algorithm-----
q: 41    primitive root(alpha): 3
For User A: X_a= 34, Y_a= 9 K_a= 9
For User B: X_b= 29, Y_b= 38 K_b= 9
```

Multiplicative Inverse using Extended Euclidean Algorithm

```
def gcd(a,b):
    while (b!=0):
        r= a - a//b * b     # a- int(a/b)*b
        a= b;       b=r
    return a

def multiplicative_inv(a,b):
    """returns multiplicative inverse of a mod b"""
    if gcd(a,b) == 1:
        x1, x2 = 1, 0
        while b != 0:
            r = a % b;
            q = a // b
            a = b; b = r
            x = x1 - q * x2
            x1 = x2;          x2 = x
            # print(f"q: {q},a:{a}, b={b}, x1={x1}, x2={x2}, x={x} ")
        return x1
if __name__ == "__main__":
    print(f"-----Multiplicative inverse using Extended Euclidean Algorithm-----")
    # A= random.randint(1,100); M= random.randint(1,100)
    A= int(input("Enter dividend: "))
    M = int(input("Enter Modulo: "))
    mul_inv= multiplicative_inv(A,M)

    if mul_inv == None:
        print("GCD(A,B) is not 1. So they are not coprime and there is not multiplicative inverse.")
    elif mul_inv < 0:
        i=1
        while i*M + mul_inv < 0:    # as mul_inv is -ve ---> % operator can do so
            i+= 1
        mul_inv= i*M + mul_inv
    print(f"Multiplicative inverse of {A} mod {M} is {mul_inv}")
```

```
-----Multiplicative inverse using Extended Euclidean Algorithm-----
Enter dividend: 23
Enter Modulo: 35
Multiplicative inverse of 23 mod 35 is 32

Process finished with exit code 0
```

elGamal Algorithm

```
import random
from mul_inv import multiplicative_inv
from RSA import random_prime
from diffie_hellman import primitive_root

def generate_key():
    X_a = random.randint(2, q-1)  # less than q-1
    global Y_a
    Y_a = pow(a, X_a) % q  # public a, X_a, q
    return X_a

def decryption():
    X_a= generate_key()
    c1,c2= encryption()
    K= pow(c1, X_a) % q
    M= (c2* multiplicative_inv(K,q)) % q
    print(f"Decryption: K={K} \n\tMsg: {M} \n\tC2: {c2}")

def encryption():
    M = random.randint(0, q)  # message < q
    k = random.randint(0, q)  # less than q
    K = pow(Y_a, k) % q
    C1= pow(a,k) % q
    C2= (K*M) % q
    # cipher (C1, C2) public
    print(f"Encryption: \n\tMsg: {M} \n\tK:{K} \n\t(C1,C2): {C1,C2}")
    return C1, C2

global Y_a
q= random_prime()
a= primitive_root(q)   # primitive root of q
if __name__ == '__main__':
    print('------Elgamal CryptoSystem------')
    decryption()
```

```
------Elgamal CryptoSystem------
Encryption:
    Msg: 55
    K:26
    (C1,C2): (6, 8)
Decryption: K=26
    Msg: 55
    C2: 8
```