EXPERIMENT NUMBER: 01

CIPHER PROFILE: -

NAME - CAESER CYPHER

CATEGORY- Mono Alphabetic cypher

DESCRIPTION- Each letter of a given text is replaced by a letter some fixed number of positions down the alphabet. Here it is 3.
Thus to cipher a given text we need an integer value, known as shift which indicates the number of positions each letter of the text has been moved down.

mathematically it can be represented as

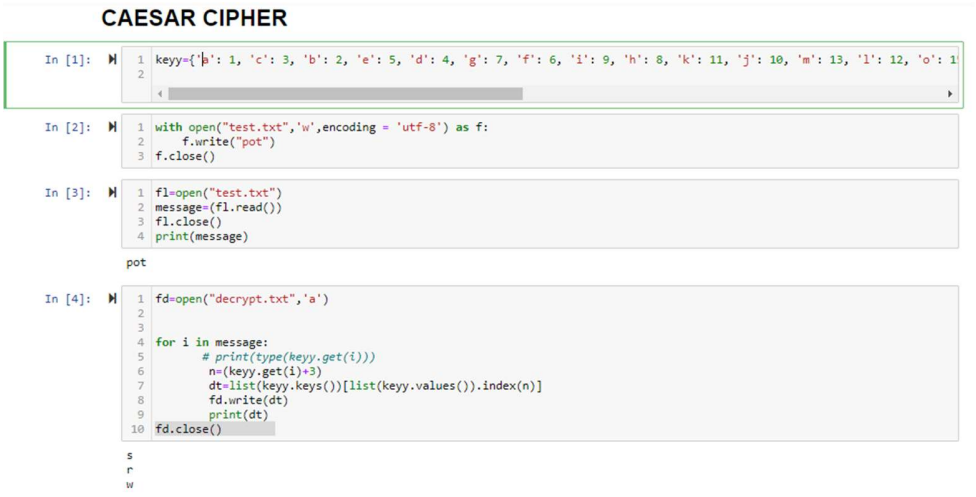
$$E_n(x) = (x + n) mod\ 26$$
(Encryption Phase with shift n)

$$D_n(x) = (x - n) mod\ 26$$
(Decryption Phase with shift n)

Here “n” is 3.

CODE-

ENCRYPTION:



DECRYPTION :

In [5]:

```
fd=open("decrypt.txt",'a')


for i in message:
       # print(type(keyy.get(i)))
        n=(keyy.get(i)-3)
        dt=list(keyy.keys())[list(keyy.values()).index(n)]
        fd.write(dt)
        print(dt)
fd.close()
```

```
m
l
q
```

==========

EXPERIMENT NUMBER: 02

CIPHER PROFILE: -

NAME – VIGENER CYPHER

CATEGORY- Poly Alphabetic cypher

DESCRIPTION- It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets .The encryption of the original text is done using the Vigenère square or Vigenère table.

- The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers.
- At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
- The alphabet used at each point depends on a repeating keyword.

ENCRYPTION-

VIGENER CIPHER

In [2]:
```
key=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v
k=['s','u','n','f','l','o','w','e','r']
```

In [3]:
```
m=input("write a message: ")
```

```
write a message: elephant
```

In [5]:
```
#ENCRYPTION
c=0
kk=[]
mm=[]
cc=[]
for i in  m:
    ky=key.index(i)
    kk.append(ky)
    #print(ky)
    ms=key.index(k[c%len(k)])
    mm.append(ms)
    #print(c)
    a=ky+ms
    cc.append(a)
    print(key[a%26])

    c +=1

```

```
w
f
r
u
s
o
j
x
```

DECRYPTION:

In [6]:

```
#DECRYPTION
cnt=0
for j in cc:
    dm=cc[cnt]-mm[cnt]
    print(key[dm%26])
    cnt +=1
```

```
e
l
e
p
h
a
n
t
```

==========

EXPERIMENT NUMBER: 03

CIPHER PROFILE: -

NAME- SUBSTITUTION CYPHER

CATEGORY-Mono Alphabetic cypher

DESCRIPTON- In a Substitution cipher, any character of plain text from the given fixed set of characters is substituted by some other character from the same set depending on a key. For example with a shift of 1, A would be replaced by B, B would become C, and so on.

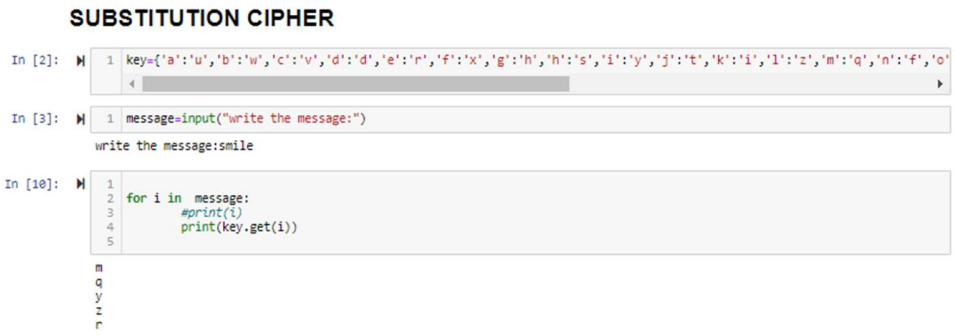
it can mathematically written as

$$E_n(x) = (x + n) mod\ 26$$
(Encryption Phase with shift n)

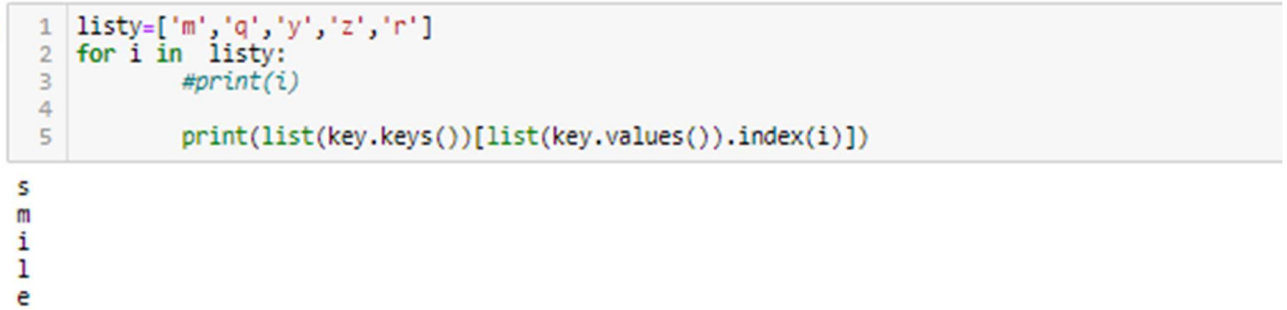
$$D_n(x) = (x - n) mod\ 26$$
(Decryption Phase with shift n)

CODE-

ENCRYPTION:



DECRYPTION:



==========

EXPERIMENT NUMBER: 04

CIPHER PROFILE: -

NAME- TRANSPOSITION CYPHER

CATEGORY- Private key cryptography

DESCRIPTION-In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

1. The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.
2. Width of the rows and the permutation of the columns are usually defined by a keyword.
3. For example, the word HACK is of length 4 (so the rows are of length 4), and the permutation is defined by the alphabetical order of the letters in the keyword. In this case, the order would be "3 1 2 4".
4. Any spare spaces are filled with nulls or left blank or placed by a character (Example: _).
5. Finally, the message is read off in columns, in the order specified by the keyword.

Encryption

Given text = Geeks for Geeks

Keyword = HACK **Length of Keyword** = 4 (no of rows) **Order of Alphabets in HACK** = 3124

H	A	C	K
3	1	2	4
G	e	e	k
s	_	f	o
r	_	G	e
e	k	s	_

Print Characters of column 1,2,3,4

Encrypted Text = e kefGsGsrekoe_

Decryption

1. To decipher it, the recipient has to work out the column lengths by dividing the message length by the key length.
2. Then, write the message out in columns again, then re-order the columns by reforming the key word.

CODE-

a) Encryption:

```
import math

key = "trans"

# 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
```

b) Decryption:

```
# 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.")
```

OUTPUT-

```
    # 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 = "Elephant"

cipher = encryptMessage(msg)
print("Encrypted Message: {}".
                format(cipher))

print("Decryped Message: {}".
      format(decryptMessage(cipher)))
```

```
Encrypted Message: etp_lnh_Ea
Decryped Message: Elephant
```

==========

EXPERIMENT NUMBER: 05

CIPHER PROFILE: -

NAME- PLAY FAIR CYPHER

CATEGORY- Symmetric key cryptography

DESCRIPTION- The Playfair cipher uses a 5 by 5 table containing a key word or phrase.

To generate the key table, one would first fill in the spaces in the table (a modified Polybius square) with the letters of the keyword (dropping any duplicate letters), then fill the remaining spaces with the rest of the letters of the alphabet in order (usually omitting "J" or "Q" to reduce the alphabet to fit; other versions put both "I" and "J" in the same space). The key can be written in the top rows of the table, from left to right, or in some other pattern, such as a spiral beginning in the upper-left-hand corner and ending in the center. The keyword together with the conventions for filling in the 5 by 5 table constitute the cipher key.

To encrypt a message, one would break the message into digrams (groups of 2 letters) such that, for example, "HelloWorld" becomes "HE LL OW OR LD". These digrams will be substituted using the key table. Since encryption requires pairs of letters, messages with an odd number of characters usually append an uncommon letter, such as "X", to complete the final digram. The two letters of the digram are considered opposite corners of a rectangle in the key table. To perform the substitution, apply the following 4 rules, in order, to each pair of letters in the plaintext:

1. If both letters are the same (or only one letter is left), add an "X" after the first letter. Encrypt the new pair and continue. Some variants of Playfair use "Q" instead of "X", but any letter, itself uncommon as a repeated pair, will do.
2. If the letters appear on the same row of your table, replace them with the letters to their immediate right respectively (wrapping around to the left side of the row if a letter in the original pair was on the right side of the row).
3. If the letters appear on the same column of your table, replace them with the letters immediately below respectively (wrapping around to the top side of the column if a letter in the original pair was on the bottom side of the column).
4. If the letters are not on the same row or column, replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair. The order is important – the first letter of the encrypted pair is the one that lies on the same row as the first letter of the plaintext pair.

To decrypt, use the inverse (opposite) of the last 3 rules, and the first as-is (dropping any extra "X"s or "Q"s that do not make sense in the final message when finished).

CODE-

OUTPUT-

```
key=['K','H','O','R','G','O','S','H']
```

```
import numpy as np
```

```
# 2D array
rows, cols = (5,5)
arr = [[0 for i in range(cols)] for j in range(rows)]
print(arr)
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

```
alpha=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y
k=[]
k=key+alpha
k = list(dict.fromkeys(k))
print(k)
```

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

```
listy = [[] for i in range(5)]
c=0
for i in range(5):

    for j in range(5):
        listy[i].append(k[c])
        c+=1
        if c == len(k):
            break
print(listy)
```

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

```
c = np.matrix(listy)
print(c)

mat=c.tolist()
print(mat)
```

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

```
pt=input("ENTER THE PLAIN TEXT ..PLEASE INSERT WITH CAPITAL LETTERS ANd please avoid speces :) ")
```

```
ENTER THE PLAIN TEXT ..PLEASE INSERT WITH CAPITAL LETTERS ANd please avoid speces :) MAN
```

```
a=[]
for i in pt:
    i,j = np.where(c==i)
    print(i,j)
    a.append(i)
    a.append(j)
print(a)
#print(type(a))
```

```
[2] [4]
[1] [1]
[3] [0]
[0] [4]
[0] [2]
[array([2], dtype=int32), array([4], dtype=int32), array([1], dtype=int32), array([1], dtype=int32), array([3], dtype=int32), a
rray([0], dtype=int32), array([0], dtype=int32), array([4], dtype=int32), array([0], dtype=int32), array([2], dtype=int32)]
```

```
bnd=len(a)
```

```
bnd=len(a)
```

```
#print(type(a[11]))
```

```
for i in range(5):
    ii=(3+i*4)
    jj=(0+i*4)
    iii=(1+i*4)
    jjj=(2+i*4)
    mi=int(a[ii])
    mii=int(a[iii])
    mjj=int(a[jjj])
    mj=int(a[jj])
    print(mat[mii][mjj])
    print(mat[mi][mj])

```

```
W
B
K
```

```
dt=input("ENTER THE ENCRYPTED TEXT")
```

```
ENTER THE ENCRYPTED TEXTWBK
```

```
a=[]
for i in dt:
    i,j = np.where(c==i)
    print(i,j)
    a.append(i)
    a.append(j)
print(a)
print(type(a))

for i in range(5):
    ii=(3+i*4)
    jj=(0+i*4)
    iii=(1+i*4)
    jjj=(2+i*4)
    mi=int(a[ii])
    mii=int(a[iii])
    mjj=int(a[jjj])
    mj=int(a[jj])
    print(mat[mi][mj])
    print(mat[mii][mjj])

```

```
[4] [1]
[1] [2]
[0] [0]
[4] [3]
[array([4], dtype=int32), array([1], dtype=int32), array([1], dtype=int32), array([2], dtype=int32), array([0], dtype=int32), a
rray([0], dtype=int32), array([4], dtype=int32), array([3], dtype=int32)]
<class 'list'>
M
A
N
```

==========

EXPERIMENT NUMBER: 05

CIPHER PROFILE: -

NAME- EUCLIDIAN ALGO

CATEGORY- For finding GCD of Numbers

DESCRIPTION-

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers, a and b.

Formal description of the Euclidean algorithm

- Input Two positive integers, a and b.
- Output The greatest common divisor, g, of a and b.
- Internal computation
    1. If a<b, exchange a and b.
    2. Divide a by b and get the remainder, r. If r=0, report b as the GCD of a and b.
    3. Replace a by b and replace b by r. Return to the previous step.

CODE-

```
def gcd(a,b):
    if a > b :
        r = a%b
        if r > 0 :
            a = b
            b = r
            gcd(a,b)
        else:
            print(b)
gcd(1750,550)
```

OUTPUT-

```
>>>
========
50
>>>
```

==========

EXPERIMENT NUMBER: 06

CIPHER PROFILE: -

NAME- EXTENDED EUCLEDIAN ALGO

CATEGORY-

DESCRIPTION- It is used for finding the greatest common divisor of two positive integers a and b and writing this greatest common divisor as an integer linear combination of a and b .

The extended Euclidean algorithm is particularly useful when a and b are coprime (or gcd is 1). Since x is the modular multiplicative inverse of "a modulo b", and y is the modular multiplicative inverse of "b modulo a". In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

CODE-

```
x=[1,0]
y=[0,1]
q = []
r = []
def gcd(a,b):
    temp = (a*x[0])+(b*y[0])
    r.append(temp)
    temp = (a*x[1])+(b*y[1])
    r.append(temp)
    print("r\tq\tx\ty")
    print(r[-2],"\t\t",x[-2],"\t",y[-2])
    print(r[-1],"\t\t",x[-1],"\t",y[-1])
    while True:
        temp = int(r[-2] % r[-1])
        r.append(temp)
        temp = int(r[-3] / r[-2])
        q.append(temp)
        temp = x[-2] - (q[-1] * x[-1])
        x.append(temp)
        temp = y[-2] - (q[-1] * y[-1])
        y.append(temp)
        print(r[-1],"\t",q[-1],"\t",x[-1],"\t",y[-1])
        if r[-1] == 0:
            print("\n\n\nx : ",x[-2]," y : ",y[-2])
            break

gcd(1759,550)
```

OUTPUT-

```
========== RESTART: C
r       q       x       y
1759             1       0
550              0       1
109      3       1       -3
5        5       -5      16
4        21      106     -339
1        1       -111    355
0        4       550     -1759



x :  -111  y :  355
>>> |
```

==========

EXPERIMENT NUMBER: 07

CIPHER PROFILE: -

NAME- DIFFIE HELMAN

CATEGORY- Public key cryptography

DESCRIPTION- Diffie Hellman (DH) key exchange algorithm is a method for securely exchanging cryptographic keys over a public communications channel. Keys are not actually exchanged – they are jointly derived. It is named after their inventors Whitfield Diffie and Martin Hellman.

In static-static mode, both Alice and Bob retain their private/public keys over multiple communications. Therefore, the OUTPUT shared secret will be the same every time. In ephemeral-static mode one party will generate a new private/public key every time, thus a new shared secret will be generated.

CODE-

Step 1: Alice and Bob get public numbers P = 23, G = 9

Step 2: Alice selected a private key a = 4 and Bob selected a private key b = 3

Step 3: Alice and Bob compute public values Alice: x =(9^4 mod 23) = (6561 mod 23) = 6 Bob: y = (9^3 mod 23) = (729 mod 23) = 16

Step 4: Alice and Bob exchange public numbers

Step 5: Alice receives public key y =16 and Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys Alice: ka = y^a mod p = 65536 mod 23 = 9 Bob: kb = x^b mod p = 216 mod 23 = 9

Step 7: 9 is the shared secret.

For allice

```
p=23 g=9
a=4 x=(g**a)%p print(x)
```

for bob

```
b=3
y=(g**b)%p
print(y)
```

16

alice

```
pubkA=y
symkeyA=(pubkA**a )%p
print(symkeyA)
```

9

bob

```
pubkB=x
symkeyB=(pubkB**b )%p
print(symkeyB)
```

9

==========

EXPERIMENT NUMBER: 08

CIPHER PROFILE: -

NAME- HILL CYPHER

CATEGORY- Polygraphy substitution cypher or block cypher

DESCRIPTION- Hill cipher is a polygraphy substitution cipher based on linear algebra. Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

CODE-

```
keyy={'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'g': 6, 'f': 5, 'i': 8, 'h': 7, 'k': 10, 'j': 9, 'm': 12, 'l': 11, 'o': 14, 'r
```

```
pt=input("enter the plain text use lower case: ")
```

```
enter the plain text use lower case: vuy
```

```
a=len(pt)
```

```
txt=[]
for i in pt:
    txt.append(keyy[i])

```

```
import random
aa=[]

while 1:
    b=random.randint(1,16)
    if b not in aa:
        aa.append(b)
    if len(aa)==(a*a):
        break
```

```
listy = [[] for i in range(a)]
c=0
for m in range(a):

    for n in range(a):
        listy[m].append(aa[c])
        c+=1
        if c == len(aa):
            break
print(listy)
```

```
[[15, 14, 6], [7, 1, 16], [11, 5, 8]]
```

```
cols = (a)
ary = [0 for i in range(cols)]
for i in range(len(listy)):
    for j in range(len(txt)):
        ary[j] += (listy[i][j] * txt[j] )
anss = [0 for i in range(cols)]
for ii in range (a):
    anss[ii] +=ary[ii]%26
print(anss)

```

```
[17, 10, 18]
```

```
ans = [0 for i in range(cols)]
```

```
# list out keys and values separately
key_list = list(keyy.keys())
val_list = list(keyy.values())
for jj in anss:
     print(key_list[val_list.index(jj)])



```

```
r
k
s
```

#decryption

```
import numpy as np
m = np.array(listy)
print("Original matrix:")
print(m)
result =  np.linalg.inv(m)
print("Inverse of the said matrix:")
print(result)
```

```
Original matrix:
[[15 14  6]
 [ 7  1 16]
 [11  5  8]]
Inverse of the said matrix:
[[-0.09677419 -0.11021505  0.29301075]
 [ 0.16129032  0.07258065 -0.26612903]
 [ 0.03225806  0.1061828  -0.11155914]]
```

```
mat=result.tolist()
print(mat)

rows, cols = (a,a)
dky = [[0 for i in range(cols)] for j in range(rows)]
for i in range(a):
    print(i)
    for j in range(a):
        dky[i][j] += (mat[i][j] % 26 )
print(dky)
```

```
0
1
2
[[25.903225806451612, 25.88978494623656, 0.2930107526881721], [0.16129032258064518, 0.07258064516129033, 25.733870967741936],
[0.03225806451612904, 0.10618279569892473, 25.888440860215052]]
```

```
et=input("Enter the encrypted message")
```

```
Enter the encrypted messagerks
```

```
dtxt=[]
for i in pt:
    dtxt.append(keyy[i])
```

```
cols = (a)
aryy = [0 for i in range(cols)]
for i in range(len(dky)):
    for j in range(len(dtxt)):
        aryy[j] += (dky[i][j] * dtxt[j] )
danss = [0 for i in range(cols)]
for ii in range (a):
    danss[ii] +=ary[ii]%26
print(danss)
```

```
[0, 0, 0]
```

==========