Ex:no:1 Caeser Cipher

Aim: To Implement caeser cipher using python programming language.

Algorithm:

1. Encryption.
   - choose a shift value k (key)
   - For each letter in plaintext
     * shift it forward by k positions in the alphabet
     * Wrap around to the beginning if the end of the alphabet is reacher.

2. Decryption
   - To decrypt, shift each letter in the ciphertext backward by k position

3. The shift is applied only to alphabetic characters, and the case is preserved. Non-alphetic characters remain unchanged.

Program.

```
def caesar_encrypt(text, shift):
    result = ""
    for char in text:
        if char.isupper():
            result += chr((ord(char) - 65 + shift) % 26) + 65)
        elif char.islower():
            result += chr((ord(char) - 97 + shift) % 26) + 97)
        else:
```

```
            result += char
        return result

def caeser_decrypt(text, shift):
    result = ""
    for char in text:
        if char.isupper():
            result += chr((ord(char) - 65 - shift) % 26 + 65)
        elif char.islower():
            result += chr((ord(char) - 97 - shift) % 26 + 97)
        else:
            result += char
    return result

plaintext = input("Enter text:")
shift = 3
encrypted_text = caeser_encrypt(plaintext, shift)
decrypted_text = caeser_decrypt(encrypted_text, shift)
print(f"Plaintext : {plaintext}")
print(f"Encrypted : {encrypted_text}")
```

Output:

Enter text: Hello, world!
Encrypted : Khoor, Zruog!

Result:

Hence the caeser cipher has been executed successfully and the encrypted text is received

Ex:No:2 Basic Mono Alphabetic Cipher

Aim:

To implement the basic monoalphabetic cipher using python program.

Algorithm:

1. Generate cipher key to take the shift value as input and generate a monoalphabetic cipher key
2. The alphabet combined with remaining part of alphabet create a one to one mapping between orginal and shipted letter
3. Encrypt message the key generates by the previous function. It iterates throug each character this resulting encrypted message is returned.
4. Decrypt cipher text similar to the encryption
5. It creates a reverse key by swapping key and value of orginal key.

Program:

```
import string
class Monoalphabetic Cipher:
    def __init__(self, key):
        self.key = key
        self.alphabet = string.ascii_lowercase
    def encrypt(self, plaintext):
        translation_table = str.maketrans(self.alphabet, self.key)
```

```
        ciphertext = plaintext.lower().translate(translation_table)
        return ciphertext

    def decrypt(self, ciphertext):
        reverse_translation_table = str.maketrans(self.key, self.
                                                  alphabet)
        plaintext = ciphertext.translate(reverse_translation_table)
        return plaintext

if __name__ == "__main__":
    key = "qwertyuiopasdfghjklzxcvbnm"
    cipher = MonoalphabeticCipher(key)
    plaintext = "helloworld"
    ciphertext = cipher.encrypt(plaintext)
    decrypted_text = cipher.decrypt(ciphertext)
    print(f"Plaintext: {plaintext}")
    print(f"Ciphertext: {ciphertext}")
    print(f"Decrypted: {decrypted_text}")
```

Output:

```
Plaintext: hello world
Ciphertext: itssg vgksr
Decrypted: hello world
```

Result:

To implement the basic monoalphabetic cipher has implement in python program successfully.

Ex:No:3 Hill cipher

Aim:

To implement the hill cipher algorithm for encrypting a plaintext message using matrix-based substitution.

Algorithm:

1. Input
   - A plaintext message (length should be a multiple of the matrix size, or padded)
   - A square key matrix (e.g. 2x2, 3x3)
   - Ensure the key matrix is invertible modulo 26
2. Preprocessing
   - Convert plaintext into numerical equivalent
   - Pad the plaintext with 'X' if its length is not a multiple of the matrix size.
3. Encryption
   - Divide plaintext into blocks matching the size of the key matrix
   - Multiply each block vector with the key matrix (mod 26)
   - Convert the resulting numerical vectors back to letters
4. Decryption
   - Calculate the inverse of the key matrix modulo 26
   - Multiply each ciphertext block with the inverse key matrix (mod 26) to retrieve plaintext.

Example:

PlainText: "HELP"

Cipher Text: ?

Key: DCDF

Step 1: Convert key matrix into Numerical matrix form

$$DCDF: \begin{bmatrix} D & D \\ C & F \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

Step 2:

Convert plainText into Matrix form

$$HELP \Rightarrow \begin{bmatrix} H \\ E \end{bmatrix} \Rightarrow \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} L \\ P \end{bmatrix} \Rightarrow \begin{bmatrix} 11 \\ 15 \end{bmatrix}$$

Step 3: Encryption formula : $C = (P * K) \bmod 26$

$$\begin{bmatrix} 7 \\ 4 \end{bmatrix} \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 33 \\ 34 \end{bmatrix} \bmod 26 = \begin{bmatrix} 7 \\ 8 \end{bmatrix} \Rightarrow \begin{bmatrix} H \\ I \end{bmatrix}$$

$$\begin{bmatrix} 11 \\ 15 \end{bmatrix} \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 78 \\ 97 \end{bmatrix} \bmod 26 = \begin{bmatrix} 0 \\ 19 \end{bmatrix} \Rightarrow \begin{bmatrix} A \\ T \end{bmatrix}$$

Cipher Text = HIAT

Program:

```
import numpy as np
def hill_cipher_encrypt(plaintext, key):
    key = np.array(key)
    n = key.shape[0]
    assert key.shape[0] == key.shape[1], 'key Matrix must be square'
    det = int(round(np.linalg.det(key))) % 26
    if np.gcd(det, 26) != 1:
        raise ValueError("key Matrix not invertable")
    plaintext = plaintext.upper().replace(" ", "")
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    plaintext_nums = [alphabet.index(c) for c in plaintext]
    while len(plaintext_nums) % n != 0:
        plaintext_nums.append(alphabet.index('x'))
```

```
    plaintext_matrix = np.array(plaintext_nums).reshape(-1,3)
    ciphertext_matrix = (np.dot(plaintext_matrix, key) % 26).astype(int)
    ciphertext = "".join(alphabet[num] for num in ciphertext_matrix.flatten())
    return ciphertext

plaintext = input("Enter the plaintext:")
key = [[6,24,1], [13,16,10], [20,17,15]]
ciphertext = hill_cipher_encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
```

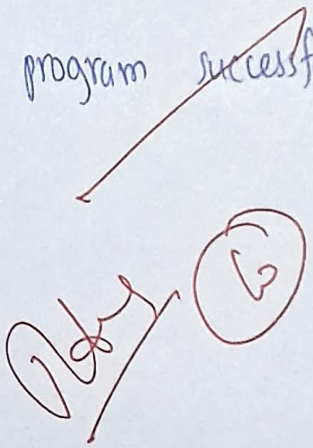
Output :

Enter the plaintext : HELLO

ciphertext : DXTUML

Result :

The program successfully implements the hill cipher



Ex:No:4 Vigenere Cipher

Aim:

To implement the Vigenere Cipher algorithm for encrypting a plaintext message using vigenere table

Algorithm:

Encryption

1. Input
   - Plaintext message P
   - A Keyword K
2. Repeat keyword: Extend K to match the length of P
3. Encryption Rule:
   - For each letter in P:
     - Use the Vigenere table to find the intersection of the row corresponding to the plaintext letter and the column corresponding to the keyword letter.
     - Replace the plaintext letter with the letter at the intersection

Decryption

1. Input
   - Ciphertext C
   - Keyword K
2. Repeat keyword: Extend K to match the length of C
3. Decryption Rule:
   - For each letter in C:
     - Use the Vigenere table to find the row corresponding to the keyword letter.
     - Locate the ciphertext letter in this row and determine the column index. The column index corresponds to the plaintext letter.

Example:

Vigenere table

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

PlainText: Hello

Keyword: key

Encryption:

Raw (Keyword) - K E Y K E

Column (Plaintext) - H E L L O

Ciphertext (Intersection) - R I J V S

Explanation :

1. H (Plaintext) + K (Keyword).
   - Row = K
   - Column = H
   - Intersection = R
2. E (Plaintext) → E (Keyword):
   - Row = E
   - Column = E
   - Intersection = I
3. L (Plaintext) + Y (Keyword):
   - Row = Y
   - Column = L
   - Intersection = J
4. L (Plaintext) + K (Keyword):
   - Row = K
   - Column = L
   - Intersection = V
5. O (Plaintext) + E (Keyword):
   - Row = E
   - Column = O
   - Intersection = S

Program:

```
def generate_vigenere_table():
    table = []
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    for i in range(26):
        table.append(alphabet[i:] + alphabet[:i])
    return table

def vigenere_encrypt(plaintext, keyword):
    plaintext = plaintext.upper().replace(" ", "")
    keyword = keyword.upper()
```

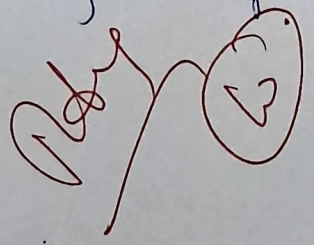
```
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
extended_keyword = (keyword * (len(ciphertext) // len(keyword) + 1))[:len(
                                                        ciphertext)]
plaintext = ""
for c, k in zip(ciphertext, extended_keyword):
        c_idx = alphabet.index(c)
        k_idx = alphabet.index(k)
        p_idx = (c_idx - k_idx + 26) % 26
        plaintext += alphabet[p_idx]
    return plaintext
plaintext = input("Enter the plaintext:")
keyword = "KEY"
ciphertext = vigenere_encrypt(plaintext, keyword)
print("Cipher text :", ciphertext)
decrypted_text = vigenere_decrypt(ciphertext, keyword)
print("Decrypted Text: ", decrypted_text)
```

Output:

```
Enter the plaintext: HI
Ciphertext : RM
Decrypted Text : HI
```

Result:

The program successfully encrypts and decrypts messages using the Vigenère cipher based on the Vigenère table.



Ex. No:

Data Encryption Standard (DES)

Aim: To implement the data Encryption Standard (DES) algorithm for encrypting and decrypting messages

Algorithm:

1. Key Generation
   - The initial 64-bit key is reduced to 56 bits using a permutation choice (PC-1)
   - The key is split into two 28-bit halves
   - for 16 rounds, both halves are shifted left and compressed to 48 bits using PC-2
2. Initial Permutation (IP)
   - The 64-bit plaintext undergoes an initial permutation (IP)
3. Round Function (16 Rounds)

   For each round:
   - The 64-bit block is split into left (L) and right (R) halves
   - The right half (R) goes through:
     1. Expansion (E) box: Expands 32-bit R to 48 bits
     2. Key mixing: XOR with a 48-bit subkey.
     3. Substitution (S-boxes): Compressed back to 32 bits
     4. Permutation (P-box): Rearranged for diffusion.
     5. XOR with Left half (L).
     6. Swap L and R.
4. Final Permutation (FP).
   - After 16 rounds, the left and right halves are combined and undergo the final permutation (FP) to produce the 64-bit ciphertext

Example:

i) Key = 10100 00010

P_{10} (Permutate)

IP	1	2	3	4	5	6	7	8	9	10
OP	3	5	2	7	4	10	1	9	8	6

P8 (Permutate)

I/P	1	2	3	4	5	6	7	8	9	10
O/P	6	3	7	4	8	5	10	9	-	-

IP

I/P	1	2	3	4	5	6	7	8
O/P	2	6	3	1	4	8	5	7

EP

I/P	1	2	3	4				
O/P	4	1	2	3	2	3	4	1

P4

I/P	1	2	3	4
O/P	2	4	3	1

$$S_0 = \begin{bmatrix} 01 & 00 & 11 & 10 \\ 11 & 10 & 01 & 00 \\ 00 & 10 & 01 & 11 \\ 11 & 01 & 11 & 10 \end{bmatrix}$$

$$S_1 = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 10 & 00 & 01 & 11 \\ 11 & 00 & 01 & 00 \\ 10 & 01 & 00 & 11 \end{bmatrix}$$

Plain Text = 01110010

Soln:

Key = 10100 00010

P10 = 10000 (LHK) 01100 (RHK)

LS -1 = 00001 11000

P8 = 10100100

Key 1 = 10100100

Plaintext = 0111 0010

IP = 1010 (LH) 1001 (RH)

EP = 1100 0011

Key1 = 1010 0111

XOR = 0110 0111

S0 = 0110 => 10

S1 = 0111 => 11

P4 = 0111

LH = 1010

XOR = 1101

Ciphertext : 10011101

Program:

```
from Crypto.Cipher import DES
import binascii
def pad(text):
    while len(text) % 8 != 0:
        text += ' '
    return text
def des_encrypt(plaintext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    padded_text = pad(plaintext)
    ciphertext = cipher.encrypt(padded_text.encode())
    return binascii.hexlify(ciphertext).decode()
def des_decrypt(ciphertext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_text = cipher.decrypt(binascii.unhexlify(ciphertext))
    return decrypted_text.decode().strip()

plaintext = input("Enter :")
key = b"8BYTEKEY"
print("Ciphertext:", des_encrypt(plaintext, key))
print("Decrypted Text :", des_decrypt(ciphertext, key))
```

Output:

```
Please enter any 8 character string: hellodes
Ciphertext: 2f21ad1eb16c516f
Decrypted text: hellodes
```

Result:

The DES algorithm successfully encrypts and decrypts plaintext using 64-bit blocks and a 56-bit key (8-byte key with parity bits).

Ex: No:

Advanced Encryption Standard (AES)

Aim:

To implement the AES (Advanced Encryption Standard) algorithm for encrypting and decrypting message using a symmetric key.

Algorithm:

1. Key Expansion
   - The key is expanded into multiple round keys using a key schedule
2. Initial Round
   - Add Round key: XOR the plaintext Block with the first round kg.
3. Main Rounds
   - Subbytes: A non-linear substitution using the S-Box
   - ShiftRows: Circular byte shifting in each row
   - Mixcolumns: A mathematical mixing operation (except in the last round)
   - AddRoundKey: XOR with the corresponding round key.
4. Final Round.
   - Subbytes
   - Shift Rows.
   - Add Round key.

Example:

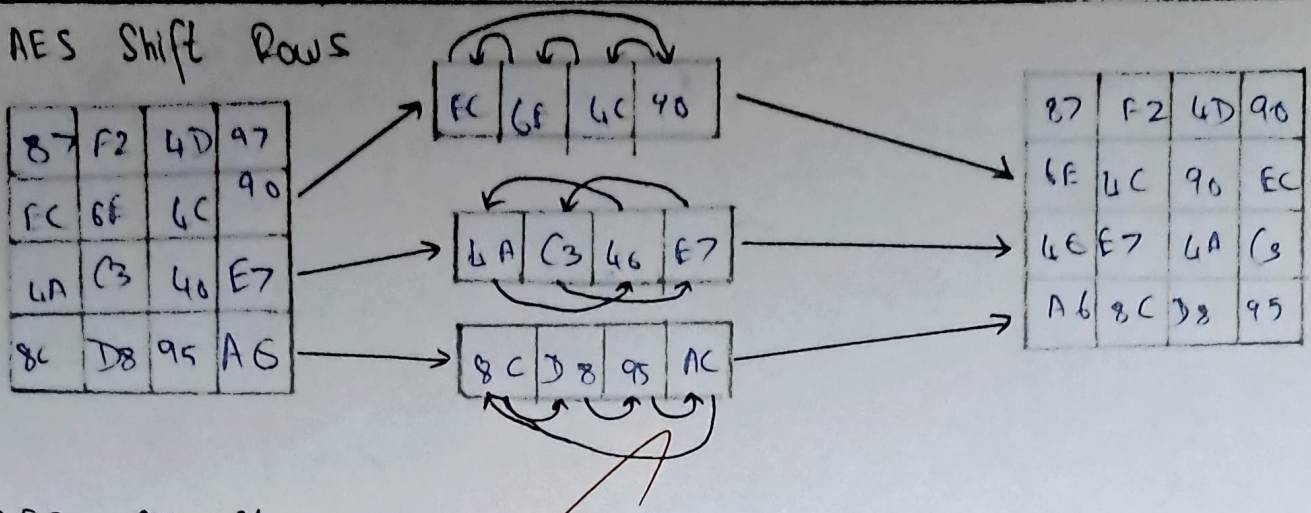
1) AES Sub Bytes.

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

⇒

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

2. AES Shift Rows



3. AES Mix Column

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

*

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

=

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

4. AES Add Round key

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

⊕

AC	19	28	57
77	FA	D1	5C
66	DC	29	00
F3	21	41	6A

=

EB	59	8B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D6

Program,

```
from Cryptodome.Cipher import AES
import binascii
def pad(text):
    while len(text) % 16 != 0:
        text += " "
    return text
def aes_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    padded_text = pad(plaintext)
```

```
    ciphertext = cipher.encrypt(padded_text.encode())
    return binascii.hexlify(ciphertext).decode()
def aes_decrypt(ciphertext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted_text = cipher.decrypt(binascii.unhexlify(ciphertext))
    return decrypted_text.decode().strip()
plaintext = input("Enter 16 byte string:")
key = b'mysecretkey12345'
ciphertext = aes_encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
decrypted_text = aes_decrypt(ciphertext, key)
print("Decrypted Text:", decrypted_text)
```

Output:

```
Enter 16 byte string: HELLOAES1234567
Ciphertext: 22dd503ca3f794f7d97ca982e4fd333
Decrypted Text: HELLOAES1234567
```

Result:

The AES algorithm successfully encrypts and decrypts the plaintext message using 128-bit encryption in ECB mode.

Ex. No:

Vernam Cipher

Aim: To implement the Vernam Cipher algorithm for encrypting and decrypting messages using a one-time pad key.

Algorithm.

1. Input the plaintext and key
2. Convert plaintext and key into their ASCII values
3. Encryption:
   - Ciphertext = Plaintxt ⊕ key (bitwise XOR operation)
4. Decryption
   - Plaintext = Ciphertext ⊕ key (bitwise XOR restores the original text)
5. Key Requirements.
   - The key must be as long as the plaintext
   - for perfect secrecy, the key must be random and used only once (one-time pad)

Example:

Plaintext: "HELLO"

Key: "XMCKL"

Plaintext	H	E	L	L	O
Key	X	M	C	K	L
Ascii (plaintext)	72	69	76	76	79
Ascii (Key)	88	77	67	75	76
XOR	16	8	15	7	5
Ciphertext	Q	I	P	H	D

Program:

```
def vernam_encrypt(plaintext, key):
    ciphertext = ""
    for p,k in zip(plaintext, key):
        cipher_char = chr(ord(p) ^ ord(k))
        ciphertext += cipher_char
    return ciphertext

def vernam_decrypt(ciphertext, key):
    plaintext = ""
    for c,k in zip(ciphertext, key):
        plain_char = chr(ord(c) ^ ord(k))
        plaintext += plain_char
    return plaintext

plaintext = "HELLO"
key = "XMCKL"
ciphertext = vernam_encrypt(plaintext, key)
print("Ciphertext:", ciphertext.encode())
decrypted_text = vernam_decrypt(ciphertext, key)
print("Decrypted Text:", decrypted_text)
```

Output:

Ciphertext : QIPHD

Decrypted Text: HELLO

Result:

The vernam cipher successfully encrypts and decrypts the given plaintext using a onetime pad.

Ex: No:

Double Columnar Cipher

Aim: To implement the Double Columnar Transposition Cipher algorithm for encrypting and decrypting messages using two different keys

Algorithm:

Encryption:

1. Write the plaintext into a grid
2. Rearrange the columns based on the first key.
3. Write the result into a second grid
4. Rearrange the columns again based on the second key
5. Read the ciphertext from the final grid row by row

Decryption:

1. Reverse the second columnar transposition using the second key
2. Reverse the first columnar transposition using the first key.
3. Reconstruct the original plaintext

Example:

Plaintext: "WE ARE DISCOVERED SAVE YOURSELF"

Key_1: "ZEBRAS"

Key_2: "KEYWORD"

Soln:

Z	E	B	R	A	S
W	E	A	R	E	D
I	S	C	O	V	E
R	E	D	S	A	V
E	Y	O	U	R	S
E	L	F	X	X	X

Encryption 1: EVARXACDOFESEYLROSUXDEVSXWIREE

K	E	Y	W	O	R	D
E	V	A	R	X	A	C
D	O	F	E	S	E	Y
L	R	O	S	U	X	D
E	V	S	X	W	I	R
E	E	X	X	X	X	X

Ciphertext: CYDRXVORVE EDLEEXSUWXAEXIXRESXXAFOX

Program:

```
import math
def columnar_transposition(text, key):
    key_order = sorted(list(key))
    col_order = [key.index(k) for k in key_order]
    num_cols = len(key)
    num_rows = math.ceil(len(text)/num_cols)
    grid = ['' for _ in range(num_cols)]
    for i, char in enumerate(text):
        grid[i%num_cols] += char
    cipher = ""
    for index in col_order:
        cipher += grid[index]
    return cipher
def double_columnar_encrypt(plaintxt, key1, key2):
    plaintext = plaintext.replace(" ", "")
    first_pass = columnar_transposition(plaintext, key1)
    second_pass = columnar_transposition(first_pass, key2)
    return second_pass
plaintext = input("Enter:")
```

```
key1 = "ZEBRAS"
key2 = "KEYWORD"
ciphertext = double_columnar_encrypt(plaintext, key1, key2)
print("Ciphertext:", ciphertext)
```

Output:

Enter: WE ARE DISCOVERED SAVE YOURSELF

Ciphertext: CYDRXVORUE ED ZEEXSU WX AEX IX RESXXAFOSX

Result:

The Double columnar transposition cipher successfully encrypts the given plaintext by performing two columnar transposition using two different keys.