**DAY-2**

**Q5:** Write a C program for generalization of the Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.

a. Are there any limitations on the value of b?

b. Determine which values of a are not allowed.

**PROGRAM:**

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

else:

return x % m

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(' ', '') ])

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 ])

def main():

text = 'TWENTY FIFTEEN'

key = [17, 20]

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

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

print('Decrypted Text: {}'.format

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

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

main()

**OUTPUT:**



**Q6:** Write a High level code for ciphertext has been generated with an affine cipher. The most frequent letter of the ciphertext is “B,” and the second most frequent letter of the ciphertext is “U.”Break this code.

**PROGRAM:**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class AffineCipherBreaker {

private static final String ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

public static void main(String[] args) {

Scanner read = new Scanner(System.in);

String ciphertext = read.nextLine();

Map<Character, Integer> frequencyMap = new HashMap<>();

for (char c : ciphertext.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

char mostFrequent = 'A';

char secondMostFrequent = 'A';

int highestFrequency = 0;

int secondHighestFrequency = 0;

for (char c : frequencyMap.keySet()) {

int frequency = frequencyMap.get(c);

if (frequency > highestFrequency) {

secondMostFrequent = mostFrequent;

secondHighestFrequency = highestFrequency;

mostFrequent = c;

highestFrequency = frequency;

} else if (frequency > secondHighestFrequency) {

secondMostFrequent = c;

secondHighestFrequency = frequency;

}

}

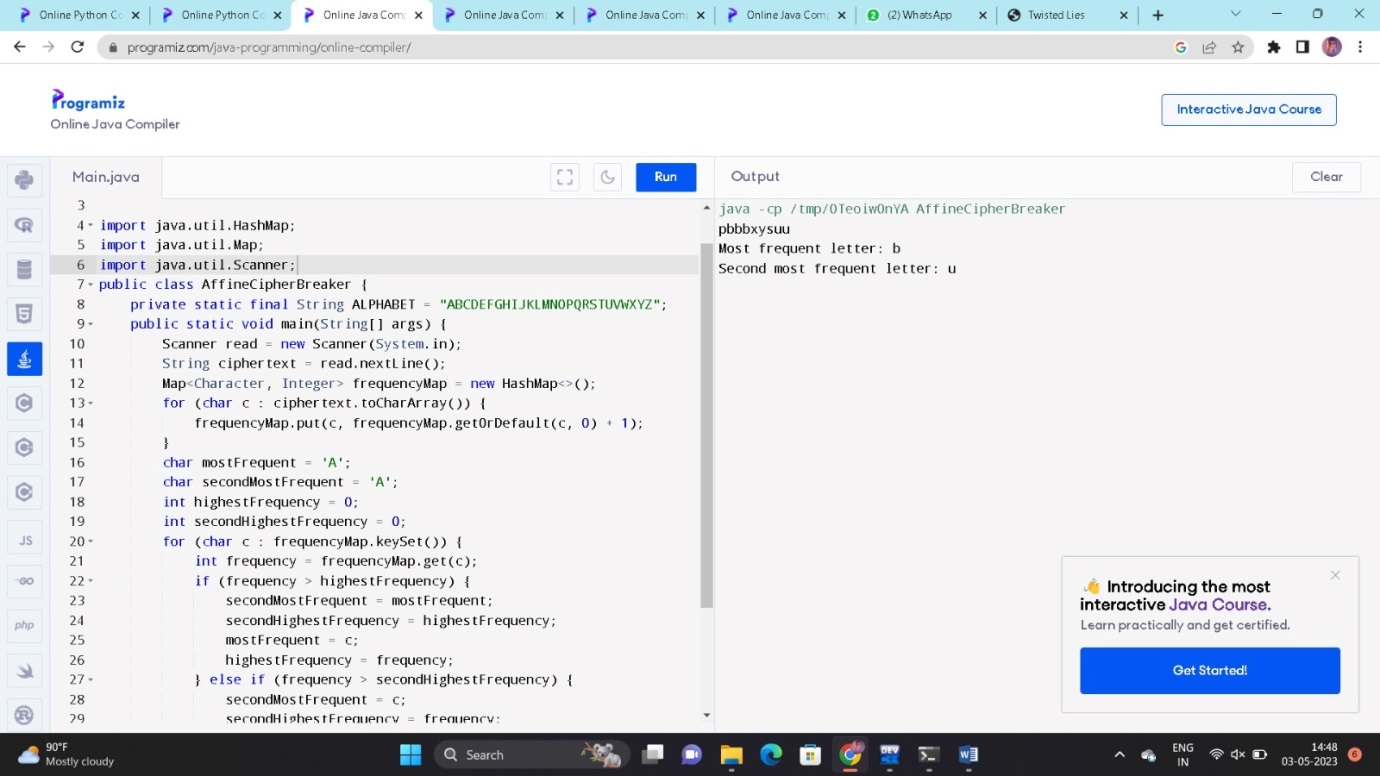
System.out.println("Most frequent letter: " + mostFrequent);

System.out.println("Second most frequent letter: " + secondMostFrequent);

    }

}

**OUTPUT:**

****

**Q7:** Write a C program for monoalphabetic cipher is that both sender and receiver must commit the permuted cipher sequence to memory. A common technique for avoiding this is to use a keyword from which the cipher sequence can be generated.

For example, using the keyword CIPHER, write out the keyword followed by unused letters in normal order and match this against the plaintext letters:

plain: 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

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

**PROGRAM:**

import string

all\_alphabets = list(string.ascii\_uppercase)

def encoder(key):

encoded = ""

arr = [False]\*26

for i in range(len(key)):

if key[i] >= 'A' and key[i] <= 'Z':

if arr[ord(key[i]) - 65] == False:

encoded += key[i]

arr[ord(key[i]) - 65] = True

elif key[i] >= 'a' and key[i] <= 'z':

if arr[ord(key[i]) - 97] == False:

encoded += chr(ord(key[i]) - 32)

arr[ord(key[i]) - 97] = True

for i in range(26):

if arr[i] == False:

arr[i] = True

encoded += (chr(i + 65))

return encoded

def decipheredIt(msg, encoded):

plaintext = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

decipher = ""

enc = {}

for i in range(len(encoded)):

enc[encoded[i]] = i

for i in range(len(msg)):

if msg[i] >= 'a' and msg[i] <= 'z':

pos = enc.get((chr)(msg[i]-32))

decipher += plaintext[pos]

elif msg[i] >= 'A' and msg[i] <= 'Z':

pos = enc.get(msg[i])

decipher += plaintext[pos]

else:

decipher += msg[i]

return decipher

key = "CIPHER"

print("Keyword : " + key)

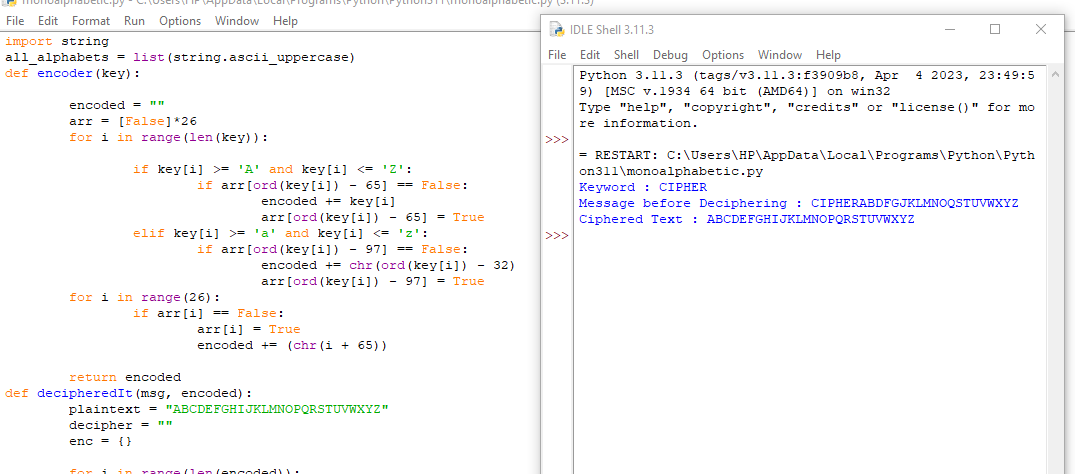
decoded = encoder(list(key))

message = "CIPHERABDFGJKLMNOQSTUVWXYZ"

print("Message before Deciphering : " + message)

print("Ciphered Text : " + decipheredIt(message, decoded))

**OUTPUT:**



**Q8:** Write a C program for Playfair matrix:

M F H I/J K

U N O P Q

Z V W X Y

E L A R G

D S T B C

Encrypt this message: Must see you over

**PROGRAM:**

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)

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 = 'MUST SEE YOU OVER'

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

PlainTextList = Diagraph(FillerLetter(text\_Plain))

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

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

key = "MFHIKUNOPQZVWXYELARGDSTBC"

print("Key text:", key)

key = toLowerCase(key)

Matrix = generateKeyTable(key, list1)

print("Plain Text:", text\_Plain)

CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

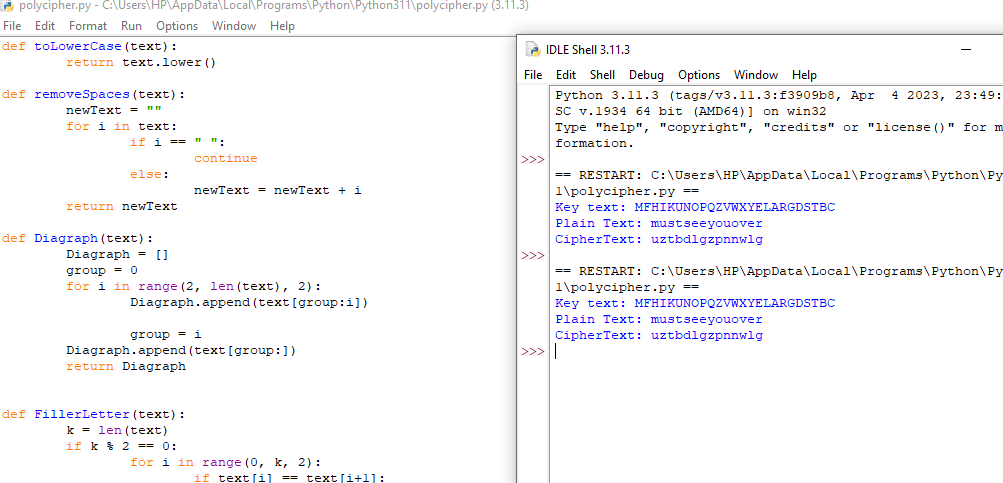
CipherText = ""

for i in CipherList:

CipherText += i

print("CipherText:", CipherText)

**OUTPUT:**



**Q9:** Write a high-level code for possible keys does the Playfair cipher have? Ignore the fact that some keys might produce identical encryption results. Express your answer as an approximate power of 2.

**PROGRAM:**

import itertools

def find\_keyword():

alphabet = 'ABCDEFGHIKLMNOPQRSTUVWXYZ'

combinations = itertools.combinations(alphabet, 25)

for keyword in combinations:

matrix = [[0]\*5 for \_ in range(5)]

for i, letter in enumerate(keyword):

row = i // 5

col = i % 5

matrix[row][col] = letter

valid = True

for row in range(5):

for col in range(5):

if matrix[row][col] == 0:

valid = False

break

if matrix[row][col] == 'I' or matrix[row][col] == 'J':

matrix[row][col] = 'IJ'

if matrix[row][col] in matrix[row][col+1:] + [matrix[i][col] for i in range(row+1, 5)]:

valid = False

break

if not valid:

break

if valid:

return keyword

return None

keyword = find\_keyword()

if keyword is not None:

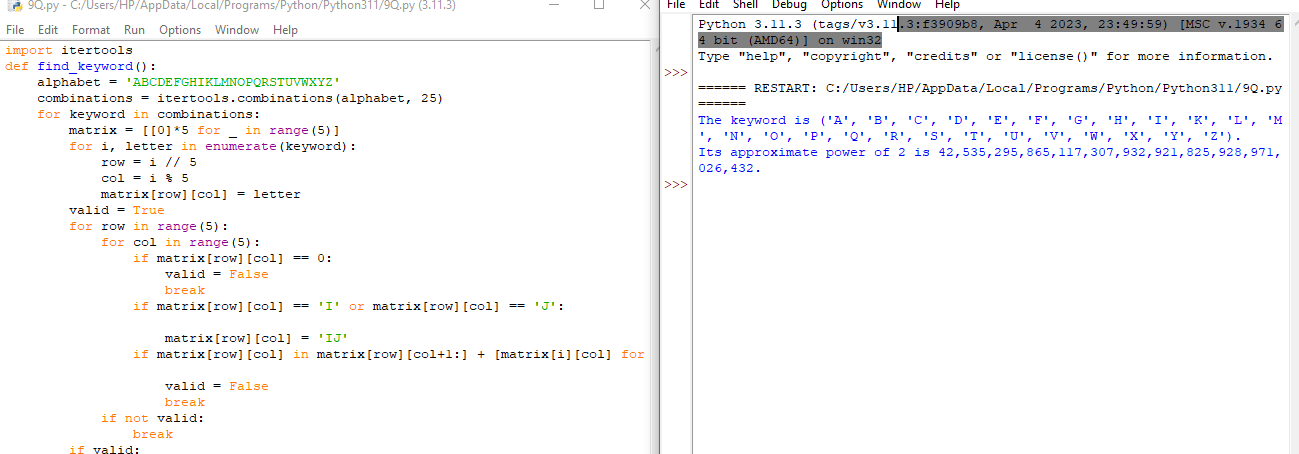
print(f"The keyword is {keyword}.")

print(f"Its approximate power of 2 is {2\*\*(len(keyword)\*5):,.0f}.")

else:

print("No valid keyword was found.")

**OUTPUT:**



**Q10**: Write a high level code to Encrypt the message “meet me at the usual place at ten rather than eight oclock” using the Hill cipher with the key.

9 4

5 7

a. Show your calculations and the result.

b. Show the calculations for the corresponding decryption of the ciphertext to recover the original plaintext.

**PROGRAM:**

# Define the key matrix

key = [[9, 4], [5, 7]]

# Define the plaintext message

plaintext = "meet me at the usual place at ten rather than eight oclock"

# Convert the plaintext to uppercase and remove spaces

plaintext = plaintext.upper().replace(" ", "")

# Pad the plaintext with "X" if necessary to make the length a multiple of 2

if len(plaintext) % 2 != 0:

plaintext += "X"

# Split the plaintext into pairs of 2 letters and convert each pair to a vector

plaintext\_vectors = []

for i in range(0, len(plaintext), 2):

pair = plaintext[i:i+2]

vector = [ord(pair[0])-65, ord(pair[1])-65]

plaintext\_vectors.append(vector)

# Multiply each plaintext vector by the key matrix to get the corresponding ciphertext vector

ciphertext\_vectors = []

for vector in plaintext\_vectors:

ciphertext\_vector = [(key[0][0]\*vector[0] + key[0][1]\*vector[1]) % 26, (key[1][0]\*vector[0] + key[1][1]\*vector[1]) % 26]

ciphertext\_vectors.append(ciphertext\_vector)

# Convert each ciphertext vector back to a pair of letters

ciphertext = ""

for vector in ciphertext\_vectors:

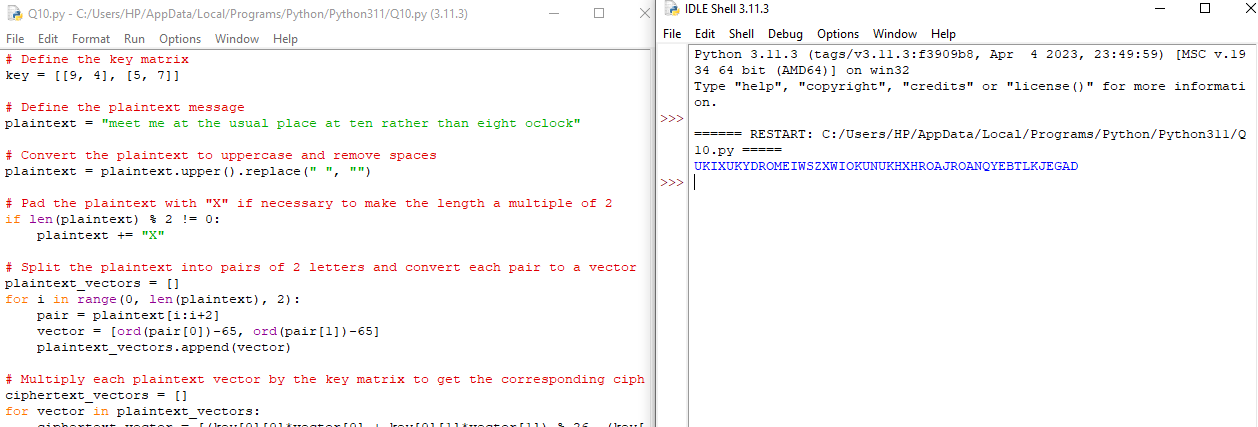
pair = chr(vector[0]+65) + chr(vector[1]+65)

ciphertext += pair

# Print the ciphertext

print(ciphertext)

**OUTPUT:**



**Q11:** Write a high level language program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 1 and 26. For example, if the key is 3 19 5 . . . , then the first letter of the plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.

Encrypt the plaintext send more money with the key stream

9 0 1 7 23 15 21 14 11 11 2 8 9

**PROGRAM:**

def vigenere\_otp\_encrypt(plaintext, key\_stream):

ciphertext = ""

key\_index = 0

for char in plaintext:

shift = key\_stream[key\_index]

if char.isalpha():

if char.isupper():

ciphertext += chr((ord(char) - 65 + shift) % 26 + 65)

else:

ciphertext += chr((ord(char) - 97 + shift) % 26 + 97)

key\_index = (key\_index + 1) % len(key\_stream)

else:

ciphertext += char

return ciphertext

# Example usage

plaintext = "sendmoremoney"

key\_stream = [9, 0, 1, 7, 23, 15, 21, 14, 11, 11, 2, 8, 9]

ciphertext = vigenere\_otp\_encrypt(plaintext, key\_stream)

print(ciphertext)

**OUTPUT:**

