# MANUAL

**NETWORK SECURITY AND CRYPTOGRAPHY**

# CODE: 631

# NAME: ZAYAHAN HASAN SHAH

# SEAT NUMBER: EB21102135

# SUBMITED TO: SIR BARI AHMED KHAN

# SIGNATURE\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO** | **DATE** | **TOPIC** | **SIGNATURE** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# **LAB: CEASER CIPHER**

**Objective:** Encode the given input and then decode by using cease cipher technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:** A Caesar cipher is a simple method of encoding of messages. Caesar ciphers use a substitution method where letters in the alphabet are shifted by some fixed number of spaces to yield an encoding alphabet.

**Method:** Shift right three the given input for the encoding and shift left three for the decoding.

**Code:**

import tkinter as tk # for GUI

root=tk.Tk()

root.geometry("400x250")

root.resizable(True,False)

title=root.title("Encode/Decode")

decode = []

encode = []

Upper = ['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','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']

def Quit1():

root.destroy()

def Decode():

global Quit

for i in range(len(encode)):

if encode[i] == 'A':

decode.append('X')

elif encode[i] == 'B':

decode.append('Y')

elif encode[i] == 'C':

decode.append('Z')

elif encode[i] == 'a':

decode.append('x')

elif encode[i] == 'b':

decode.append('y')

elif encode[i] == 'c':

decode.append('z')

else:

if encode[i].islower():

# print(i)

next\_elem = encode[i]

# print("elem", next\_elem)

index = Upper.index(next\_elem)

# print(index)

shifted\_index = (index - 3) % 52

# print(shifted\_index)

decode.append(Upper[shifted\_index])

else:

# print(i)

next\_elem = encode[i]

# print("elem", next\_elem)

index = Upper.index(next\_elem)

# print(index)

shifted\_index = (index - 3) % 52

# print(shifted\_index)

decode.append(Upper[shifted\_index])

cs()

decoded=tk.Label(root,text=f"Decoded:{decode}")

decoded.pack() Quit=tk.Button(root,text="Quit",height=2,width=6,command=Quit1)

Quit.pack()

def Encode():

global Quit

user=inp.get()

for j in range(len(user)):

if user[j] == 'X':

encode.append('A')

elif user[j] == 'Y':

encode.append('B')

elif user[j] == 'Z':

encode.append('C')

elif user[j] == 'x':

encode.append('a')

elif user[j] == 'y':

encode.append('b')

elif user[j] == 'z':

encode.append('c')

else:

if user[j].islower() :

# print(j)

next\_elem = user[j]

# print("elem", next\_elem)

index = Upper.index(next\_elem)

# print(index)

shifted\_index = (index + 3) % 52

# print(shifted\_index)

encode.append(Upper[shifted\_index])

else:

# print(j)

next\_elem = user[j]

# print("elem", next\_elem)

index = Upper.index(next\_elem)

# print(index)

shifted\_index = (index + 3) % 26

# print(shifted\_index)

encode.append(Upper[shifted\_index])

cs()

encoded=tk.Label(root,text=f"Encoded:{encode}")

encoded.pack(pady=40)

decode=tk.Button(root,text="Decode",height=2,width=6,command=Decode)

decode.place(x=40,y=80) Quit=tk.Button(root,text="Quit",height=2,width=6,command=Quit1)

Quit.place(x=100,y=80)

def cs():

cs=tk.Button(root,height=20,width=10000)

cs.place(x=-10,y=-10)

inp=tk.Entry(root)

inp.place(x=40,y=40)

Encode=tk.Button(root,text="Encode" ,height=1,width=16,command=Encode)

Encode.place(x=40,y=80)

**Calculation:**

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | HELLO | KHOOR |
| DECODE | KHOOR | HELLO |

**Output:**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Conclusion**:

The code above is shift right three the give input into encoding and shift left three for decoding.

# **LAB: VIGENERE CIPHER**

**Objective:** Encode the given input and then decode by using cease cipher technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:** Vigenere Cipher is a method of encrypting alphabetic text. 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.

**Method:** The Vigenère cipher is a method of encrypting alphabetic text where each letter of the plaintext is encoded with a different Caesar cipher, whose increment is determined by the corresponding letter of another text, the key.

**Code:**

import tkinter as tk

import random

root=tk.Tk()

root.resizable(False,False)

root.geometry("400x400")

title=root.title("Encoder/Decode")

label=tk.Label(root,text="Enter text to encode",font=("Arial", 25))

label.pack(side="top")

ui=tk.Entry(root,font=("Arial", 25))

ui.pack(side="top")

def Start(CF=False):

global decode

pt=(ui.get())

pt = pt.upper()

print(pt)

keyword = []

for i in range(random.randint(4,5)):

keyword.append(chr(random.randint(65,90)))

key = ''.join(keyword)

print(key)

button\_e.config(text="Decode",command=ET\_)

def equalkey(key, pt):

key\_len = len(key) # length of keyword

pt\_len = len(pt) # length of plain text

key\_repeat = '' # repeated keyword

if key\_len == pt\_len:

return key

elif key\_len < pt\_len:

repeat = pt\_len // key\_len # need to be repeat the keyword

reminder = pt\_len % key\_len # additional character after repeatition

key\_repeat += key \* repeat

key\_repeat += key[:reminder]

return key\_repeat

elif key\_len > pt\_len:

key\_repeat += key[:pt\_len]

return key\_repeat

def generate\_vigenere\_table():

global repeate\_key

global vigenere\_table

Capital = [] # ascii 65 to 90

for i in range(26):

Capital.append([])

for j in range(26):

Capital[i].append(chr(((i + j) % 26) + 65))

for row in Capital:

print(" ".join(row))

return Capital

# Generate and print the Vigenère table

vigenere\_table = generate\_vigenere\_table()

repeate\_key = equalkey(key, pt)

print(repeate\_key)

# encryption , encryption is done by matrix method

def encrypt(pt, repeate\_key, vigenere\_table):

global ET

pt = list(pt)

encrypted = []

for i in range(len(pt)):

row = ord(repeate\_key[i]) - 65 # Calculate the row index in the Vigenère table

print(row)

col = ord(pt[i]) - 65 # Calculate the column index in the Vigenère table

print(col)

encrypted\_char = vigenere\_table[row][col] # Get the character from the Vigenère table

encrypted.append(encrypted\_char) # Add the encrypted character to the list

return ''.join(encrypted)

encode = encrypt(pt, repeate\_key, vigenere\_table)

print('===============')

ET.config(text=f"Encoded:{encode}")

print(encode)

# decryption, decryption is done by Mod method

def decrypt(encode, vigenere\_table, repeate\_key):

global decode

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

encode = list(encode)

decrypted = ''

for i in range(len(encode)):

result = ( ord(encode[i]) - ord(repeate\_key[i]) ) % 26

decrypted += alphabet[result]

return decrypted

decode = decrypt(encode, vigenere\_table, repeate\_key)

print(decode)

def ET\_():

ET.config(text=f"Defcoded:{decode}") quit\_=tk.Button(root,text="Quit",height=2,width=8,command=Quit,font=("Arial", 25))

quit\_.pack(side="bottom")

def Quit():

exit()

button\_e=tk.Button(root,text="Encode",height=2,width=8,command=Start,font=("Arial", 25))

button\_e.pack(side="top")

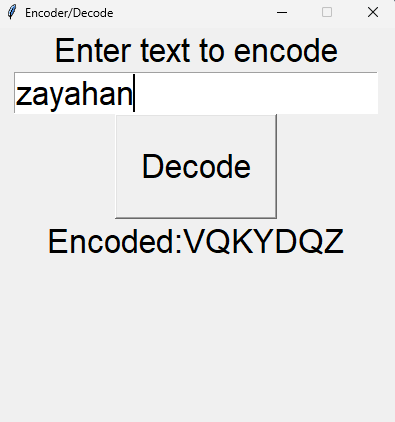
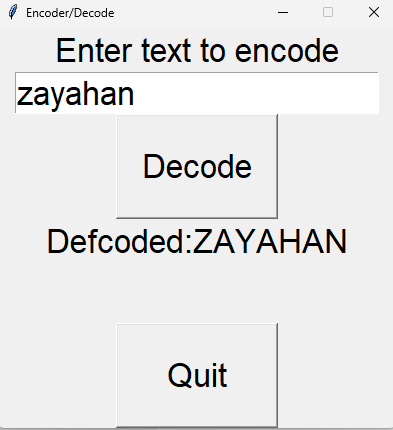
ET=tk.Label(root,text="",font=("Arial", 25))

ET.pack(side="top")

**Calculation:**

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | ZAYAHAN | VQKYDQZ |
| DECODE | VQKYDQZ | ZAYAHAN |

**Output:**

**** ****

**Conclusion**:

The code above is running when the user enter the plaint text and it convert into cipher text diagonally with the help of key.

**LAB: ONE TIME PAD CIPHER**

**Objective:** Encode the given input and then decode by using one time pad technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:** the one-time pad (OTP) is an encryption technique that cannot be cracked, but requires the use of a single use pre-shared key that is equal to the size of the message being sent. In this technique, a plaintext is paired with a random secret key (also referred to as a one-time pad).

**Method:** one-time pad is a system in which a randomly generated private key is used only once to encrypt a message that is then decrypted by the receiver using a matching one-time pad and key.

**Code:**

import tkinter as tk

import random

root=tk.Tk()

##root.geometry("320x210")

root.resizable(True,True)

title=root.title("ONE TIME PAD CIPHER")

label=tk.Label(root,text="Enter text to encode",font=("Ariel",25))

label.pack(side="top")

ui=tk.Entry(root,font=("Ariel",25))

ui.pack(side="top")

def Start():

global decryptrd\_message

plaintext = (ui.get())

key\_length = len(plaintext)

key = generate\_random\_key(key\_length)

print(f"Plaintext: {plaintext}")

KT.config(text=f"Key: {key}",font=("Ariel",25))

plaintext\_binary = plaintext\_to\_binary(plaintext)

key\_repeated = repeat\_key(key, key\_length)

key\_binary = key\_to\_binary(key\_repeated)

print(f"plain text in binary is : {plaintext\_binary}")

print(f"Binary key: {key\_binary}")

encrypted\_message = encrypt(plaintext\_binary, key\_binary)

EM.config(text=f"Encrypted(Binary): {encrypted\_message}",font=("Ariel",25)

button\_e.config(text="Decode",command=Decode)

decryptrd\_message = decrypt(encrypted\_message, key\_binary)

def Decode():

EM.config(text=f"decoded(binary): {decryptrd\_message}")

button\_e.config(text="Encode",command=Start) quit\_=tk.Button(root,text="Quit",height=2,width=8,command=Quit,font=("Ariel",15))

quit\_.pack(side="top"

def generate\_random\_key(length):

return ''.join([chr(random.randint(65, 90)) for i in range(length)])

# plain text to binary conversion

def plaintext\_to\_binary(pt):

return [format(ord(char), '08b') for char in pt]

# repeating key to the length of plain text

def repeat\_key(key, length):

repeats = length // len(key)

remainder = length % len(key)

return (key \* repeats) + key[:remainder]

# key to binary conversion

def key\_to\_binary(key):

return [format(ord(char), '08b') for char in key]

# encrypting plaintext using XOR gate

def encrypt(plaintext\_binary, key\_binary):

encrypted = []

for pt, key in zip(plaintext\_binary, key\_binary):

encrypted.append(''.join(str(int(pt\_bit) ^ int(key\_bit)) for pt\_bit, key\_bit in zip(pt, key)))

## encrypted = [text for text in encrypted

return encrypted

# decrypting cipher text using XOR gate

def decrypt(encrypted\_message, key\_binary):

decrypted = []

for em, key in zip(encrypted\_message, key\_binary):

decrypted.append(''.join(str(int(enc\_bit) ^ int(key\_bit)) for enc\_bit, key\_bit in zip(em,key)))

return decrypted

def Quit():

exit()

button\_e=tk.Button(root,text="Encode",height=2,width=8,command=Start,font=("Ariel",25))

button\_e.pack(side="top")

KT=tk.Label(root,text="")

KT.pack(side="top")

EM=tk.Label(root,text="")

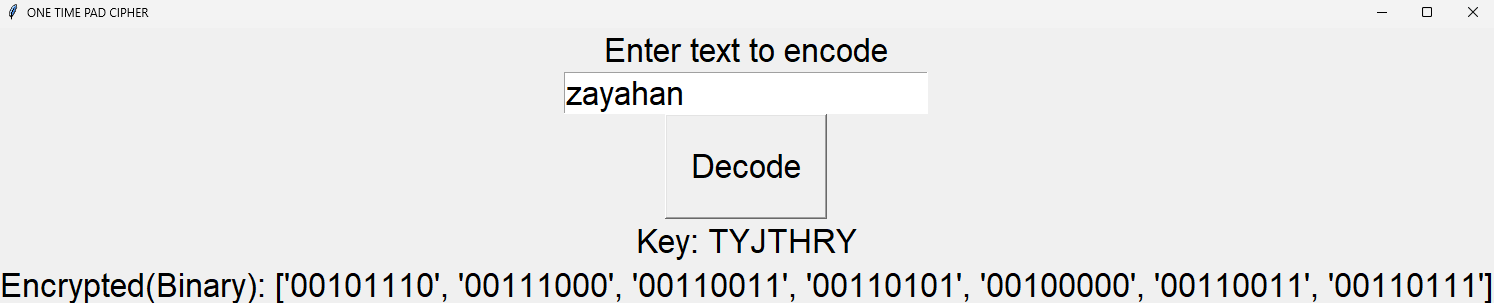
EM.pack(side="top", fill='both')

**CALCULATION:**

Key: TYJTHRY (Random Key)

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | ZAYAHAN | Output in the image |
| DECODE | Cipher text in the image | Output in the image(ZAYAHAN) |

**OUTPUT:**



A screenshot of a computer

Description automatically generated

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into binary (both in encryption and decryption) with the help of random generation of key.

**LAB: HILL CIPHER**

**Objective:** Encode the given input and then decode by using hill cipher technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:**  the **Hill cipher** is a polygraphic substitution cipher based on linear algebra. Invented by Lester S. Hill in 1929.

**Method**: To encrypt a message, each block of *n* letters is multiplied by an invertible n x n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption. Both encryption and decryption done with key.

**Code:**

import tkinter as tk

import random

import math

root=tk.Tk()

root.resizable(False,False)

title=root.title("Encoder/Decode")

label=tk.Label(root,text="Enter text to encode",font=("Ariel",25))

label.pack(side="top")

ui=tk.Entry(root,font=("Ariel",25))

ui.pack(side="top")

def Start():

global decrypt

pt=ui.get()

pt = pt.upper()

print("Plaint text : ", pt)

print("==================")

encrypted\_ans = []

decrypted\_ans = []

alphabets = ['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']

break\_three = []

spliting\_word = []

ascii\_pt = []

key\_pt\_mult = []

mode\_for\_pt = [] # not a 2d array

mode\_for\_pt\_3d = []

encrypt = []

expand\_key =[]

adjoint = []

inv\_adj = []

key\_inverese = []

dec\_key\_mult = []

dec\_sum = []

decrypt = []

# removing spaces from the plain text

if " " in pt:

pt = pt.replace(" ", "")

# len of plain text

print(len(pt))

print("==================")

pt\_break = 0

if len(pt) > 4 :

if(len(pt)%2 == 0):

pt\_break = 3

start = 0

# breaking string in the pair of 3

for i in range(len(pt)):

break\_three.append(pt[start:start+pt\_break])

start += pt\_break

break\_three = [text for text in break\_three if text != ""]

print("breaking words into 3 words : ", break\_three)

print("==================")

# filtering 'x'

for i in range(len(break\_three)):

if len(break\_three[i]) != 3:

while len(break\_three[i]) != 3:

break\_three[i] += 'X'

if len(break\_three[i]) == 3:

break

print("filtering x : ", break\_three)

print("==================")

# spliting words

for word in break\_three:

temp\_word = []

for chars in word:

temp\_word.append(chars)

spliting\_word.append(temp\_word)

print("Spliting words : ")

for i in range(len(spliting\_word)):

for j in range(len(spliting\_word[i])):

print(spliting\_word[i][j], end= " ")

print()

print("==================")

# ascii characters (0-25) of alphabets

for i in range(len(spliting\_word)):

temp\_ascii\_pt =[]

for j in range(len(spliting\_word[i])):

split = spliting\_word[i][j]

assci = alphabets.index(split)

temp\_ascii\_pt.append(assci)

ascii\_pt.append(temp\_ascii\_pt)

print("Index of plain text")

for i in range(len(ascii\_pt)):

for j in range(len(ascii\_pt[i])):

print(ascii\_pt[i][j], end=" ")

print(

print("==================")

# key

# hard coded key

key = [[17,17,5],[21,18,21],[2,2,19]]

PK=tk.Label(root,text="Key : ",font=("Ariel",25))

PK.pack(side="top")

print("key")

for i in range(3):

for j in range(3):

key1= ""

key1 += " " + str(key[i][j])

PK.config(text=PK.cget("text") + key1)

## print(key[i][j], end = " ")

print()

print("==================")

# multiplcation of key and plain text

for i in range(len(ascii\_pt)):

for j in range(len(key)):

temp\_solution = []

for k in range(len(key)):

print(f'{ascii\_pt[j][k]} \* {key[k][j]}', end = " ")

temp\_solution.append(ascii\_pt[i][k] \* key[k][j])

key\_pt\_mult.append(temp\_solution)

print()

print("==================")

print("just multiplcation of key and plain text(index)")

for i in range(len(key\_pt\_mult)):

for j in range(len(key\_pt\_mult[i])):

print(key\_pt\_mult[i][j], end = " ")

print()

print("==================")

for i in range(len(key\_pt\_mult)):

print(f'sum of : {key\_pt\_mult[i]} is {sum(key\_pt\_mult[i])} and mode 26 is {sum(key\_pt\_mult[i])%26}')

mode\_for\_pt.append(sum(key\_pt\_mult[i])%26)

print("==================")

for i in range(len(mode\_for\_pt)):

print(mode\_for\_pt[i])

on = 0 # start

off = 3 # sto

for i in range(len(mode\_for\_pt)):

mode\_for\_pt\_3d.append(mode\_for\_pt[on:on+off])

on += off

print("==================")

# 3d array of solution

print("Encryption in the form of numbers")

mode\_for\_pt\_3d = [subarray for subarray in mode\_for\_pt\_3d if subarray]

for i in range(len(mode\_for\_pt\_3d)):

for j in range(len(mode\_for\_pt\_3d[i])):

if mode\_for\_pt\_3d[i][j] is None:

continue

else:

print(mode\_for\_pt\_3d[i][j], end= " ")

print()

print("==================")

# Encryption in the form of text

for i in range(len(mode\_for\_pt\_3d)):

for j in range(len(mode\_for\_pt\_3d[i])):

text = mode\_for\_pt\_3d[i][j]

print(f'{mode\_for\_pt\_3d[i][j]} --> {alphabets[text]}')

encrypt.append(alphabets[text]

print("==================")

print("Encryption of the plain text is : ", "".join(encrypt))

print("==================")

ET.config(text=f"Encryption of the plain text is : {''.join(encrypt)}")

button\_e.config(text="Decode",command=decode\_)

# determinant of key

a = key[0][0] \* (key[1][1] \* key[2][2] - key[1][2] \* key[2][1])

b = key[0][1] \* (key[1][0] \* key[2][2] - key[1][2] \* key[2][0])

c = key[0][2] \* (key[1][0] \* key[2][1] - key[1][1] \* key[2][0])

determinant = (a-b+c) % 26

print(determinant)

print("==================")

if determinant == 0:

print("Can't solve further")

quit()

elif determinant < 0 and determinant < 26:

determinant += 26

print(determinant)

else:

# expanding key matrix

print("Expanded key : ")

expand\_key = [

[key[0][0], key[0][1], key[0][2], key[0][0], key[0][1]],

[key[1][0], key[1][1], key[1][2], key[1][0], key[1][1]],

[key[2][0], key[2][1], key[2][2], key[2][0], key[2][1]],

[key[0][0], key[0][1], key[0][2], key[0][0], key[0][1]],

[key[1][0], key[1][1], key[1][2], key[1][0], key[1][1]]

]

print(len(expand\_key))

print("==================")

for i in range(len(expand\_key)):

for j in range(len(expand\_key)):

print(expand\_key[i][j], end= " ")

print()

print("==================")

# solving for ajoint of key

# row 1

r1c1 = (expand\_key[1][1] \* expand\_key[2][2] - expand\_key[1][2] \* expand\_key[2][1]) % 26

r1c2 = (expand\_key[2][1] \* expand\_key[3][2] - expand\_key[2][2] \* expand\_key[3][1]) % 26

r1c3 = (expand\_key[3][1] \* expand\_key[4][2] - expand\_key[3][2] \* expand\_key[4][1]) % 26

# row 2

r2c1 = (expand\_key[1][2] \* expand\_key[2][3] - expand\_key[1][3] \* expand\_key[2][2]) % 26

r2c2 = (expand\_key[2][2] \* expand\_key[3][3] - expand\_key[2][3] \* expand\_key[3][2]) % 26

r2c3 = (expand\_key[3][2] \* expand\_key[4][3] - expand\_key[3][3] \* expand\_key[4][2]) % 26

# row 3

r3c1 = (expand\_key[1][3] \* expand\_key[2][4] - expand\_key[1][4] \* expand\_key[2][3]) % 26

r3c2 = (expand\_key[2][3] \* expand\_key[3][4] - expand\_key[2][4] \* expand\_key[3][3]) % 26

r3c3 = (expand\_key[3][3] \* expand\_key[4][4] - expand\_key[3][4] \* expand\_key[4][3]) % 26

print(r1c1, r1c2, r1c3, r2c1, r2c2, r2c3, r3c1, r3c2, r3c3 )

print("Adjoint")

adjoint = [

[r1c1, r1c2, r1c3],

[r2c1, r2c2, r2c3],

[r3c1, r3c2, r3c3]

]

for i in range(3):

for j in range(3):

print(adjoint[i][j], end = " ")

print()

print("==================")

# inverse of determinant

inv\_der = 0

if determinant%2 == 0:

for i in range(1,100):

mo = (determinant \* i + 1) % 26

if mo == 1:

inv\_der = i

break

print(inv\_der)

else:

for i in range(1,100):

mo = (determinant \* i) % 26

if mo == 1:

inv\_der = i

break

print('inverse of determinant : ', inv\_der)

print("==================")

# multiplying adjoint with inverse of determinant

print('multiplying adjoint with inverse of determinant')

for i in range(3):

temp = []

for j in range(3):

print(f'{adjoint[i][j]} \* {inv\_der} = {adjoint[i][j] \* inv\_der}', end=" ")

temp.append(adjoint[i][j] \* inv\_der)

inv\_adj.append(temp)

print()

print("==================")

# inverse of key

for i in range(3):

temp\_mod = []

for j in range(3):

print(f'{inv\_adj[i][j]} % {26} = {inv\_adj[i][j] % 26}', end=" ")

temp\_mod.append(inv\_adj[i][j] % 26)

key\_inverese.append(temp\_mod)

print()

print("==================")

print("key inverse :")

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

for j in range(len(key\_inverese)):

print(key\_inverese[i][j], end = " ")

print()

print("==================")

for i in range(len(mode\_for\_pt\_3d)):

for j in range(len(key\_inverese)):

temp\_dec\_num = []

for k in range(len(key\_inverese)):

print(f'{(mode\_for\_pt\_3d[i][k] \* key\_inverese[k][j]) % 26}', end = " ")

temp\_dec\_num.append((mode\_for\_pt\_3d[i][k] \* key\_inverese[k][j]) % 26)

dec\_key\_mult.append(temp\_dec\_num)

print(

print("==================")

for i in range(len(dec\_key\_mult)):

print(f'sum of : {dec\_key\_mult[i]} is {sum(dec\_key\_mult[i])} and mode 26 is {sum(dec\_key\_mult[i])%26}')

dec\_sum.append(sum(dec\_key\_mult[i])%26)

print("=================="

for i in range(len(dec\_sum)):

print(alphabets[dec\_sum[i]] , " is decrypted answer")

if alphabets[dec\_sum[i]] != 'X':

decrypt.append(alphabets[dec\_sum[i]])

print("Decrypted answer is : ", "".join(decrypt))

def decode\_():

ET.config(text=f"Decrypted answer is : {''.join(decrypt)}")

quit\_=tk.Button(root,text="Quit",height=2,width=8,command=Quit,font=("Ariel",25))

quit\_.pack(side="top")

def Quit():

exit()

button\_e=tk.Button(root,text="Encode",height=2,width=8,command=Start, font=("Ariel", 25))

button\_e.pack(side="top")

ET=tk.Label(root,text="",font=("Ariel",25))

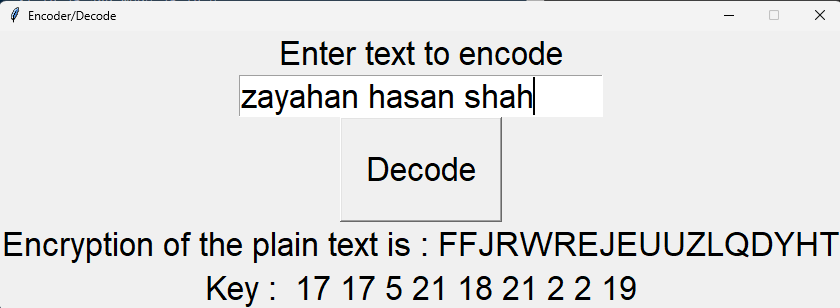
ET.pack(side="top")

**CALCULATION:**

Key: 17 17 5 21 18 21 2 2 19 (It’s pre-defined, but it can be a Random Key)

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | Zayahan hasan shah | FFJRWREJEUUZLQDYHT |
| DECODE | FFJRWREJEUUZLQDYHT | Zayahan hasan shah |

**OUTPUT:**

****

**A screenshot of a computer

Description automatically generated**

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into cipher text with the help of matrix of key(can be random).

**LAB: PLAY FAIR CIPHER**

**Objective:** Encode the given input and then decode by using play fair cipher technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:** The ciphertext character that replaces a particular plaintext character in the encryption will depend in part on an adjacent character in the plaintext.

**Method**: The Algorithm consists of 2 steps: 

1. **Generate the key Square (5×5):**
   * The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
   * The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
2. **Algorithm to encrypt the plain text:** The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.

**Code:**

def createPlayFairMatrix(key):

key = key.upper().replace("J", "I")

seen = set()

preparedKey = []

for char in key:

if char not in seen and char.isalpha():

seen.add(char)

preparedKey.append(char)

alphabet = 'ABCDEFGHIKLMNOPQRSTUVWXYZ'

for char in alphabet:

if char not in seen:

preparedKey.append(char)

matrix = []

for i in range(0, 25, 5):

matrix.append(preparedKey[i:i+5])

return matrix

def findPosition(char, matrix):

for i, row in enumerate(matrix):

if char in row:

return i, row.index(char)

raise ValueError(f'Character {char} not found in matrix')

def preProcessText(text):

text = re.sub(r'[^A-Z]', '', text.upper().replace("J", "I"))

processedText = ""

i = 0

while i < len(text):

char1 = text[i]

if i + 1 < len(text):

char2 = text[i + 1]

else:

char2 = 'X'

if char1 == char2:

processedText += char1 + "X"

i += 1

else:

processedText += char1 + char2

i += 2

if len(processedText) % 2 != 0:

processedText += "X"

return processedText

def encrypt(plainText, matrix):

plainText = preProcessText(plainText)

cipherText = ""

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

char1, char2 = plainText[i], plainText[i + 1]

row1, col1 = findPosition(char1, matrix)

row2, col2 = findPosition(char2, matrix)

if row1 == row2:

cipherText += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]

elif col1 == col2:

cipherText += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]

else:

cipherText += matrix[row1][col2] + matrix[row2][col1]

return cipherText

def decrypt(cipherText, matrix):

plaintext = ""

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

char1, char2 = cipherText[i], cipherText[i + 1]

row1, col1 = findPosition(char1, matrix)

row2, col2 = findPosition(char2, matrix)

if row1 == row2:

plaintext += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]

elif col1 == col2:

plaintext += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]

else:

plaintext += matrix[row1][col2] + matrix[row2][col1]

return plaintext

def printMatrix(matrix):

La1=[]

for row in matrix:

PL3=(" ".join(row))

La1.append(PL3)

L3.config(text=f"{La1[0]}\n{La1[1]}\n{La1[2]}\n{La1[3]}\n{La1[4]}")

def checkPlaintext(plaintext):

bot = False

specialChars = r"[\.\\\^\$\\*\+\?\{\}\[\]\|()]"

if re.findall(specialChars, plaintext):

return not bot

# raise ValueError(f'Do not enter the special character!')

return bot

key = "playfair example"

matrix = createPlayFairMatrix(key)

def Code():

global ciphertext

plaintext = str(Entry.get())

checker = checkPlaintext(plaintext)

if checker == True:

print("Enter the text for encryption : ")

plaintext = input("")

L2.config(text="Playfair Matrix:")

printMatrix(matrix)

print("==============================")

ciphertext = encrypt(plaintext, matrix)

L4.config(text=f"Encrypted Text: {ciphertext}") Quit=tk.Button(root,text="Quit",height=2,width=8,command=Quit\_)

Quit.pack(side="bottom")

ec.config(text="Decrypt",command=Decode)

def Decode():

global ciphertext

decrypted\_text = decrypt(ciphertext, matrix)

L4.config(text=f"Decrypted Text: {decrypted\_text}")

L2.pack\_forget()

L3.pack\_forget()

def Quit\_():

exit()

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

import tkinter as tk

import re

root=tk.Tk()

root.resizable(False,False)

Label1=tk.Label(text="Enter the text for encryption ")

Label1.pack(side="top")

Entry=tk.Entry()

Entry.pack() ec=tk.Button(root,text="Encrypt",height=2,width=8,command=Code)

ec.pack(side="top")

L2=tk.Label()

L2.pack()

L3=tk.Label()

L3.pack()

L4=tk.Label()

L4.pack()

**CALCULATION:**

Key: PLAYFAIREXAMPLE

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | Zayahan hasan shah | VFFYDFSCFOLOSFD |
| DECODE | VFFYDFSCFOLOSFD | Zayahan hasan shah |

**Output:**

**** **A screenshot of a computer

Description automatically generated**

**Conclusion:**

The code above is running when the user enter the plaint text and it convert into cipher text with the help of key(unique key, no repetition of key).

**LAB: ROW COLUMNAR TRANSPOSITION CIPHER**

**Objective:** Encode the given input and then decode by using row column cipher technique.

**Software:** Code Editor: Vs Code, Language: Python

**Description:** A cipher in which encryption that rearranges the characters in the plaintext according to a certain system (a pattern key), while maintaining their identity.

**Method:** The message is written out in rows of a fixed length, and then read out again column by column, and the column are chosen in some scrambled order.

**Code:**

def encrypt\_transposition(plain\_text: str, key: str) -> str:

global cipher\_text

table = []

key\_lst = sorted(key)

k = 0

total\_rows = (len(plain\_text) + len(key) - 1) // len(key) # ceil(len(plain\_text) / len(key))

for \_ in range(total\_rows):

row = []

for \_ in range(len(key)):

if k > len(plain\_text) - 1:

row.append("-")

else:

row.append(plain\_text[k])

k += 1

table.append(row)

cipher = ""

for k\_indx in range(len(key)):

curr\_idx = key.index(key\_lst[k\_indx])

for row in table:

cipher += row[curr\_idx]

return cipher

def decrypt\_transposition(cipher: str, key: str) -> str:

total\_rows = (len(cipher) + len(key) - 1) // len(key) # ceil(len(cipher) / len(key))

total\_columns = len(key)

key\_lst = sorted(key)

table = [[""] \* total\_columns for \_ in range(total\_rows)]

k = 0

for col in range(total\_columns):

original\_index = key.index(key\_lst[col])

for row in range(total\_rows):

if k < len(cipher):

table[row][original\_index] = cipher[k]

k += 1

plain\_text = ""

for row in table:

plain\_text += "".join(row)

plain\_text = plain\_text.rstrip('-')

return plain\_text

def Code():

global cipher\_text,key

plain\_text = Entry.get()

key = "SECRET"

cipher\_text = encrypt\_transposition(plain\_text.replace(" ", "").upper(), key.upper())

L2.config(text=f"Cipher Text: {cipher\_text}")

ec.config(text="Decrypt",command=Decode) Quit=tk.Button(root,text="Quit",height=2,width=8,command=Quit\_)

Quit.pack(side="bottom")

def Decode():

decrypted\_text = decrypt\_transposition(cipher\_text, key.upper())

L2.config(text=f"Decrypted Text: {decrypted\_text}")

def Quit\_():

quit()

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

import tkinter as tk

import re

root=tk.Tk()

root.resizable(False,False)

Label1=tk.Label(text="Enter the text for encryption ")

Label1.pack(side="top")

Entry=tk.Entry()

Entry.pack() ec=tk.Button(root,text="Encrypt",height=2,width=8,command=Code)

ec.pack(side="top")

L2=tk.Label()

L2.pack()

**Calculation:**

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | zayahan | Y-A-A-A-ZNA- |
| DECODE | Y-A-A-A-ZNA- | zayahan |

**Output:**

**** **A screenshot of a computer

Description automatically generated**

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into cipher text with transposition of the original message.

**LAB: DES ALGORITHM**

**Objective:** Encode the given input and then decode by using DES technique.

**Description:** A symmetric key block cipher created in the early 1970s by an IBM team and adopted by a National Institute of Standards and Technology (NIST).

**Method:** It operates on plain text of a given size (64-bits) and returns ciphertext blocks of the same size. Each block is divided into two blocks of 32 bit each.

**Code:**

def Code():

#initail permutation

ip\_table = [

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

]

# PC1 permutation table

pc1\_table = [

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

]

# Define the left shift schedule for each round

shift\_schedule = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1]

# PC2 permutation table

pc2\_table = [

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

]

#expension

e\_box\_table = [

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

]

# S-box tables for DES

s\_boxes = [

# S-box 1

[

[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]

],

# S-box 2

[

[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]

],

# S-box 3

[

[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]

],

# S-box 4

[

[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]

],

# S-box 5

[

[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]

],

# S-box 6

[

[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]

],

# S-box 7

[

[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]

],

# S-box 8

[

[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]

]

]

p\_box\_table = [

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

]

ip\_inverse\_table = [

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 str\_to\_bin(user\_input):

# Convert the string to binary

binary\_representation = ''

for char in user\_input:

# Get ASCII value of the character and convert it to binary

binary\_char = format(ord(char), '08b')

binary\_representation += binary\_char

binary\_representation = binary\_representation[:64]

# Pad or truncate the binary representation to 64 bits

binary\_representation = binary\_representation[:64].ljust(64, '0')

# Print the binary representation

# print("Binary representation of input string: ", binary\_representation)

# print(len(binary\_representation), 'bits of input string')

return binary\_representation

def binary\_to\_ascii(binary\_str):

ascii\_str = ''.join([chr(int(binary\_str[i:i+8], 2)) for i in range(0, len(binary\_str), 8)])

return ascii\_str

def ip\_on\_binary\_rep(binary\_representation):

ip\_result = [None] \* 64

for i in range(64):

ip\_result[i] = binary\_representation[ip\_table[i] - 1]

# Convert the result back to a string for better visualization

ip\_result\_str = ''.join(ip\_result)

return ip\_result\_str

def key\_in\_binary\_conv():

# Original key (can be changed but it should be 8 char)

original\_key = 'abcdefgh'

binary\_representation\_key = ''

for char in original\_key:

# Convert the characters to binary and concatenate to form a 64-bit binary string

binary\_key = format(ord(char), '08b')

binary\_representation\_key += binary\_key

return binary\_representation\_key

def generate\_round\_keys():

# Key into binary

binary\_representation\_key = key\_in\_binary\_conv()

pc1\_key\_str = ''.join(binary\_representation\_key[bit - 1] for bit in pc1\_table)

# Split the 56-bit key into two 28-bit halves

c0 = pc1\_key\_str[:28]

d0 = pc1\_key\_str[28:]

round\_keys = []

for round\_num in range(16):

# Perform left circular shift on C and D

c0 = c0[shift\_schedule[round\_num]:] + c0[:shift\_schedule[round\_num]]

d0 = d0[shift\_schedule[round\_num]:] + d0[:shift\_schedule[round\_num]]

# Concatenate C and D

cd\_concatenated = c0 + d0

# Apply the PC2 permutation

round\_key = ''.join(cd\_concatenated[bit - 1] for bit in pc2\_table)

# Store the round key

round\_keys.append(round\_key)

return round\_keys

def encryption(user\_input):

binary\_rep\_of\_input = str\_to\_bin(user\_input)

# Initialize lists to store round keys

round\_keys = generate\_round\_keys()

ip\_result\_str = ip\_on\_binary\_rep(binary\_rep\_of\_input)

# the initial permutation result is devided into 2 halfs

lpt = ip\_result\_str[:32]

rpt = ip\_result\_str[32:]

# Assume 'rpt' is the 32-bit right half, 'lpt' is the 32-bit left half, and 'round\_keys' is a list of 16 round keys

for round\_num in range(16):

# Perform expansion (32 bits to 48 bits)

expanded\_result = [rpt[i - 1] for i in e\_box\_table]

# Convert the result back to a string for better visualization

expanded\_result\_str = ''.join(expanded\_result)

# Round key for the current round

round\_key\_str = round\_keys[round\_num]

xor\_result\_str = ''

for i in range(48):

xor\_result\_str += str(int(expanded\_result\_str[i]) ^ int(round\_key\_str[i]))

# Split the 48-bit string into 8 groups of 6 bits each

six\_bit\_groups = [xor\_result\_str[i:i+6] for i in range(0, 48, 6)]

# Initialize the substituted bits string

s\_box\_substituted = ''

# Apply S-box substitution for each 6-bit group

for i in range(8):

# Extract the row and column bits

row\_bits = int(six\_bit\_groups[i][0] + six\_bit\_groups[i][-1], 2)

col\_bits = int(six\_bit\_groups[i][1:-1], 2)

# Lookup the S-box value

s\_box\_value = s\_boxes[i][row\_bits][col\_bits]

# Convert the S-box value to a 4-bit binary string and append to the result

s\_box\_substituted += format(s\_box\_value, '04b')

# Apply a P permutation to the result

p\_box\_result = [s\_box\_substituted[i - 1] for i in p\_box\_table]

# # Convert the result back to a string for better visualization

# p\_box\_result\_str = ''.join(p\_box\_result)

# Convert LPT to a list of bits for the XOR operation

lpt\_list = list(lpt)

# Perform XOR operation

new\_rpt = [str(int(lpt\_list[i]) ^ int(p\_box\_result[i])) for i in range(32)]

# Convert the result back to a string for better visualization

new\_rpt\_str = ''.join(new\_rpt)

# Update LPT and RPT for the next round

lpt = rpt

rpt = new\_rpt\_str

# Print or use the RPT for each round

print('\n')

# At this point, 'lpt' and 'rpt' contain the final left and right halves after 16 rounds

# After the final round, reverse the last swap

final\_result = rpt + lpt

# Perform the final permutation (IP-1)

final\_cipher = [final\_result[ip\_inverse\_table[i] - 1] for i in range(64)]

# Convert the result back to a string for better visualization

final\_cipher\_str = ''.join(final\_cipher)

# Print or use the final cipher(binary)

# print("Final Cipher binary:", final\_cipher\_str, len(final\_cipher\_str))

# Convert binary cipher to ascii

final\_cipher\_ascii = binary\_to\_ascii(final\_cipher\_str)

L2.config(text=f"Final Cipher text: {final\_cipher\_ascii} {len(final\_cipher\_ascii)}")

ec.config(text="Decrypt",command=Decode) Quit=tk.Button(root,text="Quit",height=2,width=8,command=Quit\_)

Quit.pack(side="bottom",pady="5")

## print("Final Cipher text:", final\_cipher\_ascii , len(final\_cipher\_ascii))

return final\_cipher\_ascii

# decryption of cipher to origional

def decryption(final\_cipher):

global final\_cipher\_str

# Initialize lists to store round keys

round\_keys = generate\_round\_keys()

# Apply Initial Permutation

ip\_dec\_result\_str = ip\_on\_binary\_rep(final\_cipher)

lpt = ip\_dec\_result\_str[:32]

rpt = ip\_dec\_result\_str[32:]

for round\_num in range(16):

# Perform expansion (32 bits to 48 bits)

expanded\_result = [rpt[i - 1] for i in e\_box\_table]

# Convert the result back to a string for better visualization

expanded\_result\_str = ''.join(expanded\_result)

# print(expanded\_result\_str)

# Round key for the current round

round\_key\_str = round\_keys[15-round\_num]

# XOR between key and expanded result

xor\_result\_str = ''

for i in range(48):

xor\_result\_str += str(int(expanded\_result\_str[i]) ^ int(round\_key\_str[i]))

# Split the 48-bit string into 8 groups of 6 bits each

six\_bit\_groups = [xor\_result\_str[i:i+6] for i in range(0, 48, 6)]

# Initialize the substituted bits string

s\_box\_substituted = ''

# Apply S-box substitution for each 6-bit group

for i in range(8):

# Extract the row and column bits

row\_bits = int(six\_bit\_groups[i][0] + six\_bit\_groups[i][-1], 2)

col\_bits = int(six\_bit\_groups[i][1:-1], 2)

# Lookup the S-box value

s\_box\_value = s\_boxes[i][row\_bits][col\_bits]

# Convert the S-box value to a 4-bit binary string and append to the result

s\_box\_substituted += format(s\_box\_value, '04b')

# Apply a P permutation to the result

p\_box\_result = [s\_box\_substituted[i - 1] for i in p\_box\_table]

# Convert the result back to a string for better visualization

# p\_box\_result\_str = ''.join(p\_box\_result)

# Convert LPT to a list of bits for the XOR operation

lpt\_list = list(lpt)

# Perform XOR operation

new\_rpt = [str(int(lpt\_list[i]) ^ int(p\_box\_result[i])) for i in range(32)

# Convert the result back to a string for better visualization

new\_rpt\_str = ''.join(new\_rpt)

# Update LPT and RPT for the next round

lpt = rpt

rpt = new\_rpt\_str

# Print or use the RPT for each round

print('\n')

final\_result = rpt + lpt

# Perform the final permutation (IP-1)

final\_cipher = [final\_result[ip\_inverse\_table[i] - 1] for i in range(64)]

# Convert the result back to a string for better visualization

final\_cipher\_str = ''.join(final\_cipher)

# Print or use the final cipher

def Decode():

# binary cipher string to ascii

final\_cipher\_ascii = binary\_to\_ascii(final\_cipher\_str)

L2.config(text=f"Decryption of Cipher :{final\_cipher\_ascii}")

## print("Decryption of Cipher :", final\_cipher\_ascii)

return final\_cipher\_ascii

# Start

# user input

user\_input = Entry.get()

# Encryption

enc = encryption(user\_input)

# Decyption

# First we'll convert Final Cipher text into binary

enc\_to\_binary = str\_to\_bin(enc)

# we'll call the decryption function

dec = decryption(enc\_to\_binary)

def Quit\_():

quit()

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

import tkinter as tk

root=tk.Tk()

root.resizable(False,False)

Label1=tk.Label(text="Enter a String: ")

Label1.pack()

Entry=tk.Entry()

Entry.pack() ec=tk.Button(root,text="Encrypt",height=2,width=8,command=Code)

ec.pack(side="top",pady="5")

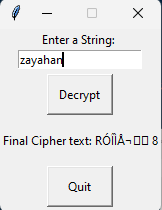
L2=tk.Label()

L2.pack(pady="10")

**Calculation:**

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | zayahan | Output in image |
| DECODE | Output in image | zayahan |

**Output:**

**** **A screenshot of a computer

Description automatically generated**

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into cipher text with key of 48-bits.

**LAB: RAIL FENCE CIPHER**

**Objective:** Encode the given input and then decode by using Rail Fence Cipher technique.

**Description:** A simple transposition technique cipher that encrypts plaintext by rearranging the characters in a zigzag pattern.

**Method:** A simple transposition technique in which plaintext is written zigzag manner and read in row-wise format.

**Code**

def encrypt\_rail\_fence(text, rails):

# Initialize rail matrix

rail\_matrix = [['' for j in range(len(text))] for i in range(rails)

# Direction variables for zigzag pattern

direction\_down = False

row, col = 0, 0

# Fill the rail matrix

for char in text:

if row == 0 or row == rails - 1:

direction\_down = not direction\_down

rail\_matrix[row][col] = char

col += 1

if direction\_down:

row += 1

else:

row -= 1

# Print the rail matrix for visualization

tl1=""

tl2=""

tl3=""

TL=[]

TL1=[]

TL2=[]

TL3=[]

L2.config(text="Rail Fence Zigzag Table (Encryption):")

for i in range(rails):

for j in range(len(text)):

if rail\_matrix[i][j] != '':

## print(rail\_matrix[i][j], end=' ')

TL.append(rail\_matrix[i][j])

else:

## print('=', end=' ')

TL.append("=")

## print()

if i == 0:

TL1=TL

TL=[]

if i == 1:

TL2=TL

TL=[]

if i == 2:

TL3=TL

TL=[]

for x in range(len(TL1)):

tl1+=TL1[x]+" "

tl2+=TL2[x]+" "

tl3+=TL3[x]+" "

L3.config(text=f"{tl1}\n{tl2}\n{tl3}")

ec.config(text="Decrypt",command=Decode)

# Read ciphertext row by row

cipher\_text = ''

for i in range(rails):

for j in range(len(text)):

if rail\_matrix[i][j] != '':

cipher\_text += rail\_matrix[i][j]

return cipher\_text

def decrypt\_rail\_fence(cipher\_text, rails):

# Initialize rail matrix

rail\_matrix = [['' for j in range(len(cipher\_text))] for i in range(rails)]

# Direction variables for zigzag pattern

direction\_down = None

row, col = 0, 0

# Fill the rail matrix with placeholders

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

rail\_matrix[row][col] = '\*'

col += 1

if row == 0:

direction\_down = True

elif row == rails - 1:

direction\_down = False

if direction\_down:

row += 1

else:

row -= 1

# Fill the rail matrix with characters from ciphertext

index = 0

for i in range(rails):

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

if rail\_matrix[i][j] == '\*' and index < len(cipher\_text):

rail\_matrix[i][j] = cipher\_text[index]

index += 1

# Print the rail matrix for visualization.

tl1=""

tl2=""

tl3=""

TL=[]

TL1=[]

TL2=[]

TL3=[]

L2.config(text="Rail Fence Zigzag Table (Decryption):")

for i in range(rails):

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

if rail\_matrix[i][j] != '':

TL.append(rail\_matrix[i][j])

## print(rail\_matrix[i][j], end=' ')

else:

TL.append("=")

## print('=', end=' ')

## print()

if i == 0:

TL1=TL

TL=[]

if i == 1:

TL2=TL

TL=[]

if i == 2:

TL3=TL

TL=[]

for x in range(len(TL1)):

tl1+=TL1[x]+" "

tl2+=TL2[x]+" "

tl3+=TL3[x]+" "

L3.config(text=f"{tl1}\n{tl2}\n{tl3}")

# Read plaintext column by column

plain\_text = ''

row, col = 0, 0

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

if row == 0:

direction\_down = True

elif row == rails - 1:

direction\_down = False

if rail\_matrix[row][col] != '\*':

plain\_text += rail\_matrix[row][col]

col += 1

if direction\_down:

row += 1

else:

row -= 1

return plain\_text

def Code():

global rails

global encrypted\_text

plaintext = str(Entry.get())

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

rails = 3

encrypted\_text = encrypt\_rail\_fence(plaintext, rails)

L4.config(text=f"Encrypted: {encrypted\_text}") Quit=tk.Button(root,text="Quit",height=2,width=8,command=Quit\_)

Quit.pack(side="bottom")

def Decode():

decrypted\_text = decrypt\_rail\_fence(encrypted\_text, rails)

L4.config(text=f"Decrypted: {decrypted\_text}")

def Quit\_():

quit()

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

import tkinter as tk

import re

root=tk.Tk()

root.resizable(False,False)

Label1=tk.Label(text="Enter the text for encryption ")

Label1.pack(side="top")

Entry=tk.Entry()

Entry.pack()

ec=tk.Button(root,text="Encrypt",height=2,width=8,command=Code)

ec.pack(side="top")

L2=tk.Label()

L2.pack()

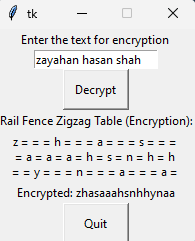
L3=tk.Label()

L3.pack()

L4=tk.Label()

L4.pack()

**Output:**

**** **A screenshot of a computer

Description automatically generated**

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into cipher text with zigzag manner of the original message.

**LAB: RSA ALGORITHM**

**Objective:** Encode the given input and then decode by using RSA technique.

**Method:** A cryptography algorithm that work on two different keys for encryption and decryption.

**Description:** A public key signature algorithm developed by Ron Rivest, Adi Shamir and Leonard Alderman.

**Code:**

def isPrime(number):

if number < 2 :

return False

for i in range(2, number // 2 + 1):

if number % i == 0:

return False

return True

def generatePrime(minValue, maxValue):

prime = random.randint(minValue, maxValue)

while not isPrime(prime):

prime = random.randint(minValue, maxValue)

return prime

def modInverse(e, phi):

for d in range(3, phi):

if (d \* e ) % phi == 1:

return d

raise ValueError("Mod inverse does not exist")

def Code():

global cipherText , d , n

p,q = generatePrime(1000, 5000), generatePrime(1000, 5000)

while p == q:

q = generatePrime(1000, 5000)

n = p \* q

phi\_n = (p-1) \* (q-1)

e = random.randint(3, phi\_n-1)

while math.gcd(e, phi\_n) != 1:

e = random.randint(3, phi\_n-1)

d = modInverse(e, phi\_n)

message = Entry.get()

message = message.lower()

message\_encoded = [ord(c) for c in message]

# encryption : (m^e) mod n = c

cipherText = [pow(c,e,n) for c in message\_encoded]

L2.config(text=cipherText)

ec.config(text="Decode",command=Decode)

Quit=tk.Button(root,text="Quit",height=2,width=8,command=Quit\_)

Quit.pack(side="bottom")

def Decode():

message\_decoded = [pow(ch, d, n) for ch in cipherText]

dec\_message = "".join(chr(ch) for ch in message\_decoded)

L2.config(text=dec\_message)

def Quit\_():

quit()

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

import tkinter as tk

import random

import math

root=tk.Tk()

root.resizable(False,False)

Label1=tk.Label(text="Enter the text for encryption ")

Label1.pack(side="top")

Entry=tk.Entry()

Entry.pack()

ec=tk.Button(root,text="Encrypt",height=2,width=8,command=Code)

ec.pack(side="top")

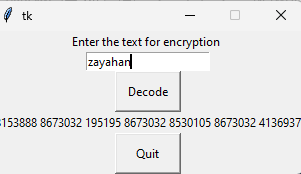
L2=tk.Label()

L2.pack()

**Calculation:**

|  |  |  |
| --- | --- | --- |
|  | INPUT | OUTPUT |
| ENCODE | zayahan | Output in image |
| DECODE | Output in image | zayahan |

**Output:**

**** **A screenshot of a computer

Description automatically generated**

**CONCLUSION:**

The code above is running when the user enter the plaint text and it convert into cipher text with help of public and private key of the original message.