## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

## Date: 11/01/2024

## Due Date for Submission: 20/01/2024

**Assignment-1.**

**IMPLEMENT CEASER CIPHER ENCRYPTION AND DECRYPTION:**

**Caesar Cipher:**

The Caesar cipher is named after Julius Caesar, who is said to have used this method to communicate with his generals. It is a type of monoalphabetic substitution cipher, meaning that each letter is replaced by another letter with a fixed shift. While the Caesar cipher is straightforward to implement and understand, it is not secure against modern cryptographic analysis. The key space is limited to 25 possible keys, making it vulnerable to brute-force attacks. Despite its lack of security, the Caesar cipher is a fundamental concept in cryptography and serves as the basis for more complex encryption algorithms.

**Code [In Python]:**

def encrypt(text,s):

result = ""

for i in range(len(text)):

char = text[i]

if(char==' '):

result += ' '

elif (char.isupper()):

result += chr((ord(char) + s-65) % 26 + 65)

else:

result += chr((ord(char) + s - 97) % 26 + 97)

return result

def decrypt(text,s):

result = ""

for i in range(len(text)):

char = text[i]

if(char==' '):

result += ' '

elif (char.isupper()):

result += chr((ord(char) - s - 65) % 26 + 65)

else:

result += chr((ord(char) - s - 97) % 26 + 97)

return result

while(True):

x = int(input("\n\nEnter\n0 for Exit: \n1 for Encrypt: \n2 for Decrypt: \n"))

if x==0:

break

elif x==1:

text = input("Text : ")

s = int(input("Shift : "))

print ("Encrypted Text: " + encrypt(text,s))

elif x==2:

text = input("Text : ")

s = int(input("Shift : "))

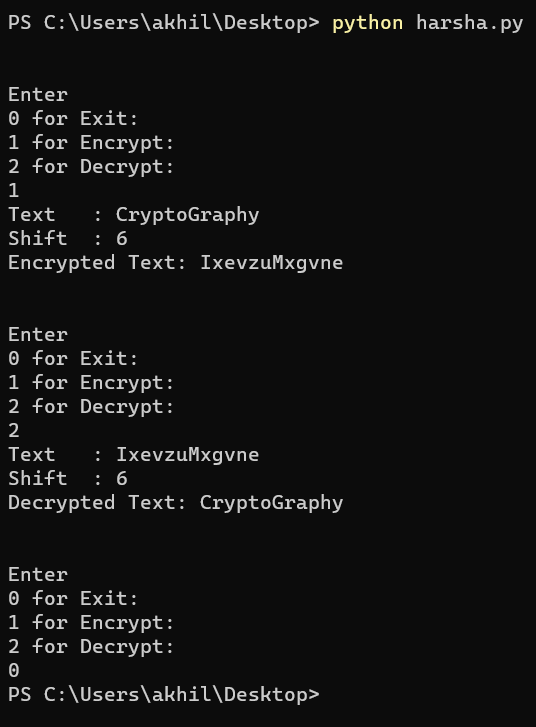
print ("Decrypted Text: " + decrypt(text,s))

else:

pass

**Output:**





**Result:**

Successfully Implemented the CEASER CIPHER ENCRYPTION AND DECRYPTION Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

## Date: 18/01/2024

## Due Date for Submission: 19/01/2024

**Assignment-2.**

**IMPLEMENT PLAYFAIR CIPHER ENCRYPTION**

**Playfair Cipher**:

Playfair cipher is an encryption algorithm to encrypt or encode a message. It is the same as a traditional cipher. The only difference is that it encrypts a digraph (a pair of two letters) instead of a single letter. It initially creates a key-table of 5\*5 matrix. The matrix contains alphabets that act as the key for the encryption of the plaintext. Note that any alphabet should not be repeated. Another point to note that there are 26 alphabets and we have only 25 blocks to put a letter inside it. Therefore, one letter is excess so, a letter will be omitted (usually J) from the matrix. Nevertheless, the plaintext contains J, then J is replaced by I. It means treat I and J as the same letter, accordingly. Since Playfair cipher encrypts the message digraph by digraph. Therefore, the Playfair cipher is an example of a digraph substitution cipher.

**Code [In Python]:**

def generate\_key\_square(key):

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

key\_square = list(key)

for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":

if char not in key:

key\_square.append(char)

key\_square = [key\_square[i:i+5] for i in range(0, 25, 5)]

return key\_square

def find\_position(key\_square, char):

for i in range(5):

for j in range(5):

if key\_square[i][j] == char:

return i, j

def encrypt(plaintext, key):

key\_square = generate\_key\_square(key)

print("Key Square:")

for row in key\_square:

print(" ".join(row))

ciphertext = ""

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

digraph = plaintext[i:i+2]

if len(digraph) == 1:

digraph += "X"

row1, col1 = find\_position(key\_square, digraph[0])

row2, col2 = find\_position(key\_square, digraph[1])

if row1 == row2:

ciphertext += key\_square[row1][(col1+1)%5] + key\_square[row2][(col2+1)%5]

elif col1 == col2:

ciphertext += key\_square[(row1+1)%5][col1] + key\_square[(row2+1)%5][col2]

else:

ciphertext += key\_square[row1][col2] + key\_square[row2][col1]

return ciphertext

def decrypt(ciphertext, key):

key\_square = generate\_key\_square(key)

plaintext = ""

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

digraph = ciphertext[i:i+2]

row1, col1 = find\_position(key\_square, digraph[0])

row2, col2 = find\_position(key\_square, digraph[1])

if row1 == row2:

plaintext += key\_square[row1][(col1-1)%5] + key\_square[row2][(col2-1)%5]

elif col1 == col2:

plaintext += key\_square[(row1-1)%5][col1] + key\_square[(row2-1)%5][col2]

else:

plaintext += key\_square[row1][col2] + key\_square[row2][col1]

return plaintext

while True:

choice = input("Enter 1 for encryption, 2 for decryption, or any other key to exit: ")

if choice == "1":

key = input("Enter the key: ")

key = key.upper()

plaintext = input("Enter the plaintext: ")

plaintext = plaintext.upper()

ciphertext = encrypt(plaintext, key)

print("Ciphertext:", ciphertext)

elif choice == "2":

key = input("Enter the key: ")

key = key.upper()

ciphertext = input("Enter the ciphertext: ")

ciphertext = ciphertext.upper()

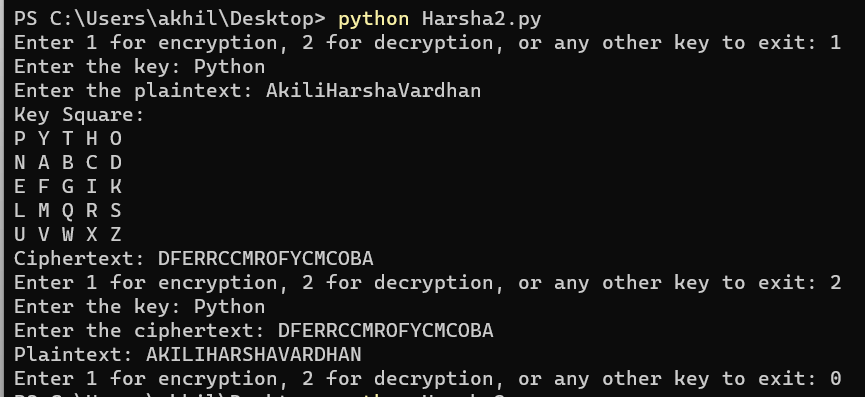
plaintext = decrypt(ciphertext, key)

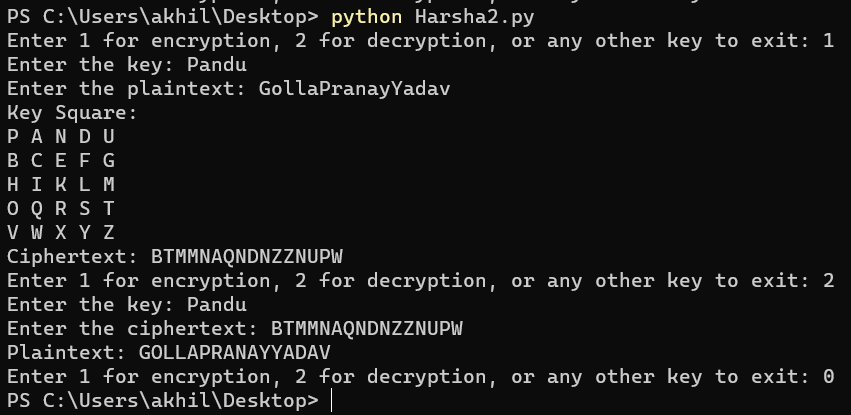
print("Plaintext:", plaintext)

else:

break

**Output:**





**Result:**

Successfully Implemented the PLAYFAIR CIPHER ENCRYPTION AND DECRYPTION Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

## Date: 20/01/2024

## Due Date for Submission: 20/01/2024

**Assignment-3.**

**IMPLEMENTING HILL CIPHER ENCRYPTION AND DECRYPTION:**

**Introduction:**

Hill cipher is a symmetric key encryption technique that operates on blocks of plaintext. It was developed by Lester S. Hill in 1929. Unlike traditional substitution or transposition ciphers, the Hill cipher employs linear algebraic operations for encryption and decryption.

The key feature of the Hill cipher is the use of matrix calculations. The plaintext is divided into blocks, and each block is then mapped into numerical values according to a predetermined scheme. These values are arranged into a matrix, known as the message matrix. The encryption key, called the key matrix, is also a square matrix. To encrypt the message matrix, matrix multiplication is performed between the message matrix and the key matrix.

Decryption is achieved by multiplying the encrypted matrix with the inverse of the key matrix. The resulting matrix is converted back to numerical values and mapped to their corresponding plaintext characters.

Some use cases of the Hill cipher include:

1. Secure Communication: Hill cipher provides a means to securely transmit sensitive information over insecure channels, such as the internet. It ensures the confidentiality of the transmitted data through encryption.

2. Military Applications: Hill cipher has been historically used in military communications and cryptography systems to protect classified information.

3. Banking and Finance: The Hill cipher can be used to secure financial transactions and protect confidential banking information.

**Important information about Hill cipher:**

1. Key Matrix Size: The size of the key matrix determines the strength of the encryption. It needs to be a square matrix with a defined size, usually 2x2 or 3x3. Larger matrix sizes provide stronger encryption but also increase computational complexity.

2. Invertibility: For decryption to be possible, the key matrix must be invertible. If the key matrix is not invertible, the decryption process will not yield the correct plaintext.

3. Modulo Arithmetic: Hill cipher often employs modulo arithmetic to bring the elements of the encrypted matrix within a specified range. This ensures that the resulting matrix can be mapped to valid characters during decryption.

4. Statistical Attacks: Hill cipher is susceptible to certain statistical attacks when the encryption block size is small. The choice of an appropriate block size is crucial to mitigate such attacks.

It is important to note that while the Hill cipher provides a level of security, modern encryption algorithms like AES and RSA are generally preferred for strong cryptographic protection.

**Code [In Python]:**

import numpy as np

from sympy import Matrix

def encrypt(message, key):

message = message.upper()

key = key.upper()

message = ''.join(i for i in message if i.isalpha())

key = ''.join(i for i in key if i.isalpha())

key\_length = len(key)

key\_root = int(np.sqrt(key\_length))

if key\_root \*\* 2 != key\_length:

print("Invalid key: not a square matrix")

return

message\_vector = [ord(i) - 65 for i in message]

key\_vector = [ord(i) - 65 for i in key]

key\_matrix = Matrix(np.reshape(key\_vector, (key\_root, key\_root)))

if key\_matrix.det() % 26 == 0:

print("Invalid key: not invertible modulo 26")

return

message\_length = len(message\_vector)

if message\_length % key\_root != 0:

message\_vector.extend([0] \* (key\_root - message\_length % key\_root))

message\_blocks = [message\_vector[i:i + key\_root] for i in range(0, message\_length, key\_root)]

cipher\_blocks = [key\_matrix \* Matrix(block) % 26 for block in message\_blocks]

cipher\_vector = [i for block in cipher\_blocks for i in block]

cipher\_text = ''.join(chr(i + 65) for i in cipher\_vector)

return cipher\_text

def decrypt(cipher\_text, key):

cipher\_text = cipher\_text.upper()

key = key.upper()

cipher\_text = ''.join(i for i in cipher\_text if i.isalpha())

key = ''.join(i for i in key if i.isalpha())

key\_length = len(key)

key\_root = int(np.sqrt(key\_length))

if key\_root \*\* 2 != key\_length:

print("Invalid key: not a square matrix")

return

cipher\_vector = [ord(i) - 65 for i in cipher\_text]

key\_vector = [ord(i) - 65 for i in key]

key\_matrix = Matrix(np.reshape(key\_vector, (key\_root, key\_root)))

if key\_matrix.det() % 26 == 0:

print("Invalid key: not invertible modulo 26")

return

key\_inverse = key\_matrix.inv\_mod(26)

cipher\_blocks = [cipher\_vector[i:i + key\_root] for i in range(0, len(cipher\_vector), key\_root)]

message\_blocks = [key\_inverse \* Matrix(block) % 26 for block in cipher\_blocks]

message\_vector = [i for block in message\_blocks for i in block]

message\_text = ''.join(chr(i + 65) for i in message\_vector)

return message\_text

while True:

choice = int(input("\nEnter 1 for Encryption OR 2 for Decryption: OR 3 for Exit: "))

if choice == 1 :

message = input("Enter the message: ")

key = input("Enter the key: ")

cipher\_text = encrypt(message, key)

print("The encrypted message is:", cipher\_text)

elif choice == 2 :

message = input("Enter the message: ")

key = input("Enter the key: ")

message\_text = decrypt(cipher\_text, key)

print("The decrypted message is:", message\_text)

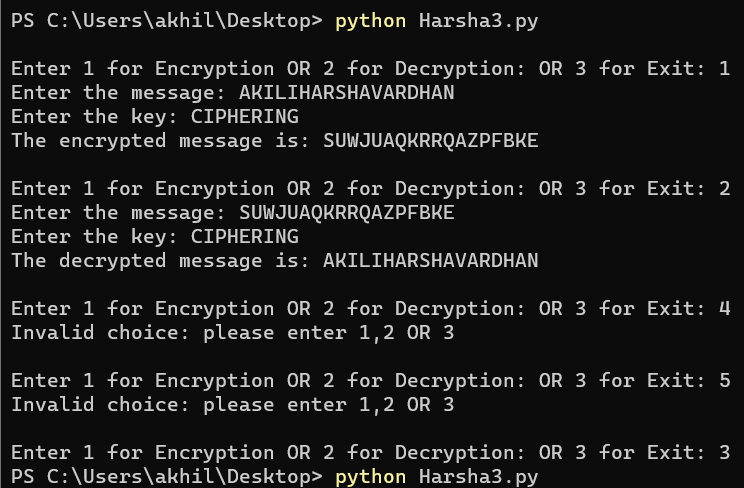
elif choice == 3:

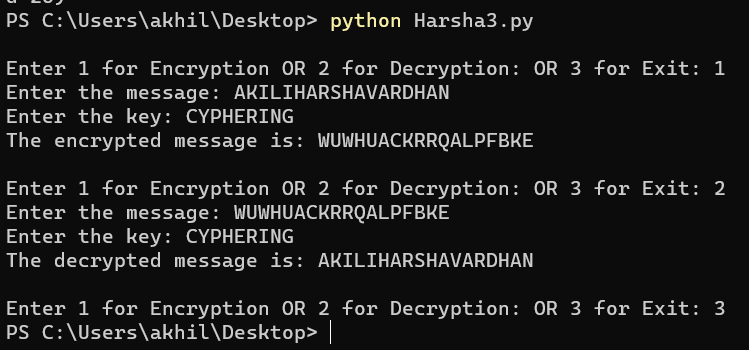
break

else:

print("Invalid choice: please enter 1,2 OR 3")

**Output:**

****

****

**Result:**

Successfully Implemented the HILL CIPHER ENCRYPTION AND DECRYPTION Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

**Assignment-4.**

**IMPLEMENTATION OF EUCLIDEAN ALGORITHM**

**Introduction:**

Extended Euclidean Algorithm Additionally, the extended Euclidean method disovers integer coefficients x and y such that axe + by = gcd(a, b) is true. The results of the recursive call gcd(b%a, a) are used by the extended Euclidean algorithm to update the results of gcd(a, b). Let x1 and y1 be the recursively derived values for x and y. Using the equations shown below, x and y are updated.

1. ax + by = gcd(a, b)   
2. gcd(a, b) = gcd(b%a, a)   
3. gcd(b%a, a) = (b%a)x1 + ay1   
4. ax + by = (b%a)x1 + ay1   
5. ax + by = (b - [b/a] \* a)x1 + ay1   
6. ax + by = a(y1 - [b/a] \* x1) + bx1   
7. Comparing LHS and RHS,   
8. x = y1 - ⌊ ⌋ b/a \* x1   
9. y = x1

**Code [In Python]:**

def extended\_euclidean\_alg(a, b):

x = []

y = []

q\_i = int(a/b)

x.append(0)

y.append(1)

x.append(1)

y.append(-q\_i)

a, b = b, a % b

i = 2

print(f'Initial setup: x[0]={x[0]}, y[0]={y[0]}, x[1]={x[1]}, y[1]={y[1]}, a={a}, b={b}')

while (b > 0):

r\_i = a % b

q\_i = int(a/b)

a, b = b, a % b

x.append(x[i-2] - q\_i \* x[i-1])

y.append(y[i-2] - q\_i \* y[i-1])

print(f'Step {i}: x[{i}]={x[i]}, y[{i}]={y[i]}, a={a}, b={b}')

i += 1

print(f'Final result: gcd={a}, x={x[i-2]}, y={y[i-2]}')

return (a, x[i-2], y[i-2])

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

print("Enter values of a & b :")

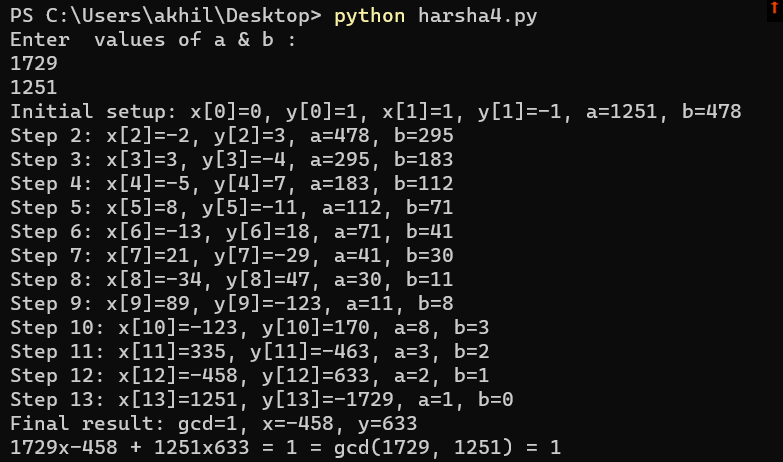
a = int(input())

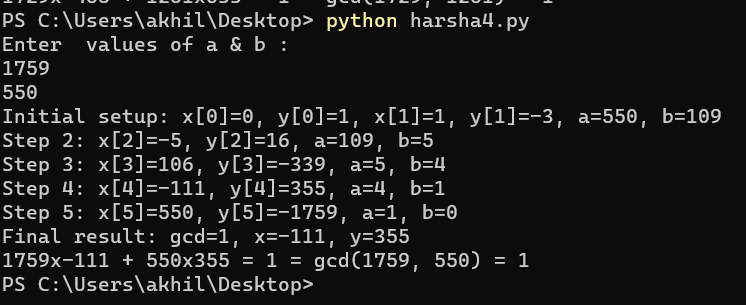
b = int(input())

d, x, y = extended\_euclidean\_alg(a, b)

print(f'{a}x{x} + {b}x{y} = {a\*x + b\*y} = gcd({a}, {b}) = {d}')

**Output:**





**Result:**

Successfully Implemented the **EUCLIDEAN ALGORITHM** Using Python.

Name : Akili Harsha Vardhan Reg No : 21BCE5353

Faculty Name: RAJESH R Lab Slot : L53+L54

Subject : Cryptography and Network Security Lab (BCSE309P)

Date: 01/02/2024

Due Date for Submission: 02/02/2024

# **Assignment-5.**

**IMPLEMENTATION of DES algorithm:**

# **Introduction:**

# Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security. Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits.

**Key Steps in Implementation:**

1. Initial Permutation (IP):
   * Rearrange the 64-bit plaintext block using a fixed permutation table.
2. Key Generation:
   * Generate 16 round keys from the 56-bit key through:
     + Permutation Choice 1 (PC-1)
     + 16 Left Circular Shifts
     + Permutation Choice 2 (PC-2)
3. 16 Rounds of Encryption:
   * Each round involves:
     + Expansion (E): Expand the 32-bit right half to 48 bits.
     + Key Mixing (XOR): XOR the expanded right half with the round key.
     + Substitution (S-boxes): Divide the 48-bit result into 8 blocks, each passing through an S-box to produce a 32-bit output.
     + Permutation (P-box): Permute the 32-bit output.
     + Swap: Swap the left and right halves (except in the last round).
4. Final Permutation (IP^-1):
   * Apply the inverse of the initial permutation to produce the 64-bit ciphertext.

**Decryption:**

* Uses the same algorithm in reverse, with round keys applied in reverse order.

Implementation Considerations:

* Programming Language: C, C++, Java, Python, and others are suitable.
* Libraries: Some languages provide built-in cryptographic libraries
* Key Management: Securely generate, store, and exchange keys.
* Security Concerns:
  + DES is considered insecure due to its small key size, making it vulnerable to brute-force attacks.
  + Use Triple DES or more modern alternative like AES for secure encryption.

## **Code [In Python]:**

def hex2bin(s):

mp = {'0': "0000",

'1': "0001",

'2': "0010",

'3': "0011",

'4': "0100",

'5': "0101",

'6': "0110",

'7': "0111",

'8': "1000",

'9': "1001",

'A': "1010",

'B': "1011",

'C': "1100",

'D': "1101",

'E': "1110",

'F': "1111"}

bin = ""

for i in range(len(s)):

bin = bin + mp[s[i]]

return bin

# Binary to hexadecimal conversion

def bin2hex(s):

mp = {"0000": '0',

"0001": '1',

"0010": '2',

"0011": '3',

"0100": '4',

"0101": '5',

"0110": '6',

"0111": '7',

"1000": '8',

"1001": '9',

"1010": 'A',

"1011": 'B',

"1100": 'C',

"1101": 'D',

"1110": 'E',

"1111": 'F'}

hex = ""

for i in range(0, len(s), 4):

ch = ""

ch = ch + s[i]

ch = ch + s[i + 1]

ch = ch + s[i + 2]

ch = ch + s[i + 3]

hex = hex + mp[ch]

return hex

# Binary to decimal conversion

def bin2dec(binary):

binary1 = binary

decimal, i, n = 0, 0, 0

while(binary != 0):

dec = binary % 10

decimal = decimal + dec \* pow(2, i)

binary = binary//10

i += 1

return decimal

# Decimal to binary conversion

def dec2bin(num):

res = bin(num).replace("0b", "")

if(len(res) % 4 != 0):

div = len(res) / 4

div = int(div)

counter = (4 \* (div + 1)) - len(res)

for i in range(0, counter):

res = '0' + res

return res

# Permute function to rearrange the bits

def permute(k, arr, n):

permutation = ""

for i in range(0, n):

permutation = permutation + k[arr[i] - 1]

return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

s = ""

for i in range(nth\_shifts):

for j in range(1, len(k)):

s = s + k[j]

s = s + k[0]

k = s

s = ""

return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

ans = ""

for i in range(len(a)):

if a[i] == b[i]:

ans = ans + "0"

else:

ans = ans + "1"

return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table

initial\_perm = [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]

# Expansion D-box Table

exp\_d = [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]

# Straight Permutation Table

per = [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]

# S-box Table

sbox = [[[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]],

[[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]],

[[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]],

[[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]],

[[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]],

[[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]],

[[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]],

[[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]]]

# Final Permutation Table

final\_perm = [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 encrypt(pt, rkb, rk):

pt = hex2bin(pt)

# Initial Permutation

pt = permute(pt, initial\_perm, 64)

print("After initial permutation: ", bin2hex(pt))

# Splitting

left = pt[0:32]

right = pt[32:64]

for i in range(0, 16):

# Expansion D-box: Expanding the 32 bits data into 48 bits

right\_expanded = permute(right, exp\_d, 48)

# XOR RoundKey[i] and right\_expanded

xor\_x = xor(right\_expanded, rkb[i])

# S-boxex: substituting the value from s-box table by calculating row and column

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(

int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

# Straight D-box: After substituting rearranging the bits

sbox\_str = permute(sbox\_str, per, 32)

# XOR left and sbox\_str

result = xor(left, sbox\_str)

left = result

# Swapper

if(i != 15):

left, right = right, left

#print("Round ", i + 1, " ", bin2hex(left)," ", bin2hex(right), " ", rk[i])

# Combination

combine = left + right

# Final permutation: final rearranging of bits to get cipher text

cipher\_text = permute(combine, final\_perm, 64)

return cipher\_text

while(1):

xx = int(input("\n\n1 for Encrption, 2 for Decryption or 3 for Exit: "))

if (xx==1):

#pt = "123456ABCD132536"

pt = input("\nEnter Plain Text: ")

key = input("Enter Key : ")

#key = "AABB09182736CCDD"

elif (xx==2):

pt = input("\nEnter Cipher Text: ")

key = input("Enter Key : ")

elif (xx==3):

break

else:

print("\nEnter Correct Choice: ")

pass

if(xx==1 or xx==2):

# Key generation

# --hex to binary

key = hex2bin(key)

# --parity bit drop table

keyp = [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]

# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

# Number of bit shifts

shift\_table = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits

key\_comp = [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]

# Splitting

left = key[0:28] # rkb for RoundKeys in binary

right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []

rk = []

for i in range(0, 16):

# Shifting the bits by nth shifts by checking from shift table

left = shift\_left(left, shift\_table[i])

right = shift\_left(right, shift\_table[i])

# Combination of left and right string

combine\_str = left + right

# Compression of key from 56 to 48 bits

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

if (xx==1):

print("\nEncryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

print("Cipher Text : ", cipher\_text)

elif (xx==2):

print("\nDecryption")

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

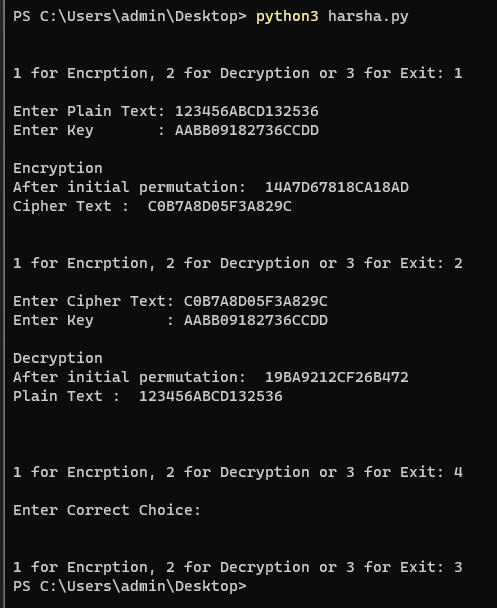
text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

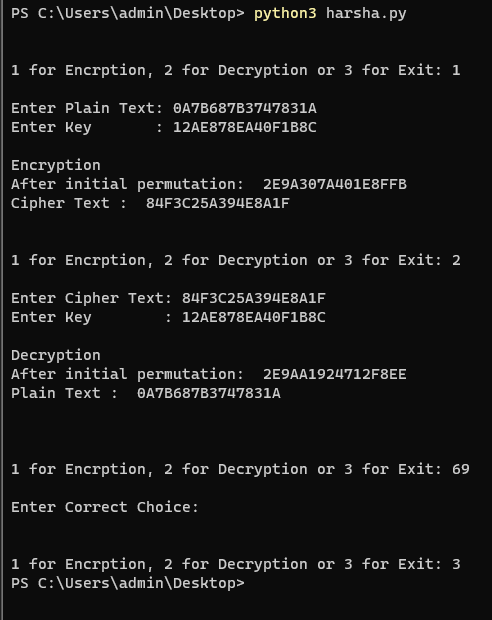
print("Plain Text : ", text,end='\n\n')

else:

pass

## **Output:**





**Result:**

Successfully Implemented the DES ALGORITHM Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

**Assignment-6.**

**IMPLEMENTATION OF RSA ALGORITHM**

**Introduction:**

RSA encryption algorithm is a type of public-key encryption algorithm. To better understand RSA, lets first understand what is public-key encryption algorithm.

Public key encryption algorithm:

Public Key encryption algorithm is also called the Asymmetric algorithm. Asymmetric algorithms are those algorithms in which sender and receiver use different keys for encryption and decryption. Each sender is assigned a pair of keys:

**Public key**

**Private key**

The Public key is used for encryption, and the Private Key is used for decryption. Decryption cannot be done using a public key. The two keys are linked, but the private key cannot be derived from the public key. The public key is well known, but the private key is secret and it is known only to the user who owns the key. It means that everybody can send a message to the user using user's public key. But only the user can decrypt the message using his private key.

**Example:**

* Choose p = 3 and q = 11
* Compute n = p \* q = 3 \* 11 = 33
* Compute φ(n) = (p - 1) \* (q - 1) = 2 \* 10 = 20
* Choose e such that 1 < e < φ(n) and e and φ (n) are coprime. Let e = 7
* Compute a value for d such that (d \* e) % φ(n) = 1. One solution is d = 3 [(3 \* 7) % 20 = 1]
* Public key is (e, n) => (7, 33)
* Private key is (d, n) => (3, 33)
* The encryption of m = 2 is c = 27 % 33 = 29
* The decryption of c = 29 is m = 293 % 33 = 2

**Code [In Python]:** **(for a=1 to z=26):**

import math

def str\_iarr(msg):

return [ord(char) - ord('a')+1 if 'a' <= char <= 'z' else 0 if char == 'a' else 26 if char == 'z' else char for char in msg.lower()]

def iarr\_str(iarr):

return ''.join(chr(i + ord('a')-1) if 0 <= i <= 25 else chr(i + ord('A')-1) - 32 for i in iarr)

def gcd(a, h):

temp = 0

while(1):

temp = a % h

if (temp == 0):

return h

a = h

h = temp

p = int(input("Enter p: "))

q = int(input("Enter q: "))

n = p\*q

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

print("phi = ",phi)

et = int(input("\nEnter '1' for Custom 'e' or else Random 'e' will be calculated: "))

if et!=1:

e = 2

while (e < phi):

# e must be co-prime to phi and

# smaller than phi.

if(gcd(e, phi) == 1):

break

else:

e = e+1

print(f"Public\_Key <{e},{n}>")

else:

e = int(input("Enter e: "))

print(f"Public\_Key <{e},{n}>")

# Private key (d stands for decrypt)

# choosing d such that it satisfies

# d\*e = 1 + k \* totient

dt = int(input("\nEnter '1' for Custom 'd' or else 'd' will be calculated: "))

if dt!=1:

for k in range(1,phi):

z = (k\*e)%phi

#print(k,k\*e,z)

if z==1:

d=k

break

d=math.ceil(d)

print(f"Private\_Key <{d},{n}>")

else:

d = int(input("Enter d: "))

print(f"Private\_Key <{d},{n}>")

# Message to be encrypted

msg\_type = int(input ("\nEnter '1' for Text\_Type or else Int\_Type will be taken: "))

msg=[]

if msg\_type==1:

msg = input("Enter Msg(str) : ")

msg = str\_iarr(msg)

print("Converted\_msg :",msg)

else:

msg2 = input("Enter Msg(int) : ")

msg2 = msg2.split(" ")

for i in msg2:

msg.append(int(i))

print('\nEncrypted data : ',end='')

# Encryption c = (msg ^ e) % n

c =[]

cc=[]

for i in msg:

a = i\*\*e

a = a%n

c.append(a)

print(a,end=" ")

for j in c:

cc.append(j%26)

if msg\_type==1:

print("\nConverted\_Encrypted:",iarr\_str(cc))

msg3=[]

print('\nOriginal Message : ',end='')

# Decryption m = (c ^ d) % n

for i in c:

m = i\*\*d

m = m%n

msg3.append(m)

print(m,end=" ")

if msg\_type==1:

msg4=[]

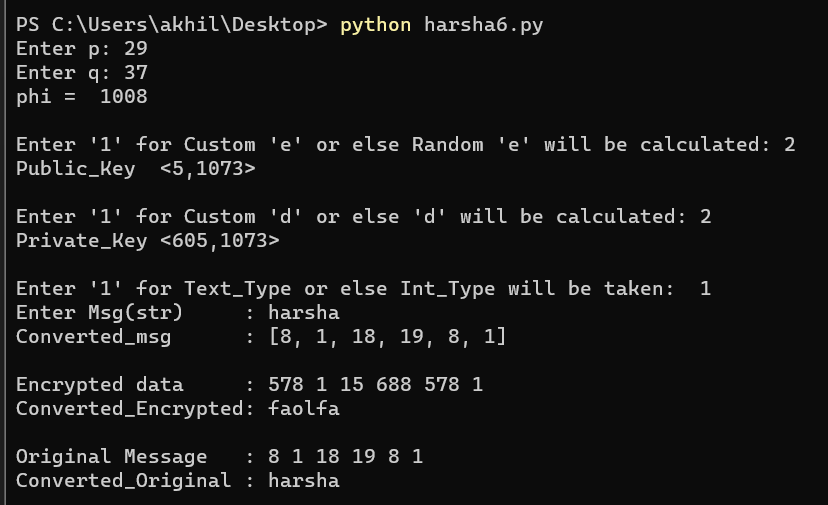
for j in msg3:

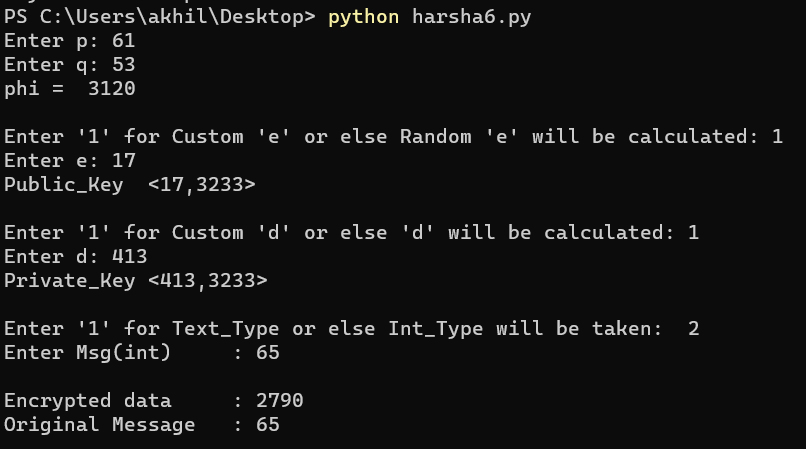
msg4.append(j%26)

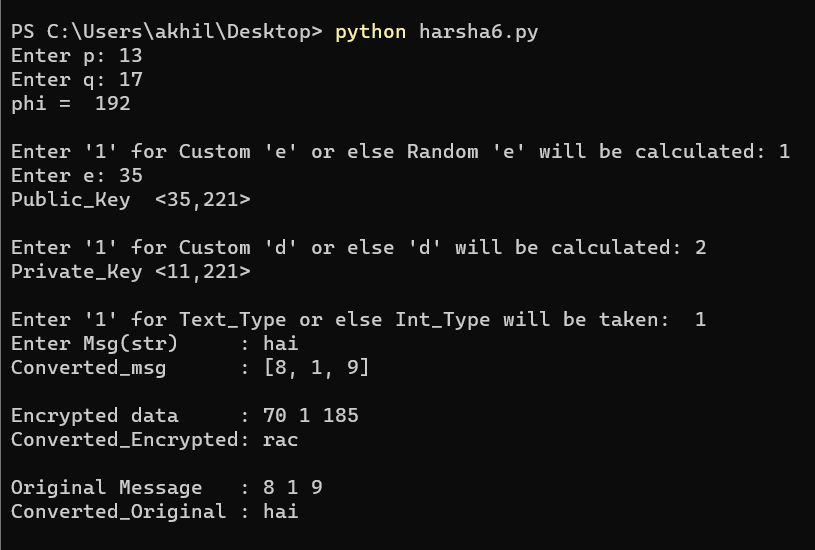
print("\nConverted\_Original :",iarr\_str(msg4))

print("\n")

**Output:**

****





**Code\_2 (for a=0 to z=25):**

import math

def str\_iarr(msg):

return [ord(char) - ord('a') if 'a' <= char <= 'z' else 0 if char == 'a' else 26 if char == 'z' else char for char in msg.lower()]

def iarr\_str(iarr):

return ''.join(chr(i + ord('a')) if 0 <= i <= 25 else chr(i + ord('A')) - 32 for i in iarr)

def gcd(a, h):

temp = 0

while(1):

temp = a % h

if (temp == 0):

return h

a = h

h = temp

p = int(input("Enter p: "))

q = int(input("Enter q: "))

n = p\*q

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

print("phi = ",phi)

et = int(input("\nEnter '1' for Custom 'e' or else Random 'e' will be calculated: "))

if et!=1:

e = 2

while (e < phi):

# e must be co-prime to phi and

# smaller than phi.

if(gcd(e, phi) == 1):

break

else:

e = e+1

print(f"Public\_Key <{e},{n}>")

else:

e = int(input("Enter e: "))

print(f"Public\_Key <{e},{n}>")

# Private key (d stands for decrypt)

# choosing d such that it satisfies

# d\*e = 1 + k \* totient

dt = int(input("\nEnter '1' for Custom 'd' or else 'd' will be calculated: "))

if dt!=1:

for k in range(1,phi):

z = (k\*e)%phi

#print(k,k\*e,z)

if z==1:

d=k

break

d=math.ceil(d)

print(f"Private\_Key <{d},{n}>")

else:

d = int(input("Enter d: "))

print(f"Private\_Key <{d},{n}>")

# Message to be encrypted

msg\_type = int(input ("\nEnter '1' for Text\_Type or else Int\_Type will be taken: "))

msg=[]

if msg\_type==1:

msg = input("Enter Msg(str) : ")

msg = str\_iarr(msg)

print("Converted\_msg :",msg)

else:

msg2 = input("Enter Msg(int) : ")

msg2 = msg2.split(" ")

for i in msg2:

msg.append(int(i))

print('\nEncrypted data : ',end='')

# Encryption c = (msg ^ e) % n

c =[]

cc=[]

for i in msg:

a = i\*\*e

a = a%n

c.append(a)

print(a,end=" ")

for j in c:

cc.append(j%26)

if msg\_type==1:

print("\nConverted\_Encrypted:",iarr\_str(cc))

msg3=[]

print('\nOriginal Message : ',end='')

# Decryption m = (c ^ d) % n

for i in c:

m = i\*\*d

m = m%n

msg3.append(m)

print(m,end=" ")

if msg\_type==1:

msg4=[]

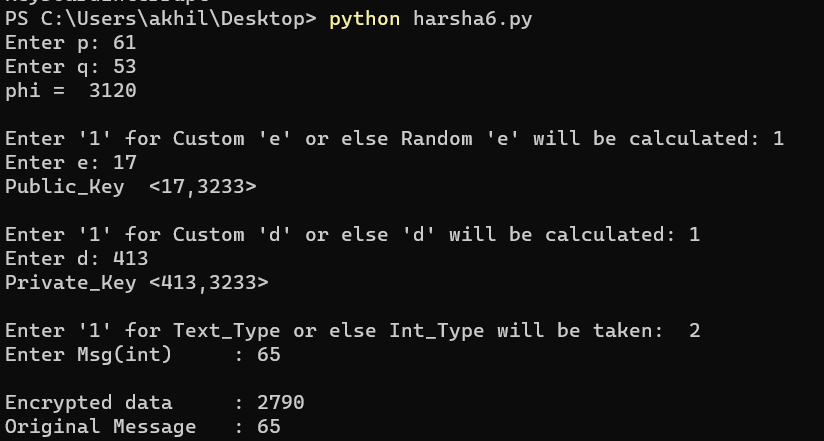
for j in msg3:

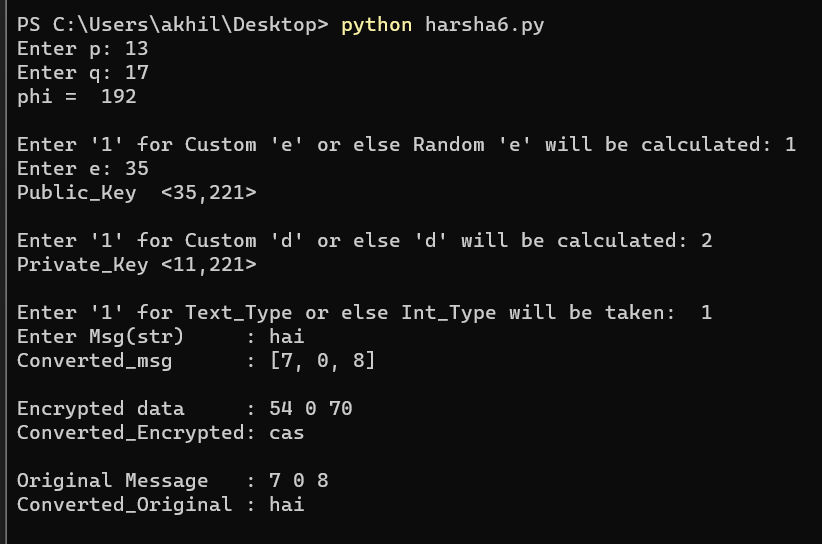
msg4.append(j%26)

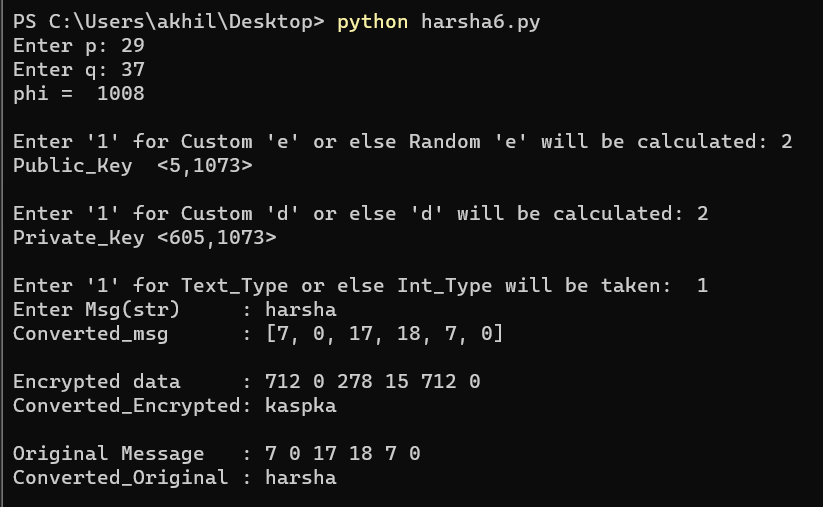
print("\nConverted\_Original :",iarr\_str(msg4))

print("\n")

**Output\_2:**







**Result:**

Successfully Implemented the RSA ALGORITHM’s ENCRYPTION AND DECRYPTION Using Python.

Name : Akili Harsha Vardhan Reg No : 21BCE5353

Faculty Name: RAJESH R Lab Slot : L53+L54

Subject : Cryptography and Network Security Lab (BCSE309P)

# **Assignment-7:**

**IMPLEMENTATION of Man in The Middle Attack with Diffie-Hellman Key Exchange:**

# **Introduction:**

# Diffie-Hellman Key Exchange algorithm is an advanced cryptographic method used to establish a shared secret (or shared secret key) that can be used to perform secret communication on a public network between Alice and Bob while preventing Eve (eavesdropper), who can eavesdrop on all their communication, from learning the generated secret.

# The key exchange procedure has two steps:

# One-time setup: We define some public parameters that are used by everyone forever.

# Protocol: To generate new secret keys, run a two-message key exchange protocol. This process is done using some simple algebra, prime numbers, and properties of modular arithmetic.

A malicious Darth, that has a MitM (man in the middle) position, can manipulate the   
communications between Alice and Bob, and break the security of the key exchange.

**Key Steps in Implementation:**

Step 1: Selected public numbers p and g, p is a prime number, called the “modulus” and g   
 is called the base.

Step 2: Selecting private numbers. Let Alice pick a private random number a and let Bob   
 pick a private random number b, Darth picks 2 random numbers c and d.

Step 3: Intercepting public values, Darth intercepts Alice’s public value (ga(mod p)), block   
 it from reaching Bob, and instead sends Bob her own public value (gc(modp)) and   
 Darth intercepts Bob’s public value (gb(mod p)), block it from reaching Alice, and   
 instead sends Alice her own public value (gd (modp))

Step 4: Computing secret key Alice will compute a key S1=gda(mod p), and Bob will   
 compute a different key, S2=gcb(mod p)

Step 5: If Alice uses S1 as a key to encrypt a later message to Bob, Darth can decrypt it,   
 re-encrypt it using S2, and send it to Bob. Bob and Alice won’t notice any problem   
 and may assume their communication is encrypted, but in reality, Darth can   
 decrypt, read, modify, and then re-encrypt all their conversations.

## C:\Users\student\Pictures\Screenshots\Screenshot 2024-02-29 174929.png

## **Code [In Python]:**

import random

# public keys are taken

# p is a prime number

# g is a primitive root of p

p = int(input('\nEnter a prime number : '))

g = int(input('Enter a number : '))

class A:

def \_\_init\_\_(self):

# Generating a random private number selected by alice

self.n = random.randint(1, p)

def publish(self):

# generating public values

return (g\*\*self.n)%p

def compute\_secret(self, gb):

# computing secret key

return (gb\*\*self.n)%p

class B:

def \_\_init\_\_(self):

# Generating a random private number selected for alice

self.a = random.randint(1, p)

# Generating a random private number selected for bob

self.b = random.randint(1, p)

self.arr = [self.a,self.b]

def publish(self, i):

# generating public values

return (g\*\*self.arr[i])%p

def compute\_secret(self, ga, i):

# computing secret key

return (ga\*\*self.arr[i])%p

alice = A()

bob = A()

eve = B()

# Printing out the private selected number by Alice and Bob

print(f'\nAlice selected (Xa) : {alice.n}')

print(f'Bob selected (Xb) : {bob.n}')

print(f'\nEve selected private number for Alice (Xd1) : {eve.a}')

print(f'Eve selected private number for Bob (Xd2) : {eve.b}\n')

# Generating public values

ga = alice.publish()

gb = bob.publish()

gea = eve.publish(0)

geb = eve.publish(1)

print(f'Alice published (Ya): {ga}')

print(f'Bob published (Yb): {gb}')

print(f'\nEve published value for Alice (Yd1): {gea}')

print(f'Eve published value for Bob (Yd2): {geb}\n')

# Computing the secret key

sa = alice.compute\_secret(gea)

sea = eve.compute\_secret(ga,0)

sb = bob.compute\_secret(geb)

seb = eve.compute\_secret(gb,1)

print(f'Alice computed (K1) : {sa}')

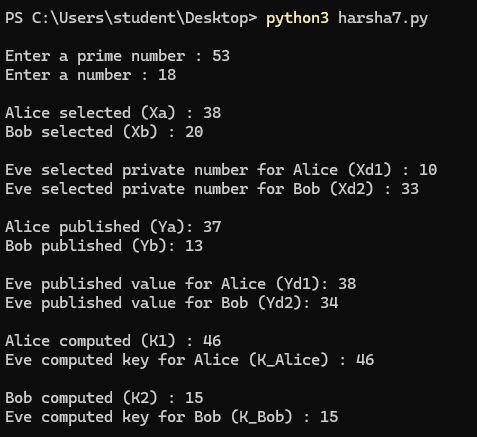
print(f'Eve computed key for Alice (K\_Alice) : {sea}\n')

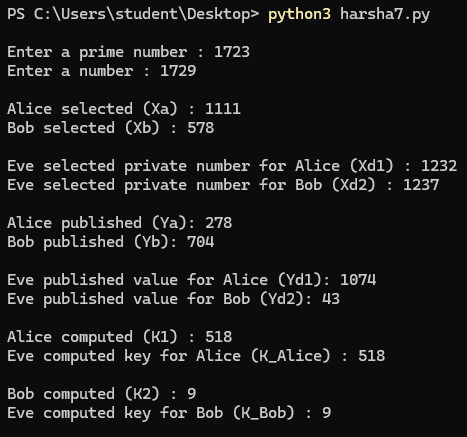
print(f'Bob computed (K2) : {sb}')

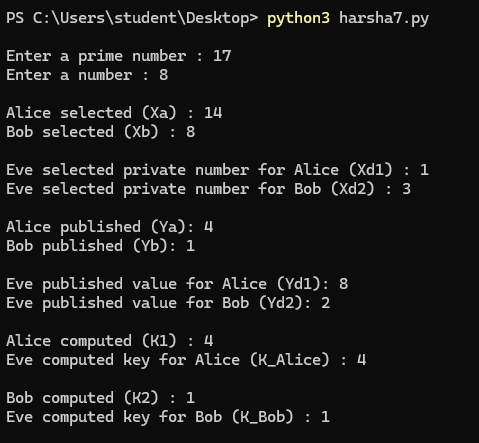
print(f'Eve computed key for Bob (K\_Bob) : {seb}\n')

## 

## **Output:**







**Result:**

Successfully Implemented the **Man in The Middle Attack with Diffie-Hellman Key   
 Exchange** Using Python.

Name : Akili Harsha Vardhan Reg No : 21BCE5353

Faculty Name: RAJESH R Lab Slot : L53+L54

Subject : Cryptography and Network Security Lab (BCSE309P)

# **Assignment-8:**

**IMPLEMENTATION of ElGamal Encryption and Decryption:**

# **Introduction:**

# ElGamal encryption is a public-key cryptosystem. It uses asymmetric key encryption for communicating between two parties and encrypting the message. This cryptosystem is based on the difficulty of finding discrete logarithm in a cyclic group that is even if we know ga and gk, it is extremely difficult to compute gak.

# Idea of ElGamal cryptosystem:

# Suppose Alice wants to communicate with Bob.

# Bob generates public and private keys:

# Bob chooses a very large number q and a cyclic group Fq.

# From the cyclic group Fq, he choose any element g and

# an element a such that gcd(a, q) = 1.

# Then he computes h = ga.

# Bob publishes F, h = ga, q, and g as his public key and retains a as private key.

# Alice encrypts data using Bob’s public key :

# Alice selects an element k from cyclic group F

# such that gcd(k, q) = 1.

# Then she computes p = gk and s = hk = gak.

# She multiples s with M.

# Then she sends (p, M\*s) = (gk, M\*s).

# Bob decrypts the message :

# Bob calculates s′ = pa = gak.

# He divides M\*s by s′ to obtain M as s = s′.

# The key exchange procedure has two steps:

## **Code [In Python]:**

import random

from math import pow

a = random.randint(2, 10)

def gcd(a, b):

if a < b:

return gcd(b, a)

elif a % b == 0:

return b;

else:

return gcd(b, a % b)

# Generating large random numbers

def gen\_key(q):

key = random.randint(pow(10, 20), q)

while gcd(q, key) != 1:

key = random.randint(pow(10, 20), q)

return key

# Modular exponentiation

def power(a, b, c):

x = 1

y = a

while b > 0:

if b % 2 != 0:

x = (x \* y) % c;

y = (y \* y) % c

b = int(b / 2)

return x % c

# Asymmetric encryption

def encrypt(msg, q, h, g):

en\_msg = []

k = gen\_key(q)# Private key for sender

s = power(h, k, q)

p = power(g, k, q)

for i in range(0, len(msg)):

en\_msg.append(msg[i])

print("g^k used : ", p)

print("g^ak used : ", s)

for i in range(0, len(en\_msg)):

en\_msg[i] = s \* ord(en\_msg[i])

return en\_msg, p

def decrypt(en\_msg, p, key, q):

dr\_msg = []

h = power(p, key, q)

for i in range(0, len(en\_msg)):

dr\_msg.append(chr(int(en\_msg[i]/h)))

return dr\_msg

def main():

msg = input("Enter the message to encrypt: ")

print("Original Message :", msg)

q = random.randint(pow(10, 20), pow(10, 50))

g = random.randint(2, q)

key = gen\_key(q)# Private key for receiver

h = power(g, key, q)

print("g used : ", g)

print("g^a used : ", h)

en\_msg, p = encrypt(msg, q, h, g)

dr\_msg = decrypt(en\_msg, p, key, q)

dmsg = ''.join(dr\_msg)

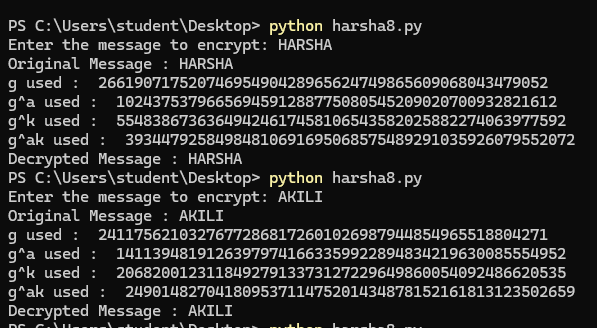
print("Decrypted Message :", dmsg);

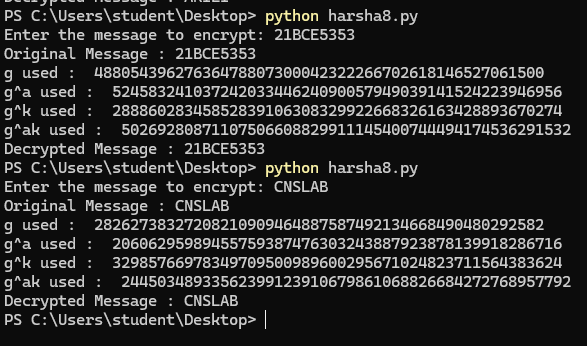
if \_\_name\_\_ == '\_\_main\_\_':

main()

## 

## **Output:**





**Result:**

Successfully Implemented the **ElGamal Encryption and Decryption** Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

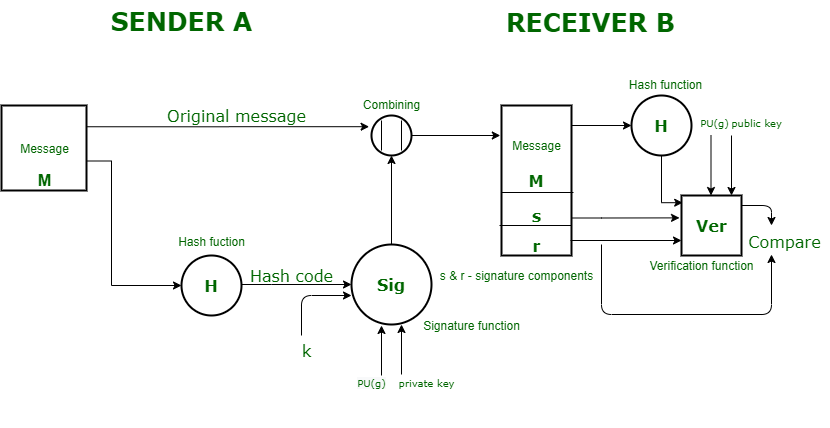
## Subject : Cryptography and Network Security Lab (BCSE309P)

**Assignment-9.**

**DIGITAL SIGNATURE STANADARD IMPLEMENTATION**

**Introduction:**

As we have studied, signature is a way of authenticating the data coming from a trusted individual. Similarly, digital signature is a way of authenticating a digital data coming from a trusted source. Digital Signature Standard (DSS) is a Federal Information Processing Standard (FIPS) which defines algorithms that are used to generate digital signatures with the help of Secure Hash Algorithm (SHA) for the authentication of electronic documents. DSS only provides us with the digital signature function and not with any encryption or key exchanging strategies.



**Sender Side:** In DSS Approach, a hash code is generated out of the message and following inputs are given to the signature function –

1. The hash codes.
2. The random number ‘k’ generated for that particular signature.
3. The private key of the sender i.e., PR(a).
4. A global public key (which is a set of parameters for the communicating principles) i.e., PU(g).

These input to the function will provide us with the output signature containing two components – ‘s’ and ‘r’. Therefore, the original message concatenated with the signature is sent to the receiver.

**Receiver Side:** At the receiver end, verification of the sender is done. The hash code of the sent message is generated. There is a verification function which takes the following inputs –

1. The hash code generated by the receiver.
2. Signature components ‘s’ and ‘r’.
3. Public key of the sender.
4. Global public key.

The output of the verification function is compared with the signature component ‘r’. Both the values will match if the sent signature is valid because only the sender with the help of it private key can generate a valid signature.

**Code [In Python]:**

import random

import hashlib

import sys

# Helper functions for modular exponentiation and modular inverse

def mod\_exp(base, exponent, modulus):

result = 1

while exponent > 0:

if exponent % 2 == 1:

result = (result \* base) % modulus

base = (base \* base) % modulus

exponent = exponent // 2

return result

def egcd(a, b):

x0, x1, y0, y1 = 1, 0, 0, 1

while b != 0:

q, a, b = a // b, b, a % b

x0, x1 = x1, x0 - q \* x1

y0, y1 = y1, y0 - q \* y1

return a, x0, y0

def mod\_inv(a, m):

g, x, y = egcd(a, m)

if g != 1:

raise Exception('Modular inverse does not exist')

else:

return x % m

# RSA key generation

def generate\_keypair(bits):

def is\_prime(n, k=5):

if n <= 1:

return False

if n <= 3:

return True

if n % 2 == 0:

return False

r = 0

s = n - 1

while s % 2 == 0:

r += 1

s //= 2

for \_ in range(k):

a = random.randrange(2, n - 1)

x = pow(a, s, n)

if x == 1 or x == n - 1:

continue

for \_ in range(r - 1):

x = pow(x, 2, n)

if x == n - 1:

break

else:

return False

return True

def generate\_prime(bits):

while True:

num = random.getrandbits(bits)

if is\_prime(num):

return num

p = generate\_prime(bits)

q = generate\_prime(bits)

n = p \* q

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

while True:

e = random.randrange(2, phi)

if egcd(e, phi)[0] == 1:

break

d = mod\_inv(e, phi)

return ((n, e), (n, d))

# Signature generation and verification

def sign(message, private\_key):

n, d = private\_key

hashed\_message = int.from\_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

signature = mod\_exp(hashed\_message, d, n)

return signature

def verify(message, signature, public\_key):

n, e = public\_key

hashed\_message = int.from\_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

decrypted\_signature = mod\_exp(signature, e, n)

return hashed\_message == decrypted\_signature

# Example usage

message = input("Enter the message :")

private\_key, public\_key = generate\_keypair(1024)

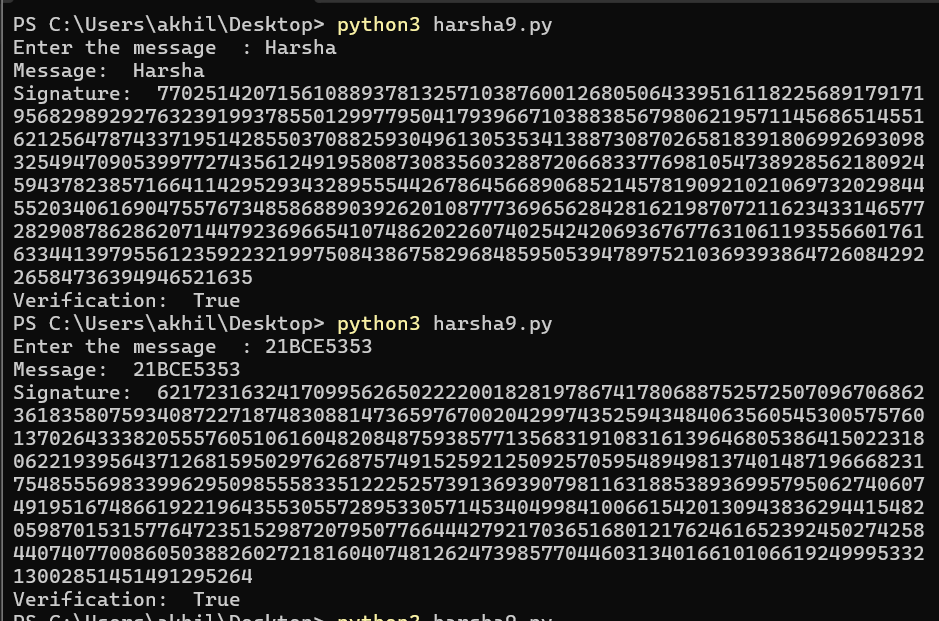
signature = sign(message, private\_key)

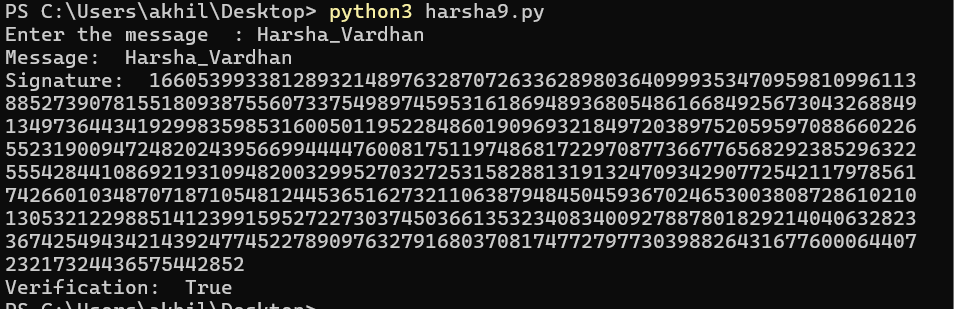
print("Message:", message)

print("Signature:", signature)

print("Verification:", verify(message, signature, public\_key))

**Output:**

****

****

**Result:**

Successfully Implemented the DIGITAL SIGNATURE STANADARD ENCRYPTION AND DECRYPTION Using Python.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

**Assignment-10.**

**To explore different features of the software WireShark which is used for tracing packet on a particular interface.**

**1)Capture the packets using Wireshark and depict the TCP connection establishment and termination process.**

**PROCEDURE:**

a.Open the capture menu and select the interface which is currently   
 connected.

b.Now select the promiscuous monitoring option to monitor all the   
 packets coming to the interface.

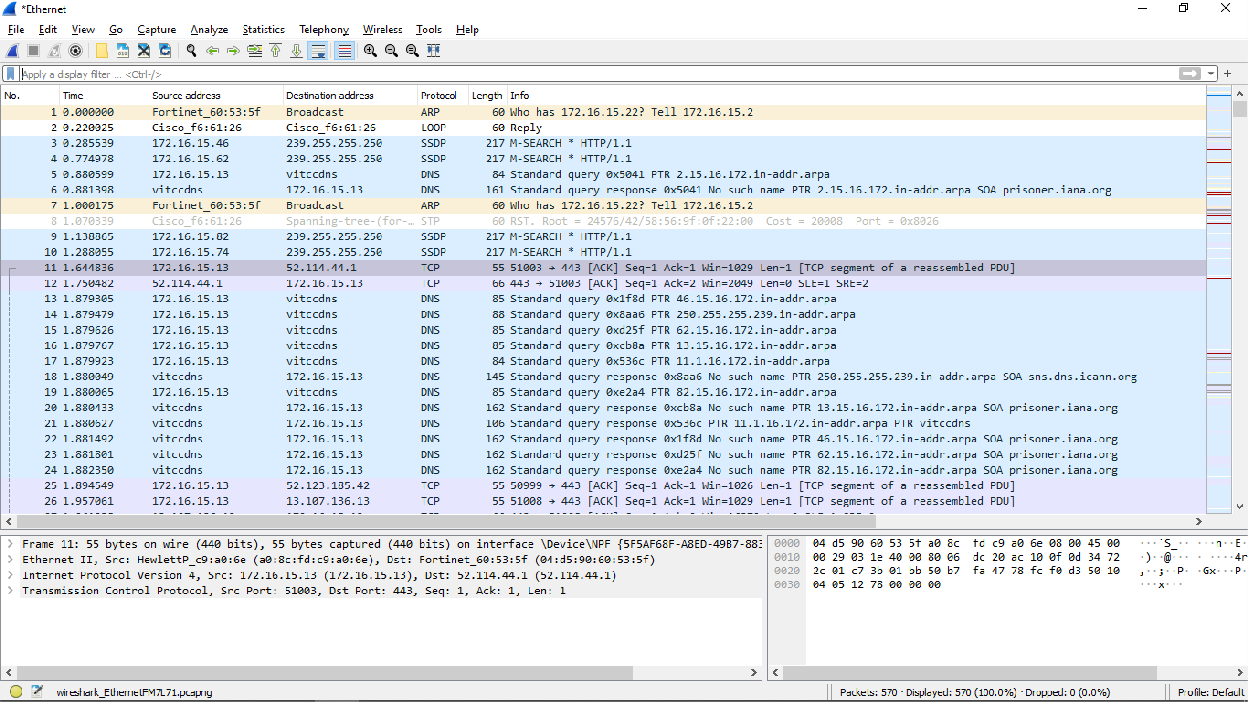
c.Now click the start option on the left bottom of dialog box.

d.It will start capturing the packets

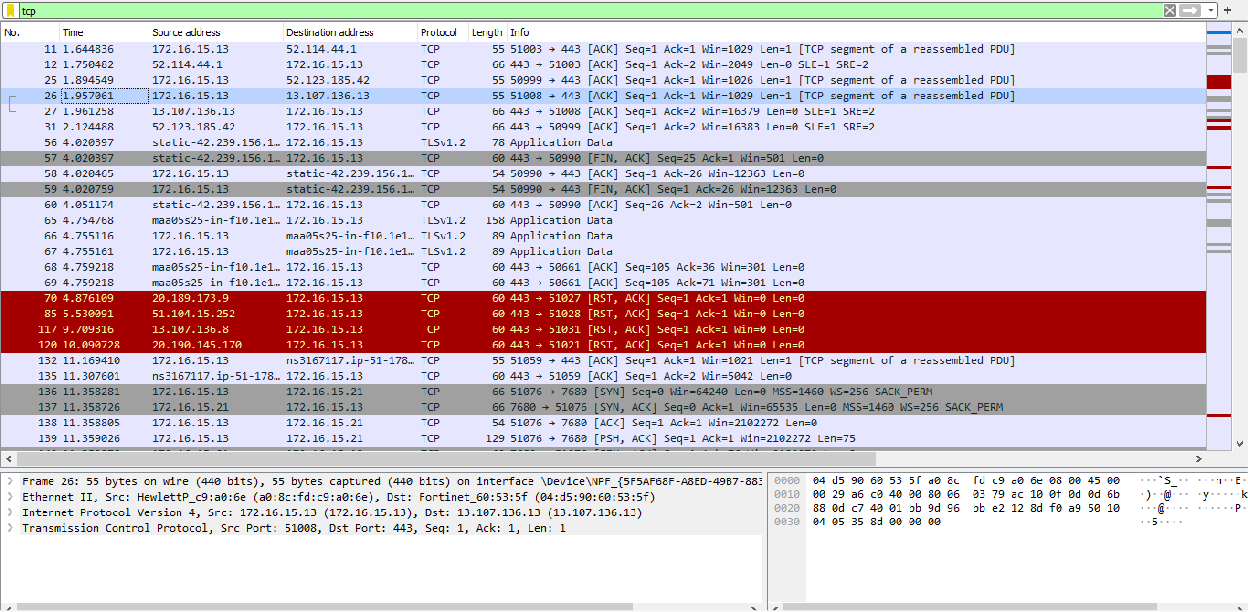
e.Stop the capturing of the packets.

f.Now type tcp in the display filter and press enter.

**Capturing the Ethernet packets using promiscuous monitoring:**



**Filtering the TCP connections out of all the packets received:**

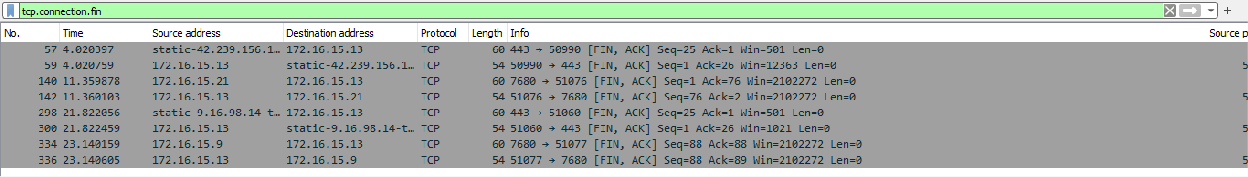


**2)Filter the TCP packets that contain the request to terminate.**

**PROCEDURE:**

a.For Filtering the tcp packets that contain the request to terminate we   
 have to choose fin packets which stands for finished.

b.So to filter out the packets which contains request to terminate type   
 tcp.connection.fin in the display filter and press enter.



**3)Depict the flowgraph and the I/O graph of TCP, UDP, ICMP, and ARP.**

**PROCEDURE:**

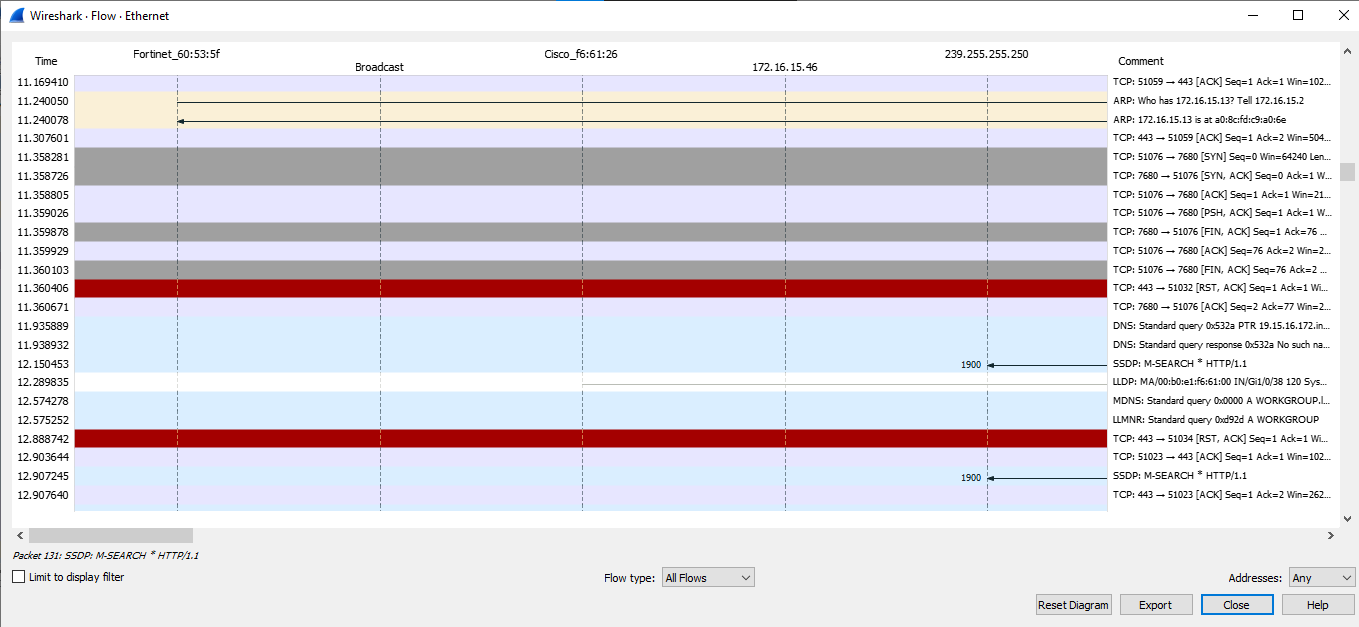
a.Go to Statistics menu and select the Flowgraph option.

b.A dialog box wil appear

c.Select All Flows in the Flow Type option

d.Observe the graph pattern and close the dialog box.

**Flowgraph of TCP, UDP and ARP:**



**PROCEDURE:**

a.Enter all the name of the packets you want to plot graph off.

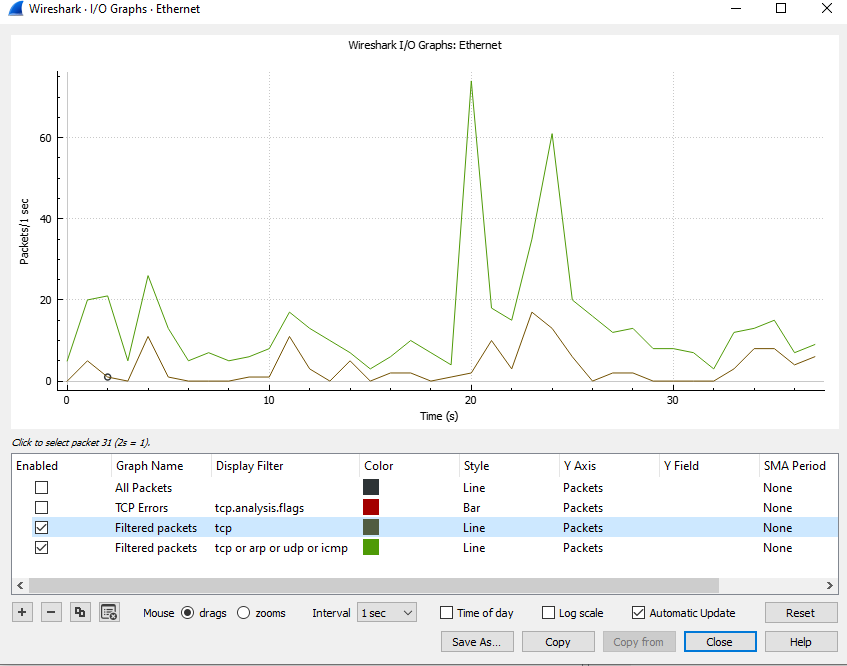
b.In this case tcp or udp or arp or icmp

c.Go to Statistics menu and select the I/O Graph option.

d.A dialog box will appear select the Filtered packets option.

e.Observe the graph pattern and close the dialog box.

**I/O Graph of TCP, ARP, UDP and ICMP:**



**RESULTS:**

Hence, we explored different features of WireShark and analysed the captured packets successfully.

## Name : Akili Harsha Vardhan

## Reg No : 21BCE5353

## Faculty Name: RAJESH R

## Lab Slot : L53+L54

## Subject : Cryptography and Network Security Lab (BCSE309P)

**Assignment-11 :** **JSON Web Token**

* **Aim:** : Develop a web application that implements JSON web token.
* **Procedure :**

Backend Setup (Choose one based on your preferred framework):

* 1. Project Setup: Create a new project in your chosen backend framework (e.g., Node.js with Express, Python with Flask, etc.).
  2. Dependency Management: Install necessary libraries for working with JWTs. Popular options include jsonwebtoken (Node.js) or PyJWT (Python).

User Management:

* 1. User Database: Implement a user database to store user credentials (username, password) securely. This can be a relational database (MySQL, PostgreSQL) or a NoSQL database (MongoDB).
  2. User Registration: Develop an API endpoint for user registration where new users can sign up with their credentials. Hash passwords securely before storing them (e.g., using bcrypt).

Authentication with JWT:

* 1. Login Endpoint: Create an API endpoint for user login. Users will provide their credentials for authentication.
  2. User Verification: Verify the username and password against the user database.
  3. Payload Creation: If login is successful, generate a JWT payload containing user information (e.g., user ID, username).
  4. Secret Key Setup: Define a strong, secret key for signing and verifying JWTs. Store this key securely (environment variables recommended).
  5. Token Signing: Use the JWT library and the secret key to sign the payload, creating the JWT token.

Token Management:

* + 1. Token Distribution: Send the generated JWT token back to the client-side application (typically in the response body or header).

Client-Side Integration:

* + 1. JWT Storage: Securely store the received JWT token on the client-side (e.g., browser local storage with appropriate security measures like HttpOnly flag).

Secured API Endpoints:

* + 1. iddleware Implementation: Develop middleware in your backend framework to validate JWTs before accessing protected API endpoints.
    2. Token Extraction: Extract the JWT token from the request headers (commonly sent in the "Authorization" header with a "Bearer" prefix).
    3. Token Verification: Use the JWT library and the secret key to verify the signature and integrity of the received JWT.
    4. User Authorization: If the token is valid, extract user information from the payload and use it for authorization checks to determine user access to specific resources.
* **Outputs :**

**Token Generation :**

"name": "harsha", "email":"[akhilharsha78@gmail.com](mailto:akhilharsha78@gmail.com)", "password":"harsha123", "picturePath":"test.jpg"

"name": "harsha",

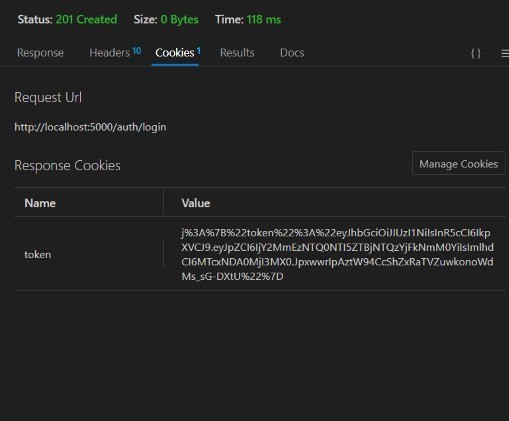
"email": "[akhilharsha78@gmail.com](mailto:akhilharsha78@gmail.com)", "password": "$2b$10$vL8rvj

.XfGXDZXPdy864LukybTMDMfSCPXL6RU6Mn4ryjx3fGFeJK", "picturePath": "test.jpg",

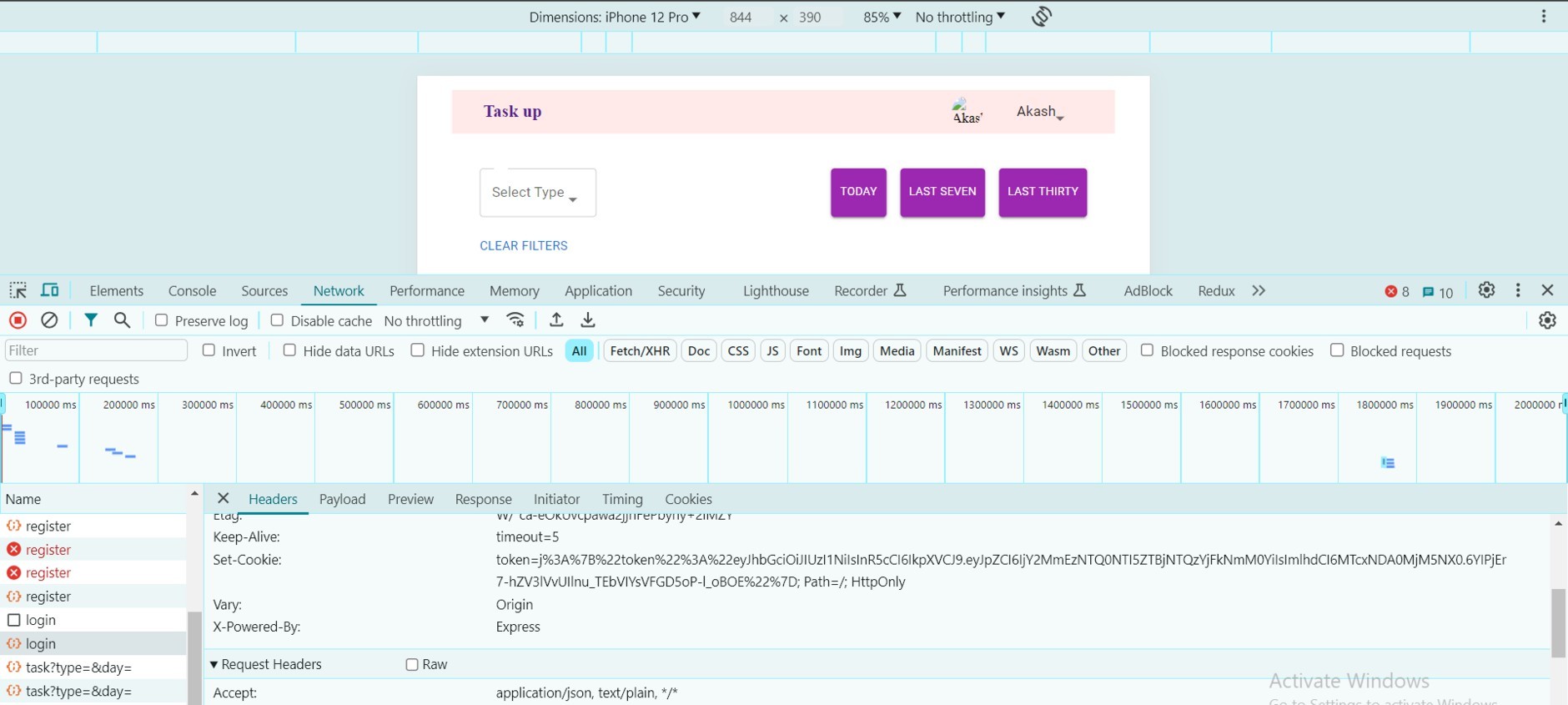
"\_id": "662a3544529e0c543b1d6c4b", "createdAt": "2024-04-25T10:49:40.870Z", "updatedAt": "2024-04-25T10:49:40.870Z",

**Cookie Stored in Database Successfully:**

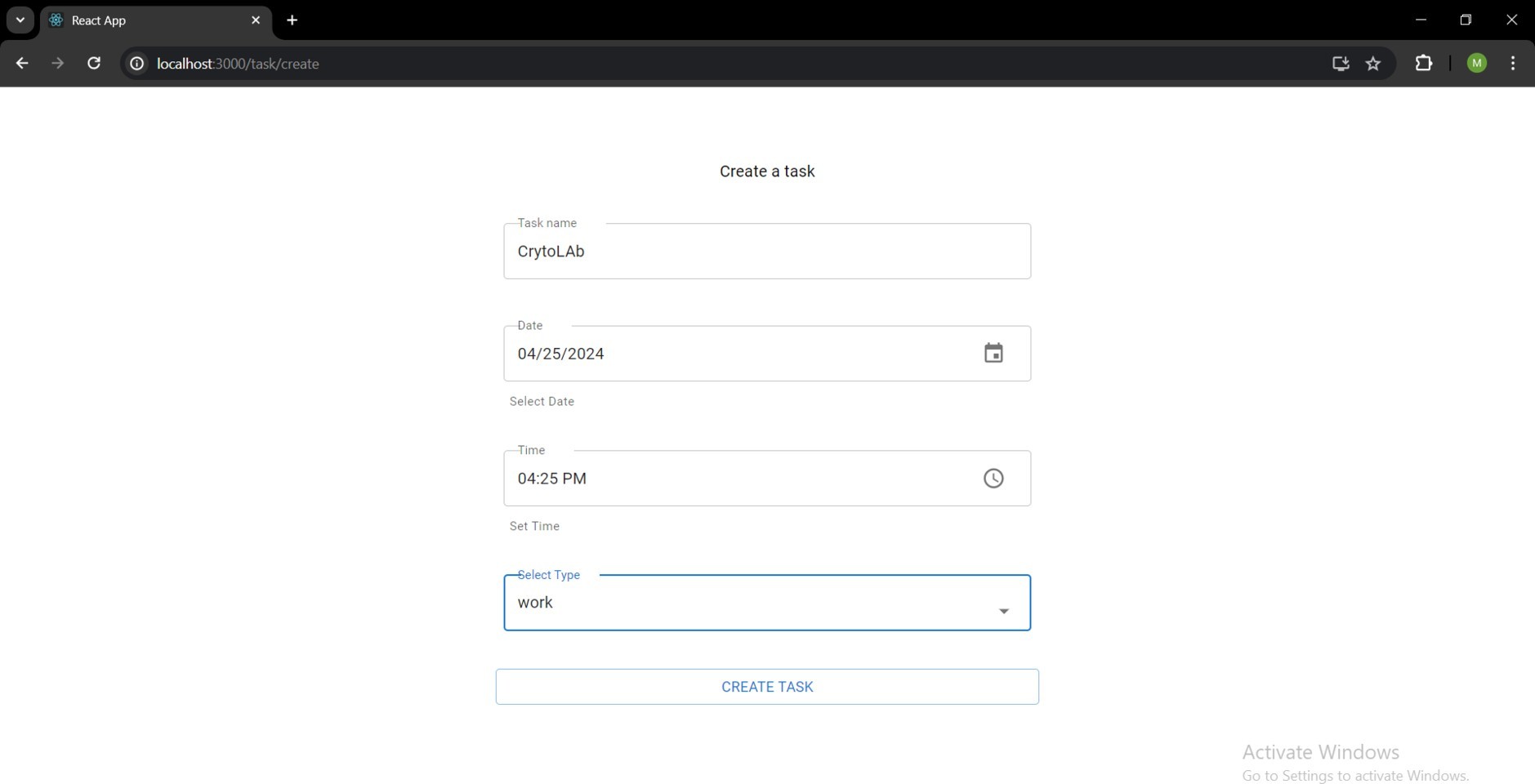
**Login Successful:**



**Cookie Saved the JSON web token:**



**Task Creation:**



**Task stored stored in the data base:**

