**Assignment 1**

**PRN: 21510074**

**Name :Yash Nawale**

**1. Perform encryption, decryption using the following substitution techniques:**

**a. Ceaser cipher**

**Ans:**

The Caesar Cipher is a simple encryption technique where each letter in a message is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, "A" becomes "D," "B" becomes "E," and so on. It's one of the oldest known ciphers and is easy to implement but also easy to break.

**Python Code:**

def caesar\_encrypt(text, shift):

    """

    Encrypt the plain text using Caesar cipher.

    Parameters:

    text (str): The input text to be encrypted.

    shift (int): The number of positions to shift each character.

    Returns:

    str: The encrypted text.

    """

    encrypted\_text = ""

    for char in text:

        if char.isalpha():

            shift\_amount = shift % 26

            if char.islower():

                new\_char = chr((ord(char) - ord('a') + shift\_amount) % 26 + ord('a'))

            else:

                new\_char = chr((ord(char) - ord('A') + shift\_amount) % 26 + ord('A'))

            encrypted\_text += new\_char

        else:

            encrypted\_text += char

    return encrypted\_text

def caesar\_decrypt(text, shift):

    """

    Decrypt the encrypted text using Caesar cipher.

    Parameters:

    text (str): The input text to be decrypted.

    shift (int): The number of positions to shift each character back.

    Returns:

    str: The decrypted text.

    """

    return caesar\_encrypt(text, -shift)

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nCaesar Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ")

            shift = int(input("Enter the shift value: "))

            encrypted\_text = caesar\_encrypt(plain\_text, shift)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            encrypted\_text = input("Enter the encrypted text: ")

            shift = int(input("Enter the shift value: "))

            decrypted\_text = caesar\_decrypt(encrypted\_text, shift)

            print(f"Decrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

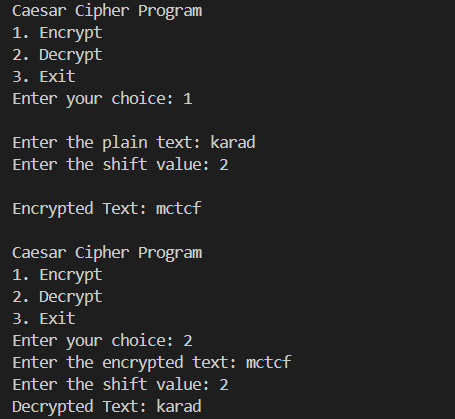
            break

        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output: **

**Advantages:**

* **Simplicity**: Easy to understand and implement.
* **Efficiency**: Fast encryption and decryption.

**Disadvantages:**

* **Weak Security**: Vulnerable to frequency analysis and brute-force attacks (only 25 possible shifts).
* **Predictability**: Does not change much between different texts.

**b. Playfair cipher**

**Ans:**

The Playfair Cipher is a digraph substitution cipher that encrypts pairs of letters. It uses a 5x5 matrix of letters created from a keyword. To encrypt, locate each letter pair in the matrix and swap or substitute based on their positions. It’s more secure than simple substitution ciphers because it encodes pairs of letters rather than individual letters.

**Python code:**

def generate\_playfair\_matrix(key):

    """

    Generate a 5x5 matrix for the Playfair cipher based on the provided key.

    Parameters:

    key (str): The key to generate the matrix.

    Returns:

    list: A 5x5 matrix for the Playfair cipher.

    """

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

    matrix = []

    used = set()

    for char in key:

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

            used.add(char)

            matrix.append(char)

    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":

        if char not in used:

            used.add(char)

            matrix.append(char)

    return [matrix[i:i + 5] for i in range(0, 25, 5)]

def find\_position(matrix, char):

    """

    Find the row and column of a character in the Playfair matrix.

    Parameters:

    matrix (list): The 5x5 matrix for the Playfair cipher.

    char (str): The character to find in the matrix.

    Returns:

    tuple: The row and column of the character in the matrix.

    """

    for row in range(5):

        for col in range(5):

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

                return row, col

    return None

def playfair\_encrypt(text, key):

    """

    Encrypt the plain text using the Playfair cipher.

    Parameters:

    text (str): The input text to be encrypted.

    key (str): The key for the Playfair cipher.

    Returns:

    str: The encrypted text.

    """

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

    if len(text) % 2 != 0:

        text += "X"

    matrix = generate\_playfair\_matrix(key)

    encrypted\_text = ""

    for i in range(0, len(text), 2):

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

        if char1 == char2:

            char2 = 'X'

        row1, col1 = find\_position(matrix, char1)

        row2, col2 = find\_position(matrix, char2)

        if row1 == row2:

            encrypted\_text += matrix[row1][(col1 + 1) % 5]

            encrypted\_text += matrix[row2][(col2 + 1) % 5]

        elif col1 == col2:

            encrypted\_text += matrix[(row1 + 1) % 5][col1]

            encrypted\_text += matrix[(row2 + 1) % 5][col2]

        else:

            encrypted\_text += matrix[row1][col2]

            encrypted\_text += matrix[row2][col1]

    return encrypted\_text

def playfair\_decrypt(text, key):

    """

    Decrypt the encrypted text using the Playfair cipher.

    Parameters:

    text (str): The input text to be decrypted.

    key (str): The key for the Playfair cipher.

    Returns:

    str: The decrypted text.

    """

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

    matrix = generate\_playfair\_matrix(key)

    decrypted\_text = ""

    for i in range(0, len(text), 2):

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

        row1, col1 = find\_position(matrix, char1)

        row2, col2 = find\_position(matrix, char2)

        if row1 == row2:

            decrypted\_text += matrix[row1][(col1 - 1) % 5]

            decrypted\_text += matrix[row2][(col2 - 1) % 5]

        elif col1 == col2:

            decrypted\_text += matrix[(row1 - 1) % 5][col1]

            decrypted\_text += matrix[(row2 - 1) % 5][col2]

        else:

            decrypted\_text += matrix[row1][col2]

            decrypted\_text += matrix[row2][col1]

    return decrypted\_text

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nPlayfair Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ")

            key = input("Enter the key: ")

            encrypted\_text = playfair\_encrypt(plain\_text, key)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            encrypted\_text = input("\nEnter the encrypted text: ")

            key = input("Enter the key: ")

            decrypted\_text = playfair\_decrypt(encrypted\_text, key)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

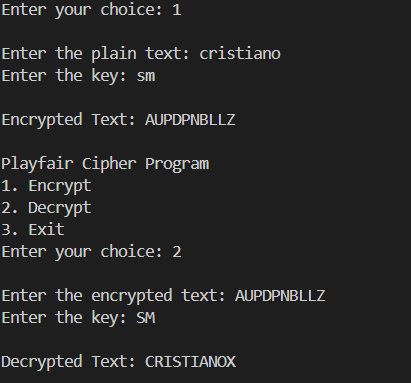
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages:**

* **Improved Security**: More secure than Caesar Cipher as it encrypts digraphs (pairs of letters).
* **Simplicity**: Slightly more complex but still relatively easy to implement.

**Disadvantages:**

* **Key Management**: Requires a good keyword and matrix setup.
* **Vulnerability**: Can still be broken with modern techniques like frequency analysis of digraphs.

**c. Hill Cipher**

**Ans:**

The Hill Cipher is a polygraphic substitution cipher that uses linear algebra. It encrypts blocks of text (usually 2x2 or 3x3 matrices) by multiplying them with a key matrix. The key matrix must be invertible for decryption. This method allows for more complex encryption compared to simple substitution ciphers.

**Python code:**

import numpy as np

def mod\_inverse(matrix, modulus):

    """

    Calculate the modular inverse of a matrix under a given modulus.

    Parameters:

    matrix (numpy.ndarray): The matrix to invert.

    modulus (int): The modulus value.

    Returns:

    numpy.ndarray: The modular inverse of the matrix.

    """

    det = int(np.round(np.linalg.det(matrix)))

    det\_inv = pow(det, -1, modulus)

    matrix\_modulus\_inv = (

        det\_inv \* np.round(det \* np.linalg.inv(matrix)).astype(int) % modulus

    )

    return matrix\_modulus\_inv

def hill\_encrypt(text, key):

    """

    Encrypt the plain text using the Hill cipher.

    Parameters:

    text (str): The input text to be encrypted.

    key (list of int): The key for the Hill cipher as a flat list.

    Returns:

    str: The encrypted text.

    """

    size = int(len(key) \*\* 0.5)

    key\_matrix = np.array(key).reshape(size, size)

    modulus = 26

    text\_vector = np.array([ord(char) - ord('A') for char in text])

    text\_vector = text\_vector.reshape(-1, size).T

    encrypted\_vector = (np.dot(key\_matrix, text\_vector) % modulus).T

    encrypted\_text = ''.join(chr(num + ord('A')) for num in encrypted\_vector.flatten())

    return encrypted\_text

def hill\_decrypt(text, key):

    """

    Decrypt the encrypted text using the Hill cipher.

    Parameters:

    text (str): The input text to be decrypted.

    key (list of int): The key for the Hill cipher as a flat list.

    Returns:

    str: The decrypted text.

    """

    size = int(len(key) \*\* 0.5)

    key\_matrix = np.array(key).reshape(size, size)

    modulus = 26

    key\_matrix\_inv = mod\_inverse(key\_matrix, modulus)

    text\_vector = np.array([ord(char) - ord('A') for char in text])

    text\_vector = text\_vector.reshape(-1, size).T

    decrypted\_vector = (np.dot(key\_matrix\_inv, text\_vector) % modulus).T

    decrypted\_text = ''.join(chr(int(num) + ord('A')) for num in decrypted\_vector.flatten())

    return decrypted\_text

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nHill Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text (length multiple of key matrix size): ").upper().replace(" ", "")

            key = input("Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): ")

            key\_matrix = list(map(int, key.split(',')))

            size = int(len(key\_matrix) \*\* 0.5)

            if len(plain\_text) % size != 0:

                print("Error: The length of the plain text must be a multiple of the key matrix size.")

                continue

            encrypted\_text = hill\_encrypt(plain\_text, key\_matrix)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            encrypted\_text = input("\nEnter the encrypted text: ").upper().replace(" ", "")

            key = input("Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): ")

            key\_matrix = list(map(int, key.split(',')))

            size = int(len(key\_matrix) \*\* 0.5)

            if len(encrypted\_text) % size != 0:

                print("Error: The length of the encrypted text must be a multiple of the key matrix size.")

                continue

            decrypted\_text = hill\_decrypt(encrypted\_text, key\_matrix)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

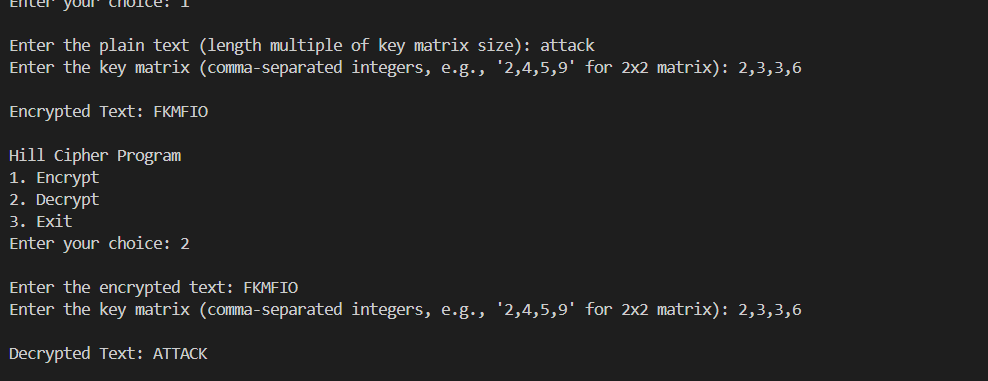
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages:**

* **Polyalphabetic**: Uses linear algebra for encryption, making it stronger than monoalphabetic ciphers.
* **Higher Complexity**: More resistant to frequency analysis due to the use of matrices.

**Disadvantages:**

* **Complexity**: Requires matrix inversion and modular arithmetic, which can be cumbersome.
* **Key Size**: Key matrix must be invertible, and the length of the plaintext must be a multiple of the matrix size.

**d. Vigenere cipher**

**Ans:**

The Vigenère Cipher is a method of encrypting text using a keyword. It works by shifting each letter in the plaintext by an amount determined by the corresponding letter in the keyword. The key repeats itself if it's shorter than the plaintext.

**How It Works:**

1. **Keyword**: Choose a keyword (e.g., "KEY").
2. **Encryption**:
   * Write the keyword repeatedly above the plaintext.
   * Shift each letter in the plaintext by the position of the corresponding letter in the keyword (A=0, B=1, ..., Z=25).
3. **Decryption**:
   * Use the same keyword to reverse the shifts and recover the plaintext.

**Python Code:**

def vigenere\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using the Vigenere cipher.

    Parameters:

    plain\_text (str): The input text to be encrypted.

    key (str): The key for the Vigenere cipher.

    Returns:

    str: The encrypted text.

    """

    plain\_text = plain\_text.upper().replace(" ", "")

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

    key\_length = len(key)

    encrypted\_text = ""

    for i, char in enumerate(plain\_text):

        if char.isalpha():

            shift = ord(key[i % key\_length]) - ord('A')

            encrypted\_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))

            encrypted\_text += encrypted\_char

        else:

            encrypted\_text += char

    return encrypted\_text

def vigenere\_decrypt(cipher\_text, key):

    """

    Decrypt the cipher text using the Vigenere cipher.

    Parameters:

    cipher\_text (str): The input text to be decrypted.

    key (str): The key for the Vigenere cipher.

    Returns:

    str: The decrypted text.

    """

    cipher\_text = cipher\_text.upper().replace(" ", "")

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

    key\_length = len(key)

    decrypted\_text = ""

    for i, char in enumerate(cipher\_text):

        if char.isalpha():

            shift = ord(key[i % key\_length]) - ord('A')

            decrypted\_char = chr((ord(char) - ord('A') - shift + 26) % 26 + ord('A'))

            decrypted\_text += decrypted\_char

        else:

            decrypted\_text += char

    return decrypted\_text

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nVigenere Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ")

            key = input("Enter the key: ")

            encrypted\_text = vigenere\_encrypt(plain\_text, key)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            encrypted\_text = input("\nEnter the encrypted text: ")

            key = input("Enter the key: ")

            decrypted\_text = vigenere\_decrypt(encrypted\_text, key)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

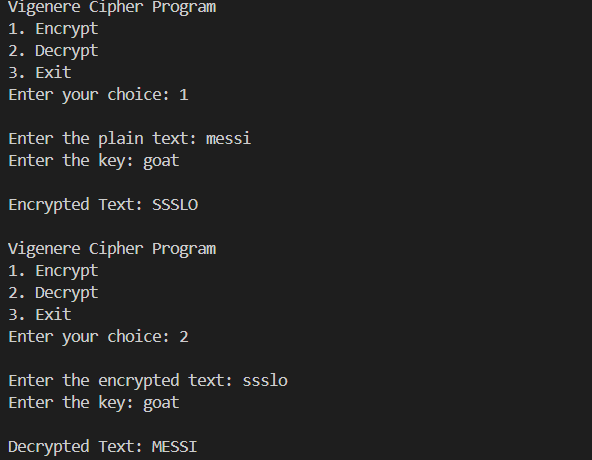
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages:**

* **Polyalphabetic**: Uses a keyword to shift letters, making it more secure than Caesar Cipher.
* **Improved Security**: Harder to crack with frequency analysis if the keyword is long and complex.

**Disadvantages:**

* **Keyword Management**: Security depends on the keyword length and complexity.
* **Vulnerabilities**: Can be broken with techniques like the Kasiski examination or frequency analysis if the keyword is short.

**ASSIGNMENT 2**

**1. Perform encryption and decryption using following transposition techniques**

**a. Rail fence**

**Ans:**

The Rail Fence Cipher is a type of transposition cipher where the plain text is written in a zigzag pattern across multiple "rails" (rows) and then read row by row to create the cipher text. Decryption involves reconstructing the zigzag pattern to retrieve the original message.

**Python code:**

def rail\_fence\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using the Rail Fence cipher.

    Parameters:

    plain\_text (str): The input text to be encrypted.

    key (int): The number of rails (rows) for the Rail Fence cipher.

    Returns:

    str: The encrypted text.

    """

    # Create a list of strings to represent each rail

    rail = ['' for \_ in range(key)]

    row, direction = 0, 1

    # Distribute the characters across the rails in a zigzag pattern

    for char in plain\_text:

        rail[row] += char

        row += direction

        # Reverse direction when we reach the top or bottom rail

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

            direction \*= -1

    # Concatenate all the rails to get the encrypted text

    return ''.join(rail)

def rail\_fence\_decrypt(cipher\_text, key):

    """

    Decrypt the cipher text using the Rail Fence cipher.

    Parameters:

    cipher\_text (str): The input text to be decrypted.

    key (int): The number of rails (rows) for the Rail Fence cipher.

    Returns:

    str: The decrypted text.

    """

    # Determine the length of each rail in the zigzag pattern

    pattern = [0] \* len(cipher\_text)

    row, direction = 0, 1

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

        pattern[i] = row

        row += direction

        # Reverse direction when we reach the top or bottom rail

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

            direction \*= -1

    # Reconstruct the rails from the cipher text

    rail\_lengths = [pattern.count(i) for i in range(key)]

    rail\_chars = ['' for \_ in range(key)]

    pos = 0

    for i in range(key):

        rail\_chars[i] = cipher\_text[pos:pos + rail\_lengths[i]]

        pos += rail\_lengths[i]

    # Reconstruct the original message by following the zigzag pattern

    result = []

    row\_pointers = [0] \* key

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

        result.append(rail\_chars[pattern[i]][row\_pointers[pattern[i]]])

        row\_pointers[pattern[i]] += 1

    return ''.join(result)

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nRail Fence Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ").replace(" ", "")

            key = int(input("Enter the number of rails: "))

            encrypted\_text = rail\_fence\_encrypt(plain\_text, key)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            cipher\_text = input("\nEnter the encrypted text: ").replace(" ", "")

            key = int(input("Enter the number of rails: "))

            decrypted\_text = rail\_fence\_decrypt(cipher\_text, key)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

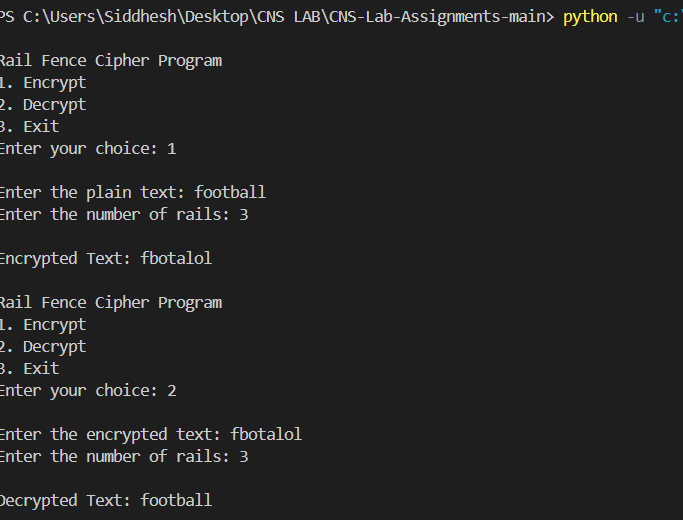
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages:**

* **Simplicity**: Easy to understand and implement.
* **Low Computation**: Requires minimal computational resources for encryption and decryption.

**Disadvantages:**

* **Weak Security**: Very easy to break with simple analysis or known-plaintext attacks.
* **Pattern Recognition**: The regular zigzag pattern makes it susceptible to pattern recognition, which can be exploited to decode the message.

**b. row and Column Transformation**

**Ans:**

Row and column transformation is a type of transposition cipher where the message is written in a grid (matrix) and the order of rows and columns is changed according to a key.

**Row Transposition**: Encrypts text by writing it into rows of a grid, then permuting the columns according to a specific key.

**Column Transposition**: Encrypts text by writing it into columns of a grid, then permuting the rows according to a specific key.

**How It Works**:

1. **Write** the plaintext into a grid according to the number of rows or columns.
2. **Permute** the rows or columns based on the key.
3. **Read** off the text in the new order to get the ciphertext.

**Python code:**

import math

def create\_matrix(text, key\_len):

    """

    Create a matrix from the text with the specified number of columns (key length).

    """

    rows = math.ceil(len(text) / key\_len)

    matrix = [['' for \_ in range(key\_len)] for \_ in range(rows)]

    k = 0

    for i in range(rows):

        for j in range(key\_len):

            if k < len(text):

                matrix[i][j] = text[k]

                k += 1

            else:

                matrix[i][j] = 'X'  # Padding with 'X' if the matrix is not full

    return matrix

def row\_column\_encrypt(plain\_text, row\_key, col\_key):

    """

    Encrypt the plain text using row and column transformation.

    Parameters:

    plain\_text (str): The input text to be encrypted.

    row\_key (list): The key to rearrange rows.

    col\_key (list): The key to rearrange columns.

    Returns:

    str: The encrypted text.

    """

    plain\_text = plain\_text.replace(" ", "")

    key\_len = len(col\_key)

    # Create the matrix from the plain text

    matrix = create\_matrix(plain\_text, key\_len)

    # Apply the row key

    row\_matrix = [matrix[i] for i in row\_key]

    # Apply the column key

    encrypted\_text = ""

    for row in row\_matrix:

        encrypted\_row = [row[j] for j in col\_key]

        encrypted\_text += ''.join(encrypted\_row)

    return encrypted\_text

def row\_column\_decrypt(cipher\_text, row\_key, col\_key):

    """

    Decrypt the cipher text using row and column transformation.

    Parameters:

    cipher\_text (str): The input text to be decrypted.

    row\_key (list): The key to rearrange rows.

    col\_key (list): The key to rearrange columns.

    Returns:

    str: The decrypted text.

    """

    key\_len = len(col\_key)

    rows = len(cipher\_text) // key\_len

    # Create the matrix to store the rearranged cipher text

    matrix = [['' for \_ in range(key\_len)] for \_ in range(rows)]

    k = 0

    # Arrange the cipher text in the matrix based on the column key

    for i in range(len(row\_key)):

        for j in col\_key:

            matrix[row\_key[i]][j] = cipher\_text[k]

            k += 1

    # Read the decrypted text row by row

    decrypted\_text = ""

    for i in range(rows):

        decrypted\_text += ''.join(matrix[i])

    return decrypted\_text

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nRow and Column Transformation Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ")

            row\_key = list(map(int, input("Enter the row key as a sequence of numbers (e.g., 2 0 1): ").split()))

            col\_key = list(map(int, input("Enter the column key as a sequence of numbers (e.g., 1 0 2): ").split()))

            encrypted\_text = row\_column\_encrypt(plain\_text, row\_key, col\_key)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            cipher\_text = input("\nEnter the encrypted text: ")

            row\_key = list(map(int, input("Enter the row key as a sequence of numbers (e.g., 2 0 1): ").split()))

            col\_key = list(map(int, input("Enter the column key as a sequence of numbers (e.g., 1 0 2): ").split()))

            decrypted\_text = row\_column\_decrypt(cipher\_text, row\_key, col\_key)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

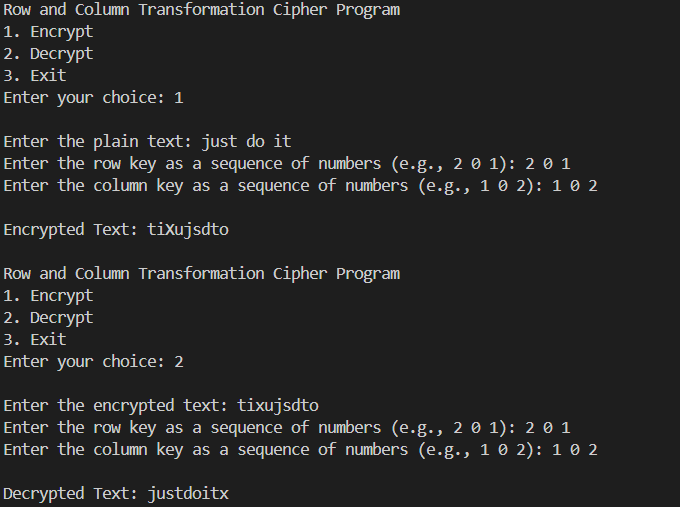
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages**:

* **Increased Security**: More complex than simple transpositions.
* **Flexibility**: Key-based rearrangement can add security.

**Disadvantages**:

* **Complexity**: Can be more complex to implement and manage compared to simple ciphers.
* **Pattern Recognition**: Still susceptible to pattern analysis if not combined with other encryption methods.

**Assignment 3**

1. **Implementation of Euclidean and Extended Euclidean Algorithm**

**Ans:**

The Euclidean and Extended Euclidean algorithms are essential for finding the greatest common divisor (GCD) of two integers. The Extended Euclidean algorithm also finds the coefficients of Bézout's identity, which are useful in solving linear Diophantine equations and in modular arithmetic.

**Euclidean Algorithm**

The Euclidean algorithm finds the GCD of two numbers by repeatedly applying the following rule: gcd(a, b) = gcd(b, a % b) until b becomes zero. The GCD is then the non-zero remainder.

**Extended Euclidean Algorithm**

The Extended Euclidean algorithm not only computes the GCD of two integers a and b, but also finds integers x and y such that ax + by = gcd(a, b).

**Python Code:**

def euclidean\_algorithm(a, b):

    """

    Compute the GCD of a and b using the Euclidean algorithm.

    Parameters:

    a (int): First integer.

    b (int): Second integer.

    Returns:

    int: The GCD of a and b.

    """

    while b != 0:

        a, b = b, a % b

    return a

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

    """

    Compute the GCD of a and b, as well as the coefficients x and y

    such that ax + by = gcd(a, b) using the Extended Euclidean algorithm.

    Parameters:

    a (int): First integer.

    b (int): Second integer.

    Returns:

    tuple: (gcd, x, y) where gcd is the GCD of a and b, and x, y are

    the coefficients of Bézout's identity.

    """

    if b == 0:

        return a, 1, 0

    else:

        gcd, x1, y1 = extended\_euclidean\_algorithm(b, a % b)

        x = y1

        y = x1 - (a // b) \* y1

        return gcd, x, y

def main():

    """

    The main function to run the program.

    """

    while True:

        print("\nEuclidean and Extended Euclidean Algorithm")

        print("1. Compute GCD using Euclidean Algorithm")

        print("2. Compute GCD and coefficients using Extended Euclidean Algorithm")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            a = int(input("\nEnter the first integer (a): "))

            b = int(input("Enter the second integer (b): "))

            gcd = euclidean\_algorithm(a, b)

            print(f"\nGCD of {a} and {b} is: {gcd}")

        elif choice == '2':

            a = int(input("\nEnter the first integer (a): "))

            b = int(input("Enter the second integer (b): "))

            gcd, x, y = extended\_euclidean\_algorithm(a, b)

            print(f"\nGCD of {a} and {b} is: {gcd}")

            print(f"Coefficients x and y are: x = {x}, y = {y}")

            print(f"\nBézout's identity: {a}\*({x}) + {b}\*({y}) = {gcd}")

        elif choice == '3':

            print("Exiting the program.")

            break

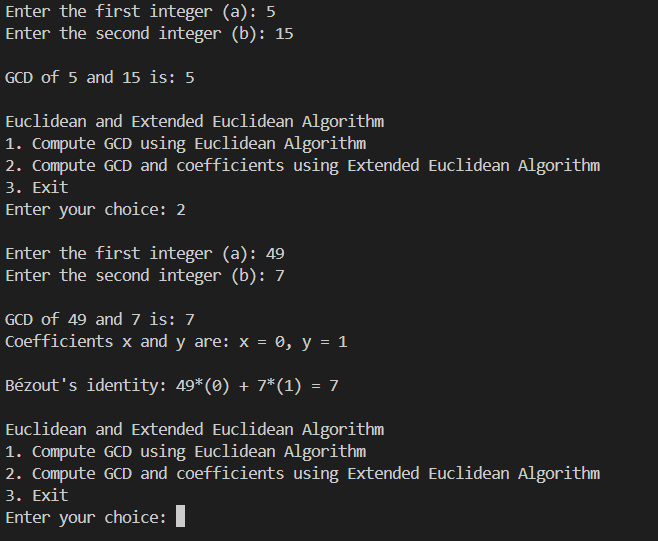
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**



This implementation of the Euclidean and Extended Euclidean algorithms is fundamental in cryptography, number theory, and algorithms related to modular arithmetic.

**Assignment 4**

1. **Implementation of Chinese Remainder Theorem (CRT)**

**Ans:**

The Chinese Remainder Theorem (CRT) is a powerful tool in number theory that provides a solution to a system of simultaneous congruences with pairwise coprime moduli. Given a system of congruences, the CRT allows us to find a unique solution modulo the product of the moduli.

**Problem Description**

Given n congruences: x≡a1 (mod m1), x≡a2 (mod m2) ⋮x ≡ an (mod mn)

Where the moduli m1, m2, …, mn are pairwise coprime, the CRT provides a unique solution modulo M=m1×m2×⋯× mn.

For each congruence x ≡ ai (mod mi), it calculates the partial solution using the formula: x ≡ ai × Mi × inverse(Mi, mi) (mod M) where Mi=M/mi

The final solution is obtained by summing all partial solutions modulo M.

**Python code:**

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

    """

    Compute the GCD of a and b, as well as the coefficients x and y

    such that ax + by = gcd(a, b) using the Extended Euclidean algorithm.

    Parameters:

    a (int): First integer.

    b (int): Second integer.

    Returns:

    tuple: (gcd, x, y) where gcd is the GCD of a and b, and x, y are

    the coefficients of Bézout's identity.

    """

    if b == 0:

        return a, 1, 0

    else:

        gcd, x1, y1 = extended\_euclidean\_algorithm(b, a % b)

        x = y1

        y = x1 - (a // b) \* y1

        return gcd, x, y

def chinese\_remainder\_theorem(a, m):

    """

    Solve the system of congruences using the Chinese Remainder Theorem.

    Parameters:

    a (list): List of remainders.

    m (list): List of moduli (must be pairwise coprime).

    Returns:

    int: The smallest non-negative solution to the system of congruences.

    """

    assert len(a) == len(m), "The number of remainders and moduli must be the same"

    # Calculate the product of all moduli

    M = 1

    for mi in m:

        M \*= mi

    # Initialize the solution

    x = 0

    # Apply the CRT

    for ai, mi in zip(a, m):

        Mi = M // mi  # M\_i = M / m\_i

        gcd, inverse, \_ = extended\_euclidean\_algorithm(Mi, mi)

        if gcd != 1:

            raise ValueError("Moduli are not pairwise coprime")

        x += ai \* inverse \* Mi

    return x % M

def main():

    """

    The main function to run the program.

    """

    while True:

        print("\nChinese Remainder Theorem (CRT)")

        print("1. Solve System of Congruences")

        print("2. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            n = int(input("\nEnter the number of congruences: "))

            a = []

            m = []

            for i in range(n):

                ai = int(input(f"\nEnter remainder a[{i+1}]: "))

                mi = int(input(f"Enter modulus m[{i+1}]: "))

                a.append(ai)

                m.append(mi)

            solution = chinese\_remainder\_theorem(a, m)

            print(f"\nThe solution to the system of congruences is: {solution}")

        elif choice == '2':

            print("Exiting the program.")

            break

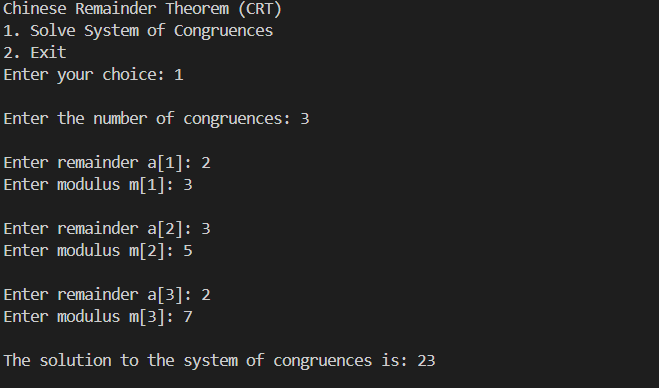
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**



**Assignment 5**

1. **Apply DES algorithm for practical applications**

**Ans:**The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Although DES is now considered insecure for many applications due to its small key size, it is still an important algorithm for understanding the basics of cryptography.

**Practical Application of DES Algorithm**

To apply the DES algorithm in a practical application, we can use the **pycryptodome** library in Python, which provides an implementation of DES. Below is an example that demonstrates how to use DES to encrypt and decrypt a message.

**Python Code:**

from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

def des\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using DES algorithm.

    Parameters:

    plain\_text (str): The text to be encrypted.

    key (bytes): The encryption key (must be 8 bytes long).

    Returns:

    bytes: The encrypted cipher text.

    """

    cipher = DES.new(key, DES.MODE\_ECB)

    padded\_text = pad(plain\_text.encode(), DES.block\_size)

    encrypted\_text = cipher.encrypt(padded\_text)

    return encrypted\_text

def des\_decrypt(cipher\_text, key):

    """

    Decrypt the cipher text using DES algorithm.

    Parameters:

    cipher\_text (bytes): The encrypted text to be decrypted.

    key (bytes): The decryption key (must be 8 bytes long).

    Returns:

    str: The decrypted plain text.

    """

    cipher = DES.new(key, DES.MODE\_ECB)

    decrypted\_text = unpad(cipher.decrypt(cipher\_text), DES.block\_size)

    return decrypted\_text.decode()

def main():

    """

    The main function to run the program.

    """

    print("\nDES Encryption and Decryption")

    # Generate a random 8-byte key for DES

    key = get\_random\_bytes(8)

    print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

    # Input plaintext

    plain\_text = input("Enter the plain text to encrypt: ")

    # Encrypt the plaintext

    encrypted\_text = des\_encrypt(plain\_text, key)

    print(f"\nEncrypted Text (in hexadecimal): {encrypted\_text.hex()}")

    # Decrypt the ciphertext

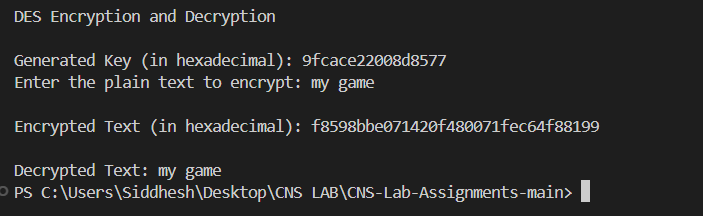
    decrypted\_text = des\_decrypt(encrypted\_text, key)

    print(f"\nDecrypted Text: {decrypted\_text}")

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

    main()

**Output:**

****

**Practical Applications:**

* **File Encryption:** DES can be used to encrypt sensitive files before storing them in insecure locations.
* **Secure Communication:** DES ensures that messages sent over a network are unreadable to unauthorized parties.
* **Password Storage:** Encrypting passwords before storing them in databases (though modern standards recommend stronger algorithms like AES).

While DES itself is outdated and not recommended for secure applications, understanding how it works is crucial for grasping more advanced encryption algorithms like AES.

**Assignment 6**

1. **Apply AES algorithm for practical applications**

Ans:

The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that is both fast and secure. It is the standard encryption algorithm used by governments, financial institutions, and many other organizations. Unlike DES, which is now considered insecure, AES is robust and provides a high level of security.

**Practical Application of AES Algorithm**

We can use the **pycryptodome** library in Python to implement AES encryption and decryption. The AES algorithm can work with key sizes of 128, 192, or 256 bits, and it operates on 128-bit blocks. In this example, we'll use AES with a 256-bit key in Cipher Block Chaining (CBC) mode.

**Python Code:**

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

def aes\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using AES algorithm.

    Parameters:

    plain\_text (str): The text to be encrypted.

    key (bytes): The encryption key (must be 16, 24, or 32 bytes long).

    Returns:

    bytes: The initialization vector (IV) and the encrypted cipher text.

    """

    cipher = AES.new(key, AES.MODE\_CBC)

    iv = cipher.iv  # Initialization vector

    padded\_text = pad(plain\_text.encode(), AES.block\_size)

    encrypted\_text = cipher.encrypt(padded\_text)

    return iv, encrypted\_text

def aes\_decrypt(iv, cipher\_text, key):

    """

    Decrypt the cipher text using AES algorithm.

    Parameters:

    iv (bytes): The initialization vector used during encryption.

    cipher\_text (bytes): The encrypted text to be decrypted.

    key (bytes): The decryption key (must be 16, 24, or 32 bytes long).

    Returns:

    str: The decrypted plain text.

    """

    cipher = AES.new(key, AES.MODE\_CBC, iv)

    decrypted\_text = unpad(cipher.decrypt(cipher\_text), AES.block\_size)

    return decrypted\_text.decode()

def main():

    """

    The main function to run the program.

    """

    print("\nAES Encryption and Decryption")

    # Generate a random 32-byte key for AES (256-bit)

    key = get\_random\_bytes(32)

    print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

    # Input plaintext

    plain\_text = input("\nEnter the plain text to encrypt: ")

    # Encrypt the plaintext

    iv, encrypted\_text = aes\_encrypt(plain\_text, key)

    print(f"\nInitialization Vector (IV) (in hexadecimal): {iv.hex()}")

    print(f"\nEncrypted Text (in hexadecimal): {encrypted\_text.hex()}")

    # Decrypt the ciphertext

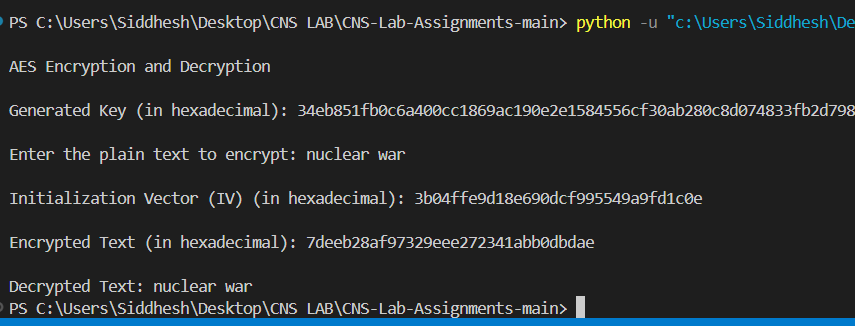
    decrypted\_text = aes\_decrypt(iv, encrypted\_text, key)

    print(f"\nDecrypted Text: {decrypted\_text}")

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

    main()

**Output:**



**Practical Applications of AES:**

* **File Encryption:** Encrypting sensitive files before storing them on disk.
* **Secure Communication:** Ensuring that data sent over the network remains confidential.
* **Data Protection in Applications:** Encrypting user data, such as passwords, to protect them from unauthorized access.

AES is widely adopted due to its strength and efficiency, and it remains the standard for securing digital data across various industries.

**Assignment 7**

**PRN:** 21510003  **Name:** siddhesh mohite

**1. Implementation of RSA Algorithm**

**Ans:**

The RSA algorithm is one of the first public-key cryptosystems and is widely used for secure data transmission. It is an asymmetric cryptographic algorithm, meaning it uses a pair of keys: a public key for encryption and a private key for decryption. It relies on the mathematical properties of prime numbers.

**How RSA Works:**

1. **Key Generation:**
   * Choose two large prime numbers p and q.
   * Compute n = p \* q.
   * Compute the totient φ(n) = (p-1) \* (q-1).
   * Choose an encryption key e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1. The integer e is the public key exponent.
   * Calculate the decryption key d such that d \* e ≡ 1 (mod φ(n)). The integer d is the private key exponent.
2. **Encryption:**
   * The public key is (n, e).
   * Given a plaintext message M, the ciphertext C is computed as:

C = M^ e mod n.

1. **Decryption:**
   * The private key is (n, d).
   * Given a ciphertext C, the plaintext M is recovered as:

M = C^d mod n

To implement the RSA algorithm **using large prime numbers with 2048 bits** and converting plaintext into numbers, we'll use the **Crypto library in Python**, which provides the **necessary tools to handle such large prime numbers and perform RSA encryption and decryption.**

**The large primes and the strong key sizes make RSA secure against most attacks when implemented correctly.**

**Python Code:**

import random

from sympy import isprime, mod\_inverse

def generate\_prime\_candidate(length):

    """Generate an odd integer randomly."""

    p = random.getrandbits(length)

    # Ensure p is odd

    p |= (1 << length - 1) | 1

    return p

def generate\_prime\_number(length):

    """Generate a prime number."""

    p = 4

    while not isprime(p):

        p = generate\_prime\_candidate(length)

    return p

def generate\_keypair(keysize):

    """Generate RSA public and private keys."""

    # Generate two large primes p and q

    p = generate\_prime\_number(keysize)

    q = generate\_prime\_number(keysize)

    print("\np: ", p)

    print("\nq: ", q)

    # Compute n = p \* q

    n = p \* q

    # Compute Euler's Totient φ(n) = (p-1)\*(q-1)

    phi = (p - 1) \* (q - 1)

    # Choose an integer e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1

    e = random.randrange(2, phi)

    g = gcd(e, phi)

    while g != 1:

        e = random.randrange(2, phi)

        g = gcd(e, phi)

    # Compute d, the modular inverse of e

    d = mod\_inverse(e, phi)

    # Public key (e, n) and Private key (d, n)

    return ((e, n), (d, n))

def gcd(a, b):

    """Compute the greatest common divisor using Euclid's algorithm."""

    while b != 0:

        a, b = b, a % b

    return a

def encrypt(public\_key, plaintext):

    """Encrypt plaintext using the public key."""

    e, n = public\_key

    cipher = [pow(ord(char), e, n) for char in plaintext]

    return cipher

def decrypt(private\_key, ciphertext):

    """Decrypt ciphertext using the private key."""

    d, n = private\_key

    plain = [chr(pow(char, d, n)) for char in ciphertext]

    return ''.join(plain)

def main():

    """Run RSA algorithm."""

    print("RSA Encryption/Decryption")

    keysize = 2048  # Keysize in bits

    # Generate public and private keys

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

    print(f"\nPublic key: {public\_key}")

    print(f"Private key: {private\_key}")

    # Input plaintext

    plaintext = input("\nEnter a message to encrypt: ")

    # Encrypt the message

    encrypted\_msg = encrypt(public\_key, plaintext)

    print(f"\nEncrypted message: {encrypted\_msg}")

    # Decrypt the message

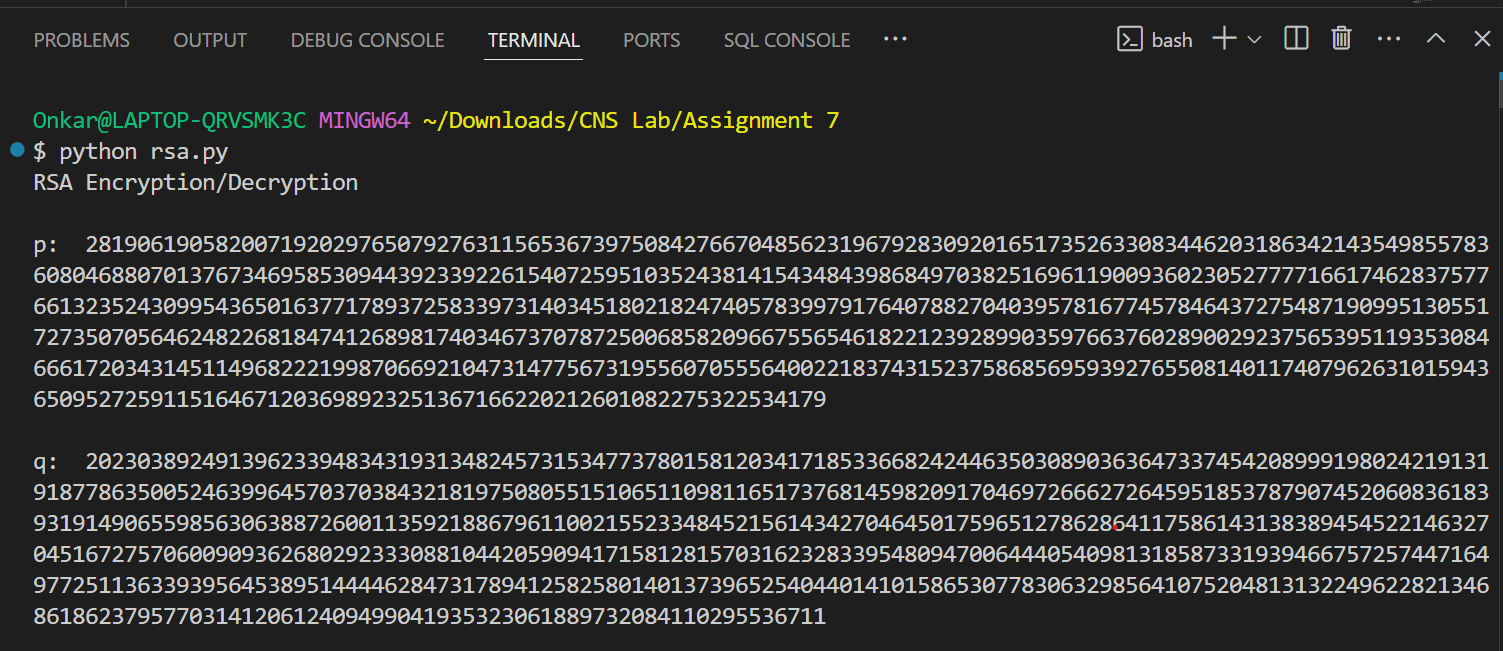
    decrypted\_msg = decrypt(private\_key, encrypted\_msg)

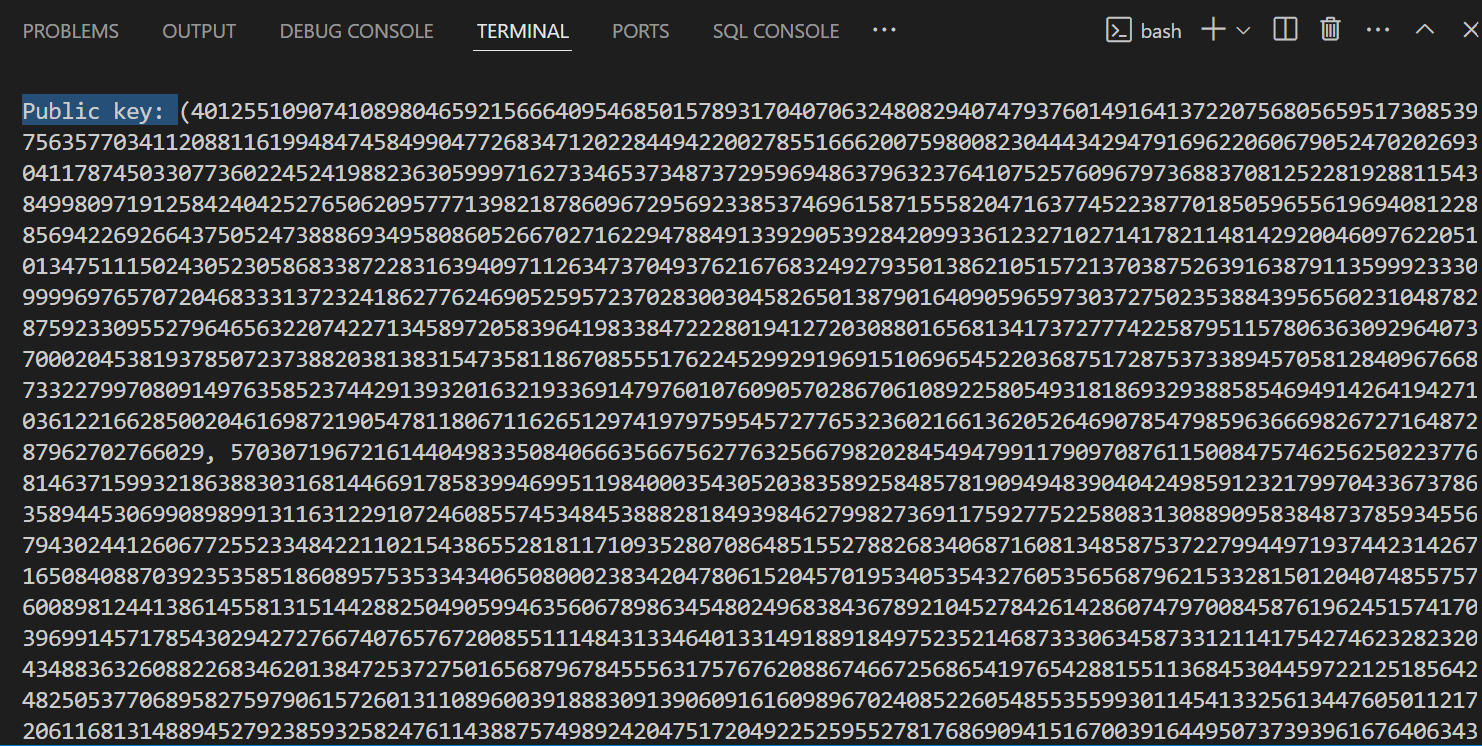
    print(f"\nDecrypted message: {decrypted\_msg}")

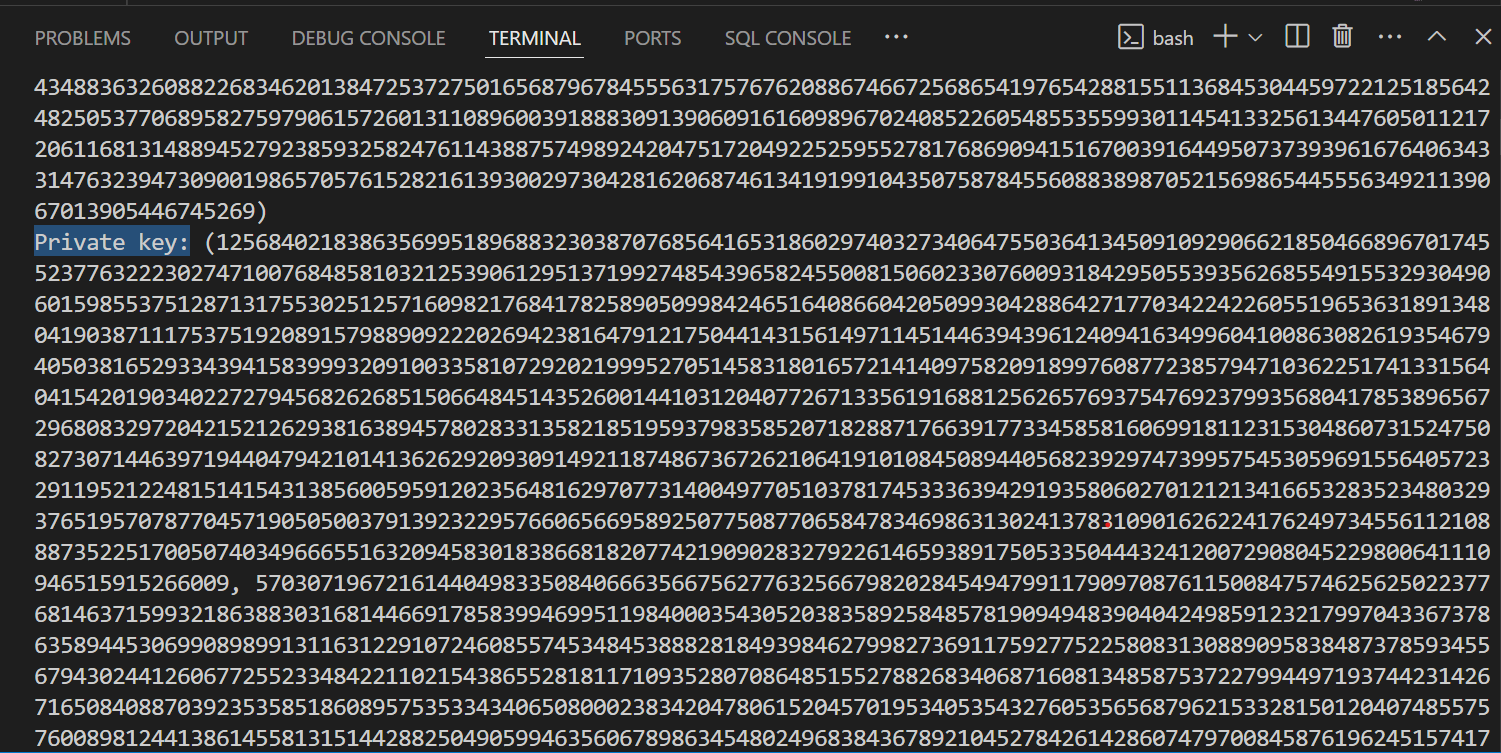
if \_\_name\_\_ == "\_\_main\_\_":

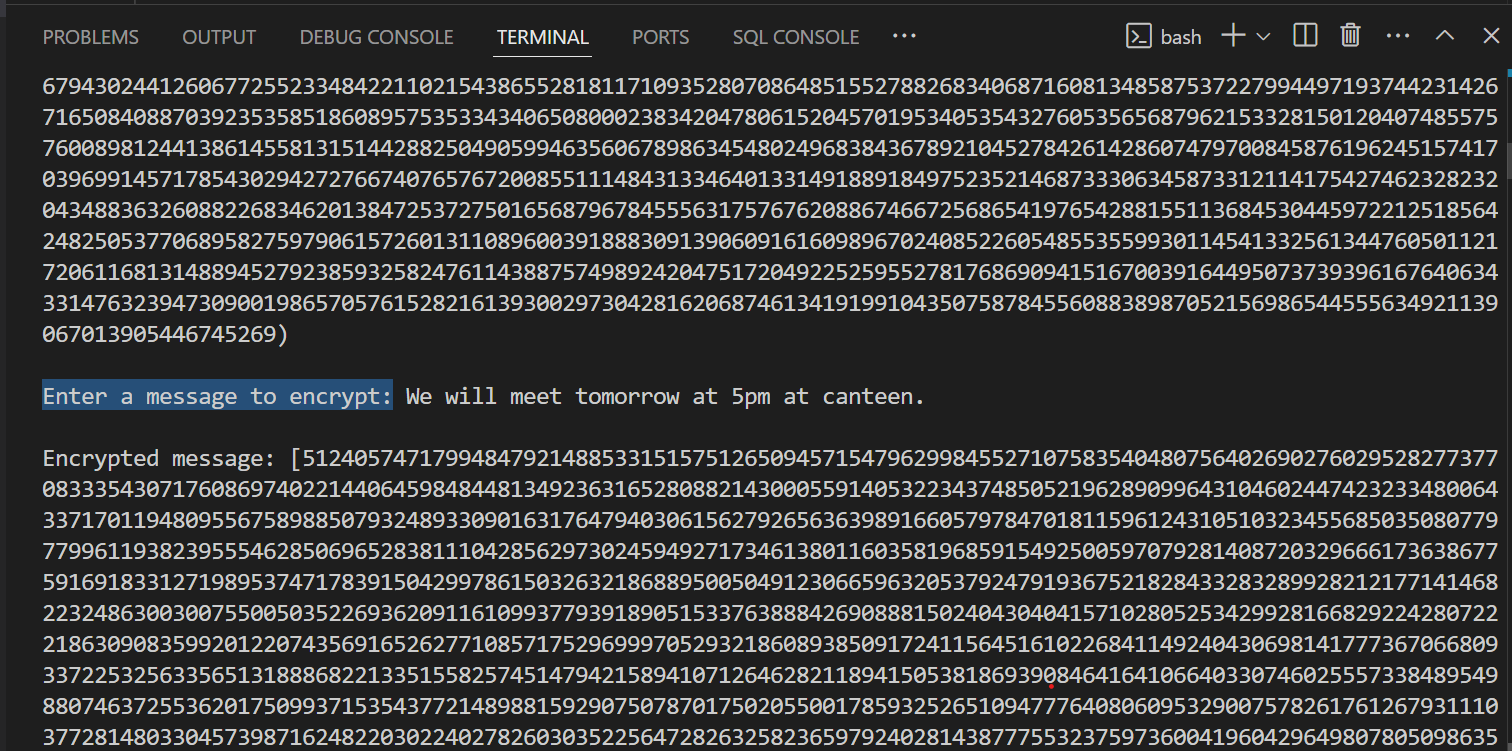
    main()

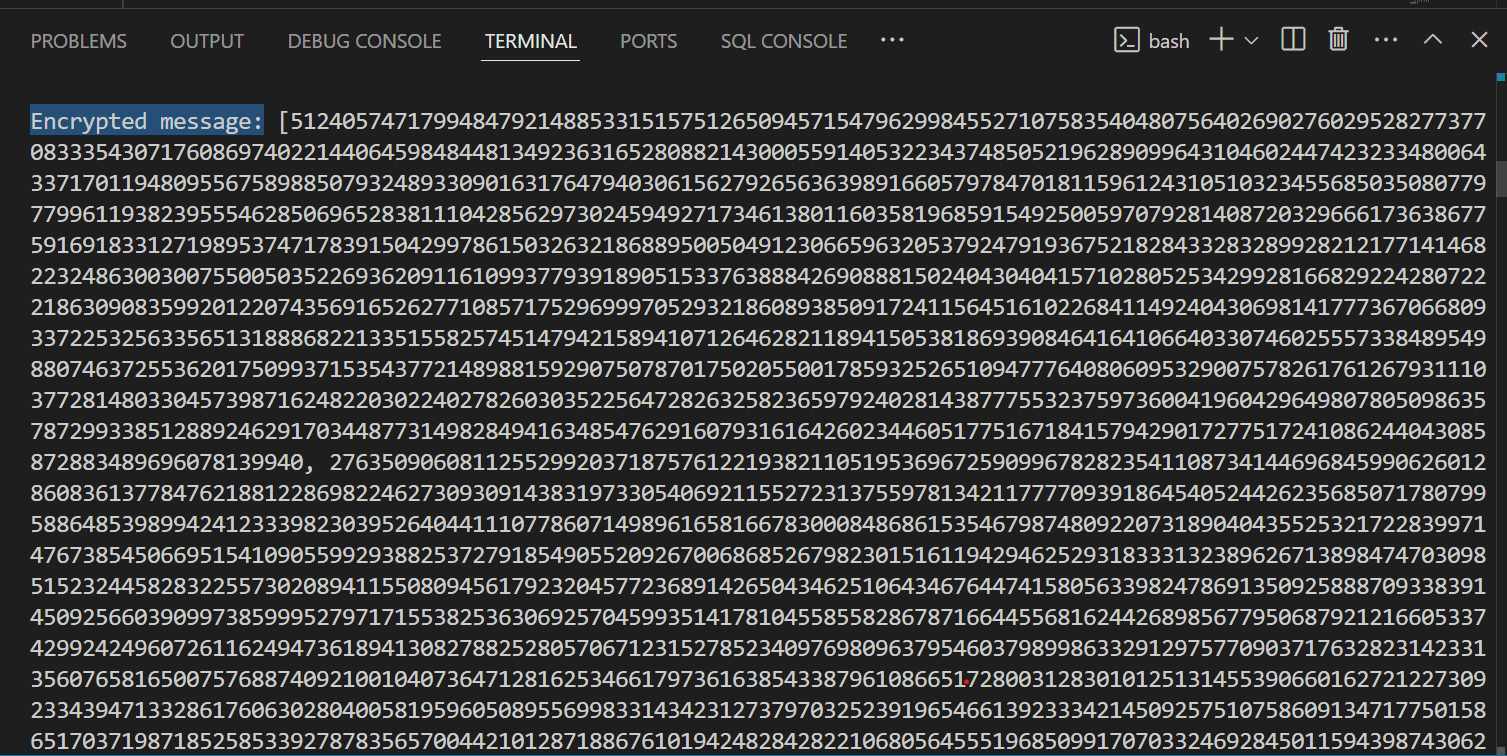
**Output:**

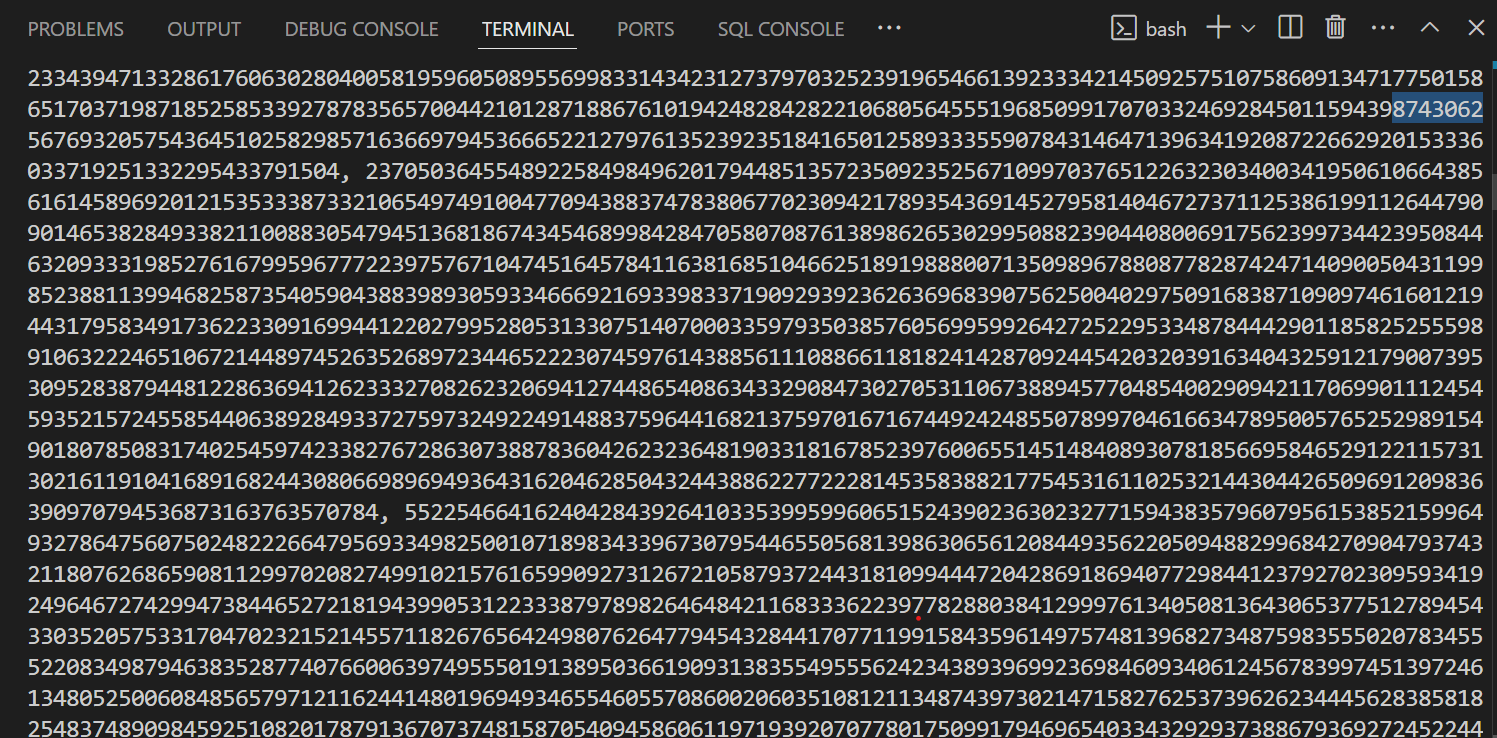


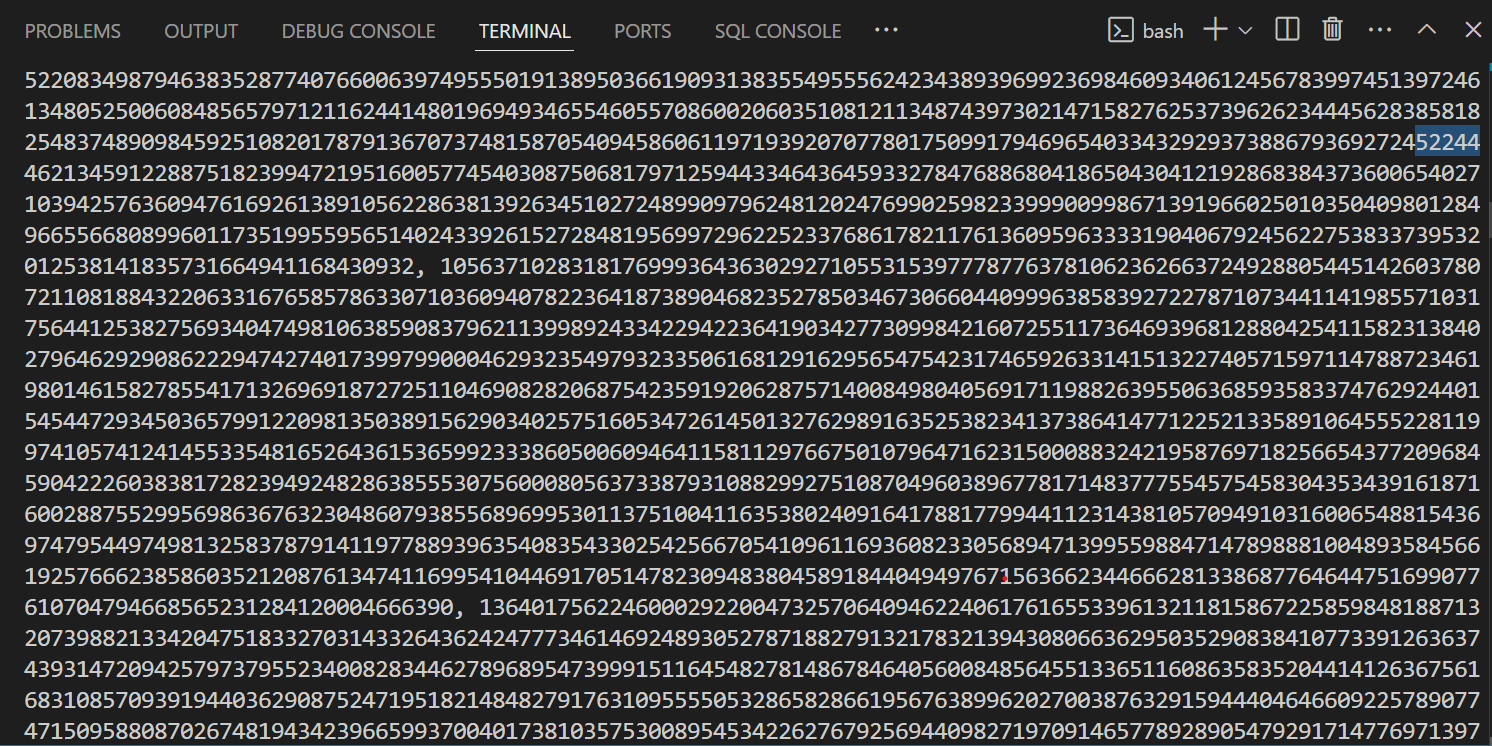


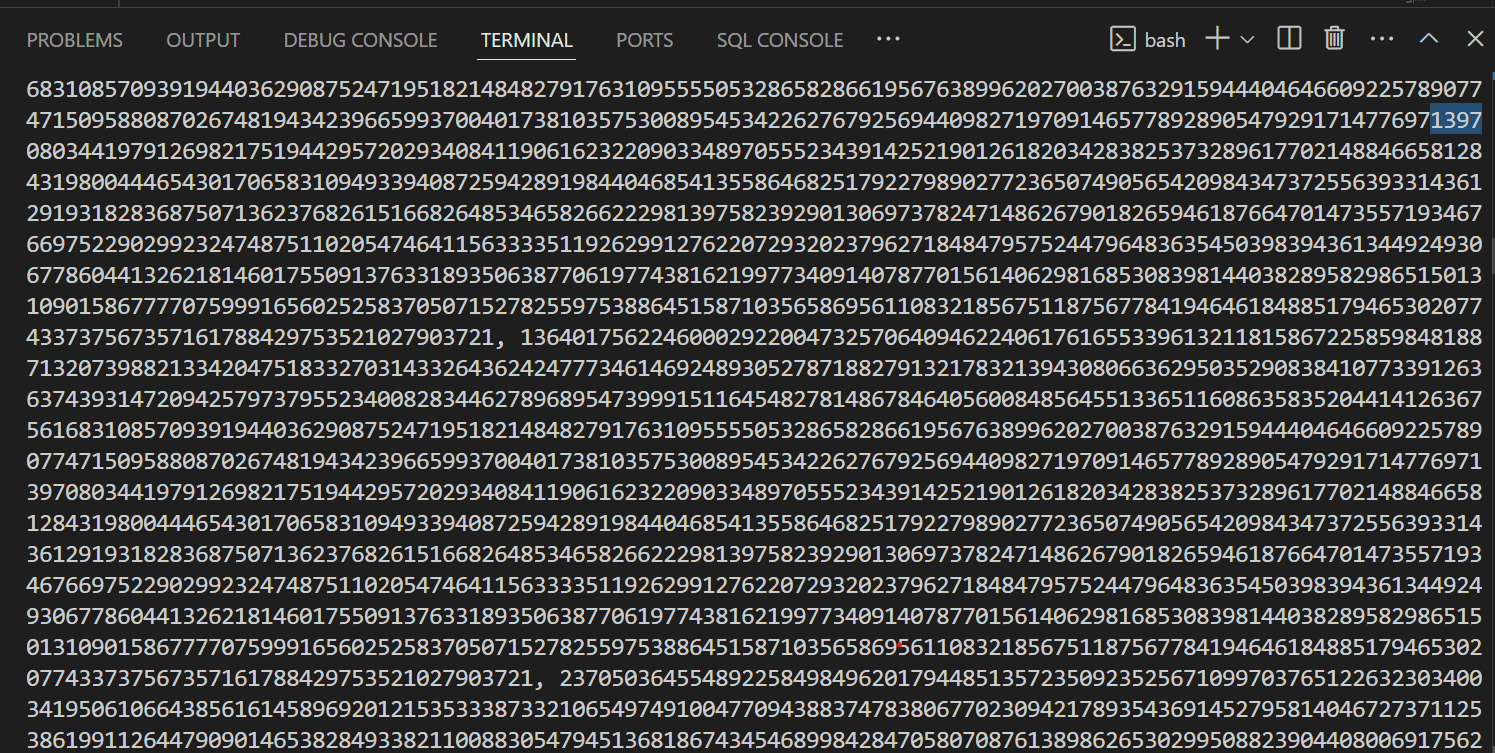


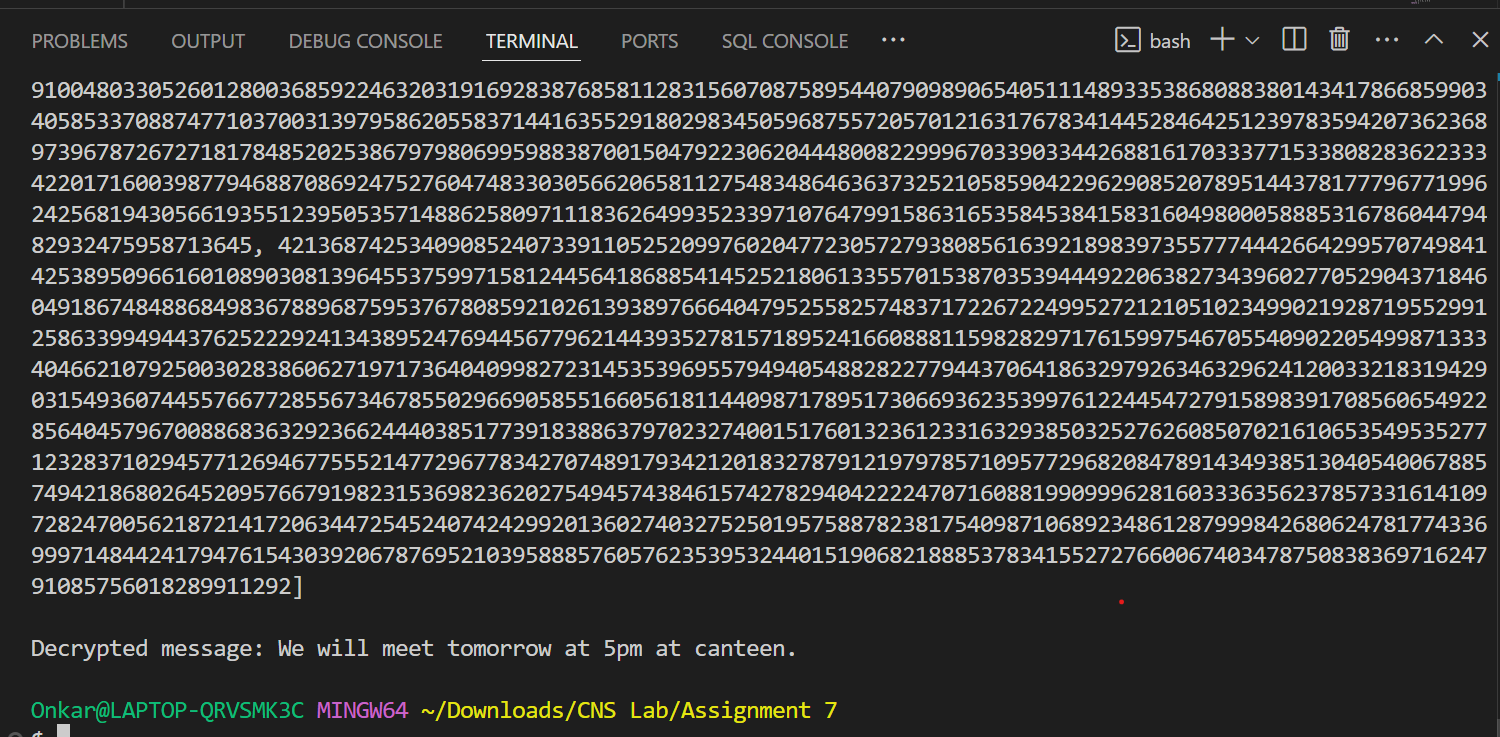












**Practical Applications of RSA**

* **Secure Communication:** Encrypting emails and messages.
* **Digital Signatures:** Verifying the authenticity of a message or document.
* **Key Exchange:** Securely exchanging keys for symmetric encryption algorithms.

RSA is widely used in various security protocols, including SSL/TLS for secure internet communications.

RSA ensures security through the difficulty of factoring large numbers. It is commonly used for securing sensitive data, digital signatures, and in SSL/TLS protocols.