

## Assignment 1

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

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:**

```
Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 1
$ python caesar_cipher.py

Caesar Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: HELLO
Enter the shift value: 3

Encrypted Text: KHOOR

Caesar Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2
Enter the encrypted text: KHOOR
Enter the shift value: 3
Decrypted Text: HELLO
```

#### 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 "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
        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:

```
Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 1  
○ $ python playfair_cipher.py  
  
Playfair Cipher Program  
1. Encrypt  
2. Decrypt  
3. Exit  
Enter your choice: 1  
  
Enter the plain text: MEET ME AT NOON  
Enter the key: MONARCHY  
  
Encrypted Text: CLKLCLRSANNA  
  
Playfair Cipher Program  
1. Encrypt  
2. Decrypt  
3. Exit  
Enter your choice: 2  
  
Enter the encrypted text: CLKLCLRSANNA  
Enter the key: MONARCHY  
  
Decrypted Text: MEETMEATNOON
```

### 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:**

```

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 1
$ python hill_cipher.py

Hill Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text (length multiple of key matrix size): WILL MEET TOMORROW
Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): 6, 1, 3, 2

Encrypted Text: KEZDYSRYHIMPHCI

Hill Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: KEZDYSRYHIMPHCI
Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): 6, 1, 3, 2

Decrypted Text: WILLMEETTOMORROW

```

### 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_text += chr(ord(char) + shift)
```

```

        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:

```

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 1
$ python vigenere_cipher.py

Vigenere Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: MEET ME AT MORNING
Enter the key: CODE

Encrypted Text: OSHXOSDXOCURKBJ

Vigenere Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: OSHXOSDXOCURKBJ
Enter the key: CODE

Decrypted Text: MEETMEATMORNING

```

### 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

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

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: ")
            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:**

```

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 2
○ $ python rail_fence.py

Rail Fence Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: HELLO FROM THE OTHER SIDE
Enter the number of rails: 3

Encrypted Text: HOMOREELFOTETESDLRHHI

Rail Fence Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: HOMOREELFOTETESDLRHHI
Enter the number of rails: 3

Decrypted Text: HELLOFROMTHEOTHERSIDE

```

### 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:

```

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 2
$ python row_column_transformation.py

Row and Column Transformation Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: HELLO WORLD
Enter the row key as a sequence of numbers (e.g., 2 0 1): 2 0 1
Enter the column key as a sequence of numbers (e.g., 1 0 2): 1 0 2

Encrypted Text: ROLEHLOLW

Row and Column Transformation Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: ROLEHLOLW
Enter the row key as a sequence of numbers (e.g., 2 0 1): 2 0 1
Enter the column key as a sequence of numbers (e.g., 1 0 2): 1 0 2

Decrypted Text: HELLOWORL

```

### 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

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

### 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:  $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$  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  $ax + by = \text{gcd}(a, b)$ .

**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  $ax + by = \text{gcd}(a, b)$  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:

```
Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 3
$ python euclidean.py

Euclidean and Extended Euclidean Algorithm
1. Compute GCD using Euclidean Algorithm
2. Compute GCD and coefficients using Extended Euclidean Algorithm
3. Exit
Enter your choice: 1

Enter the first integer (a): 48
Enter the second integer (b): 18

GCD of 48 and 18 is: 6

Euclidean and Extended Euclidean Algorithm
1. Compute GCD using Euclidean Algorithm
2. Compute GCD and coefficients using Extended Euclidean Algorithm
3. Exit
Enter your choice: 2

Enter the first integer (a): 55
Enter the second integer (b): 15

GCD of 55 and 15 is: 5
Coefficients x and y are: x = -1, y = 4

Bézout's identity: 55*(-1) + 15*(4) = 5
```

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



## Assignment 4

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

### 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:  $x \equiv a_1 \pmod{m_1}$ ,  $x \equiv a_2 \pmod{m_2}$  :  $x \equiv a_n \pmod{m_n}$

Where the moduli  $m_1, m_2, \dots, m_n$  are pairwise coprime, the CRT provides a unique solution modulo  $M = m_1 \times m_2 \times \dots \times m_n$ .

For each congruence  $x \equiv a_i \pmod{m_i}$ , it calculates the partial solution using the formula:  $x \equiv a_i \times M_i \times \text{inverse}(M_i, m_i) \pmod{M}$  where  $M_i = M/m_i$

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  $ax + by = \text{gcd}(a, b)$  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:

```
Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 4
$ python chinese_remainder_theorem.py

Chinese Remainder Theorem (CRT)
1. Solve System of Congruences
2. Exit
Enter your choice: 1

Enter the number of congruences: 3

Enter remainder a[1]: 2
Enter modulus m[1]: 3

Enter remainder a[2]: 3
Enter modulus m[2]: 5

Enter remainder a[3]: 2
Enter modulus m[3]: 7

The solution to the system of congruences is: 23
```

## Assignment 5

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

### 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: ")

```

```

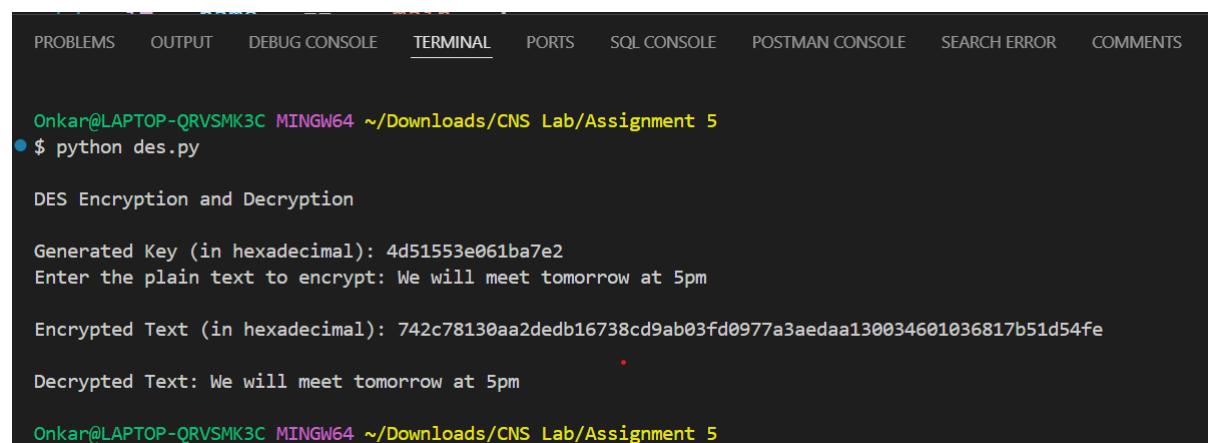
# Encrypt the plaintext
encrypted_text = des_encrypt(plain_text, key)
print(f"\nEncrypted Text (in hexadecimal):
{encrypted_text.hex()}")

# Decrypt the ciphertext
decrypted_text = des_decrypt(encrypted_text, key)
print(f"\nDecrypted Text: {decrypted_text}")

if __name__ == "__main__":
    main()

```

### Output:



The screenshot shows a terminal window with the following output:

```

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 5
$ python des.py

DES Encryption and Decryption

Generated Key (in hexadecimal): 4d51553e061ba7e2
Enter the plain text to encrypt: We will meet tomorrow at 5pm

Encrypted Text (in hexadecimal): 742c78130aa2dedb16738cd9ab03fd0977a3aedaa130034601036817b51d54fe

Decrypted Text: We will meet tomorrow at 5pm

```

### 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

**PRN:** 21510017

**Name:** Onkar Anand Yemul

---

### 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:
")

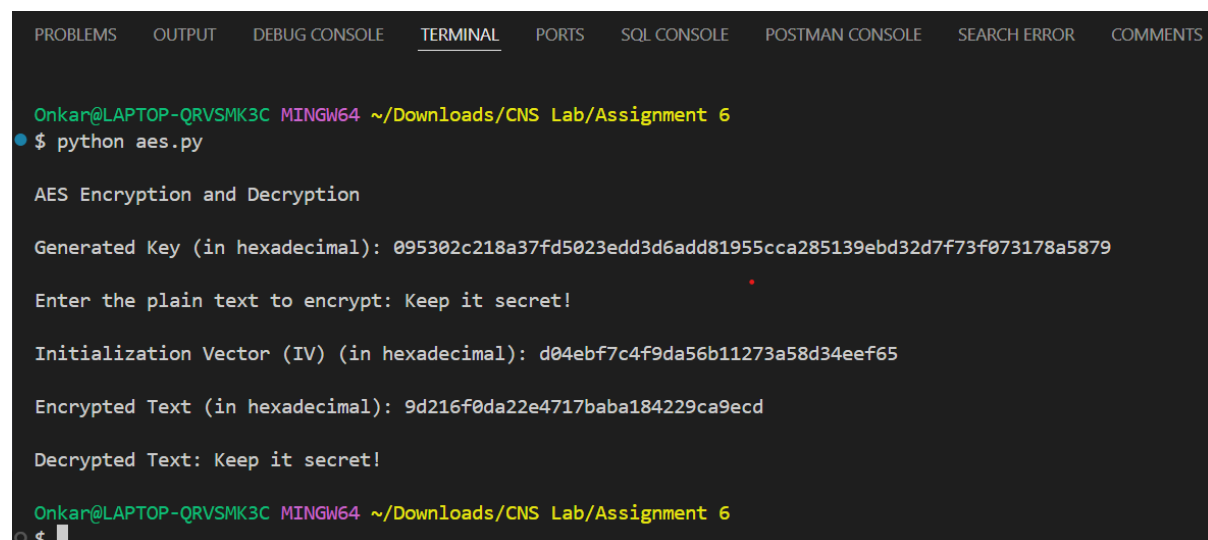
# Encrypt the plaintext
iv, encrypted_text = aes_encrypt(plain_text, key)
print(f"\nInitialization Vector (IV) (in hexadecimal):
{iv.hex()}")
print(f"\nEncrypted Text (in hexadecimal):
{encrypted_text.hex()}")

# Decrypt the ciphertext
decrypted_text = aes_decrypt(iv, encrypted_text, key)
print(f"\nDecrypted Text: {decrypted_text}")

if __name__ == "__main__":
    main()

```

## Output:



The screenshot shows a terminal window with the following content:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL CONSOLE  POSTMAN CONSOLE  SEARCH ERROR  COMMENTS

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 6
$ python aes.py

AES Encryption and Decryption

Generated Key (in hexadecimal): 095302c218a37fd5023edd3d6add81955cca285139ebd32d7f73f073178a5879

Enter the plain text to encrypt: Keep it secret!

Initialization Vector (IV) (in hexadecimal): d04ebf7c4f9da56b11273a58d34eef65

Encrypted Text (in hexadecimal): 9d216f0da22e4717baba184229ca9ecd

Decrypted Text: Keep it secret!

Onkar@LAPTOP-QRVSMK3C MINGW64 ~/Downloads/CNS Lab/Assignment 6
$

```

### **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.