Program Substitution and Transportation Ciphers
Cryptography– Lab 1

Name: Ojas Patil
Reg No: 21BAI1106

Types of Cipher

Substitution-

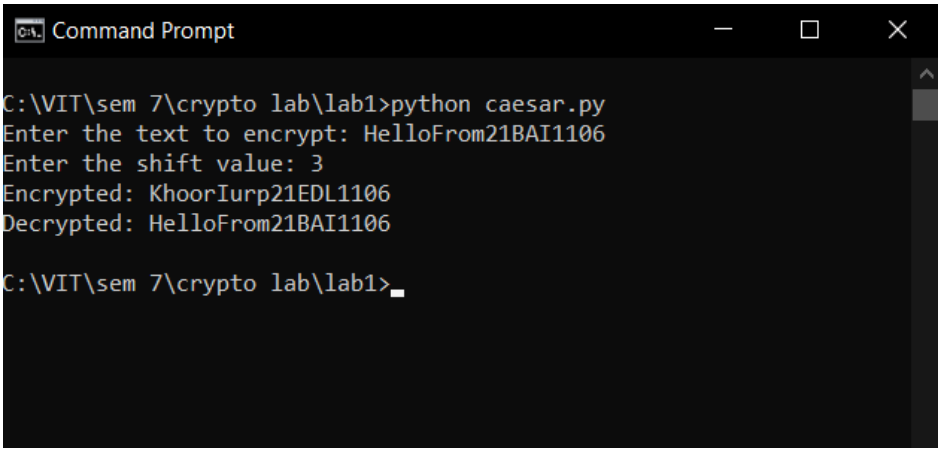
1. Caesar Cipher
2. Vigenere Cipher
3. Play fair Cipher
4. Hill Cipher

Transposition :

1. Row column Cipher
2. Rail fence Cipher

Caesar Cypher

Output Snapshot



Code Snapshot

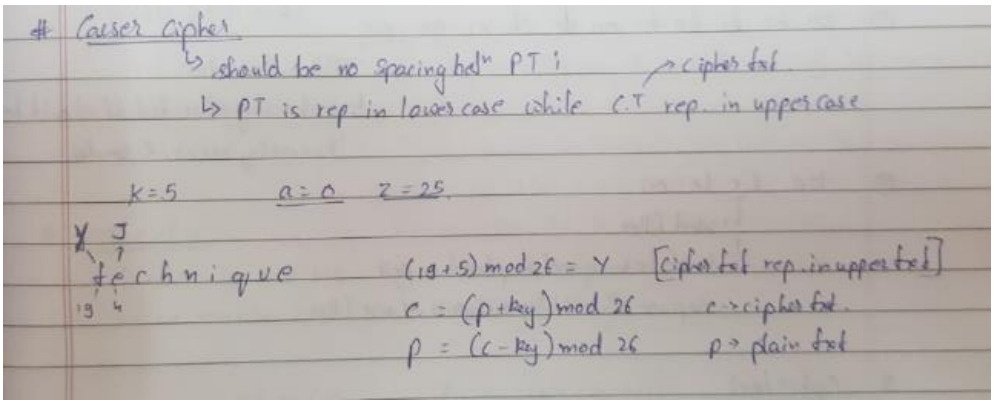
caesar.py

```python
# encryption algorithm
def encrypt(txt, s):
    result = ""
    for char in txt:
        if char.isalpha():
            shift = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - shift + s) % 26 + shift)
        else:
            result += char
    return result

# decryption algorithm
def decrypt(txt, s):
    return encrypt(txt, -s)

text = input("Enter the text to encrypt: ")
shift = int(input("Enter the shift value: "))

enc_text = encrypt(text, shift)
dec_text = decrypt(enc_text, shift)

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Command Prompt

```
C:\VIT\sem 7\crypto lab\lab1>python caesar.py
Enter the text to encrypt: HelloFrom21BAI1106
Enter the shift value: 3
Encrypted: KhoorIurp21EDL1106
Decrypted: HelloFrom21BAI1106

C:\VIT\sem 7\crypto lab\lab1>
```

Handwritten sum



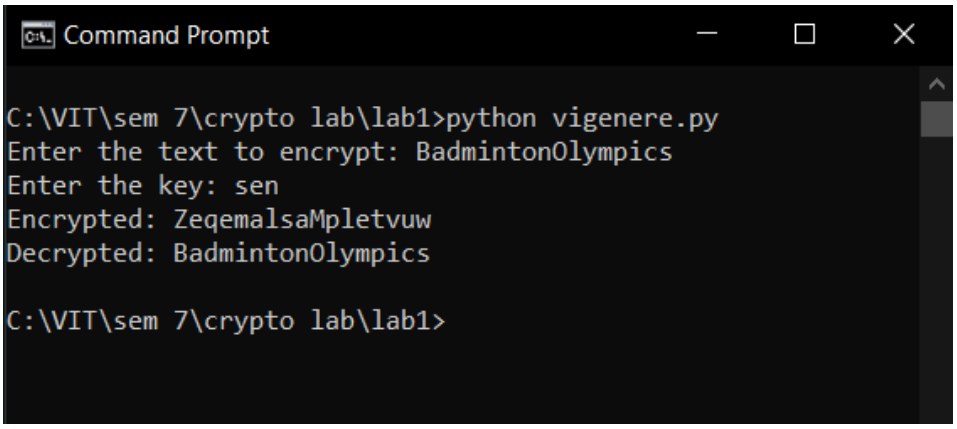
Source Code

```python
# encryption algorithm
def encrypt(txt, s):
    result = ""
    for char in txt:
```

```
        if char.isalpha():
            shift = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - shift + s) % 26 + shift)
        else:
            result += char
    return result

# decryption algorithm
def decrypt(txt, s):
    return encrypt(txt, -s)

text = input("Enter the text to encrypt: ")
shift = int(input("Enter the shift value: "))

enc_text = encrypt(text, shift)
dec_text = decrypt(enc_text, shift)

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Vigenere Cypher

Output Snapshot



Full Code Snapshot

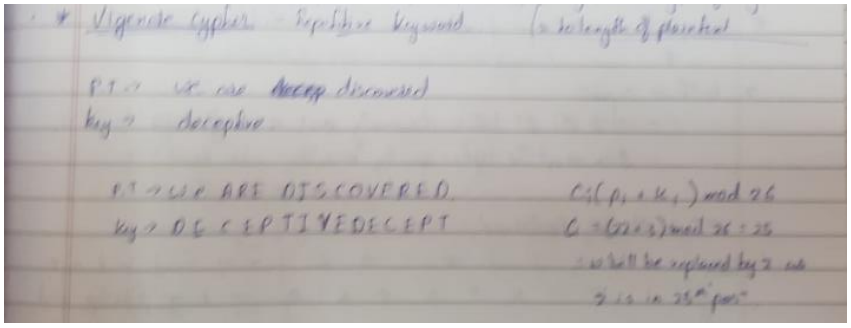
vigenere.py

```python
# encrypt the string
def encrypt(txt, key):
    result = ""
    key_len = len(key)
    key_as_int = [ord(i) for i in key]
    txt_as_int = [ord(i) for i in txt]

    for i in range(len(txt_as_int)):
        if txt[i].isalpha():
            shift = ord('A') if txt[i].isupper() else ord('a')
            value = (txt_as_int[i] - shift + key_as_int[i % key_len] - shift) % 26
            result += chr(value + shift)
        else:
            result += txt[i]
    return result

# decrypt the string
def decrypt(txt, key):
    result = ""
    key_len = len(key)
    key_as_int = [ord(i) for i in key]
    txt_as_int = [ord(i) for i in txt]

    for i in range(len(txt_as_int)):
        if txt[i].isalpha():
            shift = ord('A') if txt[i].isupper() else ord('a')
            value = (txt_as_int[i] - shift - (key_as_int[i % key_len] - shift)) % 26
            result += chr(value + shift)
        else:
            result += txt[i]
    return result
```

Command Prompt

```
C:\VIT\sem 7\crypto lab\lab1>python vigenere.py
Enter the text to encrypt: BadmintonOlympics
Enter the key: sen
Encrypted: ZeqemalsaMpletvuw
Decrypted: BadmintonOlympics

C:\VIT\sem 7\crypto lab\lab1>
```

Handwritten Sum



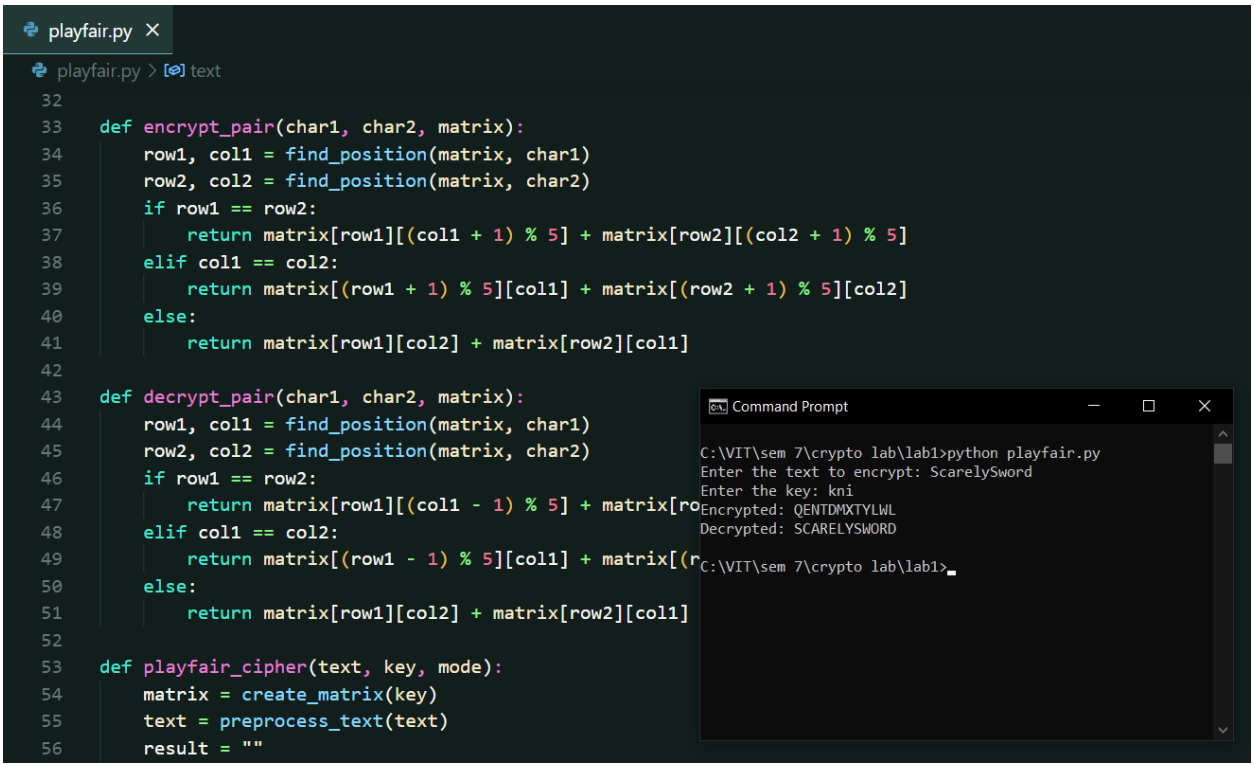
Source Code

```
# encrypt the string
def encrypt(txt, key):
    result = ""
    key_len = len(key)
    key_as_int = [ord(i) for i in key]
    txt_as_int = [ord(i) for i in txt]

    for i in range(len(txt_as_int)):
        if txt[i].isalpha():
            shift = ord('A') if txt[i].isupper() else ord('a')
            value = (txt_as_int[i] - shift + key_as_int[i % key_len] - shift) %
26
            result += chr(value + shift)
        else:
            result += txt[i]
    return result

# decrypt the string
def decrypt(txt, key):
    result = ""
    key_len = len(key)
    key_as_int = [ord(i) for i in key]
    txt_as_int = [ord(i) for i in txt]

    for i in range(len(txt_as_int)):
        if txt[i].isalpha():
            shift = ord('A') if txt[i].isupper() else ord('a')
            value = (txt_as_int[i] - shift - (key_as_int[i % key_len] - shift)) %
26
            result += chr(value + shift)
        else:
            result += txt[i]
    return result

text = input("Enter the text to encrypt: ")
key = input("Enter the key: ")

enc_text = encrypt(text, key)
dec_text = decrypt(enc_text, key)

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

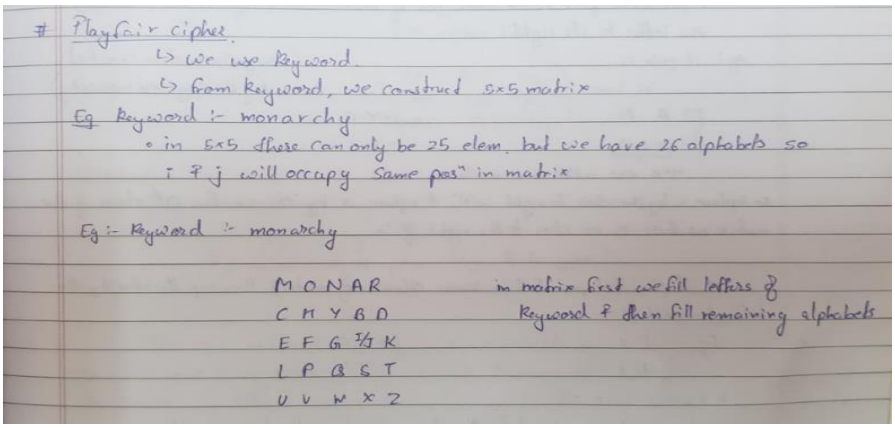
Playfair Cypher

Output Snapshot

Full Code ss



Handwritten Sum



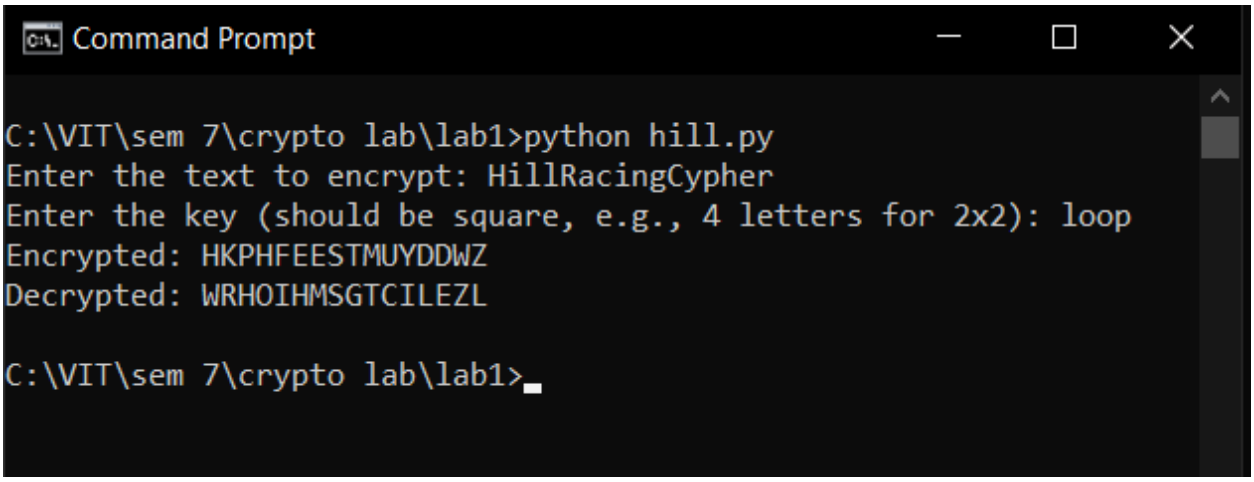
Source Code

```
def preprocess_text(text):
    text = text.upper().replace('J', 'I')
    processed_text = ''
    i = 0
    while i < len(text):
        if i + 1 < len(text) and text[i] != text[i + 1]:
            processed_text += text[i] + text[i + 1]
            i += 2
        else:
            processed_text += text[i] + 'X'
            i += 1
    return processed_text

def create_matrix(key):
    key = key.upper().replace('J', 'I')
    matrix = []
    seen = set()
    for char in key:
        if char.isalpha() and char not in seen:
            seen.add(char)
            matrix.append(char)
    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':
        if char not in seen:
            matrix.append(char)
    return [matrix[i:i + 5] for i in range(0, 25, 5)]

def find_position(matrix, char):
    for row in range(5):
        if char in matrix[row]:
            return (row, matrix[row].index(char))
```

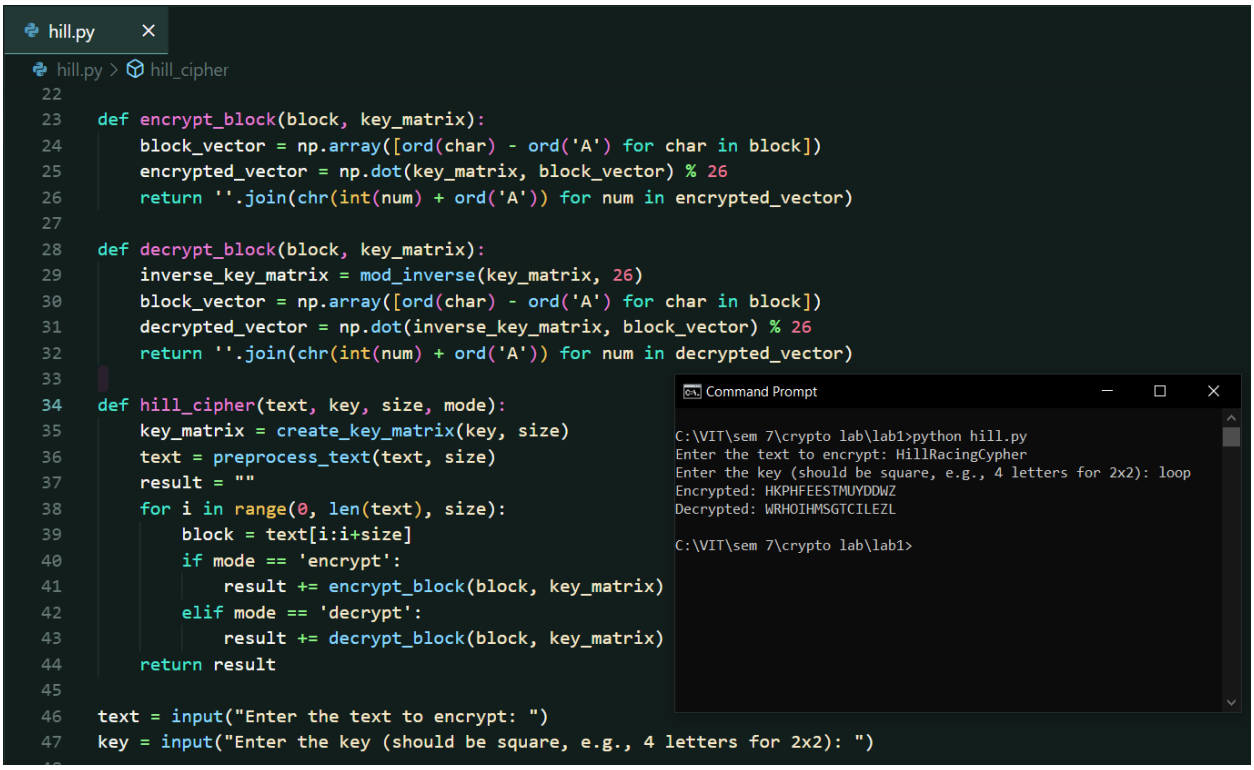
```
    return None

def encrypt_pair(char1, char2, matrix):
    row1, col1 = find_position(matrix, char1)
    row2, col2 = find_position(matrix, char2)
    if row1 == row2:
        return matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
    elif col1 == col2:
        return matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def decrypt_pair(char1, char2, matrix):
    row1, col1 = find_position(matrix, char1)
    row2, col2 = find_position(matrix, char2)
    if row1 == row2:
        return matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
    elif col1 == col2:
        return matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def playfair_cipher(text, key, mode):
    matrix = create_matrix(key)
    text = preprocess_text(text)
    result = ""
    for i in range(0, len(text), 2):
        if mode == 'encrypt':
            result += encrypt_pair(text[i], text[i + 1], matrix)
        elif mode == 'decrypt':
            result += decrypt_pair(text[i], text[i + 1], matrix)
    return result

text = input("Enter the text to encrypt: ")
key = input("Enter the key: ")

enc_text = playfair_cipher(text, key, 'encrypt')
dec_text = playfair_cipher(enc_text, key, 'decrypt')

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Hill Cypher

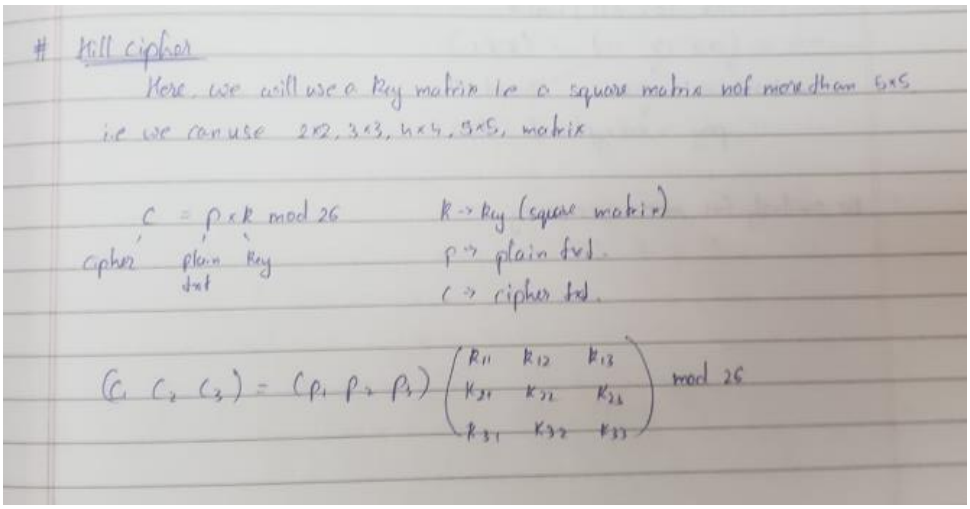
Output Snapshot



Full Code ss



Handwritten Sum



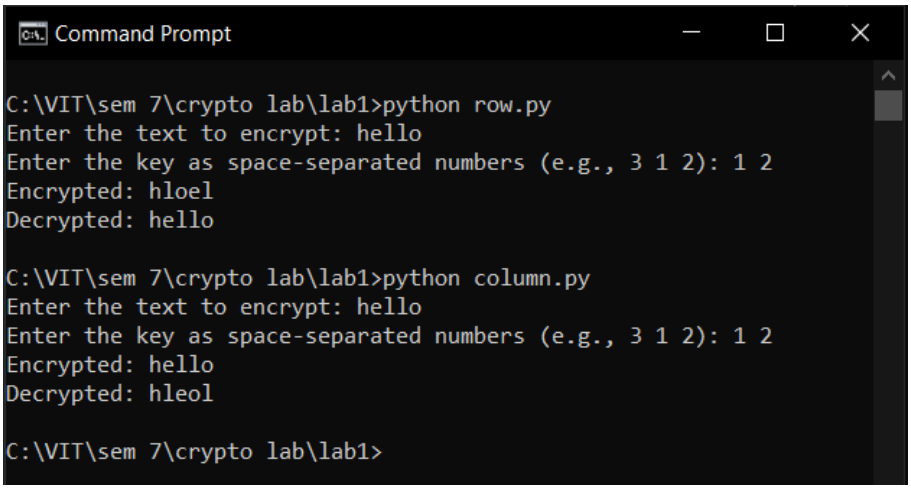
Source Code

```
import numpy as np

def preprocess_text(text, size):
    text = text.upper().replace(' ', '')
    if len(text) % size != 0:
        text += 'X' * (size - len(text) % size)
    return text

def create_key_matrix(key, size):
    key = key.upper().replace(' ', '')
    key_matrix = []
    for i in range(size):
        row = [ord(char) - ord('A') for char in key[i*size:(i+1)*size]]
        key_matrix.append(row)
    return np.array(key_matrix)

def mod_inverse(matrix, modulus):
    determinant = int(np.round(np.linalg.det(matrix))) % modulus
    inverse_determinant = pow(determinant, -1, modulus)
    adjugate_matrix = np.round(np.linalg.inv(matrix) * determinant).astype(int) %
modulus
    return (inverse_determinant * adjugate_matrix) % modulus

def encrypt_block(block, key_matrix):
    block_vector = np.array([ord(char) - ord('A') for char in block])
    encrypted_vector = np.dot(key_matrix, block_vector) % 26
    return ''.join(chr(int(num) + ord('A')) for num in encrypted_vector)
```

```python
def decrypt_block(block, key_matrix):
    inverse_key_matrix = mod_inverse(key_matrix, 26)
    block_vector = np.array([ord(char) - ord('A') for char in block])
    decrypted_vector = np.dot(inverse_key_matrix, block_vector) % 26
    return ''.join(chr(int(num) + ord('A')) for num in decrypted_vector)

def hill_cipher(text, key, size, mode):
    key_matrix = create_key_matrix(key, size)
    text = preprocess_text(text, size)
    result = ""
    for i in range(0, len(text), size):
        block = text[i:i+size]
        if mode == 'encrypt':
            result += encrypt_block(block, key_matrix)
        elif mode == 'decrypt':
            result += decrypt_block(block, key_matrix)
    return result

text = input("Enter the text to encrypt: ")
key = input("Enter the key (should be square, e.g., 4 letters for 2x2): ")
size = int(len(key)**0.5)

enc_text = hill_cipher(text, key, size, 'encrypt')
dec_text = hill_cipher(enc_text, key, size, 'decrypt')

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Transportation - Row and Column Cypher

Output Snapshot



Full Code ss

row.py

```python
def row_transpose_encrypt(text, key):
    rows = len(key)
    cols = (len(text) + rows - 1) // rows
    grid = ['' for _ in range(rows)]

    for i, char in enumerate(text):
        grid[i % rows] += char

    # Create the encrypted message by reading
    the grid row by row
    return ''.join(grid[key.index(i + 1)] for i
    in range(rows))

def row_transpose_decrypt(ciphe
    rows = len(key)
    cols = (len(cipher) + rows
    grid = ['' for _ in range(r

    # Fill the grid by columns
    idx = 0
    for i in range(rows):
        k = key.index(i + 1)
        grid[k] = cipher[idx:id
        idx += cols

    # Read the grid column by column to decrypt
    result = ''
    for i in range(cols):
        for j in range(rows):
            if i < len(grid[j]):
                result += grid[j][i]
    return result
```

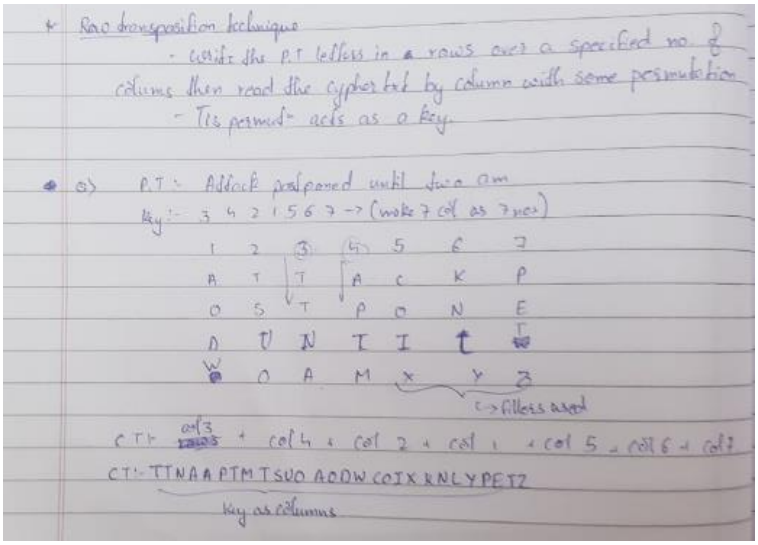
Command Prompt

```
C:\VIT\sem 7\crypto lab\lab1>python row.py
Enter the text to encrypt: hello
Enter the key as space-separated numbers (e.g., 3 1 2): 1 2
Encrypted: hloel
Decrypted: hello

C:\VIT\sem 7\crypto lab\lab1>python column.py
Enter the text to encrypt: hello
Enter the key as space-separated numbers (e.g., 3 1 2): 1 2
Encrypted: hello
Decrypted: hleol

C:\VIT\sem 7\crypto lab\lab1>
```

column.py

```python
def column_transpose_encrypt(text, key):
    rows = (len(text) + len(key) - 1) // len(key)
    grid = ['' for _ in range(rows)]

    # Fill the grid row by row
    for i, char in enumerate(text):
        grid[i // len(key)] += char

    # Create the encrypted message by reading
    the grid by columns based on the key
    return ''.join(grid[i // len(key)][key.index
    (i % len(key) + 1)] for i in range(len

e_decrypt(cipher, key):
her) + len(key) - 1) // len

_ in range(len(key))]

 column by column based on

len(key)):
ndex(k + 1)
 cipher[idx:idx + rows]
    idx += rows

# Read the grid row by row to decrypt
result = ''
for i in range(rows):
    for j in range(len(key)):
        if i < len(grid[j]):
```

Handwritten Sum



Source Code

```python
def row_transpose_encrypt(text, key):
    rows = len(key)
```

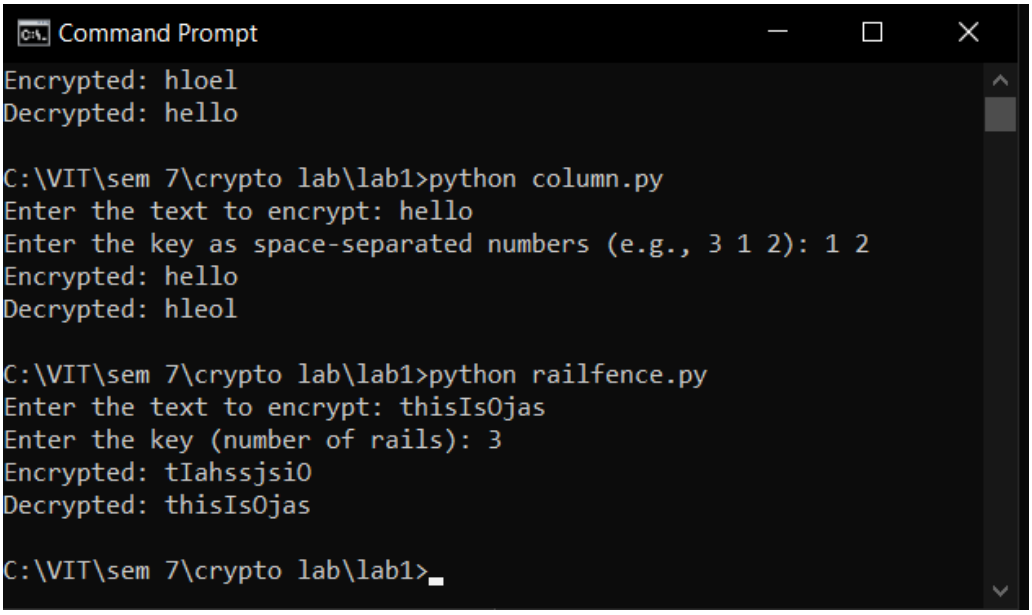
```
    cols = (len(text) + rows - 1) // rows
    grid = ['' for _ in range(rows)]

    for i, char in enumerate(text):
        grid[i % rows] += char

    # Create the encrypted message by reading the grid row by row
    return ''.join(grid[key.index(i + 1)] for i in range(rows))

def row_transpose_decrypt(cipher, key):
    rows = len(key)
    cols = (len(cipher) + rows - 1) // rows
    grid = ['' for _ in range(rows)]

    # Fill the grid by columns based on the key
    idx = 0
    for i in range(rows):
        k = key.index(i + 1)
        grid[k] = cipher[idx:idx + cols]
        idx += cols

    # Read the grid column by column to decrypt
    result = ''
    for i in range(cols):
        for j in range(rows):
            if i < len(grid[j]):
                result += grid[j][i]
    return result

text = input("Enter the text to encrypt: ")
key = list(map(int, input("Enter the key as space-separated numbers (e.g., 3 1
2): ").split()))

enc_text = row_transpose_encrypt(text, key)
dec_text = row_transpose_decrypt(enc_text, key)

print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

```
def column_transpose_encrypt(text, key):
    rows = (len(text) + len(key) - 1) // len(key)
```

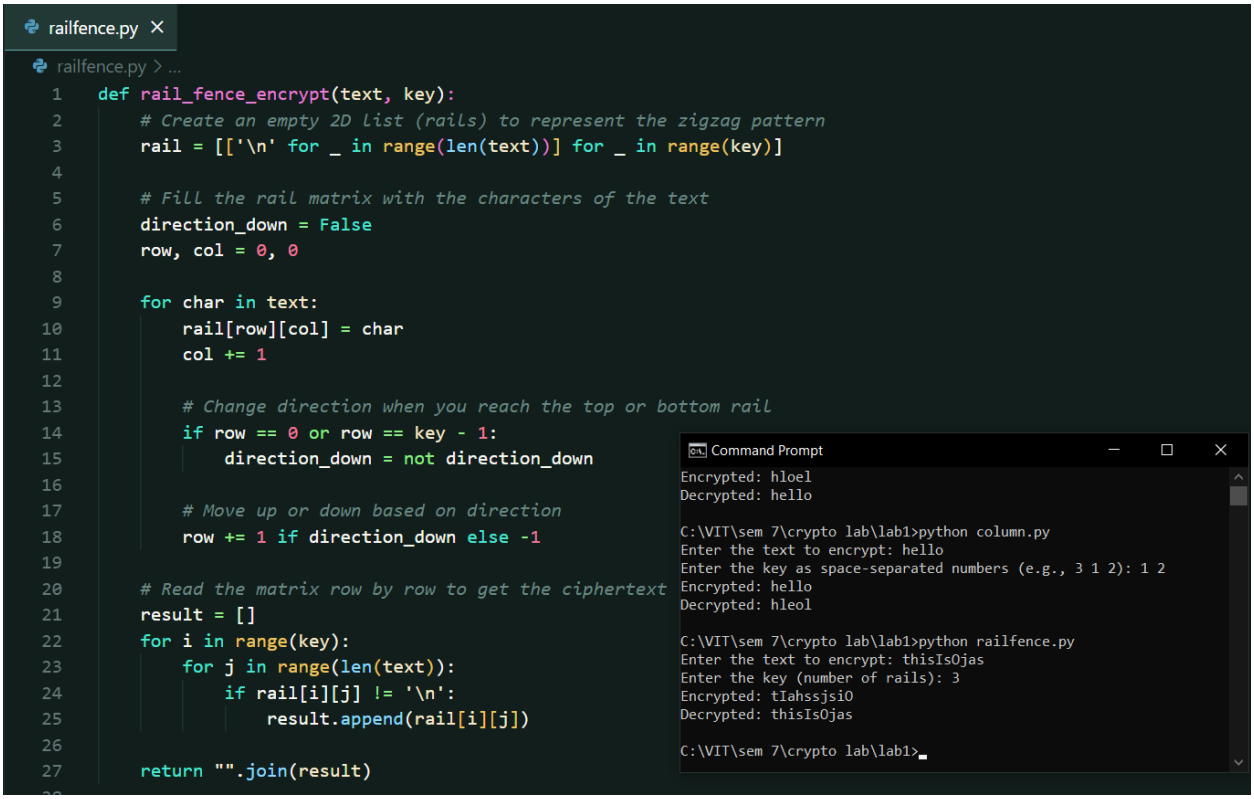
```
    grid = ['' for _ in range(rows)]

    # Fill the grid row by row
    for i, char in enumerate(text):
        grid[i // len(key)] += char

    # Create the encrypted message by reading the grid by columns based on the
key
    return ''.join(grid[i // len(key)][key.index(i % len(key) + 1)] for i in
range(len(text)))

def column_transpose_decrypt(cipher, key):
    rows = (len(cipher) + len(key) - 1) // len(key)
    grid = ['' for _ in range(len(key))]

    # Fill the grid column by column based on the key
    idx = 0
    for k in range(len(key)):
        col = key.index(k + 1)
        grid[col] = cipher[idx:idx + rows]
        idx += rows

    # Read the grid row by row to decrypt
    result = ''
    for i in range(rows):
        for j in range(len(key)):
            if i < len(grid[j]):
                result += grid[j][i]
    return result

# Take user input
text = input("Enter the text to encrypt: ")
key = list(map(int, input("Enter the key as space-separated numbers (e.g., 3 1
2): ").split()))

# Encrypt and decrypt the input text
enc_text = column_transpose_encrypt(text, key)
dec_text = column_transpose_decrypt(enc_text, key)

# Print the results
print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Railfence Cypher

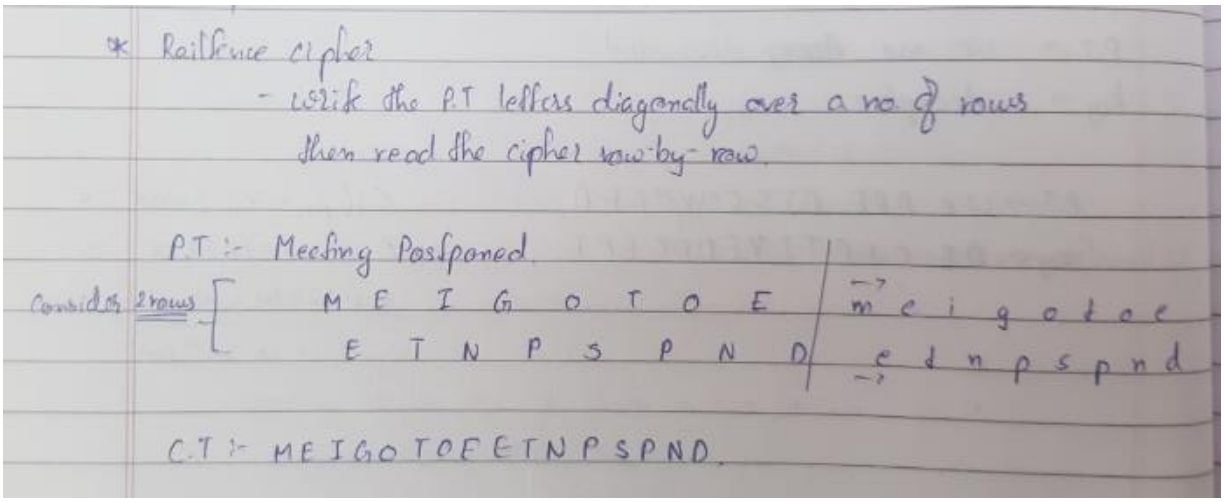
Output Snapshot



Full Code ss



Handwritten Sum



Source Code

```
def rail_fence_encrypt(text, key):
    # Create an empty 2D list (rails) to represent the zigzag pattern
    rail = [['\n' for _ in range(len(text))] for _ in range(key)]

    # Fill the rail matrix with the characters of the text
    direction_down = False
    row, col = 0, 0

    for char in text:
        rail[row][col] = char
        col += 1

        # Change direction when you reach the top or bottom rail
        if row == 0 or row == key - 1:
            direction_down = not direction_down

        # Move up or down based on direction
        row += 1 if direction_down else -1

    # Read the matrix row by row to get the ciphertext
    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
```

```
    return "".join(result)

def rail_fence_decrypt(cipher, key):
    # Create an empty 2D list (rails) to represent the zigzag pattern
    rail = [['\n' for _ in range(len(cipher))] for _ in range(key)]

    # Mark the positions where characters will go
    direction_down = None
    row, col = 0, 0

    for _ in range(len(cipher)):
        # Place a marker '*' at the positions to fill later
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False

        rail[row][col] = '*'
        col += 1
        row += 1 if direction_down else -1

    # Now, fill the markers with the actual cipher text
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if rail[i][j] == '*' and index < len(cipher):
                rail[i][j] = cipher[index]
                index += 1

    # Read the matrix to reconstruct the plaintext
    result = []
    row, col = 0, 0
    for _ in range(len(cipher)):
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False

        if rail[row][col] != '\n':
            result.append(rail[row][col])
            col += 1

        row += 1 if direction_down else -1

    return "".join(result)
```

```
# Take user input
text = input("Enter the text to encrypt: ")
key = int(input("Enter the key (number of rails): "))

# Encrypt and decrypt the input text
enc_text = rail_fence_encrypt(text, key)
dec_text = rail_fence_decrypt(enc_text, key)

# Print the results
print(f"Encrypted: {enc_text}")
print(f"Decrypted: {dec_text}")
```

Conclusion

We explored substitution and transposition ciphers, understanding how they alter or rearrange plaintext to secure information. These concepts were demonstrated through various Python implementations, highlighting their encryption and decryption processes.