Practical 2

Playfair Cipher

Problem:
The task is to implement a Playfair Cipher, a type of substitution cipher in which pairs of letters are encrypted using a key matrix. This involves creating a program that can both encrypt and decrypt messages using this cipher.

Description:
The Playfair Cipher encrypts pairs of letters (digraphs) in the plaintext. If both letters in the pair are the same or if there's only one letter left at the end, an 'X' is inserted between them or at the end. Non-alphabetic characters are ignored during encryption. The matrix for encryption is generated using a keyword.

Code:

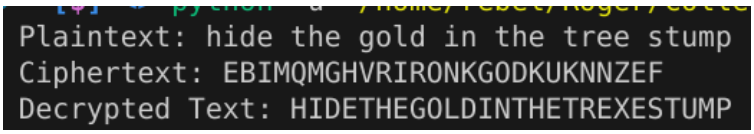
1. playfair-basic.py

```
import string

# function to create the key matrix using the keyword
def create_matrix(keyword):
    keyword = "".join(dict.fromkeys(keyword))  # remove duplicates
    keyword = keyword.replace("J", "I").upper()  # treat 'J' as 'I' and uppercase
    alphabet = string.ascii_uppercase.replace("J", "")

    key_matrix = []
    used_chars = set()

    for char in keyword:
        if char not in used_chars:
            key_matrix.append(char)
            used_chars.add(char)

    for char in alphabet:
        if char not in used_chars:
            key_matrix.append(char)
            used_chars.add(char)

    matrix = [key_matrix[i:i+5] for i in range(0, 25, 5)]  # create 5x5 matrix
    return matrix

# function to preprocess the text
def preprocess_text(text):
    text = text.upper().replace("J", "I")  # replace 'J' with 'I' and convert to uppercase
    processed_text = ""

    i = 0
    while i < len(text):
        if text[i] not in string.ascii_uppercase:  # skip non-alphabet characters
            i += 1
```

```
            continue
        processed_text += text[i]
        if i + 1 < len(text) and text[i] == text[i + 1]:
            processed_text += "X"  # insert 'X' between duplicate letters
        elif i + 1 < len(text) and text[i + 1] in string.ascii_uppercase:
            processed_text += text[i + 1]
            i += 1
        i += 1

    if len(processed_text) % 2 != 0:
        processed_text += "X"  # add 'X' if the length is odd

    return processed_text

# function to find the position of a character in the matrix
def find_position(matrix, char):
    for i, row in enumerate(matrix):
        for j, matrix_char in enumerate(row):
            if matrix_char == char:
                return i, j
    return None

# function to perform playfair cipher encryption or decryption
def playfair_cipher(text, matrix, mode='encrypt'):
    text = preprocess_text(text)
    result = ""

    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 is None or row2 is None:
            raise ValueError(f"Characters {char1} or {char2} not found in matrix")

        if row1 == row2:  # same row
            if mode == 'encrypt':
                result += matrix[row1][(col1 + 1) % 5]
                result += matrix[row2][(col2 + 1) % 5]
            else:
                result += matrix[row1][(col1 - 1) % 5]
                result += matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:  # same column
            if mode == 'encrypt':
                result += matrix[(row1 + 1) % 5][col1]
                result += matrix[(row2 + 1) % 5][col2]
            else:
                result += matrix[(row1 - 1) % 5][col1]
                result += matrix[(row2 - 1) % 5][col2]
        else:  # rectangle swap
            result += matrix[row1][col2]
            result += matrix[row2][col1]
```

```
    return result

# function to encrypt plaintext using playfair cipher
def encrypt(plaintext, keyword):
    matrix = create_matrix(keyword)
    return playfair_cipher(plaintext, matrix, 'encrypt')

# function to decrypt ciphertext using playfair cipher
def decrypt(ciphertext, keyword):
    matrix = create_matrix(keyword)
    return playfair_cipher(ciphertext, matrix, 'decrypt')

# example usage
keyword = "playfair"
plaintext = "hide the gold in the tree stump"
ciphertext = encrypt(plaintext, keyword)
decrypted_text = decrypt(ciphertext, keyword)

print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
print("Decrypted Text:", decrypted_text)
```

Output:



Code Explanation:

1. **Matrix Generation:**
- Constructs a 5x5 matrix for the Playfair cipher using a keyword. The matrix is populated with unique characters from the keyword, followed by the remaining letters of the alphabet (excluding 'J'). Characters are added only once.

2. **Plaintext Formatting:**
- Prepares plaintext for encryption by converting it to uppercase, replacing 'J' with 'I', removing non-alphabetic characters, inserting 'X' between duplicate letters, and ensuring the text length is even.

3. **Encryption:**
- Encrypts the formatted plaintext using the Playfair cipher rules. Characters in the same row are replaced with the next character; in the same column, with the character below; and in different rows and columns, they are swapped diagonally.

4. **Decryption:**
- Decrypts the ciphertext using the reverse of the encryption rules. Characters are substituted according to their positions in the matrix.

5. **Main Block:**

- Demonstrates encryption and decryption of a sample plaintext. It prints the plaintext, ciphertext, and decrypted text.

Peformance Analysis:

- **Space Complexity:**

➢ **Text Storage:** O(n), where n is the length of the input text.
➢ **Matrix and Dictionary:** O(1) since the matrix size is fixed at 5x5.

- **Time Complexity:**

➢ **Matrix Generation:** O(1), as the matrix size is constant.
➢ **Text Formatting, Encryption, and Decryption:** O(n), where n is the length of the input text.

Modified Version:

Code:

1. playfair-modified.py

```
import string

# matrix generation code block

def generate_playfair_matrix(key):
    # we use set because it removes the duplicate entries
    # using lambda to sort the characters in their original order rather than some random order
        key = "".join(sorted(set(key), key=lambda x: key.index(x)))  # Remove duplicates while preserving order

    # storing all ascii uppercase letters but omitting J
    alphabet = string.ascii_uppercase.replace("J", "")  # Playfair cipher traditionally omits 'J'

    # further we just store all the characters in key which are not present in the key
    key += ''.join([char for char in alphabet if char not in key])

    matrix = []
    char_positions = {}  # Dictionary to store character coordinates
    for i in range(5):
    # storing the slices of keys in an order of 5 keys
    # ex- 0-5, 5-10 etc.
        row = key[i*5:(i+1)*5]
        matrix.append(list(row))
        for j in range(5):
            char_positions[row[ j]] = (i, j)  # Store the coordinates

    return matrix, char_positions
```

```
def format_plaintext(plaintext):
    # converting all plaintext to uppercase and removing J with I and remove space in the string
    plaintext = plaintext.upper().replace("J", "I").replace(" ", "")
    formatted_text = ""

    i = 0
    while i < len(plaintext):
        formatted_text += plaintext[i]
        # if two consecutive letters are same it will replace the next one with 'X'
        if i + 1 < len(plaintext) and plaintext[i] == plaintext[i + 1]:
            formatted_text += "X"
        elif i + 1 < len(plaintext):
            formatted_text += plaintext[i + 1]
            i += 1
        i += 1

        # if formatted string is in odd length we have to change it to even length
        # we need even length string to make diagraphs
    if len(formatted_text) % 2 != 0:
        formatted_text += "X"
    return formatted_text

def playfair_encrypt(plaintext, matrix, char_positions):
    ciphertext = ""

    for i in range(0, len(plaintext), 2):
        # extracting coordinates from the dictionary for two consecutive letters
        row1, col1 = char_positions[plaintext[i]]
        row2, col2 = char_positions[plaintext[i + 1]]

        # if letters are in same row then move it by forward one unit in the column
        if row1 == row2:
            ciphertext += matrix[row1][(col1 + 1) % 5]
            ciphertext += matrix[row2][(col2 + 1) % 5]

        # if letters are in same column then move it forward by one unit in the row
        elif col1 == col2:
            ciphertext += matrix[(row1 + 1) % 5][col1]
            ciphertext += matrix[(row2 + 1) % 5][col2]

        else:
            # if not present in the same row/column, replace it with diagonally opposite letters in the
matrix
            ciphertext += matrix[row1][col2]
            ciphertext += matrix[row2][col1]

    return ciphertext

def playfair_decrypt(ciphertext, matrix, char_positions):
    plaintext = ""

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

```
        row1, col1 = char_positions[ciphertext[i]]
        row2, col2 = char_positions[ciphertext[i + 1]]

        # if letters are in same row then move it backwards by one unit in the column
        if row1 == row2:
            plaintext += matrix[row1][(col1 - 1) % 5]
            plaintext += matrix[row2][(col2 - 1) % 5]

        # if letters are in same column then move it backwards by one unit in the row
        elif col1 == col2:
            plaintext += matrix[(row1 - 1) % 5][col1]
            plaintext += matrix[(row2 - 1) % 5][col2]

        else:
        # diagonally exchange the elements
            plaintext += matrix[row1][col2]
            plaintext += matrix[row2][col1]

    return plaintext

# main block

def main():
    while True:
        choice = input("Do you want to Encrypt (E), Decrypt (D), or Quit (Q)? ").upper()
        if choice == 'Q':
            break
        elif choice in ['E', 'D']:
            # converting the string in upper case by default
            key = input("Enter the keyword: ").upper()
            text = input("Enter the text: ").upper()

            # matrix -> list, char_positions -> dict
            matrix, char_positions = generate_playfair_matrix(key)

            if choice == 'E':
                formatted_text = format_plaintext(text)
                result = playfair_encrypt(formatted_text, matrix, char_positions)
                print(f"Encrypted Text: {result}")

            elif choice == 'D':
                result = playfair_decrypt(text, matrix, char_positions)
                print(f"Decrypted Text: {result}")
        else:
            print("Invalid choice. Please enter E for Encrypt, D for Decrypt, or Q for Quit.")

if __name__ == "__main__":
    main()
```

Output:

```
Do you want to Encrypt (E), Decrypt (D), or Quit (Q)? E
Enter the keyword: keyword
Enter the text: PDEU
Encrypted Text: NADE
Do you want to Encrypt (E), Decrypt (D), or Quit (Q)? D
Enter the keyword: keyword
Enter the text: NADE
Decrypted Text: PDEU
Do you want to Encrypt (E), Decrypt (D), or Quit (Q)?
```

Code Explanation:

1. **Matrix Generation Code Block:**

- **generate_playfair_matrix:** Creates a 5x5 matrix using the keyword and the remaining alphabet (excluding 'J'). It also creates a dictionary to store character coordinates.

2. **Plaintext Formatting:**

- **format_plaintext:** Converts plaintext to uppercase, replaces 'J' with 'I', removes spaces, and ensures the length is even by inserting 'X' where necessary.

3. **Encryption Function:**

- **playfair_encrypt:** Encrypts the formatted plaintext using the Playfair Cipher rules (same row, same column, or rectangle swap).

4. **Decryption Function:**

- **playfair_decrypt:** Decrypts the ciphertext using the reverse process of the Playfair Cipher rules.

5. **Main Block:**

- Continuously prompts the user to encrypt or decrypt messages or quit the program.
- Generates the key matrix and character positions based on user input.
- Formats the text for encryption or directly uses it for decryption.
- Displays the encrypted or decrypted result based on user choice.

Performance Analysis:

- **Space Complexity:**

➢ The space complexity is O(n) for the input text storage and O(1) for the 5x5 matrix and character positions dictionary.

- **Time Complexity:**

➢ The time complexity is O(n) for the formatting, encryption, and decryption processes, where n is the length of the input text. The matrix generation is O(1) since it's constant size.