21CSE1003 ASHISH SINGH

1. Write the Encryption and decryption procedure for

I. Column Transposition

1. **Input:**
    - Plaintext: plain_text (a string of text to be encrypted)
    - Key: key (a string used for determining column order)
2. **Convert Key to Numeric Order:**
    - Sort the characters of the key alphabetically and assign each character its corresponding position (1-based index) in the sorted order.
    - Example: For the key "ZEBRAS", the numeric key becomes "632415".
3. **Preprocessing:**
    - Remove any spaces from the plain_text.
    - Calculate the number of columns (col) based on the length of the key.
    - Calculate the number of rows (row) required to fit the entire plain_text into the grid, using ceil(len(plain_text) / col).
4. **Create Grid:**
    - Initialize an empty grid (matrix) of size row x col.
    - Fill the grid row-wise with characters from the plain_text.
    - If the length of plain_text is not a perfect fit, pad the remaining cells with random lowercase letters.
5. **Columnar Transposition:**
    - Sort the columns based on the numeric order of the key.
    - Read the grid column-wise in the order specified by the sorted key, appending characters to cipher_text.
6. **Output:**
    - Return the cipher_text.

Decryption:

1. **Input:**
    - Ciphertext: cipher_text (a string to be decrypted)
    - Key: key (a string used for determining column order)

2. **Convert Key to Numeric Order:**
    - Convert the key to its numeric order as done during encryption.
3. **Determine Grid Dimensions:**
    - Calculate the number of columns (col) based on the length of the key.
    - Calculate the number of rows (row) using ceil(len(cipher_text) / col).
4. **Determine Column Sizes:**
    - Each column in the grid should have row characters, except for some columns which will have one less character if the ciphertext doesn't perfectly fit the grid.
5. **Rebuild Grid:**
    - Fill the grid column-wise by placing characters from cipher_text into columns, according to the numeric order of the key.
6. **Reconstruct Plaintext:**
    - Read the grid row-wise to extract the original plain_text.
7. **Output:**
    - Return the plain_text.

LAB_7\column_transposition.py

```python

# 21CSE1003 Ashish Singh
# Q1. Write the Encryption and decryption procedure for Column Transposition

import math, random, string

# Function to generate a random lowercase letter
def random_lowercase():
    return random.choice(string.ascii_lowercase)

# Function to convert the key from text to numerical order
def convert_key_to_numeric(key):
    sorted_key = sorted(key)
    key_order = {char: str(sorted_key.index(char) + 1) for char in key}
    return ''.join(key_order[char] for char in key)

# Encryption function
def encrypt_columnar_transposition(plain_text, key):
    # Convert the key to numeric order
    key = convert_key_to_numeric(key)

    # Remove spaces from the plain_text
    plain_text = plain_text.replace(" ", "")

    # Calculate number of columns and rows
    col = len(key)
    row = math.ceil(len(plain_text) / col)

    # Create a grid (matrix) to write the plain text row-wise
    grid = [[''] * col for _ in range(row)]

    index = 0
    for r in range(row):
        for c in range(col):
            if index < len(plain_text):
                grid[r][c] = plain_text[index]
                index += 1
            else:
                grid[r][c] = random_lowercase()  # Padding with random lowercase letters

    # Create a sorted key order
    sorted_key_order = sorted(list(enumerate(key)), key=lambda x: x[1])

    # Read the grid column-wise in the order specified by the sorted key
    cipher_text = ''
    for col_idx, _ in sorted_key_order:
        for r in range(row):
            cipher_text += grid[r][col_idx]
```

```
49
50      return cipher_text
51
52  # Decryption function
53  def decrypt_columnar_transposition(cipher_text, key):
54      # Convert the key to numeric order
55      key = convert_key_to_numeric(key)
56
57      # Calculate number of columns and rows
58      col = len(key)
59      row = math.ceil(len(cipher_text) / col)
60
61      # Create a sorted key order to determine column positions
62      sorted_key_order = sorted(list(enumerate(key)), key=lambda x: x[1])
63
64      # Rebuild the grid with empty characters
65      grid = [[''] * col for _ in range(row)]
66
67      # Calculate how many characters in each column
68      col_lengths = [row] * col
69      total_chars = len(cipher_text)
70      short_cols = col * row - total_chars
71      for i in range(short_cols):
72          col_lengths[sorted_key_order[col - 1 - i][0]] -= 1
73
74      # Fill the grid column by column based on the key order
75      index = 0
76      for col_idx, _ in sorted_key_order:
77          for r in range(col_lengths[col_idx]):
78              if index < len(cipher_text):
79                  grid[r][col_idx] = cipher_text[index]
80                  index += 1
81
82      # Read the grid row-wise to reconstruct the plain text
83      plain_text = ''
84      for r in range(row):
85          for c in range(col):
86              plain_text += grid[r][c]
87      return plain_text
88
89  # Example usage
90  plain_text = "WE ARE DISCOVERED FLEES AT ONCE"
91  key = "ZEBRAS"
92  # Encrypt
93  cipher_text = encrypt_columnar_transposition(plain_text, key)
94  print(f"Ciphertext: {cipher_text}")
95  # Decrypt
96  decrypted_text = decrypt_columnar_transposition(cipher_text, key)
97  print(f"Decrypted Text: {decrypted_text}")
```

II. Route cipher

Encryption:

1. **Input:**
    - Plaintext: plain_text (the text to be encrypted)
    - Number of Rows: row (the number of rows in the grid)
    - Number of Columns: col (the number of columns in the grid)
2. **Create and Fill the Grid:**
    - Initialize a grid (matrix) of size row x col.
    - Fill the grid with characters from the plain_text row-wise.
    - If the plain_text is shorter than row * col, pad the remaining cells with sequential uppercase letters (A, B, C, ...).
3. **Spiral Encryption:**
    - Define the boundaries: up, down, left, right.
    - Read the grid in a spiral order:
        - Traverse down along the rightmost column.
        - Traverse left along the bottom row.
        - Traverse up along the leftmost column (if applicable).
        - Traverse right along the top row (if applicable).
    - Append the characters to form the encrypted_text.
4. **Output:**
    - Return the encrypted_text.

Decryption:

1. **Input:**
    - Ciphertext: encrypted_text (the text to be decrypted)
    - Number of Rows: row (the number of rows in the grid)
    - Number of Columns: col (the number of columns in the grid)
2. **Create an Empty Grid:**
    - Initialize a grid (matrix) of size row x col.
3. **Spiral Filling of the Grid:**
    - Define the boundaries: up, down, left, right.

- **Fill the grid in spiral order with characters from encrypted_text:**
    - Traverse down along the rightmost column.
    - Traverse left along the bottom row.
    - Traverse up along the leftmost column (if applicable).
    - Traverse right along the top row (if applicable).

4. **Reconstruct the Plaintext:**
    - Read the grid column-wise to retrieve the original plain_text.
    - Remove any extra padding characters if necessary.
5. **Output:**
    - Return the decrypted_text.

LAB_7\route_cipher.py

```python
# Q1. Write the Encryption and decryption procedure for Route cipher

import string

# Function to generate grid and fill with plain text or padding
def create_grid(plain_text, row, col):
    grid = [[''] * col for _ in range(row)]

    p, q = 0, 0
    h = 0
    index = 0

    for i in range(row * col):
        if p == row:
            q += 1
            p = 0

        if index >= len(plain_text):
            grid[p][q] = chr(65 + h)  # Padding with 'A', 'B', 'C', ...
            h += 1
        else:
            grid[p][q] = plain_text[index]
            index += 1

        p += 1

    return grid

# Function to display the grid (for debugging/visualization)
def display_grid(grid):
    for row in grid:
        print(' '.join(row))
    print()

# Function to encrypt text using the route cipher (spiral inward)
def encrypt_route_cipher(plain_text, row, col):
    grid = create_grid(plain_text, row, col)

    # Display the filled grid
    print("Grid:")
    display_grid(grid)

    # Spiral reading from the grid
    encrypted_text = ''
    up, down = 0, row - 1
    left, right = 0, col - 1

    while up <= down and left <= right:
```

```
        # Traverse down the right side
        for i in range(up, down + 1):
            encrypted_text += grid[i][right]
        right -= 1

        # Traverse left on the bottom side
        for i in range(right, left - 1, -1):
            encrypted_text += grid[down][i]
        down -= 1

        # Traverse up the left side
        if left <= right:
            for i in range(down, up - 1, -1):
                encrypted_text += grid[i][left]
            left += 1

        # Traverse right on the top side
        if up <= down:
            for i in range(left, right + 1):
                encrypted_text += grid[up][i]
            up += 1

    return encrypted_text

# Function to decrypt text using the route cipher (spiral inward)
def decrypt_route_cipher(encrypted_text, row, col):
    # Create an empty grid
    grid = [[''] * col for _ in range(row)]

    # Define the boundaries
    up, down = 0, row - 1
    left, right = 0, col - 1

    index = 0

    # Fill the grid with the cipher text in spiral order
    while up <= down and left <= right:
        # Traverse down the right side
        for i in range(up, down + 1):
            grid[i][right] = encrypted_text[index]
            index += 1
        right -= 1

        # Traverse left on the bottom side
        for i in range(right, left - 1, -1):
            grid[down][i] = encrypted_text[index]
            index += 1
        down -= 1

        # Traverse up the left side
```

```
        if left ⩽ right:
            for i in range(down, up - 1, -1):
                grid[i][left] = encrypted_text[index]
                index += 1
            left += 1

        # Traverse right on the top side
        if up ⩽ down:
            for i in range(left, right + 1):
                grid[up][i] = encrypted_text[index]
                index += 1
            up += 1

    # Now read the grid column by column to get the original plaintext
    decrypted_text = ''
    for p in range(col):
        for q in range(row):
            decrypted_text += grid[q][p]

    return decrypted_text.strip()

# Example usage
plain_text = input("Enter the plain text: ").replace(" ", "")
row = int(input("Enter number of rows: "))
col = int(input("Enter number of columns: "))

# Encrypt the plain text
encrypted_text = encrypt_route_cipher(plain_text, row, col)
print(f"Encrypted Text: {encrypted_text}")

# Decrypt the cipher text
decrypted_text = decrypt_route_cipher(encrypted_text, row, col)
print(f"Decrypted Text: {decrypted_text}")

```

III. Row column transposition

Encryption:

1. **Input:**
    - Plaintext: plain_text (the text to be encrypted)
    - Number of Rows: row (the number of rows in the grid)
    - Number of Columns: col (the number of columns in the grid)
2. **Create and Fill the Grid:**
    - Initialize a grid (matrix) of size row x col.
    - Fill the grid with characters from the plain_text row-wise.
    - If the plain_text is shorter than row * col, pad the remaining cells with the character 'X'.
3. **Column-wise Reading for Encryption:**
    - Initialize an empty string encrypted_text.
    - Read the grid column-wise:
        - For each column, iterate through all rows and append the characters to encrypted_text.
4. **Output:**
    - Return the encrypted_text.

Decryption:

1. **Input:**
    - Ciphertext: cipher_text (the text to be decrypted)
    - Number of Rows: row (the number of rows in the grid)
    - Number of Columns: col (the number of columns in the grid)
2. **Create an Empty Grid:**
    - Initialize a grid (matrix) of size row x col.
3. **Column-wise Filling of the Grid:**
    - Fill the grid with characters from cipher_text column-wise:
        - For each column, iterate through all rows and place characters in the grid from cipher_text.
4. **Row-wise Reading for Decryption:**
    - Initialize an empty string decrypted_text.

- Read the grid row-wise:
    - For each row, iterate through all columns and append the characters to decrypted_text.
- Remove any padding characters ('X') from the decrypted_text.

5. **Output:**
    - Return the decrypted_text.

LAB_7\row_column_transposition.py

```python
# Q1. Write the Encryption and decryption procedure for Row column transposition


import math

# Function to create a grid for row-column transform
def create_grid(plain_text, row, col):
    grid = [[''] * col for _ in range(row)]

    index = 0
    for r in range(row):
        for c in range(col):
            if index < len(plain_text):
                grid[r][c] = plain_text[index]
                index += 1
            else:
                grid[r][c] = 'X'  # Padding with 'X' if necessary

    return grid

# Function to display the grid (for debugging/visualization)
def display_grid(grid):
    for row in grid:
        print(' '.join(row))
    print()

# Function to encrypt text using row-column transform cipher
def encrypt_row_column(plain_text, row, col):
    # Remove spaces from plain text
    plain_text = plain_text.replace(" ", "")

    # Create the grid
    grid = create_grid(plain_text, row, col)

    # Display the filled grid
    print("Grid:")
    display_grid(grid)

    # Read the grid column by column to generate ciphertext
    encrypted_text = ''
    for c in range(col):
        for r in range(row):
            encrypted_text += grid[r][c]

    return encrypted_text

# Function to decrypt text using row-column transform cipher
def decrypt_row_column(cipher_text, row, col):
```

```
    # Create an empty grid
    grid = [[''] * col for _ in range(row)]

    # Fill the grid column by column with the cipher text
    index = 0
    for c in range(col):
        for r in range(row):
            if index < len(cipher_text):
                grid[r][c] = cipher_text[index]
                index += 1

    # Display the filled grid
    print("Decryption Grid:")
    display_grid(grid)

    # Read the grid row by row to generate plaintext
    decrypted_text = ''
    for r in range(row):
        for c in range(col):
            decrypted_text += grid[r][c]

    return decrypted_text.strip('X')  # Remove padding 'X' characters

# Example usage
plain_text = input("Enter the plain text: ").replace(" ", "")
row = int(input("Enter number of rows: "))
col = int(input("Enter number of columns: "))

# Encrypt the plain text
encrypted_text = encrypt_row_column(plain_text, row, col)
print(f"Encrypted Text: {encrypted_text}")

# Decrypt the cipher text
decrypted_text = decrypt_row_column(encrypted_text, row, col)
print(f"Decrypted Text: {decrypted_text}")

```

IV. Rail fence cipher

Encryption:

1. **Input:**
    - Plaintext: plain_text (the text to be encrypted)
    - Key: key (the number of rails or lines in the fence)
2. **Create the Grid:**
    - Initialize a grid (matrix) with dimensions key x len(plain_text) filled with placeholder characters (e.g., '0').
3. **Fill the Grid in Zigzag Manner:**
    - For each character in the plain_text, determine the appropriate row in the grid based on the current index:
        - If the current index falls in an "up" direction of the zigzag (even iteration of the complete cycle):
            - Calculate the row as row = i % (key - 1).
        - If it falls in a "down" direction (odd iteration of the complete cycle):
            - Calculate the row as row = (key - 1) - (i % (key - 1)).
    - Place the character in the determined row and the column corresponding to the index.
4. **Collect the Encrypted Text:**
    - Initialize an empty string encrypted_text.
    - Read the grid row by row:
        - For each row, append characters to encrypted_text only if they are not placeholders ('0').
5. **Output:**
    - Return the encrypted_text.

LAB_7\rail_fence_cipher.py

```python
# Q1. Write the Encryption and decryption procedure for Rail fence cipher

def encrypt_rail_fence(plain_text, key):
    # Create a grid with '0' as placeholders
    grid = [['0' for _ in range(len(plain_text))] for _ in range(key)]

    # Populate the grid in the zigzag manner (like in the C++ version)
    for i, char in enumerate(plain_text):
        if (i // (key - 1)) % 2 == 0:
            row = i % (key - 1)
        else:
            row = (key - 1) - (i % (key - 1))

        grid[row][i] = char

    # Collect the encrypted text by reading row by row
    encrypted_text = ""
    for r in range(key):
        for c in range(len(plain_text)):
            if grid[r][c] != '0':
                encrypted_text += grid[r][c]

    return encrypted_text

# Example usage
plain_text = input("Enter the plain text: ").replace(" ", "")
key = int(input("Enter the key: "))

# Encrypt the plain text
encrypted_text = encrypt_rail_fence(plain_text, key)
print(f"Encrypted Text: {encrypted_text}")

```

2. **Write a program to encrypt and decrypt the text using Affine cipher.**

Encryption:

1. **Input:**
    - Plaintext: plain_text (the text to be encrypted)
    - Key a: (must be coprime with the size of the alphabet, m)
    - Key b: (a constant added to the result)
2. **Initialize:**
    - Set m = 26, which is the number of letters in the English alphabet.
    - Initialize an empty string cipher_text to store the encrypted result.
3. **Encryption Process:**
    - For each character in plain_text:
        - Check if the character is alphabetic:
            - Convert the character to uppercase and calculate its numeric equivalent x (0 for 'A', 1 for 'B', ..., 25 for 'Z').
            - Apply the affine encryption formula: encrypted_char=(a·x+b)mod m\text{encrypted_char} = (a \cdot x + b) \mod mencrypted_char=(a·x+b)modm
            - Convert the result back to a character and append it to cipher_text.
        - If the character is non-alphabetic, append it unchanged to cipher_text.
4. **Output:**
    - Return the cipher_text.

Decryption:

1. **Input:**
    - Ciphertext: cipher_text (the text to be decrypted)
    - Key a: (the same key used for encryption)
    - Key b: (the same constant added during encryption)
2. **Initialize:**
    - Set m = 26 (the number of letters in the alphabet).
    - Find the modular inverse of a under modulo m using the function mod_inverse.
    - Initialize an empty string plain_text to store the decrypted result.
3. **Decryption Process:**

- For each character in cipher_text:
    - Check if the character is alphabetic:
        - Convert the character to uppercase and calculate its numeric equivalent y.
        - Apply the affine decryption formula: decrypted_char=(ainv⋅(y−b))mod m\text{decrypted_char} = (a_{inv} \cdot (y - b)) \mod mdecrypted_char=(ainv⋅(y−b))modm
        - Convert the result back to a character and append it to plain_text.
    - If the character is non-alphabetic, append it unchanged to plain_text.

4. **Output:**
    - Return the plain_text.

LAB_7\affine_cipher.py

```python

# Q2. Write a program to encrypt and decrypt the text using Affine cipher.

# Function to find the modular inverse of a under modulo m
def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    raise ValueError(f"Inverse of {a} mod {m} does not exist")

# Function to encrypt using Affine Cipher
def affine_encrypt(plain_text, a, b):
    cipher_text = ""
    m = 26  # Size of the alphabet

    for char in plain_text:
        if char.isalpha():  # Encrypt only alphabetic characters
            x = ord(char.upper()) - 65
            encrypted_char = (a * x + b) % m
            cipher_text += chr(encrypted_char + 65)
        else:
            cipher_text += char  # Non-alphabetic characters remain unchanged

    return cipher_text

# Function to decrypt using Affine Cipher
def affine_decrypt(cipher_text, a, b):
    plain_text = ""
    m = 26  # Size of the alphabet
    a_inv = mod_inverse(a, m)  # Find modular inverse of 'a'

    for char in cipher_text:
        if char.isalpha():  # Decrypt only alphabetic characters
            y = ord(char.upper()) - 65
            decrypted_char = (a_inv * (y - b)) % m
            plain_text += chr(decrypted_char + 65)
        else:
            plain_text += char  # Non-alphabetic characters remain unchanged

    return plain_text

# Example usage
plain_text = input("Enter the plain text: ").replace(" ", "")
a = int(input("Enter key 'a' (must be coprime with 26): "))
b = int(input("Enter key 'b': "))

# Encrypt the plain text
encrypted_text = affine_encrypt(plain_text, a, b)
```

```
print(f"Encrypted Text: {encrypted_text}")

# Decrypt the cipher text
decrypted_text = affine_decrypt(encrypted_text, a, b)
print(f"Decrypted Text: {decrypted_text}")

```

3. Write a program to encrypt and decrypt the text using Vigenère cipher.

Encryption:

1. **Input:**
    - **plain_text: The text to be encrypted.**
    - **key: A string used to determine the shifts for each character in the plaintext.**
2. **Initialize:**
    - **Calculate the length of the key.**
    - **Extend the key to match the length of the plain_text by repeating the key and truncating as needed.**
3. **Encryption Process:**
    - **For each character in the plain_text:**
        - **If the character is alphabetic:**
            - **Convert both the character and the corresponding character from the key to uppercase.**
            - **Calculate the shift as the difference between the ASCII value of the key character and 'A'.**
            - **Encrypt the plaintext character using the formula: encrypted_char=(ASCII_value_of_plain_char−ASCII_of_A+shift)%26**
            - **Append the encrypted character to the cipher_text.**
        - **If the character is non-alphabetic, append it unchanged to the cipher_text.**
4. **Output:**
    - **Join and return the cipher_text.**

Decryption:

1. **Input:**
    - **cipher_text: The text to be decrypted.**
    - **key: The same key used for encryption.**
2. **Initialize:**
    - **Calculate the length of the key.**
    - **Extend the key to match the length of the cipher_text as in encryption.**
3. **Decryption Process:**
    - **For each character in the cipher_text:**

- **If the character is alphabetic:**
    - **Convert both the character and the corresponding character from the key to uppercase.**
    - **Calculate the shift using the ASCII value of the corresponding key character.**
    - **Decrypt the cipher_text character using the formula: decrypted_char=(ASCII_value_of_cipher_char−ASCII_of_A−shift+26)**
    - **Append the decrypted character to the plain_text.**
- **If the character is non-alphabetic, append it unchanged to the plain_text.**

4. **Output:**
    - **Join and return the plain_text.**

LAB_7\vigenere_cipher.py

```python

# Q3. Write a program to encrypt and decrypt the text using vigenere

def vigenere_encrypt(plain_text, key):
    cipher_text = []
    key_length = len(key)

    # Extend the key to match the length of the plain text
    key = (key * (len(plain_text) // key_length)) + key[:len(plain_text) % key_length]

    for i in range(len(plain_text)):
        if plain_text[i].isalpha():
            shift = ord(key[i].upper()) - ord('A')
            encrypted_char = chr((ord(plain_text[i].upper()) - ord('A') + shift) % 26 + ord('A'))
            cipher_text.append(encrypted_char)
        else:
            cipher_text.append(plain_text[i])  # Non-alphabetic characters remain unchanged

    return ''.join(cipher_text)

def vigenere_decrypt(cipher_text, key):
    plain_text = []
    key_length = len(key)

    # Extend the key to match the length of the cipher text
    key = (key * (len(cipher_text) // key_length)) + key[:len(cipher_text) % key_length]

    for i in range(len(cipher_text)):
        if cipher_text[i].isalpha():
            shift = ord(key[i].upper()) - ord('A')
            decrypted_char = chr((ord(cipher_text[i].upper()) - ord('A') - shift + 26) % 26 + ord('A'))
            plain_text.append(decrypted_char)
        else:
            plain_text.append(cipher_text[i])  # Non-alphabetic characters remain unchanged

    return ''.join(plain_text)

# Example usage for Vigenère Cipher
plain_text_vigenere = input("Enter the plain text for Vigenère Cipher: ").replace(" ", "")
key_vigenere = input("Enter the key for Vigenère Cipher: ")

# Encrypt the plain text
encrypted_text_vigenere = vigenere_encrypt(plain_text_vigenere, key_vigenere)
```

```
print(f"Vigenère Encrypted Text: {encrypted_text_vigenere}")

# Decrypt the cipher text
decrypted_text_vigenere = vigenere_decrypt(encrypted_text_vigenere, key_vigenere)
print(f"Vigenère Decrypted Text: {decrypted_text_vigenere}")

```

4. Write a program to encrypt and decrypt the text using vernam cipher.

Encryption:

1. **Input:**
    - **plain_text: The plaintext message to be encrypted (assumed to be uppercase and without spaces).**
    - **key: The encryption key (uppercase, with the same length as the plain_text).**
2. **Check Key Length:**
    - **Ensure the length of the key matches the length of the plain_text.**
3. **Encryption Process:**
    - **For each character in the plain_text:**
        - **Convert both the plain_text and key characters to their corresponding numerical positions (0 for 'A', 1 for 'B', ..., 25 for 'Z').**
        - **Apply the bitwise XOR operation (^) between the numerical values of the plain_text and key characters.**
        - **Convert the XOR result back to an uppercase character using: encrypted_char=(XOR result)%26\text{encrypted_char} = \text{(XOR result)} \% 26encrypted_char=(XOR result)%26**
        - **Append the encrypted character to the encrypted_text.**
4. **Output:**
    - **Return the encrypted_text containing the ciphertext.**

Decryption:

1. **Input:**
    - **encrypted_text: The ciphertext to be decrypted (uppercase and same length as key).**
    - **key: The same key used for encryption.**
2. **Decryption Process:**
    - **For each character in the encrypted_text:**
        - **Convert both the encrypted_text and key characters to their corresponding numerical positions.**
        - **Apply the bitwise XOR operation (^) between the numerical values of the encrypted_text and key characters.**

- **Convert the XOR result back to an uppercase character using: decrypted_char=(XOR result)%26\text{decrypted_char} = \text{(XOR result)} \% 26decrypted_char=(XOR result)%26**
- **Append the decrypted character to the decrypted_text.**

3. **Output:**
    - **Return the decrypted_text containing the original plaintext.**

LAB_7\vernam_cipher.py

```

# Q3. Write a program to encrypt and decrypt the text using vernam cipher.

def vernam_encrypt(plain_text, key):
    # Encryption
    encrypted_text = ''
    for i in range(len(plain_text)):
        x1 = ord(plain_text[i]) - ord('A')
        x2 = ord(key[i]) - ord('A')
        x = x1 ^ x2
        x = x % 26
        encrypted_text += chr(x + ord('A'))

    return encrypted_text

def vernam_decrypt(encrypted_text, key):
    # Decryption
    decrypted_text = ''
    for i in range(len(encrypted_text)):
        x1 = ord(encrypted_text[i]) - ord('A')
        x2 = ord(key[i]) - ord('A')
        x = x1 ^ x2
        x = x % 26
        decrypted_text += chr(x + ord('A'))

    return decrypted_text

# Example usage
plain_text = input("Enter the plain text (uppercase, no spaces): ").upper()
key = input("Enter the key (uppercase, same length as plain text): ").upper()

if len(plain_text) != len(key):
    print("Key must be the same length as the plain text.")
else:
    # Encrypt the plain text
    encrypted_text = vernam_encrypt(plain_text, key)
    print(f"Encrypted Text: {encrypted_text}")

    # Decrypt the cipher text
    decrypted_text = vernam_decrypt(encrypted_text, key)
    print(f"Decrypted Text: {decrypted_text}")

```