**Experiment-3**

**Aim:** Study and implement a program for Transposition (Columnar) Cipher to encrypt and decrypt the message.

**Introduction:**

Columnar transposition is a cryptographic technique used to encrypt plaintext by rearranging the letters or characters in a specific way based on a keyword or key phrase. It falls under the category of classical or historical encryption methods and relies on a simple permutation operation.

Here is how the columnar transposition encryption process works:

1. Key Selection: Choose a keyword or key phrase that will be used for encryption. This keyword determines the order in which columns are written during the transposition process.

2. Creating the Transposition Grid: Write the keyword horizontally at the top of a grid. Each letter in the keyword becomes the label for a column. Arrange the letters of the keyword in alphabetical order without repeating any letters. Fill in the remaining cells of the grid with the rest of the alphabet letters in order.

3. Encryption: Write the plaintext message row by row into the grid, filling the cells in order. If a row is not filled, pad it with dummy characters.

4. Reading the Cipher: Read the ciphertext column by column from the transposition grid, starting with the columns corresponding to the keyword letters in alphabetical order. This creates the encrypted message.

Since the arrangement of the columns is determined by the keyword, changing the keyword will result in a different permutation of the columns, and thus a different ciphertext.

**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

string encrypt(string plainText, string keyword){

    string EncryptedText;

    double columns = keyword.length();

    double rows = ceil(plainText.length() / columns);

    char matrix[int(rows)][int(columns)];

    int stringIndexPointer = 0;

    //inserting characters into the table(row-wise)

    for (int i=0;i<rows;i++){

        for (int j=0;j<columns;j++){

            if (stringIndexPointer < plainText.length()){

                matrix[i][j] = plainText[stringIndexPointer];

                stringIndexPointer++;

            }

            //when plaingText traversal is over but table is still left to fill

            //fill it with '\_'

            else{

                matrix[i][j] = '\_';

            }

        }

    }

    // //for printing the table

    // for (int i=0;i<rows;i++){

    //     for (int j=0;j<columns;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //finding the index of highest ASCII value character in the keyword (used later)

    int highestASCIIValueindex = 0;

    for(int i=0;i<keyword.length();i++){

        if (keyword[i] >= keyword[highestASCIIValueindex]){

            highestASCIIValueindex = i;

        }

    }

    //for deciding order of reading the columns//

    //order will be stored in the below array

    int orderArr[keyword.length()];

    //for keeping track whether character is traversed or not

    int hashArr[keyword.length()] = {0};

    //for inserting into the orderArr

    int pointerForInsert = 0;

    //finding smallest ascii value in the keyword and storing its index in the orderArr each time.

    for (int i=0;i<keyword.length();i++){

        int smallestASCIIValueIndex = highestASCIIValueindex;

        for (int j=0;j<keyword.length();j++){

            if (keyword[j] <= keyword[smallestASCIIValueIndex] && hashArr[j] == 0){

                smallestASCIIValueIndex = j;

            }

        }

        orderArr[pointerForInsert++] = smallestASCIIValueIndex;

        hashArr[smallestASCIIValueIndex] = 1;

    }

    //forming the decryptedText

    for (int i=0;i<keyword.length();i++){

        int columnNumber = orderArr[i];

        for (int j=0;j<rows;j++){

            if(matrix[j][columnNumber] != '\_'){

                EncryptedText += matrix[j][columnNumber];

            }

        }

    }

    return EncryptedText;

}

string decrypt(string encryptedText, string keyword){

    string decryptedText;

    double columns = keyword.length();

    double rows = ceil(encryptedText.length() / columns);

    char matrix[int(rows)][int(columns)];

    //inserting characters into the table (column-wise)

    int stringIndexPointer = 0;

    for (int i=0;i<columns;i++){

        for (int j=0;j<rows;j++){

            if (stringIndexPointer < encryptedText.length()){

                matrix[j][i] = encryptedText[stringIndexPointer];

                stringIndexPointer++;

            }

            //when plaingText traversal is over but table is still left to fill

            //fill it with '\_'

            else{

                matrix[j][i] = '\_';

            }

        }

    }

    // //for printing the table

    // for (int i=0;i<rows;i++){

    //     for (int j=0;j<columns;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //finding the index of highest ASCII value character in the keyword (used later)

    int highestASCIIValueindex = 0;

    for(int i=0;i<keyword.length();i++){

        if (keyword[i] >= keyword[highestASCIIValueindex]){

            highestASCIIValueindex = i;

        }

    }

    //for deciding order of reading the columns//

    //order will be stored in the below array

    int orderArr[keyword.length()];

    //for keeping track whether character is traversed or not

    int hashArr[keyword.length()] = {0};

    //for inserting into the orderArr

    int pointerForInsert = 0;

    //finding smallest ascii value in the keyword and storing its index in the orderArr each time.

    for (int i=0;i<keyword.length();i++){

        int smallestASCIIValueIndex = highestASCIIValueindex;

        for (int j=0;j<keyword.length();j++){

            if (keyword[j] <= keyword[smallestASCIIValueIndex] && hashArr[j] == 0){

                smallestASCIIValueIndex = j;

            }

        }

        orderArr[pointerForInsert++] = smallestASCIIValueIndex;

        hashArr[smallestASCIIValueIndex] = 1;

    }

    //but since reading order (for decryption) will be different, actual order will be different

    //finding actual order

    int actualOrder[keyword.length()];

    int pointer = 0;

    for(int i=0;i<keyword.length();i++){

        //finding i in orderArr, and appending its index to actualOrder

        for(int j=0;j<keyword.length();j++){

            if (i == orderArr[j]){

                // j will give the actual order

                actualOrder[pointer++] = j;

            }

        }

    }

    //forming the decryptedText

    for (int i=0;i<rows;i++){

        for (int j=0;j<keyword.length();j++){

            int columnNumber = actualOrder[j];

            if (matrix[i][columnNumber] != '\_'){

                decryptedText += matrix[i][columnNumber];

            }

        }

    }

    return decryptedText;

}

int main(){

string plainText = "My name is Ravi";

string keyword = "PDEU";

    cout<<"Plain Text: "<<plainText;

    cout<<"\nKeyword: "<<keyword;

    string encryptedText = encrypt(plainText, keyword);

    cout<<"\nEncrypted Text: "<<encryptedText;

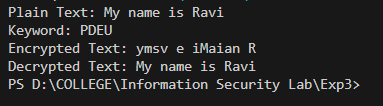
    string decryptedText = decrypt(encryptedText, keyword);

    cout<<"\nDecrypted Text: "<<decryptedText;

    return 0;

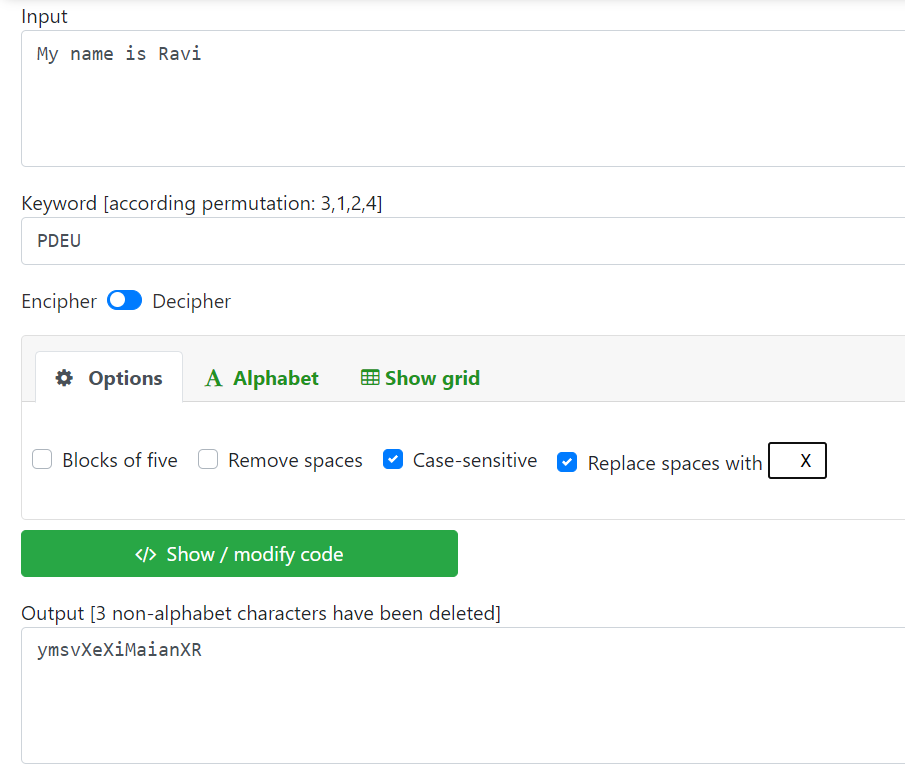
}

**Output (Program):**

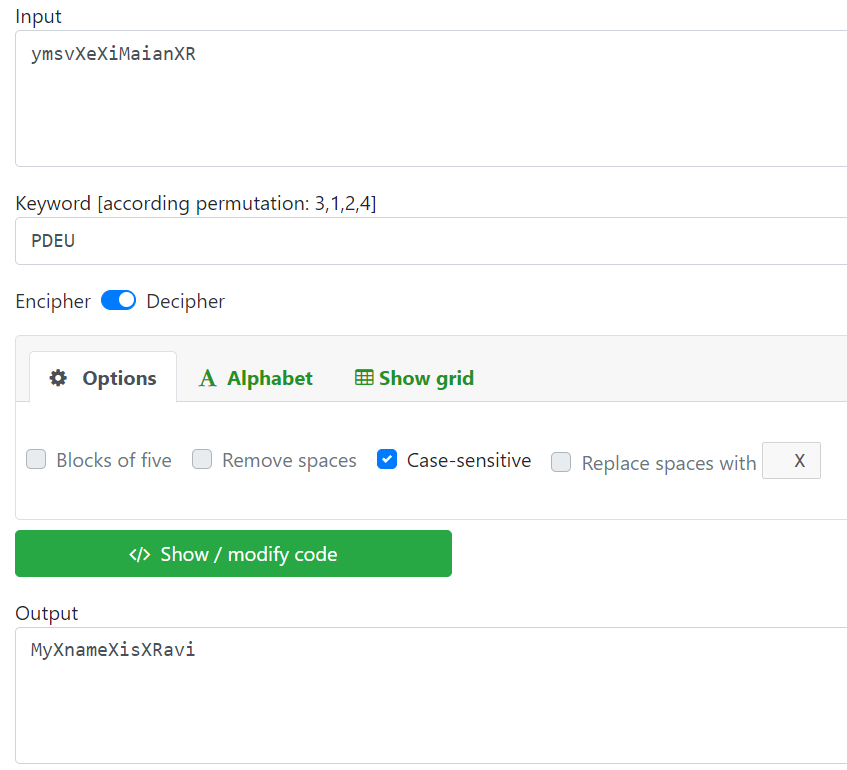
****

**Output (Cryptool):**

**Encryption:**

****

**Decryption:**



**Cryptanalysis:**

**1. Brute Force Attack:** This can be done if we know the length of the keyword. This involves trying all possible column orders based on the key length. Since the key is a permutation of column numbers, the number of possible keys can grow rapidly with the number of columns. However, for small key lengths, this attack can be feasible.

**2. Dictionary Attack:** The columnar transposition cipher is almost always keyed with a word or short phrase, so we may not need to test all possible transposition keys, we may only need to test common words. This involves having a large list of dictionary words including place names, famous people, mythological names, historical names etc. From this we generate a text file of possible keys. This method can work if the keyword was one of the words that we included in the dictionary, but it fails to work if the keyword is the mixture of more than one word.

**3. Hill Climbing:** We first assume the key length is 10, then choose a random starting keyword of this length. This is called the parent key. Child keys are generated by making random swaps in the parent keyword, and if any of the swaps lead to an increased fitness, we replace the parent with the child that beat it. If after many tries the correct key is not found, it is time to increment the key length to 11 and rerun everything.

**Applications:**

Columnar transposition, despite its lack of robust security in modern contexts, can still find applications in certain scenarios or as part of more complex encryption methods. Here are a few potential applications:

1. **Educational Purposes**: Columnar transposition is often used to teach the basics of encryption and decryption. It helps students understand fundamental concepts like permutation and the importance of encryption keys.
2. **Puzzle and Games**: Columnar transposition can be used as a basis for creating puzzles or games that involve solving simple cryptographic challenges. These can be entertaining and educational for people interested in cryptography or logic puzzles.
3. **Historical Reenactments**: Since columnar transposition is a classical encryption method, it can be used in historical reenactments, escape room activities, or events that aim to showcase historical methods of encryption and codebreaking.
4. **Obfuscation and Basic Protection**: While not suitable for serious security applications, columnar transposition can still be used for basic obfuscation of text. For example, it can make casual reading of a message more challenging for unintended recipients who are not familiar with the method.
5. **Part of Hybrid Encryption**: In some cases, columnar transposition could be used as one step in a multi-layered encryption process. By combining it with other encryption techniques, it might contribute to increasing the overall complexity of encryption.
6. **Code Words and Simple Encodings**: In non-critical applications, columnar transposition could be used to create code words or encodings for fun or artistic purposes. It can add a layer of mystique to messages without requiring highly secure encryption.
7. **Text Transformations**: Columnar transposition can be used as a text transformation method in data processing or manipulation. For example, it might be used to shuffle the order of words in a sentence for creative writing purposes.

It is worth noting that for securing sensitive information, modern encryption methods like AES (Advanced Encryption Standard) are highly recommended due to their well-established security properties and resistance to various attacks. Columnar transposition is considered relatively weak against sophisticated cryptanalysis techniques, so caution is necessary when using it for any security-related tasks.

**References:**

1. W3Schools
2. GeeksforGeeks
3. www.practicalcryptography.com