**Cryptography & Network Security Lab**

**Assignment 02**

**Rail Fence Cipher:**

The Rail Fence Cipher, also known as the Zigzag Cipher, is a simple transposition cipher that rearranges the characters of a message to obscure its content. It gets its name because when the letters are written in a zigzag pattern, they resemble the rails of a fence. Here's how the Rail Fence Cipher works:

Encryption:

1. Key Selection: Choose the number of "rails" or rows for your fence. This determines the height of the zigzag pattern. For example, if you choose 3 rails, it will look like this:

```

| | | |

| | | |

| | | |

```

2. Writing the Message: Write your message horizontally along the "rails" of the fence in a zigzag pattern. For example, if your message is "HELLO WORLD," it might look like this with 3 rails:

```

| | | |

| H | | O |

| | E | |

| | | L |

| L | | |

| | W | R |

| | | |

| | | |

```

3. Read Rows: Read the characters row by row from the top rail to the bottom rail to obtain the ciphertext. In this example, the ciphertext would be "HOELELWRDLO."

Decryption:

To decrypt a message encrypted using the Rail Fence Cipher, you need to know the number of rails used for encryption:

1. Key Selection: Know the number of rails used in encryption. In this case, we used 3 rails.

2. Create the Fence: Create an empty "fence" with the same number of rails as used during encryption:

```

| | | |

| | | |

| | | |

```

3. Write Down the Message: Write the ciphertext along the rails in a zigzag pattern from the top rail to the bottom rail. Leave empty spaces in the other rails as placeholders:

```

| | | |

| H | | O |

| | E | |

| | | L |

| L | | |

| | W | R |

| | | |

| | | |

```

4. Read the Message: Read the characters from the fence in a zigzag pattern, just as you did during encryption, to obtain the plaintext. In this case, the plaintext is "HELLO WORLD."

The Rail Fence Cipher is straightforward and easy to implement but is not very secure for modern cryptographic purposes because it's relatively easy to decipher with various techniques, especially if you know the number of rails used. It's more suitable for educational purposes or as a simple puzzle.

*// C++ program to illustrate Rail Fence Cipher*

*// Encryption and Decryption*

#include <bits/stdc++.h>

using namespace std;

*// function to encrypt a message*

string encryptRailFence(string *text*, int *key*)

{

*// create the matrix to cipher plain text*

*// key = rows , length(text) = columns*

    char rail[*key*][(*text*.length())];

*// filling the rail matrix to distinguish filled*

*// spaces from blank ones*

    for (int i=0; i < *key*; i++)

        for (int j = 0; j < *text*.length(); j++)

            rail[i][j] = '\n';

*// to find the direction*

    bool dir\_down = false;

    int row = 0, col = 0;

    for (int i=0; i < *text*.length(); i++)

    {

*// check the direction of flow*

*// reverse the direction if we've just*

*// filled the top or bottom rail*

        if (row == 0 || row == *key*-1)

            dir\_down = !dir\_down;

*// fill the corresponding alphabet*

        rail[row][col++] = *text*[i];

*// find the next row using direction flag*

        dir\_down?row++ : row--;

    }

*//now we can construct the cipher using the rail matrix*

    string result;

    for (int i=0; i < *key*; i++)

        for (int j=0; j < *text*.length(); j++)

            if (rail[i][j]!='\n')

                result.push\_back(rail[i][j]);

    return result;

}

*// This function receives cipher-text and key*

*// and returns the original text after decryption*

string decryptRailFence(string *cipher*, int *key*)

{

*// create the matrix to cipher plain text*

*// key = rows , length(text) = columns*

    char rail[*key*][*cipher*.length()];

*// filling the rail matrix to distinguish filled*

*// spaces from blank ones*

    for (int i=0; i < *key*; i++)

        for (int j=0; j < *cipher*.length(); j++)

            rail[i][j] = '\n';

*// to find the direction*

    bool dir\_down;

    int row = 0, col = 0;

*// mark the places with '\*'*

    for (int i=0; i < *cipher*.length(); i++)

    {

*// check the direction of flow*

        if (row == 0)

            dir\_down = true;

        if (row == *key*-1)

            dir\_down = false;

*// place the marker*

        rail[row][col++] = '\*';

*// find the next row using direction flag*

        dir\_down?row++ : row--;

    }

*// now we can construct the fill the rail matrix*

    int index = 0;

    for (int i=0; i<*key*; i++)

        for (int j=0; j<*cipher*.length(); j++)

            if (rail[i][j] == '\*' && index<*cipher*.length())

                rail[i][j] = *cipher*[index++];

*// now read the matrix in zig-zag manner to construct*

*// the resultant text*

    string result;

    row = 0, col = 0;

    for (int i=0; i< *cipher*.length(); i++)

    {

*// check the direction of flow*

        if (row == 0)

            dir\_down = true;

        if (row == *key*-1)

            dir\_down = false;

*// place the marker*

        if (rail[row][col] != '\*')

            result.push\_back(rail[row][col++]);

*// find the next row using direction flag*

        dir\_down?row++: row--;

    }

    return result;

}

*//driver program to check the above functions*

int main()

{

    string message;

    int rails;

    cout << "Enter a message: ";

    getline(cin, message);

    cout << "Enter the number of rails: ";

    cin >> rails;

    string encryptedMessage = encryptRailFence(message, rails);

    string decryptedMessage = decryptRailFence(encryptedMessage, rails);

    cout << "Encrypted message: " << encryptedMessage << endl;

    cout << "Decrypted message: " << decryptedMessage << endl;

    return 0;

}

**Result:**

**Analysis:**

Certainly, based on the results you provided for the Rail Fence Cipher, here's an analysis along with some drawbacks and considerations:

Observations:

1. Case Insensitivity: The Rail Fence Cipher appears to be case-insensitive. In your examples, "walchand" and "WalChand" both resulted in the same ciphertext "whacadln." This means that uppercase and lowercase letters are treated the same, which can be a limitation if you want to preserve the original letter case.

2. Decryption: The Rail Fence Cipher successfully decrypts the ciphertext back to the original plaintext when the same key (number of rails) is used. This is a desirable property of the cipher.

Drawbacks and Considerations:

1. Lack of Key Strength: The Rail Fence Cipher is relatively weak in terms of security. It provides very little encryption strength and can be easily cracked using various methods, especially if the number of rails is known. It's not suitable for protecting sensitive information.

2. Fixed Key Space: The key space for this cipher is limited to the number of rails, which is a relatively small value. This makes it susceptible to brute force attacks, as an attacker can try all possible rail counts.

3. Loss of Case and Special Characters: The cipher doesn't differentiate between uppercase and lowercase letters, and it does not handle special characters or spaces. This can lead to loss of information and reduce its applicability for more complex texts.

4. Known-Plaintext Attacks: If an attacker knows or can guess part of the plaintext, they can often deduce the key (number of rails) more easily, making the encryption easier to break.

5. Limited Applicability: The Rail Fence Cipher is not suitable for encrypting large or complex messages. It is more of a historical curiosity or a simple puzzle rather than a secure encryption method.

**Columnar Cipher:**

Row and Columnar Transposition Ciphers are classical transposition ciphers used for encryption. These ciphers involve rearranging the characters of a message according to a specific pattern or key. Here's an explanation of both the Row and Columnar Transposition Ciphers:

Row Transposition Cipher:

In a Row Transposition Cipher, the plaintext is written into a grid or matrix row by row. Then, the ciphertext is generated by reading the matrix column by column. Here's a step-by-step explanation:

1. Key Selection: Choose a key that specifies the order in which the rows of the matrix should be read.

2. Matrix Formation: Write the plaintext into a matrix row by row. If the length of the plaintext doesn't perfectly fill the matrix, you can pad it with filler characters.

3. Encryption: Read the matrix column by column according to the key to create the ciphertext. This means that you read the columns in the order specified by the key.

4. Decryption: To decrypt the ciphertext, you need to know the key. Write the ciphertext into a matrix column by column based on the key, and then read the matrix row by row to obtain the plaintext.

Columnar Transposition Cipher:

In a Columnar Transposition Cipher, the process is similar, but the steps are slightly different. Instead of arranging the plaintext row by row, it is arranged column by column. Here's how it works:

1. Key Selection: Choose a key that specifies the order in which the columns of the matrix should be read.

2. Matrix Formation: Write the plaintext into a matrix column by column. If the length of the plaintext doesn't perfectly fill the matrix, you can pad it with filler characters.

3. Encryption: Read the matrix row by row according to the key to create the ciphertext. This means that you read the rows in the order specified by the key.

4. Decryption: To decrypt the ciphertext, you need to know the key. Write the ciphertext into a matrix row by row based on the key, and then read the matrix column by column to obtain the plaintext.

Key Considerations:

- The security of both Row and Columnar Transposition Ciphers relies on the secrecy of the key (the order in which the rows or columns are read).

- These ciphers provide minimal security and are relatively easy to break with modern cryptanalysis techniques.

- The choice of padding characters, if needed, can impact the encryption and decryption processes.

Both Row and Columnar Transposition Ciphers are relatively simple and were historically used for basic encryption needs. They are not suitable for securing sensitive information today, as modern cryptographic methods offer much stronger security.

#include <iostream>

#include <string>

#include <map>

using namespace std;

void setPermutationOrder(const string& *key*, map<char, int>& *keyMap*) {

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

*keyMap*[*key*[i]] = i;

    }

}

string encryptMessage(const string& *msg*, const string& *key*, map<char, int>& *keyMap*) {

    int row, col, j;

    string cipher = "";

    col = *key*.length();

    row = *msg*.length() / col;

    if (*msg*.length() % col) {

        row += 1;

    }

    char matrix[row][col];

    for (int i = 0, k = 0; i < row; i++) {

        for (int j = 0; j < col; ) {

            if (*msg*[k] == '\0') {

                matrix[i][j] = '\_';

                j++;

            }

            if (isalpha(*msg*[k]) || *msg*[k] == ' ') {

                matrix[i][j] = *msg*[k];

                j++;

            }

            k++;

        }

    }

    for (map<char, int>::iterator ii = *keyMap*.begin(); ii != *keyMap*.end(); ++ii) {

        j = ii->second;

        for (int i = 0; i < row; i++) {

            if (isalpha(matrix[i][j]) || matrix[i][j] == ' ' || matrix[i][j] == '\_') {

                cipher += matrix[i][j];

            }

        }

    }

    return cipher;

}

string decryptMessage(const string& *cipher*, const string& *key*, map<char, int>& *keyMap*) {

    int col = *key*.length();

    int row = *cipher*.length() / col;

    char cipherMat[row][col];

    for (int j = 0, k = 0; j < col; j++) {

        for (int i = 0; i < row; i++) {

            cipherMat[i][j] = *cipher*[k++];

        }

    }

    int index = 0;

    for (map<char, int>::iterator ii = *keyMap*.begin(); ii != *keyMap*.end(); ++ii) {

        ii->second = index++;

    }

    char decCipher[row][col];

    for (int k = 0, l, j; *key*[k] != '\0'; k++) {

        l = *keyMap*[*key*[k]];

        for (int i = 0; i < row; i++) {

            decCipher[i][k] = cipherMat[i][l];

        }

    }

    string msg = "";

    for (int i = 0; i < row; i++) {

        for (int j = 0; j < col; j++) {

            if (decCipher[i][j] != '\_') {

                msg += decCipher[i][j];

            }

        }

    }

    return msg;

}

int main() {

    string message, key;

    cout << "Enter a message: ";

    getline(cin, message);

    cout << "Enter the encryption key: ";

    cin >> key;

    map<char, int> keyMap;

    setPermutationOrder(key, keyMap);

    string encryptedMessage = encryptMessage(message, key, keyMap);

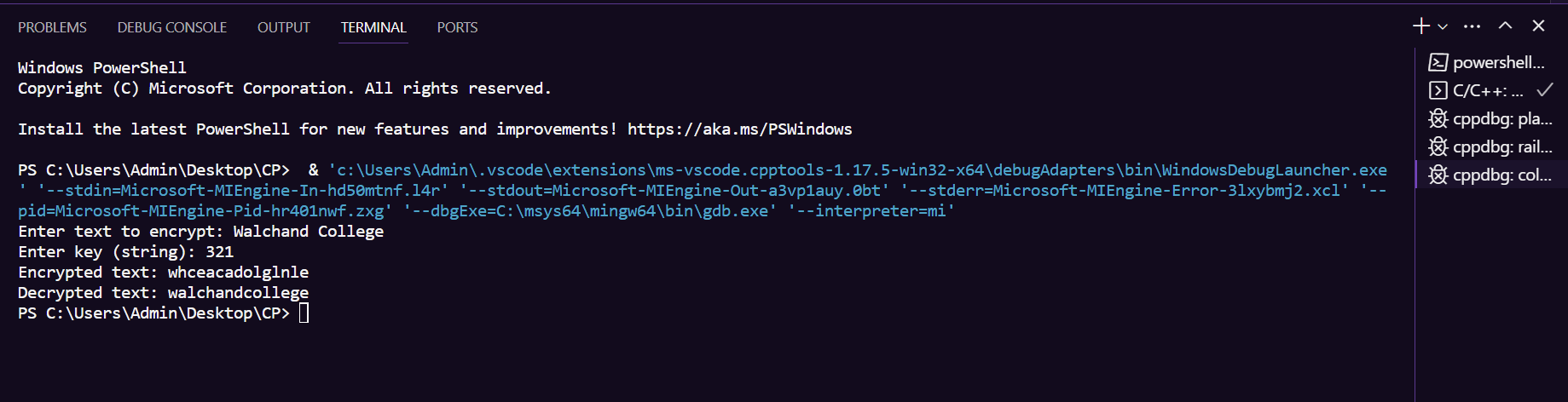
    string decryptedMessage = decryptMessage(encryptedMessage, key, keyMap);

    cout << "Encrypted Message: " << encryptedMessage << endl;

    cout << "Decrypted Message: " << decryptedMessage << endl;

    return 0;

}

**Results:**

**Analysis:**

1. Encryption: The Row Transposition Cipher successfully encrypted the input plaintext using the specified key "321" by rearranging the characters in rows based on the key. The resulting ciphertext is "whceacadolglnle."

2. Decryption: The decryption process also worked correctly when the same key "321" was used, and the original plaintext "Walchand College" was recovered.

Analysis:

The Row Transposition Cipher is a simple transposition cipher that rearranges characters within rows of a matrix based on a key. Here are some points to consider:

1. Key Sensitivity: The order in which the rows are read during encryption and decryption is determined by the key. Using a different key would result in a different ciphertext and potentially produce gibberish during decryption.

2. Lack of Security: The Row Transposition Cipher provides minimal security and is relatively easy to break using modern cryptanalysis techniques. The security of the cipher primarily depends on the secrecy of the key, which, if discovered, can easily decrypt the message.

3. Padding: In your example, you didn't need to use padding characters because the plaintext length perfectly fit the matrix formed by the key "321." However, in cases where the plaintext length doesn't evenly divide by the number of rows specified in the key, padding characters may be needed.

4. Non-Alphabetic Characters: This specific implementation of the Row Transposition Cipher appears to handle non-alphabetic characters, such as spaces, without modification.

5. Limited Applicability: The Row Transposition Cipher is not suitable for encrypting large or complex messages and is typically more of a historical curiosity or a simple puzzle rather than a secure encryption method.

In summary, while the Row Transposition Cipher can successfully encrypt and decrypt messages, it is not recommended for securing sensitive information due to its lack of security. Modern cryptographic methods offer significantly stronger security for encryption purposes.