Lab 1

Lab 1.1: Write a program that takes an integer value K (i.e. shift value between +/- 26) and a plaintext message and returns the corresponding Ceasar cipher. The program should also implement a decryption routine that reconstructs the original plaintext from the ciphertext.

Algorithm:

STEP-1: Read the plain text from the user.

STEP-2: Read the key value from the user.

STEP-3: If the key is positive then encrypt the text by adding the key with each character in the plain text.

STEP-4: Else subtract the key from the plain text.

STEP-5: Display the cipher text obtained above.

Source Code:

```
#include <iostream>
#include <string>
using namespace std;
string caesarEncrypt(int k, const string& plaintext)
{
   string ciphertext = "";
   for (char ch : plaintext)
         {
      if (isalpha(ch))
                  {
         char base = isupper(ch) ? 'A' : 'a';
         ciphertext += static_cast<char>((ch - base + k + 26) % 26 + base);
      }
                  else
                  {
         ciphertext += ch;
      }
   }
   return ciphertext;
```

```
}
string caesarDecrypt(int k, const string& ciphertext) {
    return caesarEncrypt(-k, ciphertext);
}
int main() {
    int shiftValue;
    string message;
    cout << "Enter shift value (1 - 26): ";
    cin >> shiftValue;
    if (shiftValue < -26 || shiftValue > 26) {
        cout << "Invalid shift value. Enter a value between 1 to 26." << endl;
        return 1;
    }
    cin.ignore();
    cout << "Enter plaintext: ";
    getline(cin, message);
    string encryptedMessage = caesarEncrypt(shiftValue, message);
    cout << "Cipher text: " << encryptedMessage << endl;
    string decryptedMessage = caesarDecrypt(shiftValue, encryptedMessage);
    cout << "Decrypted plaintext: " << decryptedMessage << endl;
    return 0;
}
```

Output:

```
Enter shift value (1 - 26): 4
Enter plaintext: kafle
Cipher text: oejpi
Decrypted plaintext: kafle
```

Lab 1.2: Write a program that asks user for key and plain text and displays the corresponding Vigenere cipher

Algorithm:

STEP-1: Arrange the alphabets in row and column of a 26*26 matrix.

STEP-2: Circulate the alphabets in each row to position left such that the first letter is attached to last.

STEP-3: Repeat this process for all 26 rows and construct the final key matrix.

STEP-4: The keyword and the plain text is read from the user.

STEP-5: The characters in the keyword are repeated sequentially so as to match with that of the plain text.

STEP-6: Pick the first letter of the plain text and that of the keyword as the row indices and column indices respectively.

STEP-7: The junction character where these two meet forms the cipher character.

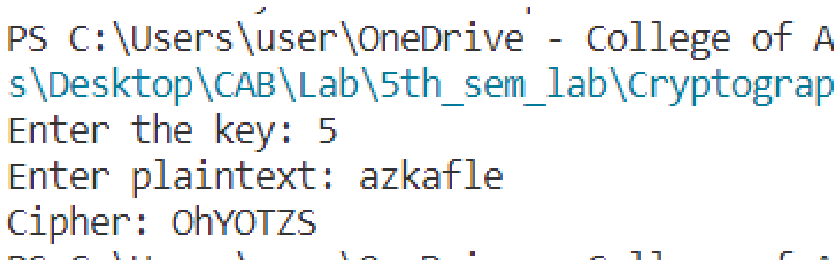
STEP-8: Repeat the above steps to generate the entire cipher text.

Source Code:

```
#include <iostream>
#include <string>
using namespace std;
string vigenereEncrypt(const string& key, const string& plaintext) {
    string ciphertext = "";
    int keyLength = key.length();
    int index = 0;
    for (char ch : plaintext) {
        if (isalpha(ch)) {
            char base = isupper(ch) ? 'A' : 'a';
            char shift = key[index % keyLength];
            shift = isupper(shift) ? shift - 'A' : shift - 'a';
            ciphertext += static_cast<char>((ch - base + shift + 26) % 26 + base);
            index++;
        } else {
            ciphertext += ch;
        }
```

```
    }
    return ciphertext;
}
int main() {
    string key, plaintext;
    cout << "Enter the key: ";
    cin >> key;
    cout << "Enter plaintext: ";
    cin.ignore();
    getline(cin, plaintext);
    string encryptedMessage = vigenereEncrypt(key, plaintext);
    cout << "Cipher: " << encryptedMessage << endl;
    return 0;
}
```

Output:



Lab 1.3: Using the Rail Fence algorithm with depth 3, write a program to encrypt the message “I love my college”.

<u>Algorithm:</u>

STEP-1: Read the Plain text.

STEP-2: Arrange the plain text in row columnar matrix format.

STEP-3: Now read the keyword depending on the number of columns of the plain text.

STEP-4: Arrange the characters of the keyword in sorted order and the corresponding columns of the plain text.

STEP-5: Read the characters row wise or column wise in the former order to get the cipher text

<u>Source Code:</u>

```
#include <stdio.h>
#include <string.h>
void railFenceEncrypt(char* message, int depth) {
    int len = strlen(message);
    char fence[depth][len];
    char encryptedMessage[len];
    int row, col;
    int direction = 1;
    for (int i = 0; i < depth; i++) {
        for (int j = 0; j < len; j++) {
            fence[i][j] = '\0';
        }
    }
    row = 0;
    col = 0;
    for (int i = 0; i < len; i++) {
        fence[row][col] = message[i];
        if (row == 0) {
            direction = 1;
        }
        else if (row == depth - 1) {
```

```
            direction = -1;
        }
        row += direction;
        col++;
    }
    int index = 0;
    for (int i = 0; i < depth; i++) {
        for (int j = 0; j < len; j++) {
            if (fence[i][j] != '\0') {
                encryptedMessage[index++] = fence[i][j];
            }
        }
    }
    encryptedMessage[len] = '\0';
    printf("Encrypted message: %s\n", encryptedMessage);
}
int main() {
    char message[] = "I love my college";
    int depth = 3;
    railFenceEncrypt(message, depth);
    return 0;
}
```

Output:

```
s\Desktop\CAB\Lab\5th_sem_lab\Cryptography\Lab_
Encrypted message: Ivyle oem olgl ce
```

Lab 1.4: Write a program to demonstrate the calculation of initial permutation of a plain text in DES algorithm.

Algorithm:

STEP 1: First provide 64-bit plain text as input.

STEP 2: Then initialize initial permutation (IP) table.

- Create a lookup table (IP table) with 64 elwements, where each element holds a unique integer value from 1 to 64, representing the new position of a corresponding bit in the permuted output.
- The specific values in the IP table are fixed as defined in the DES specification.

STEP 3: Iterate through each bit of the input plaintext block:

- Retrieve the current bit's position in the block (index).
- Use the IP table to lookup the new position for this bit.
- Place the bit at the corresponding new position in a new 64-bit output block.

STEP 4: Return the 64-bit output block as the permuted plaintext.

Source Code:

```
#include <stdio.h>
int initial_permutation_table[] = {
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
};
void initial_permutation(char* plain_text, char* initial_permuted_text) {
    for (int i = 0; i < 64; i++) {
        int bit_position = initial_permutation_table[i] - 1;
        int byte_position = bit_position / 8;
        int bit_offset = 7 - (bit_position % 8);
```

```
        char bit = (plain_text[byte_position] >> bit_offset) & 1;
        initial_permuted_text[i / 8] |= (bit << (7 - (i % 8)));
    }
}
int main() {
    char plain_text[] = "kafleaz9098";
    char initial_permuted_text[8] = {0};  // Initialize with zeros
    initial_permutation(plain_text, initial_permuted_text);
    printf("Plaintext: %s\n", plain_text);
    printf("Initial Permuted Text: ");
    for (int i = 0; i < 8; i++) {
        printf("%02X ", (unsigned char)initial_permuted_text[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
Plaintext: kafleaz9098
Initial Permuted Text: 7F C0 1C B3 00 FF C9 45
```