****

FACULTY OF ENGINEERING AND TECHNOLOGY

BACHELOR OF TECHNOLOGY

**INFORMATION AND**

**NETWORK SECURITY**

**(203105311)**

7th SEMESTER

7A13

**NAME**: **VISHAL SURESH UMAVANE**

**ENROLLMENT NUMBER**: **210306105192**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SR No.** | **Practical List** | **Start Date** | **End Date** | **Sign** | **Marks** |
| 1 | Implement Caesar cipher encryption-decryption |  |  |  |  |
| 2 | Implement Monoalphabetic cipher encryption-decryption |  |  |  |  |
| 3 | Implement Playfair cipher encryption-decryption |  |  |  |  |
| 4 | Implement Polyalphabetic cipher encryption-decryption |  |  |  |  |
| 5 | Implement Hill cipher encryption-decryption |  |  |  |  |
| 6 | Implement Simple Transposition encryption-decryption |  |  |  |  |
| 7 | Implement One time pad encryption-decryption |  |  |  |  |
| 8 | Implement Diffi-Hellmen Key exchange Method |  |  |  |  |
| 9 | Implement RSA encryption-decryption algorithm |  |  |  |  |
| 10 | Demonstrate working of Digital Signature using Cryptool |  |  |  |  |

**Index Table**

**Practical 1**

**Aim :** Implement Caesar cipher encryption-decryption

**Theory** : The Caesar cipher is a basic form of encryption where each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. It's a type of substitution cipher, named after Julius Caesar who reportedly used it to protect his military messages.

**Encryption Process**

1. Choose a shift value: This is the number of positions each letter will be shifted
2. Shift each letter: For each letter in the plaintext, replace it with the letter that is the shift
3. value positions down the alphabet. Wrap around from 'Z' to 'A' if necessary.

**Decryption Process**

1. Determine the shift value: You need to know the shift value used for encryption.
2. Shift each letter back: For each letter in the ciphertext, shift it back by the same number of positions used for encryption.

**Implementation :**

#include <iostream>

#include <string>

// Function to encrypt the text using Caesar Cipher

std::string encryptCaesarCipher(std::string text, int shift) {

std::string result = "";

// Traverse text

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

char ch = text[i];

// Encrypt uppercase letters

if (isupper(ch))

result += char(int(ch + shift - 65) % 26 + 65);

// Encrypt lowercase letters

else if (islower(ch))

result += char(int(ch + shift - 97) % 26 + 97);

// Encrypt digits

else if (isdigit(ch))

result += char(int(ch + shift - 48) % 10 + 48);

// Leave other characters unchanged

else

result += ch;

}

return result;

}

// Function to decrypt the text using Caesar Cipher

std::string decryptCaesarCipher(std::string text, int shift) {

std::string result = "";

// Traverse text

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

char ch = text[i];

// Decrypt uppercase letters

if (isupper(ch))

result += char(int(ch - shift - 65 + 26) % 26 + 65);

// Decrypt lowercase letters

else if (islower(ch))

result += char(int(ch - shift - 97 + 26) % 26 + 97);

// Decrypt digits

else if (isdigit(ch))

result += char(int(ch - shift - 48 + 10) % 10 + 48);

// Leave other characters unchanged

else

result += ch;

}

return result;

}

int main() {

std::string text;

int shift;

char choice;

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Enter the shift value: ";

std::cin >> shift;

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptCaesarCipher(text, shift) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptCaesarCipher(text, shift) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

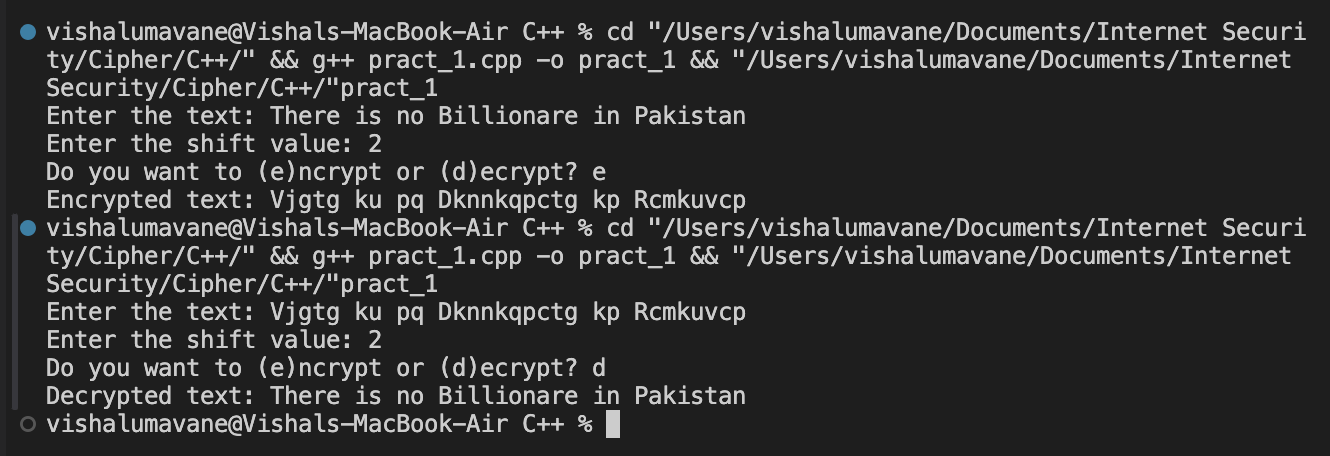
}

return 0;

}

**Outputs :**

Encryption & Decryption



**Practical 2**

**Aim**: Implement Monoalphabetic cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <unordered\_map>

#include <string>

// Function to generate the encryption and decryption maps based on the key

void generateMaps(std::string key, std::unordered\_map<char, char>& encryptMap, std::unordered\_map<char, char>& decryptMap) {

std::string alphabet = "abcdefghijklmnopqrstuvwxyz";

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

encryptMap[alphabet[i]] = key[i];

decryptMap[key[i]] = alphabet[i];

}

}

// Function to encrypt the text using Monoalphabetic Cipher

std::string encryptMonoalphabeticCipher(std::string text, std::unordered\_map<char, char>& encryptMap) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

char lower = tolower(ch);

result += isupper(ch) ? toupper(encryptMap[lower]) : encryptMap[lower];

} else {

result += ch;

}

}

return result;

}

// Function to decrypt the text using Monoalphabetic Cipher

std::string decryptMonoalphabeticCipher(std::string text, std::unordered\_map<char, char>& decryptMap) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

char lower = tolower(ch);

result += isupper(ch) ? toupper(decryptMap[lower]) : decryptMap[lower];

} else {

result += ch;

}

}

return result;

}

int main() {

std::string text, key;

char choice;

std::unordered\_map<char, char> encryptMap, decryptMap;

// Prompt for key

std::cout << "Enter the 26-letter key for the cipher (e.g., QWERTYUIOPASDFGHJKLZXCVBNM): ";

std::cin >> key;

// Generate maps

generateMaps(key, encryptMap, decryptMap);

// Clear the input buffer

std::cin.ignore();

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptMonoalphabeticCipher(text, encryptMap) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptMonoalphabeticCipher(text, decryptMap) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

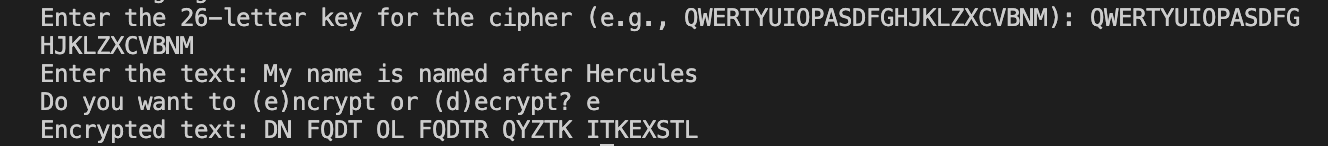
}

return 0;

}

**Outputs :**

Encryption



**Practical 3**

**Aim :** Implement Playfair cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cctype>

#include <string>

#include <unordered\_set>

// Function to generate the Playfair matrix based on the key

void generateMatrix(std::string key, char matrix[5][5]) {

std::string keyString = "";

std::unordered\_set<char> usedChars;

// Add key characters to keyString, removing duplicates and ignoring 'J'

for (char ch : key) {

ch = toupper(ch);

if (ch == 'J') ch = 'I';

if (usedChars.find(ch) == usedChars.end() && isalpha(ch)) {

keyString += ch;

usedChars.insert(ch);

}

}

// Add remaining letters to keyString

for (char ch = 'A'; ch <= 'Z'; ch++) {

if (ch == 'J') continue;

if (usedChars.find(ch) == usedChars.end()) {

keyString += ch;

usedChars.insert(ch);

}

}

// Fill the matrix

int k = 0;

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

matrix[i][j] = keyString[k++];

}

}

}

// Function to find the position of a character in the matrix

void findPosition(char matrix[5][5], char ch, int &row, int &col) {

if (ch == 'J') ch = 'I'; // Treat 'J' as 'I'

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (matrix[i][j] == ch) {

row = i;

col = j;

return;

}

}

}

}

// Function to process text by removing non-alphabetic characters and handling duplicate letters in digraphs

std::string processText(std::string text) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

ch = toupper(ch);

result += (ch == 'J') ? 'I' : ch;

}

}

// Handle duplicate letters in digraphs

for (size\_t i = 0; i < result.length(); i += 2) {

if (i + 1 < result.length() && result[i] == result[i + 1]) {

result.insert(i + 1, "X");

}

}

// If the processed text has an odd number of characters, add 'X' at the end

if (result.length() % 2 != 0) {

result += 'X';

}

return result;

}

// Function to encrypt the text using Playfair Cipher

std::string encryptPlayfairCipher(std::string text, char matrix[5][5]) {

std::string result = "";

text = processText(text);

for (size\_t i = 0; i < text.length(); i += 2) {

char first = text[i];

char second = text[i + 1];

int row1, col1, row2, col2;

findPosition(matrix, first, row1, col1);

findPosition(matrix, second, row2, col2);

if (row1 == row2) {

result += matrix[row1][(col1 + 1) % 5];

result += matrix[row2][(col2 + 1) % 5];

} else if (col1 == col2) {

result += matrix[(row1 + 1) % 5][col1];

result += matrix[(row2 + 1) % 5][col2];

} else {

result += matrix[row1][col2];

result += matrix[row2][col1];

}

}

return result;

}

// Function to decrypt the text using Playfair Cipher

std::string decryptPlayfairCipher(std::string text, char matrix[5][5]) {

std::string result = "";

text = processText(text);

for (size\_t i = 0; i < text.length(); i += 2) {

char first = text[i];

char second = text[i + 1];

int row1, col1, row2, col2;

findPosition(matrix, first, row1, col1);

findPosition(matrix, second, row2, col2);

if (row1 == row2) {

result += matrix[row1][(col1 + 4) % 5];

result += matrix[row2][(col2 + 4) % 5];

} else if (col1 == col2) {

result += matrix[(row1 + 4) % 5][col1];

result += matrix[(row2 + 4) % 5][col2];

} else {

result += matrix[row1][col2];

result += matrix[row2][col1];

}

}

return result;

}

int main() {

std::string text, key;

char choice;

char matrix[5][5];

// Prompt for key

std::cout << "Enter the key for the cipher: ";

std::getline(std::cin, key);

// Generate matrix

generateMatrix(key, matrix);

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptPlayfairCipher(text, matrix) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptPlayfairCipher(text, matrix) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

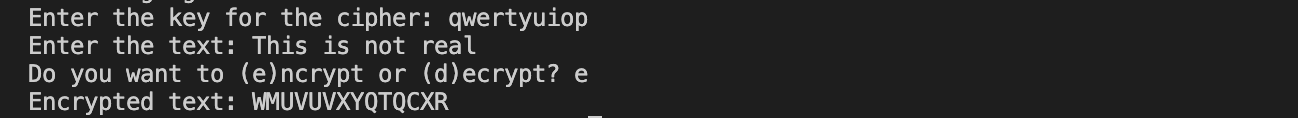
}

return 0;

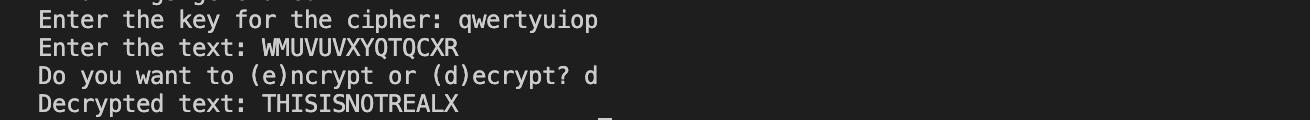
}

**Outputs :**

Encryption



Decryption



**Practical 4**

**Aim :** Implement Polyalphabetic cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

// Function to extend the key to match the length of the text

std::string extendKey(const std::string &text, const std::string &key) {

std::string extendedKey = key;

int textLength = text.length();

int keyLength = key.length();

for (int i = 0; i < textLength - keyLength; i++) {

extendedKey += key[i % keyLength];

}

return extendedKey;

}

// Function to encrypt the text using Vigenère Cipher

std::string encryptVigenereCipher(const std::string &text, const std::string &key) {

std::string encryptedText = "";

std::string extendedKey = extendKey(text, key);

for (size\_t i = 0; i < text.length(); i++) {

char ch = text[i];

if (isalpha(ch)) {

char base = isupper(ch) ? 'A' : 'a';

char keyCh = toupper(extendedKey[i]) - 'A';

encryptedText += (ch - base + keyCh) % 26 + base;

} else {

encryptedText += ch;

}

}

return encryptedText;

}

// Function to decrypt the text using Vigenère Cipher

std::string decryptVigenereCipher(const std::string &text, const std::string &key) {

std::string decryptedText = "";

std::string extendedKey = extendKey(text, key);

for (size\_t i = 0; i < text.length(); i++) {

char ch = text[i];

if (isalpha(ch)) {

char base = isupper(ch) ? 'A' : 'a';

char keyCh = toupper(extendedKey[i]) - 'A';

decryptedText += (ch - base - keyCh + 26) % 26 + base;

} else {

decryptedText += ch;

}

}

return decryptedText;

}

int main() {

std::string text, key;

char choice;

// Prompt for key

std::cout << "Enter the key for the cipher: ";

std::getline(std::cin, key);

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptVigenereCipher(text, key) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptVigenereCipher(text, key) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

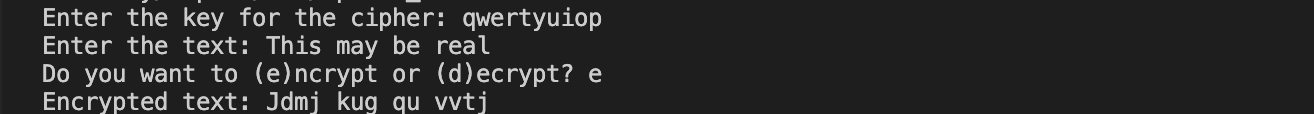
}

return 0;

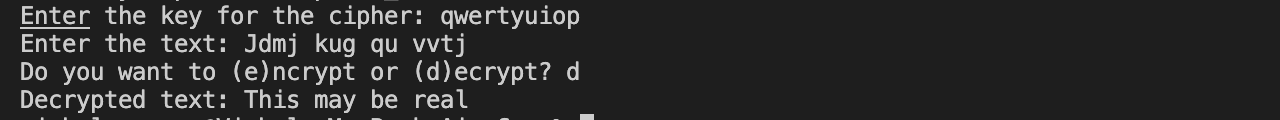
}

**Outputs :**

Encryption



Decryption



**Practical 5**

**Aim** : Implement hill cipher encryption and decryption

**Implementation :**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Function to generate the key matrix from the key string

void getKeyMatrix(string key, int keyMatrix[][3]) {

int k = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

keyMatrix[i][j] = (key[k]) % 65;

k++;

}

}

}

// Function to get the cofactor matrix

void getCofactor(int matrix[3][3], int temp[3][3], int p, int q, int n) {

int i = 0, j = 0;

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

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

if (row != p && col != q) {

temp[i][j++] = matrix[row][col];

if (j == n - 1) {

j = 0;

i++;

}

}

}

}

}

// Function to calculate the determinant of the matrix

int determinant(int matrix[3][3], int n) {

int det = 0;

if (n == 1) return matrix[0][0];

int temp[3][3];

int sign = 1;

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

getCofactor(matrix, temp, 0, i, n);

det += sign \* matrix[0][i] \* determinant(temp, n - 1);

sign = -sign;

}

return det;

}

// Function to find adjoint of a matrix

void adjoint(int matrix[3][3], int adj[3][3]) {

if (3 == 1) {

adj[0][0] = 1;

return;

}

int sign = 1, temp[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

getCofactor(matrix, temp, i, j, 3);

sign = ((i + j) % 2 == 0) ? 1 : -1;

adj[j][i] = (sign) \* (determinant(temp, 3 - 1));

}

}

}

// Function to find the modular inverse of a number

int modInverse(int a, int m) {

a = a % m;

for (int x = 1; x < m; x++) {

if ((a \* x) % m == 1)

return x;

}

return -1;

}

// Function to find the inverse of the key matrix

bool inverseKeyMatrix(int keyMatrix[3][3], int inverse[3][3]) {

int det = determinant(keyMatrix, 3);

int invDet = modInverse(det, 26);

if (invDet == -1) {

cout << "Inverse doesn't exist";

return false;

}

int adj[3][3];

adjoint(keyMatrix, adj);

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

inverse[i][j] = (adj[i][j] \* invDet) % 26;

if (inverse[i][j] < 0) inverse[i][j] += 26;

}

}

return true;

}

// Function to encrypt the message

void encrypt(int cipherMatrix[][1], int keyMatrix[][3], int messageVector[][1]) {

for (int i = 0; i < 3; i++) {

cipherMatrix[i][0] = 0;

for (int j = 0; j < 3; j++) {

cipherMatrix[i][0] += keyMatrix[i][j] \* messageVector[j][0];

}

cipherMatrix[i][0] = cipherMatrix[i][0] % 26;

}

}

// Function to decrypt the message

void decrypt(int plainMatrix[][1], int inverseKeyMatrix[][3], int cipherVector[][1]) {

for (int i = 0; i < 3; i++) {

plainMatrix[i][0] = 0;

for (int j = 0; j < 3; j++) {

plainMatrix[i][0] += inverseKeyMatrix[i][j] \* cipherVector[j][0];

}

plainMatrix[i][0] = plainMatrix[i][0] % 26;

}

}

// Function to implement Hill Cipher encryption

void HillCipherEncrypt(string message, string key) {

int keyMatrix[3][3];

getKeyMatrix(key, keyMatrix);

// Pad the message to make its length a multiple of 3

while (message.length() % 3 != 0) {

message += 'X'; // Padding with 'X'

}

string CipherText;

for (size\_t i = 0; i < message.length(); i += 3) {

int messageVector[3][1];

for (int j = 0; j < 3; j++) {

messageVector[j][0] = (message[i + j]) % 65;

}

int cipherMatrix[3][1];

encrypt(cipherMatrix, keyMatrix, messageVector);

for (int j = 0; j < 3; j++) {

CipherText += cipherMatrix[j][0] + 65;

}

}

// Print the ciphertext

cout << "Ciphertext: " << CipherText << endl;

}

// Function to implement Hill Cipher decryption

void HillCipherDecrypt(string ciphertext, string key) {

int keyMatrix[3][3];

getKeyMatrix(key, keyMatrix);

int inverseMatrix[3][3];

if (!inverseKeyMatrix(keyMatrix, inverseMatrix)) {

cout << "Key matrix is not invertible. Decryption aborted." << endl;

return;

}

string PlainText;

for (size\_t i = 0; i < ciphertext.length(); i += 3) {

int cipherVector[3][1];

for (int j = 0; j < 3; j++) {

cipherVector[j][0] = (ciphertext[i + j]) % 65;

}

int plainMatrix[3][1];

decrypt(plainMatrix, inverseMatrix, cipherVector);

for (int j = 0; j < 3; j++) {

PlainText += plainMatrix[j][0] + 65;

}

}

// Print the plaintext

cout << "Plaintext: " << PlainText << endl;

}

int main() {

string message;

string key = "GYBNQKURP";

cout << "Enter the message: ";

getline(cin, message);

HillCipherEncrypt(message, key);

string ciphertext;

cout << "Enter the ciphertext: ";

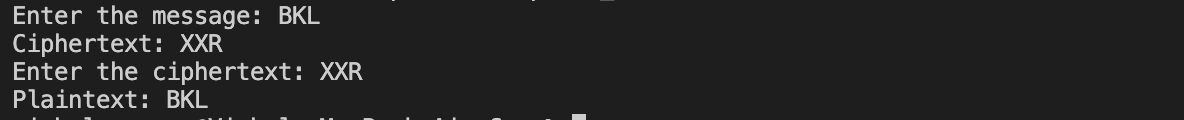
getline(cin, ciphertext);

HillCipherDecrypt(ciphertext, key);

return 0;

}

**Outputs :**



**Practical 6**

**Aim** : Implement Simple Transposition encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

#include <cstring>

using namespace std;

// Encryption function

string Encryption(int no\_rows, int len\_key, int len\_msg, string msg, int col\_val[])

{

int x = 0;

char enc\_mat[no\_rows + 1][len\_key];

// creating the matrix

for (int i = 0; i < no\_rows + 1; i++) {

for (int j = 0; j < len\_key; j++) {

// initializes the positions with '\_' after the end of message

if (x >= len\_msg) {

enc\_mat[i][j] = '\_';

}

else {

enc\_mat[i][j] = msg[x];

}

x++;

}

}

int t = 1;

string cipher = "";

// finding the cipher text according to the value of col\_val matrix

while (t <= len\_key) {

for (int i = 0; i < len\_key; i++) {

int k = col\_val[i];

if (k == t) {

for (int j = 0; j < no\_rows + 1; j++) {

cipher += enc\_mat[j][i];

}

t++;

}

}

}

return cipher;

}

// Decryption function

string Decryption(int no\_rows, int len\_key, string cipher, int col\_val[])

{

char dec\_mat[no\_rows + 1][len\_key];

int x = 0, t = 1;

// rearrange the matrix according to the col\_val

while (t <= len\_key) {

for (int i = 0; i < len\_key; i++) {

int k = col\_val[i];

if (k == t) {

for (int j = 0; j < no\_rows + 1; j++) {

dec\_mat[j][i] = cipher[x];

x++;

}

t++;

}

}

}

string message = "";

for (int i = 0; i < no\_rows + 1; i++) {

for (int j = 0; j < len\_key; j++) {

// replacing the '\_' with space

if (dec\_mat[i][j] == '\_') {

dec\_mat[i][j] = ' ';

}

message += dec\_mat[i][j];

}

}

return message;

}

int main()

{

// Taking input for the message and key from the user

string msg, key;

cout << "Enter the message to be encrypted: ";

getline(cin, msg);

cout << "Enter the key: ";

getline(cin, key);

int len\_key = key.length();

int len\_msg = msg.length();

int val = 1, count = 0, ind;

int col\_val[len\_key];

// initializing col\_val matrix with 0

memset(col\_val, 0, sizeof(col\_val));

// numbering the key alphabets according to its ASCII value

while (count < len\_key) {

int min = 999;

for (int i = 0; i < len\_key; i++) {

if ((min > int(key[i])) && (col\_val[i] == 0)) {

min = int(key[i]);

ind = i;

}

}

col\_val[ind] = val;

count++;

val++;

}

int no\_rows = len\_msg / len\_key;

// encrypted text

string cipher\_text = Encryption(no\_rows, len\_key, len\_msg, msg, col\_val);

cout << "Encrypted Message: " << cipher\_text << endl;

// decrypted text

string original\_msg = Decryption(no\_rows, len\_key, cipher\_text, col\_val);

cout << "Decrypted Message: " << original\_msg << endl;

return 0;

}

**Output :**

****

**Practical 7**

**Aim** : Implement One time pad encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

#include <algorithm> // For transform function

using namespace std;

// Method 1: Returning encrypted text

string stringEncryption(string text, string key)

{

// Initializing cipherText

string cipherText = "";

// Initialize cipher array of key length

// which stores the sum of corresponding no.'s

// of plainText and key.

int cipher[key.length()];

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

cipher[i] = text.at(i) - 'A' + key.at(i) - 'A';

}

// If the sum is greater than 25

// subtract 26 from it

// and store that resulting value

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

if (cipher[i] > 25) {

cipher[i] = cipher[i] - 26;

}

}

// Converting the no.'s into integers

// Convert these integers to corresponding

// characters and add them up to cipherText

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

int x = cipher[i] + 'A';

cipherText += (char)x;

}

// Returning the cipherText

return cipherText;

}

// Method 2: Returning plain text

static string stringDecryption(string s, string key)

{

// Initializing plain text

string plainText = "";

// Initializing integer array of key length

// which stores difference

// of corresponding no.'s of

// each character of cipherText and key

int plain[key.length()];

// Running for loop for each character

// subtracting and storing in the array

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

plain[i] = s.at(i) - 'A' - (key.at(i) - 'A');

}

// If the difference is less than 0

// add 26 and store it in the array.

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

if (plain[i] < 0) {

plain[i] = plain[i] + 26;

}

}

// Converting int to corresponding char

// add them up to plainText

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

int x = plain[i] + 'A';

plainText += (char)x;

}

// Returning plainText

return plainText;

}

// Method 3: Main driver method

int main()

{

// Declaring plain text and key

string plainText, key;

// Taking input for plain text and key from the user

cout << "Enter the message to be encrypted: ";

getline(cin, plainText);

cout << "Enter the key: ";

getline(cin, key);

// Converting plain text and key to uppercase

transform(plainText.begin(), plainText.end(), plainText.begin(), ::toupper);

transform(key.begin(), key.end(), key.begin(), ::toupper);

// Encrypting the plain text

string encryptedText = stringEncryption(plainText, key);

// Printing cipher Text

cout << "Cipher Text - " << encryptedText << endl;

// Decrypting the encrypted text

string decryptedText = stringDecryption(encryptedText, key);

// Printing the decrypted message

cout << "Decrypted Message - " << decryptedText << endl;

return 0;

}

**Output :**

