

Lab No.: 1 Date: 2080-2-12

**TITLE: WRITE A PROGRAM TO ENCRYPT AND DECRYPT THE USER INPUT MESSAGE AND KEY USING CAESAR CIPHER.**

**Caesar cipher**

The Caesar cipher is a simple encryption technique that was used by Julius Caesar to send secret messages to his allies. It works by shifting the letters in the plaintext message by a certain number of positions, known as the “shift” or “key”.

**Algorithm:**

Step 1: Start

Step 2: Convert a letter to its corresponding order in alphabet as:

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

Step 3: Compute ciphertext as:

C = E(K, P) = (P+K) mod 26

Where, 0 ≤ K ≤ 25

Step 4: Compute plaintext from ciphertext as,

P = D(K, C) = (C-K) mod 26

**IDE: Visual Studio**

**Programming Language: C**

**Source code:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

char input[100], ch;

int key, i;

printf("Encryption and decryption of input message using caesar cipher:\n");

printf("Enter the input message:");

gets(input);

printf("Enter the key(of range 0-25):");

scanf("%d", &key);

// Encryption of input message

for(i = 0; input[i] != '\0'; ++i){

ch = input[i];

if(ch >= 'a' && ch <= 'z'){

ch = ch + key;

if(ch > 'z'){

ch = ch - 'z' + 'a' - 1;

}

input[i] = ch;

}

else if(ch >= 'A' && ch <= 'Z'){

ch = ch + key;

if(ch > 'Z'){

ch = ch - 'Z' + 'A' - 1;

}

input[i] = ch;

}

}

printf("Encrypted message of input message: %s\n", input);

// Decryption for encrypted message

for(i = 0; input[i] != '\0'; ++i){

ch = input[i];

if(ch >= 'a' && ch <= 'z'){

ch = ch - key;

if(ch < 'a'){

ch = ch + 'z' - 'a' + 1;

}

input[i] = ch;

}

else if(ch >= 'A' && ch <= 'Z'){

ch = ch - key;

if(ch < 'A'){

ch = ch + 'Z' - 'A' + 1;

}

input[i] = ch;

}

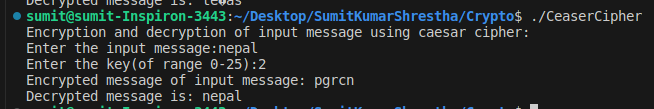
}

printf("Decrypted message is: %s\n", input);

return 0;

}

Output:





Lab No.: 2 Date: 2080-2-12

**TITLE: WRITE A PROGRAM TO ENCRYPT AND DECRYPT THE USER INPUT MESSAGE AND KEY USING PLAYFAIR CIPHER.**

**Playfair cipher**

The Playfair cipher is a manual symmetric encryption technique that encrypts pairs of letters (digraphs) instead of single letters. It uses a 5x5 matrix of letters constructed by filling in a keyword across the rows, followed by the remaining letters of the alphabet in order. Plaintext digraphs are located on the matrix, and their substitutes are determined by applying straightforward rules based on whether the two letters fall in the same row, column or neither. The use of a keyword to construct the matrix adds an extra level of cryptographic security compared to using a standard matrix. Despite its innovative digraph encryption approach, the Playfair cipher is considerably weaker than modern ciphers and is now an obsolete encryption method.

**For encryption:**

* If two symbols are in same row, substitute with immediate right circularly.
* If two symbols are in the same column, substitute with immediate upper symbol circularly.
* Otherwise, substitute with the symbols in rectangular form with corresponding row position symbol.

**For decryption:**

* If two symbols are in same row, substitute with immediate left circularly.
* If two symbols are in same column, substitute with immediate lower symbol circularly.
* Otherwise, substitute with the symbols in rectangular form with corresponding row position symbol.

**IDE: Visual Studio**

**Programming Language: C**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define SIZE 5

void prepareKeyTable(char key[], char keyTable[SIZE][SIZE]);

void formatMessage(char message[], char formatted[]);

void encrypt(char keyTable[SIZE][SIZE], char plaintext[], char ciphertext[]);

void decrypt(char keyTable[SIZE][SIZE], char ciphertext[], char plaintext[]);

void toUpper(char str[]);

void removeSpaces(char str[]);

void findPosition(char keyTable[SIZE][SIZE], char ch, int \*row, int \*col);

void replaceJwithI(char str[]);

void displayKeyTable(char keyTable[SIZE][SIZE]);

int main() {

char key[100];

char message[100];

char formattedMessage[100];

char ciphertext[100];

char decryptedMessage[100];

char keyTable[SIZE][SIZE];

printf("Enter key: ");

fgets(key, sizeof(key), stdin);

key[strcspn(key, "\n")] = '\0'; // Remove newline character

printf("Enter message: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0'; // Remove newline character

toUpper(key);

removeSpaces(key);

replaceJwithI(key);

prepareKeyTable(key, keyTable);

// Display the key table

displayKeyTable(keyTable);

formatMessage(message, formattedMessage);

toUpper(formattedMessage);

replaceJwithI(formattedMessage);

encrypt(keyTable, formattedMessage, ciphertext);

printf("Ciphertext: %s\n", ciphertext);

decrypt(keyTable, ciphertext, decryptedMessage);

printf("Decrypted message: %s\n", decryptedMessage);

return 0;

}

void prepareKeyTable(char key[], char keyTable[SIZE][SIZE]) {

int map[26] = {0};

int i, j, k, len = strlen(key);

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

if (key[i] != 'J' && map[key[i] - 'A'] == 0) {

keyTable[k / SIZE][k % SIZE] = key[i];

map[key[i] - 'A'] = 1;

k++;

}

}

for (i = 0; i < 26; i++) {

if (i != 'J' - 'A' && map[i] == 0) {

keyTable[k / SIZE][k % SIZE] = 'A' + i;

k++;

}

}

}

void formatMessage(char message[], char formatted[]) {

int i, j = 0;

for (i = 0; message[i] != '\0'; i++) {

if (isalpha(message[i])) {

formatted[j++] = toupper(message[i]);

if (j > 1 && formatted[j-1] == formatted[j-2]) {

formatted[j-1] = 'X';

formatted[j++] = toupper(message[i]);

}

}

}

if (j % 2 != 0) {

formatted[j++] = 'X';

}

formatted[j] = '\0';

}

void toUpper(char str[]) {

int i;

for (i = 0; str[i] != '\0'; i++) {

str[i] = toupper(str[i]);

}

}

void removeSpaces(char str[]) {

int i, j;

for (i = 0, j = 0; str[i] != '\0'; i++) {

if (str[i] != ' ') {

str[j++] = str[i];

}

}

str[j] = '\0';

}

void findPosition(char keyTable[SIZE][SIZE], char ch, int \*row, int \*col) {

int i, j;

for (i = 0; i < SIZE; i++) {

for (j = 0; j < SIZE; j++) {

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

\*row = i;

\*col = j;

return;

}

}

}

}

void replaceJwithI(char str[]) {

int i;

for (i = 0; str[i] != '\0'; i++) {

if (str[i] == 'J') {

str[i] = 'I';

}

}

}

void encrypt(char keyTable[SIZE][SIZE], char plaintext[], char ciphertext[]) {

int i, a, b, c, d;

for (i = 0; plaintext[i] != '\0'; i += 2) {

findPosition(keyTable, plaintext[i], &a, &b);

findPosition(keyTable, plaintext[i + 1], &c, &d);

if (a == c) {

ciphertext[i] = keyTable[a][(b + 1) % SIZE];

ciphertext[i + 1] = keyTable[c][(d + 1) % SIZE];

} else if (b == d) {

ciphertext[i] = keyTable[(a + 1) % SIZE][b];

ciphertext[i + 1] = keyTable[(c + 1) % SIZE][d];

} else {

ciphertext[i] = keyTable[a][d];

ciphertext[i + 1] = keyTable[c][b];

}

}

ciphertext[i] = '\0';

}

void decrypt(char keyTable[SIZE][SIZE], char ciphertext[], char plaintext[]) {

int i, a, b, c, d;

int len = strlen(ciphertext);

int index = 0; // Index for plaintext

for (i = 0; i < len; i += 2) {

findPosition(keyTable, ciphertext[i], &a, &b);

findPosition(keyTable, ciphertext[i + 1], &c, &d);

if (a == c) {

plaintext[index++] = keyTable[a][(b + SIZE - 1) % SIZE];

plaintext[index++] = keyTable[c][(d + SIZE - 1) % SIZE];

} else if (b == d) {

plaintext[index++] = keyTable[(a + SIZE - 1) % SIZE][b];

plaintext[index++] = keyTable[(c + SIZE - 1) % SIZE][d];

} else {

plaintext[index++] = keyTable[a][d];

plaintext[index++] = keyTable[c][b];

}

}

plaintext[index] = '\0';

// Remove 'X' characters from plaintext

int j = 0;

for (i = 0; i < index; i++) {

if (plaintext[i] != 'X') {

plaintext[j++] = plaintext[i];

}

}

plaintext[j] = '\0';

}

void displayKeyTable(char keyTable[SIZE][SIZE]) {

int i, j;

printf("Key Table:\n");

for (i = 0; i < SIZE; i++) {

for (j = 0; j < SIZE; j++) {

printf("%c ", keyTable[i][j]);

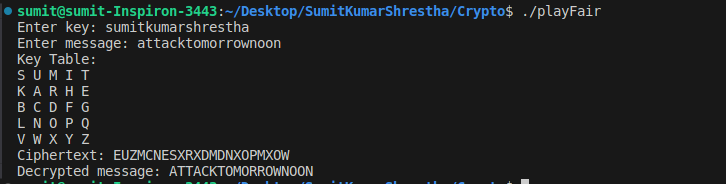
}

printf("\n");

}

}

Output:





Lab No.: 5 Date: 2080-2-22

**TITLE: WRITE A PROGRAM TO TEST IF A NUMBER IS PRIMITIVE ROOT OR NOT.**

A primitive root modulo a prime number p is an integer g such that the successive powers of g modulo p generate all the integers from 1 to p-1. In other words, the set {g1 mod p, g2 mod p, g3mod p, ..., gp-1 mod p} contains all the integers from 1 to p-1 without any repetitions.Let's consider the prime number p = 7.

We can check if 3 is a primitive root of 7 by calculating the successive powers of 3 modulo 7:

31 mod 7 = 3

32 mod 7 = 2

33 mod 7 = 6

34 mod 7 = 4

35 mod 7 = 5

36 mod 7 = 1

Since the set {3, 2, 6, 4, 5, 1} contains all the integers from 1 to 6 (which are the non-zero integers modulo 7), we can conclude that 3 is a primitive root of 7.

Primitive roots are useful in various applications, such as cryptography and number theory algorithms, where they are used to generate specific sequences of numbers or to perform certain computations efficiently.

**IDE: Visual Studio Code**

**Programming Language: C**

**Source code:**

#include <stdio.h>

#include <math.h>

void checkprimitiveRoot(int p, int g)

{

int arr[p - 1], used[p - 1];

for (int i = 0; i < p - 1; i++)

{

arr[i] = (int)pow(g, i + 1) % p;

printf("%d^%d mod %d: %d\n", g, i + 1, p, arr[i]);

}

for (int i = 0; i < p - 1; i++)

{

for (int j = i + 1; j < p - 1; j++)

{

if (arr[i] == arr[j])

{

printf("The number not is primitive root\n");

return;

}

}

}

printf("The number is primitive root\n");

}

int main()

{

int p, g;

printf("Enter the prime number p: ");

scanf("%d", &p);

printf("Enter the number g: ");

scanf("%d", &g);

checkprimitiveRoot(p, g);

return 0;

}

**Output:**

