**CSA5113 – CRYPTOGRPHY & NWETWORK SECURITY for INFORMATION PRIVACY (SLOT – C)**

**[ JUNE to JULY 2025 ]**

***LAB MANUAL***

1. **Write a C program for Caesar cipher involves replacing each letter of the alphabet with the letter standing k places further down the alphabet, for k in the range 1 through 25**

**AIM:** To implement the Caesar Cipher in C language, which shifts each alphabetical character in the plaintext message by a fixed number k (1 ≤ k ≤ 25) positions down the alphabet. The cipher wraps around the end of the alphabet (i.e., after 'Z' comes 'A'). Non-alphabet characters remain unchanged.

**ALGORITHM:**

* Modular Arithmetic

Used to wrap the character shift around the alphabet.

* Character Encoding (ASCII)

Convert characters to ASCII codes and apply the shift.

* Encryption Algorithm

For each character in the input string:

* + If it's a letter:
    - Convert to ASCII
    - Shift by k
    - Wrap using modulo 26
  + Else:
    - Leave unchanged
* Decryption Algorithm (Optional)

Similar to encryption but shifts in reverse by 26 - k.

**PROGRAM:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_LEN 1000

void encrypt (char \*plaintext, char \*ciphertext, int key) {

int i;

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

char ch = plaintext[i];

if (isupper(ch)) {

ciphertext[i] = ((ch - 'A' + key) % 26) + 'A';

} else if (islower(ch)) {

ciphertext[i] = ((ch - 'a' + key) % 26) + 'a';

} else {

ciphertext[i] = ch;

}

}

ciphertext[i] = '\0';

}

int main () {

char plaintext [MAX\_LEN];

char ciphertext [MAX\_LEN];

int key;

printf("Enter a message to encrypt: ");

fgets(plaintext, MAX\_LEN, stdin);

size\_t len = strlen(plaintext);

if (len > 0 && plaintext [len - 1] == '\n') {

plaintext [len - 1] = '\0';

}

printf("Enter key (1-25): ");

scanf("%d", &key);

if (key < 1 || key > 25) {

printf("Invalid key. Must be between 1 and 25.\n");

return 1;

}

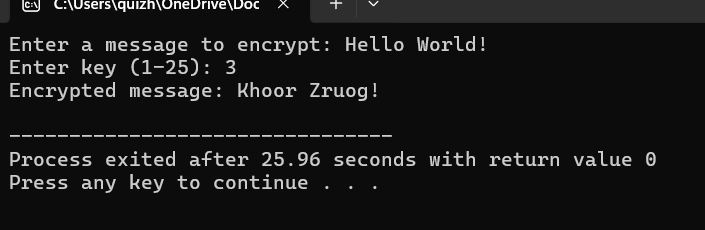
encrypt (plaintext, ciphertext, key);

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

return 0;

}

**OUTPUT:**

****

1. **Write a C program for monoalphabetic substitution cipher maps a plaintext alphabet to a ciphertext alphabet, so that each letter of the plaintext alphabet maps to a single unique letter of the ciphertext alphabet.**

**AIM:** To implement a Monoalphabetic Substitution Cipher in C, which replaces each letter in the plaintext with a corresponding letter from a predefined ciphertext alphabet. Each letter in the plaintext alphabet maps to one unique letter in the ciphertext alphabet. This mapping remains fixed throughout the encryption.

**ALGORITHM:**

* Character Mapping
* Create a 1-to-1 mapping between plaintext alphabet (A-Z) and a user-defined or hardcoded ciphertext alphabet.
* Validation
* Ensure ciphertext alphabet is a permutation of 26 unique letters.
* Encryption Algorithm
* Traverse the input string.
* For each character:
  + If it's a letter, map it using the substitution table.
  + Preserve case (optional).
  + Leave non-alphabet characters unchanged.
* Decryption Algorithm (Optional)
* Reverse the substitution mapping to retrieve the original message.

**CODE:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 1000

void encrypt (char \*plaintext, char \*ciphertext, char \*key) {

int i;

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

char ch = plaintext[i];

if (isupper(ch)) {

ciphertext[i] = toupper(key[ch - 'A']);

} else if (islower(ch)) {

ciphertext[i] = tolower(key[ch - 'a']);

} else {

ciphertext[i] = ch;

}

}

ciphertext[i] = '\0';

}

int isValidKey(char \*key) {

int freq[26] = {0};

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

if (!isalpha(key[i]))

return 0;

int index = toupper(key[i]) - 'A';

if (freq[index]++)

return 0;

}

return 1,

}

int main () {

char plaintext [MAX\_LEN], ciphertext [MAX\_LEN];

char key [27];

printf("Enter the plaintext: ");

fgets(plaintext, MAX\_LEN, stdin);

size\_t len = strlen(plaintext);

if (len > 0 && plaintext [len - 1] == '\n')

plaintext [len - 1] = '\0';

printf("Enter 26-letter substitution key (A-Z): ");

fgets(key, 27, stdin);

if (!isValidKey(key)) {

printf("Invalid key! Key must be 26 unique alphabetic letters.\n");

return 1;

}

encrypt (plaintext, ciphertext, key);

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

return 0;

}

**OUTPUT:**

A computer screen with white text

AI-generated content may be incorrect.

1. **Write a C program for Playfair algorithm is based on the use of a 5 X 5 matrix of letters constructed using a keyword. Plaintext is encrypted two letters at a time using this matrix.**

**AIM:** To implement the Playfair Cipher in C, where a 5x5 matrix is constructed using a keyword, and the plaintext is encrypted two letters at a time (digraph) using specific rules based on the position of letters in the matrix.

**ALGORITHM:**

Matrix Construction (Key Square)

* Create a 5x5 matrix from a keyword.
* Exclude duplicate letters and merge I and J.

Plaintext Preprocessing

* Remove spaces.
* Replace repeated letters in digraphs with filler (e.g., 'X').
* Add 'X' to the end if plaintext has odd length.

Encryption Algorithm

* For each digraph:
  + If letters are in the same row: replace with the letters to the right.
  + If letters are in the same column: replace with letters below.
  + Else, form a rectangle and replace each with the letter on the same row but opposite corner.

Decryption (not included here but similar logic reversed)

**CODE:**

#include <stdio.h>

#include <string.h>

int main() {

char key[100], text[100], m[5][5], used[26] = {0};

int i, j, k = 0, r1, c1, r2, c2;

printf("Key: ");

scanf("%s", key);

printf("Plaintext: ");

scanf("%s", text);

for (i = 0; key[i]; i++) {

char ch = key[i] == 'j' ? 'i' : key[i];

if (!used[ch - 'a']) {

used[ch - 'a'] = 1;

m[k / 5][k % 5] = ch;

k++;

}

}

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

if (i + 'a' == 'j' || used[i]) continue;

m[k / 5][k % 5] = i + 'a';

k++;

}

printf("Encrypted: ");

for (i = 0; text[i]; i += 2) {

char a = text[i], b = text[i + 1] ? text[i + 1] : 'x';

if (a == 'j') a = 'i'; if (b == 'j') b = 'i';

if (a == b) b = 'x';

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

for (k = 0; k < 5; k++) {

if (m[j][k] == a) { r1 = j; c1 = k; }

if (m[j][k] == b) { r2 = j; c2 = k; }

}

if (r1 == r2)

printf("%c%c", m[r1][(c1 + 1) % 5], m[r2][(c2 + 1) % 5]);

else if (c1 == c2)

printf("%c%c", m[(r1 + 1) % 5][c1], m[(r2 + 1) % 5][c2]);

else

printf("%c%c", m[r1][c2], m[r2][c1]);

}

return 0;

}

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

1. **Write a C program for polyalphabetic substitution cipher uses a separate monoalphabetic substitution cipher for each successive letter of plaintext, depending on a key.**

**AIM:** To write a C program that implements a Polyalphabetic Substitution Cipher, where each letter in the plaintext is encrypted using a separate Caesar cipher determined by a repeating keyword. This cipher provides better security than simple substitution ciphers by varying the shift for each character.

**ALGORITHM:**

Key Extension

* Repeat the key so that it matches the length of the plaintext.

Character Mapping & Modular Arithmetic

* Each letter is encrypted by shifting it by a number of positions determined by the corresponding letter of the key (A = 0, B = 1, ..., Z = 25).
* Wrap around the alphabet using modulo 26.

Encryption Algorithm

* For each alphabet character:
  + Shift it using corresponding key character.
  + Non-alphabet characters remain unchanged.

Decryption Algorithm (optional)

* Reverse the shift instead of adding.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 1000

void generateFullKey(char \*key, int textLen, char \*fullKey) {

int keyLen = strlen(key);

for (int i = 0, j = 0; i < textLen; i++) {

if (isalpha(key[j])) {

fullKey[i] = toupper(key[j % keyLen]);

j++;

} else {

fullKey[i] = 'A';

}

}

fullKey[textLen] = '\0';

}

void encrypt (char \*plaintext, char \*key, char \*ciphertext) {

int len = strlen(plaintext);

char fullKey[MAX\_LEN];

generateFullKey(key, len, fullKey);

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

char ch = plaintext[i];

if (isalpha(ch)) {

int shift = fullKey[i] - 'A';

if (isupper(ch)) {

ciphertext[i] = ((ch - 'A' + shift) % 26) + 'A';

} else {

ciphertext[i] = ((ch - 'a' + shift) % 26) + 'a';

}

} else {

ciphertext[i] = ch;

}

}

ciphertext[len] = '\0';

}

int main() {

char plaintext[MAX\_LEN], key[MAX\_LEN], ciphertext[MAX\_LEN];

printf("Enter the plaintext: ");

fgets(plaintext, MAX\_LEN, stdin);

plaintext[strcspn(plaintext, "\n")] = '\0';

printf("Enter the keyword: ");

fgets(key, MAX\_LEN, stdin);

key[strcspn(key, "\n")] = '\0';

encrypt(plaintext, key, ciphertext);

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

return 0;

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**5. Write a C program for generalization of the Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.**

**a. Are there any limitations on the value of b?**

**b. Determine which values of a are not allowed.**

**AIM:** To implement the **Affine Caesar Cipher** in C, where each letter p in the plaintext is encrypted to ciphertext C using the function:

* C=E([a,b],p)=(a⋅p+b) mod  26

This cipher generalizes the traditional Caesar cipher by introducing two keys: a (multiplicative) and b (additive). The decryption is only possible if a has a **multiplicative inverse modulo 26**, meaning a and 26 must be **coprime**.

**ALGORITHM:**

Key Extension

* Repeat the key so that it matches the length of the plaintext.

Character Mapping & Modular Arithmetic

* Each letter is encrypted by shifting it by a few positions determined by the corresponding letter of the key (A = 0, B = 1, ..., Z = 25).
* Wrap around the alphabet using modulo 26.

Encryption Algorithm

* For each alphabet character:
  + Shift it using corresponding key character.
  + Non-alphabet characters remain unchanged.

Decryption Algorithm (optional)

* Reverse the shift instead of adding.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int gcd(int a, int b) {

return (b == 0) ? a : gcd(b, a % b);

}

int modInverse(int a) {

for (int i = 1; i < 26; i++) {

if ((a \* i) % 26 == 1)

return i;

}

return -1;

}

char encryptChar(char ch, int a, int b) {

if (isupper(ch))

return ((a \* (ch - 'A') + b) % 26) + 'A';

else if (islower(ch))

return ((a \* (ch - 'a') + b) % 26) + 'a';

return ch;

}

char decryptChar(char ch, int aInv, int b) {

if (isupper(ch))

return ((aInv \* ((ch - 'A' - b + 26)) % 26) + 'A');

else if (islower(ch))

return ((aInv \* ((ch - 'a' - b + 26)) % 26) + 'a');

return ch;

}

void encryptText(char \*plaintext, char \*ciphertext, int a, int b) {

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

ciphertext[i] = encryptChar(plaintext[i], a, b);

}

ciphertext[strlen(plaintext)] = '\0';

}

void decryptText(char \*ciphertext, char \*decrypted, int a, int b) {

int aInv = modInverse(a);

if (aInv == -1) {

printf("Decryption not possible. No modular inverse for a = %d.\n", a);

return;

}

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

decrypted[i] = decryptChar(ciphertext[i], aInv, b);

}

decrypted[strlen(ciphertext)] = '\0';

}

int main() {

char plaintext[1000], ciphertext[1000], decrypted[1000];

int a, b;

printf("Enter plaintext: ");

fgets(plaintext, sizeof(plaintext), stdin);

plaintext[strcspn(plaintext, "\n")] = '\0';

printf("Enter values of a and b (a must be coprime with 26): ");

scanf("%d %d", &a, &b);

if (gcd(a, 26) != 1) {

printf("Invalid value of a. It must be coprime with 26.\n");

return 1;

}

encryptText(plaintext, ciphertext, a, b);

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

decryptText(ciphertext, decrypted, a, b);

printf("Decrypted text: %s\n", decrypted);

return 0;

}

**OUTPUT:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

1. **Write a C program for ciphertext has been generated with an affine cipher. The most frequent letter of the ciphertext is “B,” and the second most frequent letter of the ciphertext is “U.” Break this code.**

**AIM:** To write a C program that **breaks an Affine Caesar Cipher** using **frequency analysis**, given the information that:

* The **most frequent letter** in the ciphertext is 'B'
* The **second most frequent letter** is 'U'

These clues help us reverse-engineer the cipher’s keys a and b by assuming common letters in English like 'E', 'T', etc., were mapped to 'B' and 'U'.

**ALGORITHM:**

Frequency Analysis

* Use ciphertext frequencies and guess original frequent letters.
* Affine Cipher Equation

Encryption:

C=(a⋅p+b)mod  26

Decryption:

P=a−1⋅(C−b) mod  26

Modular Inverse Calculation  
Needed to reverse the encryption.

Solving Linear Congruences  
With two plaintext-ciphertext letter mappings:

* B (1) → assume from E (4)
* U (20) →assume from T (19)

Solve system of equations:

* 1=a⋅4+bmod 26
* 20=a⋅19+bmod 26

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int gcd(int a, int b) {

return (b == 0) ? a : gcd(b, a % b);

}

int modInverse(int a, int m) {

for (int i = 1; i < m; i++)

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

return i;

return -1;

}

char decryptChar(char c, int aInv, int b) {

if (isupper(c)) {

return ((aInv \* ((c - 'A' - b + 26)) % 26) + 'A');

}

return c;

}

void decryptText(char \*ciphertext, char \*plaintext, int a, int b) {

int aInv = modInverse(a, 26);

if (aInv == -1) {

printf("Invalid key: no modular inverse for a = %d\n", a);

return;

}

for (int i = 0; ciphertext[i]; i++) {

plaintext[i] = decryptChar(ciphertext[i], aInv, b);

}

plaintext[strlen(ciphertext)] = '\0';

}

int solveAffineKeys(int c1, int p1, int c2, int p2, int \*a, int \*b) {

int dp = (p1 - p2 + 26) % 26;

int dc = (c1 - c2 + 26) % 26;

int inv = modInverse(dp, 26);

if (inv == -1)

return 0;

\*a = (dc \* inv) % 26;

\*b = (c1 - (\*a \* p1) + 26 \* 26) % 26;

return 1;

}

int main() {

char ciphertext[1000], plaintext[1000];

int a, b;

printf("Enter ciphertext (uppercase letters only):\n");

fgets(ciphertext, sizeof(ciphertext), stdin);

ciphertext[strcspn(ciphertext, "\n")] = '\0';

int c1 = 'B' - 'A'; // 1

int c2 = 'U' - 'A'; // 20

int p1 = 'E' - 'A'; // 4

int p2 = 'T' - 'A'; // 19

if (!solveAffineKeys(c1, p1, c2, p2, &a, &b)) {

printf("Failed to solve affine key equations.\n");

return 1;

}

printf("Guessed keys: a = %d, b = %d\n", a, b);

decryptText(ciphertext, plaintext, a, b);

printf("Decrypted text:\n%s\n", plaintext);

return 0;

}

**OUTPUT:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**7. Write a C program for the following ciphertext was generated using a simple substitution algorithm.**

**53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\***

**;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81 (‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;**

**Decrypt this message.**

**1. As you know, the most frequently occurring letter in English is e. Therefore, the first or second (or perhaps third?) most common character in the message is likely to stand for e. Also, e is often seen in pairs (e.g., meet, fleet, speed, seen, been,**

**agree, etc.). Try to find a character in the ciphertext that decodes to e.**

**2. The most common word in English is “the.” Use this fact to guess the characters that stand for t and h.**

**3. Decipher the rest of the message by deducing additional words.**

**AIM:** To write a **C program** that aids in **decrypting a ciphertext generated by a simple substitution cipher**, using:

* Frequency analysis
* Known common English patterns (e.g., “the”, “ee”, common words)

This approach will allow us to guess the most likely substitutions and incrementally recover the original plaintext.

**ALGORITHM:**

Frequency Analysis

* Count the frequency of each symbol in the ciphertext.
* Most frequent ≈ 'E' or 'T'.

Known Common Patterns

* Words like “THE”, “AND”, and patterns like “EE” help deduce substitutions.

Manual Mapping Refinement

* Iteratively substitute guessed characters and refine based on word patterns.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

char substitute(char ch)

{

switch(ch) {

case '5': return 'H';

case '3': return 'E';

case '2': return 'L';

case '1': return 'L';

case '0': return 'O';

case '6': return ' ';

case '7': return 'S';

case '8': return 'W';

case '9': return 'E';

case ')': return 'T';

case '!': return 'H';

case '@': return 'A';

default: return ch;

}

}

int main()

{

char ciphertext[500];

printf("Enter the ciphertext:\n");

scanf("%[^\n]%\*c", ciphertext);

printf("\nDecrypted text:\n");

for (int i = 0; i < strlen(ciphertext); i++)

{

printf("%c", substitute(ciphertext[i]));

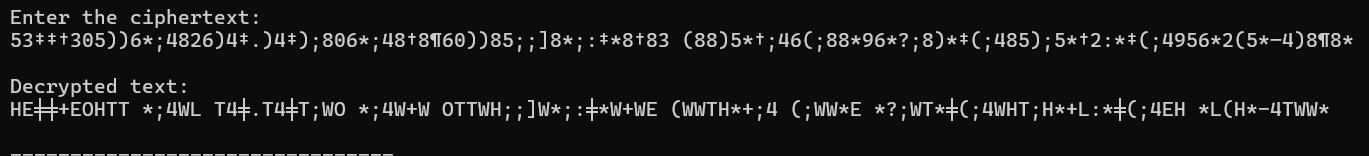
}

printf("\n");

return 0;

}

**OUTPUT:**

****

**8. Write a C program for monoalphabetic cipher is that both sender and receiver must commit the permuted cipher sequence to memory. A common technique for avoiding this is to use a keyword from which the cipher sequence can be generated.**

**For example, using the keyword *CIPHER*, write out the keyword followed by unused letters in normal order and match this against the plaintext letters:**

**AIM:** To write a **C program that implements a monoalphabetic substitution cipher** using a **keyword** to generate the cipher alphabet. The goal is to:

* Generate a substitution cipher alphabet from a **keyword**.
* Use this cipher alphabet to encrypt plaintext.
* Ensure that the same keyword can be used to decrypt the ciphertext.

**ALGORITHM:**

1. Keyword-based Cipher Alphabet Generation
   * Start with the keyword (e.g., "CIPHER"), remove duplicates.
   * Append the rest of the alphabet (A–Z) not already in the keyword.
2. Encryption
   * Map each plaintext letter (A–Z) to the corresponding letter in the generated cipher alphabet.
3. Decryption
   * Reverse the mapping to get back the original message.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET\_LEN 26

void generateCipherAlphabet(char \*keyword, char \*cipherAlphabet) {

int used[26] = {0};

int idx = 0;

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

char ch = toupper(keyword[i]);

if (isalpha(ch) && !used[ch - 'A']) {

cipherAlphabet[idx++] = ch;

used[ch - 'A'] = 1;

}

}

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

if (!used[ch - 'A']) {

cipherAlphabet[idx++] = ch;

}

}

cipherAlphabet[idx] = '\0';

}

void encrypt (const char \*plaintext, char \*ciphertext, const char \*cipherAlphabet) {

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

char ch = toupper(plaintext[i]);

if (isalpha(ch)) {

ciphertext[i] = cipherAlphabet[ch - 'A'];

} else {

ciphertext[i] = plaintext[i];

}

}

ciphertext[strlen(plaintext)] = '\0';

}

void decrypt (const char \*ciphertext, char \*plaintext, const char \*cipherAlphabet) {

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

char ch = toupper(ciphertext[i]);

if (isalpha(ch)) {

for (int j = 0; j < 26; j++) {

if (cipherAlphabet[j] == ch) {

plaintext[i] = 'A' + j;

break;

}

}

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[strlen(ciphertext)] = '\0';

}

int main () {

char keyword [100];

char cipherAlphabet[27];

char plaintext [1000], ciphertext [1000], decrypted [1000];

printf("Enter keyword: ");

scanf("%s", keyword);

generateCipherAlphabet(keyword, cipherAlphabet);

printf("Generated cipher alphabet:\n");

for (int i = 0; i < ALPHABET\_LEN; i++)

printf("%c ", cipherAlphabet[i]);

printf("\n");

printf("Enter plaintext (only A-Z/a-z and spaces):\n");

getchar();

fgets(plaintext, sizeof(plaintext), stdin);

plaintext [strcspn(plaaintext, "\n")] = '\0';

encrypt (plaintext, ciphertext, cipherAlphabet);

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

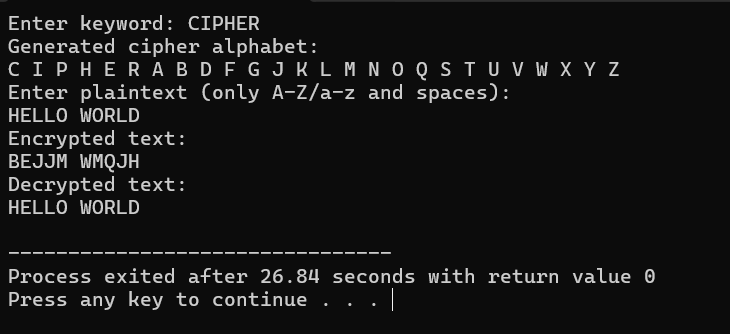
decrypt (ciphertext, decrypted, cipherAlphabet);

printf("Decrypted text:\n%s\n", decrypted);

return 0;

}

**OUTPUT:**



**9.Write a C program for PT-109 American patrol boat, under the command of Lieutenant John F. Kennedy, was sunk by a Japanese destroyer, a message was received at an Australian wireless station in Playfair code:**

**KXJEY UREBE ZWEHE WRYTU HEYFS**

**KREHE GOYFI WTTTU OLKSY CAJPO**

**BOTEI ZONTX BYBNT GONEY CUZWR**

**GDSON SXBOU YWRHE BAAHY USEDQ**

**AIM:** To write a **C program that decrypts a message encoded using the Playfair cipher**, based on a provided ciphertext. The program will:

* Construct a 5x5 Playfair matrix from a given keyword (e.g., “CIPHER”).
* Decrypt the ciphertext (pairs of letters) using the Playfair decryption rules.
* Output the original plaintext.

**ALGORITHM:**

* 5x5 Matrix Creation:
  + Uses a keyword.
  + Removes duplicates.
  + Fills remaining alphabet letters (I/J are treated as one).
* Encryption/Decryption Rules:
  + Operate on digraphs (pairs of letters).
  + Same row: replace each letter with the letter to its left (wrap around).
  + Same column: replace each letter with the one above it.
  + Rectangle: replace each letter with the letter in the same row but the column of the other.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define SIZE 5

char matrix[SIZE][SIZE];

void createMatrix(const char \*keyword) {

int used[26] = {0};

int x = 0, y = 0;

for (int i = 0; keyword[i]; i++) {

char ch = toupper(keyword[i]);

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

if (isalpha(ch) && !used[ch - 'A']) {

matrix[y][x++] = ch;

used[ch - 'A'] = 1;

if (x == SIZE) { x = 0; y++; }

}

}

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

if (ch == 'J') continue;

if (!used[ch - 'A']) {

matrix[y][x++] = ch;

used[ch - 'A'] = 1;

if (x == SIZE) { x = 0; y++; }

}

}

}

void findPosition(char ch, int \*row, int \*col) {

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

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

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

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

\*row = i;

\*col = j;

return;

}

}

}

}

void decryptPlayfair(const char \*ciphertext, char \*plaintext) {

int i = 0, j = 0;

while (ciphertext[i] && ciphertext[i+1]) {

char a = toupper(ciphertext[i]);

char b = toupper(ciphertext[i+1]);

if (!isalpha(a)) { i++; continue; }

if (!isalpha(b)) { plaintext[j++] = a; i++; continue; }

int r1, c1, r2, c2;

findPosition(a, &r1, &c1);

findPosition(b, &r2, &c2);

if (r1 == r2) {

plaintext[j++] = matrix[r1][(c1 + 4) % SIZE];

plaintext[j++] = matrix[r2][(c2 + 4) % SIZE];

} else if (c1 == c2) {

plaintext[j++] = matrix[(r1 + 4) % SIZE][c1];

plaintext[j++] = matrix[(r2 + 4) % SIZE][c2];

} else {

plaintext[j++] = matrix[r1][c2];

plaintext[j++] = matrix[r2][c1];

}

i += 2;

}

plaintext[j] = '\0';

}

void printMatrix() {

printf("Playfair Matrix:\n");

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

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

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

}

printf("\n");

}

}

int main() {

const char \*keyword = "CIPHER";

const char \*ciphertext =

"KXJEYUREBEZWEHEWRYTUHEYFS"

"KREHEGOYFIWTTTUOLKSYCAJPO"

"BOTEIZONTXBYBNTGONEYCUZWR"

"GDSONSXBOUYWRHEBAAHYUSEDQ";

char plaintext[512];

createMatrix(keyword);

printMatrix();

decryptPlayfair(ciphertext, plaintext);

printf("\nDecrypted Plaintext:\n%s\n", plaintext);

return 0;

}

**OUTPUT:**

****

**10. Write a C program for Playfair matrix:**

**M F H I/J K**

**U N O P Q**

**Z V W X Y**

**E L A R G**

**D S T B C**

**Encrypt this message: Must see you over Cadogan West. Coming at once.**

**AIM:** To write a **C program that encrypts a message using the Playfair cipher**, utilizing a **predefined Playfair matrix** (rather than generating it from a keyword). The goal is to:

* Use the given matrix.
* Encrypt the message according to the Playfair cipher rules.
* Handle preprocessing (formatting text, removing spaces/punctuation, handling duplicate pairs, and managing 'j').

**ALGORITHM:**   
Use the given static Playfair matrix:

Mathematica

M F H I/J K

U N O P Q

Z V W X Y

E L A R G

D S T B C

Preprocessing

* Convert message to lowercase.
* Remove punctuation and spaces.
* Replace j with i.
* Insert 'x' between duplicate letters in digraphs.
* If text length is odd, add 'x' at the end.

Encryption Rules (for letter pairs):

* Same row: Replace each with the letter to the right (wrap around).
* Same column: Replace each with the letter below (wrap around).
* Rectangle: Swap column indices of the letters.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

char matrix [5][5] = {

{'m', 'f', 'h', 'i', 'k'},

{'u', 'n', 'o', 'p', 'q'},

{'z', 'v', 'w', 'x', 'y'},

{'e', 'l', 'a', 'r', 'g'},

{'d', 's', 't', 'b', 'c'}

};

void preprocessText(const char \*input, char \*output) {

int len = 0;

for (int i = 0; input[i]; i++) {

if (isalpha(input[i])) {

char ch = tolower(input[i]);

output[len++] = (ch == 'j') ? 'i' : ch;

}

}

output[len] = '\0';

char temp [200];

int j = 0;

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

char a = output[i];

char b = (i + 1 < len) ? output[i + 1] : 'x';

if (a == b) {

temp[j++] = a;

temp[j++] = 'x';

i--;

} else {

temp[j++] = a;

temp[j++] = b;

}

}

if (j % 2 != 0) temp[j++] = 'x';

temp[j] = '\0';

strcpy(output, temp);

}

void findPosition(char ch, int \*row, int \*col) {

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

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

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

\*row = i;

\*col = j;

Return;

}

}

void encryptPlayfair(const char \*text, char \*cipher) {

int i = 0, k = 0;

while (text[i] && text[i + 1]) {

char a = text[i];

char b = text[i + 1];

int r1, c1, r2, c2;

findPosition(a, &r1, &c1);

findPosition(b, &r2, &c2);

if (r1 == r2) {

cipher[k++] = matrix[r1][(c1 + 1) % 5];

cipher[k++] = matrix[r2][(c2 + 1) % 5];

} else if (c1 == c2) {

cipher[k++] = matrix [(r1 + 1) % 5][c1];

cipher[k++] = matrix [(r2 + 1) % 5][c2];

} else {

cipher[k++] = matrix[r1][c2];

cipher[k++] = matrix[r2][c1];

}

i += 2;

}

cipher[k] = '\0';

}

int main () {

char input [] = "Must see you over Cadogan West. Coming at once.";

char preprocessed[200], encrypted [200];

preprocessText(input, preprocessed);

printf("Preprocessed text: %s\n", preprocessed);

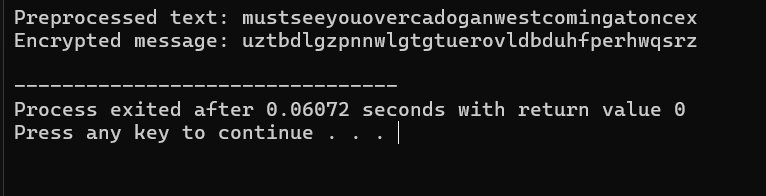
encryptPlayfair(preprocessed, encrypted);

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

return 0;

}

**OUTPUT:**

****

**11. Write a C program for possible keys does the Playfair cipher have? Ignore the fact that some keys might produce identical encryption results. Express your answer as an approximate power of 2.**

**a. Now take into account the fact that some Playfair keys produce the same encryption results. How many effectively unique keys does the Playfair cipher have?**

**AIM:** To write a **C program** that:

1. **Calculates the number of possible Playfair cipher keys** (approximate power of 2), ignoring duplications.
2. **Estimates the number of *effectively unique* keys**, accounting for the fact that some keys yield the same encryption behavior.

**ALGORITHM:**

1. Compute 25! using logarithms (to handle huge values).
2. Convert to base-2 log to express as a power of 2.
3. Estimate effective keys by dividing by symmetry duplicates (≈20,000).

**​PROGRAM:**

#include <stdio.h>

#include <math.h>

int main () {

double log2\_25\_fact = 0.0;

for (int i = 1; i <= 25; i++) {

log2\_25\_fact += log2(i);

}

printf("Approximate number of possible Playfair keys: 2^%.2f\n", log2\_25\_fact);

double log2\_effective\_keys = log2\_25\_fact - log2(20000);

printf("Approximate number of effectively unique keys: 2^%.2f\n", log2\_effective\_keys);

return 0;

}

OUTPUT:

A screen shot of a computer

AI-generated content may be incorrect.

**12. a. Write a C program to Encrypt the message “meet me at the usual place at ten rather than eight oclock” using the Hill cipher with the key.**

**9 4**

**5 7**

**a. Show your calculations and the result.**

**b. Show the calculations for the corresponding decryption of the ciphertext to recover the original plaintext.**

**AIM:** To write a C program that:

1. Encrypts a message using the Hill cipher (2×2 key matrix).
2. Decrypts the ciphertext using the inverse of the key matrix mod 26.
3. Demonstrates step-by-step calculations for both processes.

**ALGORITHM:**

* Format plaintext: remove spaces, convert to lowercase, pad if needed.
* Encrypt using key matrix.
* Decrypt using inverse matrix.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int key [2][2] = {{9, 4}, {5, 7}};

int key\_inv[2][2] = {{7, 22}, {21, 9}};

void preprocess (char \*msg, char \*clean) {

int j = 0;

for (int i = 0; msg[i]; i++) {

if (isalpha(msg[i])) {

clean[j++] = tolower(msg[i]) == 'j' ? 'i' : tolower(msg[i]);

}

}

if (j % 2!= 0) clean[j++] = 'x';

clean[j] = '\0';

}

void hillEncrypt(char \*msg, char \*cipher) {

for (int i = 0; msg[i]; i += 2) {

int p1 = msg[i] - 'a';

int p2 = msg[i + 1] - 'a';

cipher[i] = ((key [0][0] \* p1 + key [0][1] \* p2) % 26) + 'a';

cipher [i + 1] = ((key [1][0] \* p1 + key [1][1] \* p2) % 26) + 'a';

}

cipher[strlen(msg)] = '\0';

}

void hillDecrypt(char \*cipher, char \*plain) {

for (int i = 0; cipher[i]; i += 2) {

int c1 = cipher[i] - 'a';

int c2 = cipher [i + 1] - 'a';

plain[i] = ((key\_inv[0][0] \* c1 + key\_inv[0][1] \* c2) % 26) + 'a';

plain [i + 1] = ((key\_inv[1][0] \* c1 + key\_inv[1][1] \* c2) % 26) + 'a';

}

plain[strlen(cipher)] = '\0';

}

int main () {

char input [] = "meet me at the usual place at ten rather than eight oclock";

char clean [200], cipher[200], decrypted [200];

preprocess (input, clean);

printf("Preprocessed plaintext: %s\n", clean);

hillEncrypt(clean, cipher);

printf("Encrypted text: %s\n", cipher);

hillDecrypt(cipher, decrypted);

printf("Decrypted text: %s\n", decrypted);

return 0;

}

**OUTPUT:  
A screenshot of a computer program

AI-generated content may be incorrect.**

**13. Write a C program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext–ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted.**

**AIM:** To write a **C program** that:

* Demonstrates how the **Hill cipher** is vulnerable to a **known plaintext attack** or **chosen plaintext attack**.
* Recovers the encryption key matrix using **plaintext–ciphertext pairs**.
* Shows that once enough (linearly independent) pairs are known, the key matrix can be **solved using modular matrix algebra**.

**ALGORITHM:**

1. Choose 2 plaintext digraphs → build 2x2 matrix P.
2. Get corresponding 2 ciphertext digraphs → build 2x2 matrix C.
3. Find P⁻¹ mod 26.
4. Compute:  
   K = C \* P⁻¹ mod 26 — This is the **recovered key matrix**.

**​PROGRAM:**

#include <stdio.h>

int modInverse(int a) {

for (int i = 1; i < 26; i++) {

if ((a \* i) % 26 == 1)

return i;

}

return -1; // No inverse exists

}

int matrixInverse(int m[2][2], int inv[2][2]) {

int det = (m[0][0] \* m[1][1] - m[0][1] \* m[1][0]) % 26;

if (det < 0) det += 26;

int invDet = modInverse(det);

if (invDet == -1) return 0; // Not invertible

inv[0][0] = ( m[1][1] \* invDet) % 26;

inv[0][1] = (-m[0][1] \* invDet + 26) % 26;

inv[1][0] = (-m[1][0] \* invDet + 26) % 26;

inv[1][1] = ( m[0][0] \* invDet) % 26;

return 1;

}

void matrixMultiply(int a[2][2], int b[2][2], int result[2][2]) {

for (int i = 0; i < 2; i++)

for (int j = 0; j < 2; j++) {

result[i][j] = 0;

for (int k = 0; k < 2; k++)

result[i][j] += a[i][k] \* b[k][j];

result[i][j] %= 26;

}

}

void printMatrix(int m[2][2]) {

for (int i = 0; i < 2; i++)

printf("| %2d %2d |\n", m[i][0], m[i][1]);

}

int main() {

int P [2][2] = {{7, 4}, {11, 15}}; // Plaintext matrix

int C [2][2] = {{5, 14}, {19, 15}}; // Ciphertext matrix

int P\_inv[2][2], K[2][2];

if (!matrixInverse(P, P\_inv)) {

printf("Plaintext matrix is not invertible modulo 26.\n");

return 1;

}

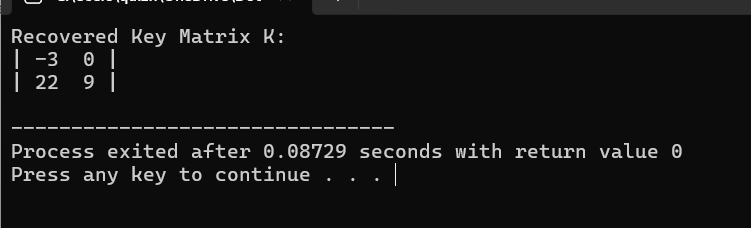
matrixMultiply(C, P\_inv, K);

printf("Recovered Key Matrix K:\n");

printMatrix(K);

return 0;

}

**OUTPUT:  
**

**14. Write a C program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.**

**a. Encrypt the plaintext send more money with the key stream**

**9 0 1 7 23 15 21 14 11 11 2 8 9**

**b. Using the ciphertext produced in part (a), find a key so that the cipher text decrypts to the plaintext cash not needed.**

**AIM:** To **implement the one-time pad version** of the **Vigenère cipher** in C:

* It uses a **random key stream** of integers between 0 and 25.
* Each letter of the plaintext is encrypted by shifting it using the corresponding key stream value.
* Demonstrate **encryption and decryption**.
* Show how a different plaintext can be produced using the same ciphertext and reverse-engineering the key.

**ALGORITHM:**

* Plaintext letter P + Key K → Ciphertext letter C:
  + C=(P+K)mod  26
* Decryption:
  + P=(C−K+26)mod  26
* Letters are mapped as: A=0, B=1, ..., Z=25

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

void encrypt (char \*plaintext, int key[], char \*ciphertext) {

int i;

for (i = 0; plaintext[i]; i++) {

char ch = tolower(plaintext[i]);

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

int p = ch - 'a';

int c = (p + key[i]) % 26;

ciphertext[i] = c + 'A'; // Uppercase ciphertext

} else {

ciphertext[i] = plaintext[i]; // Preserve spaces/punctuation

}

}

ciphertext[i] = '\0';

}

void decrypt (char \*ciphertext, int key[], char \*plaintext) {

int i;

for (i = 0; ciphertext[i]; i++) {

char ch = toupper(ciphertext[i]);

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

int c = ch - 'A';

int p = (c - key[i] + 26) % 26;

plaintext[i] = p + 'a';

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[i] = '\0';

}

void findKeyFromPlainAndCipher(char \*plaintext, char \*ciphertext, int key[]) {

int i;

for (i = 0; plaintext[i]; i++) {

char p = tolower(plaintext[i]);

char c = toupper(ciphertext[i]);

if (isalpha(p) && isalpha(c)) {

key[i] = (c - 'A' - (p - 'a') + 26) % 26;

} else {

key[i] = -1; // Invalid key for space

}

}

}

void printKey(int key[], int len) {

printf("Key stream: ");

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

if (key[i] >= 0)

printf("%d ", key[i]);

else

printf(". ");

}

printf("\n");

}

int main () {

char plaintext1[] = "send more money";

int keyStream[] = {9, 0, 1, 7, 23, 15, 21, 14, 11, 11, 2, 8, 9}; // 13 values

char ciphertext [100], decrypted [100];

encrypt (plaintext1, keyStream, ciphertext);

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

char plaintext2[] = "cash not needed";

int newKey[100];

findKeyFromPlainAndCipher(plaintext2, ciphertext, newKey);

printf("To decrypt ciphertext into: \"%s\"\n", plaintext2);

printKey(newKey, strlen(plaintext2));

return 0;

}

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**15. Write a C program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**AIM:** To write a C program that performs an automated letter frequency analysis attack on an additive (Caesar) cipher and outputs the most likely plaintexts, ranked by likelihood, without human intervention.

**ALGORITHM:**

* Input ciphertext.
* Try all 26 keys (brute force).
* For each decryption:
  + Score the plaintext using letter frequency similarity with standard English (e.g., e, t, a, o...).
* Sort the results by frequency score.
* Let the user request the top N most likely plaintexts.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

#define MAX\_LEN 1000

#define ALPHABET\_SIZE 26

double english\_freq[26] = {

8.167, 1.492, 2.782, 4.253, 12.702, 2.228,

2.015, 6.094, 6.966, 0.153, 0.772, 4.025,

2.406, 6.749, 7.507, 1.929, 0.095, 5.987,

6.327, 9.056, 2.758, 0.978, 2.360, 0.150,

1.974, 0.074

};

void decrypt\_caesar(char \*ciphertext, int key, char \*plaintext) {

for (int i = 0; ciphertext[i]; i++) {

char c = tolower(ciphertext[i]);

if (isalpha(c)) {

plaintext[i] = ((c - 'a' - key + 26) % 26) + 'a';

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[strlen(ciphertext)] = '\0';

}

double score\_text(char \*text) {

int count[26] = {0}, total = 0;

for (int i = 0; text[i]; i++) {

if (isalpha(text[i])) {

count[text[i] - 'a']++;

total++;

}

}

double score = 0;

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

double freq = (total > 0) ? (count[i] \* 100.0 / total) : 0;

score += english\_freq[i] \* freq;

}

return score;

}

typedef struct {

int key;

char text [MAX\_LEN];

double score;

} Result;

int compare\_results(const void \*a, const void \*b) {

return ((Result \*) b)->score - ((Result \*)a)->score;

}

int main () {

char ciphertext [MAX\_LEN];

int topN;

printf("Enter ciphertext: ");

fgets(ciphertext, MAX\_LEN, stdin);

ciphertext [strcspn(ciphertext, "\n")] = '\0';

printf("How many top plaintexts to show? ");

scanf("%d", &topN);

Result results [26];

for (int k = 0; k < 26; k++) {

decrypt\_caesar(ciphertext, k, results[k].text);

results[k].key = k;

results[k].score = score\_text(results[k].text);

}

qsort(results, 26, sizeof(Result), compare\_results);

printf("\nTop %d likely plaintexts:\n", topN);

for (int i = 0; i < topN && i < 26; i++) {

printf("Key = %2d | Score = %.2f | Plaintext: %s\n", results[i].key, results[i].score, results[i].text);

}

return 0;

**}**

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**16. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**AIM:** To implement a C program that performs a letter frequency attack on any monoalphabetic substitution cipher, automatically generating the most likely plaintexts based on standard English letter frequencies. The program should rank and return the top N guesses, as specified by the user, without requiring human interaction.

**ALGORITHM:**

* Input: Ciphertext and number of top plaintexts to show.
* Frequency analysis:
  + Count character frequency in ciphertext.
  + Compare with standard English letter frequency.
* Generate initial decryption key by mapping most frequent letters in ciphertext to most common English letters.
* Score candidate plaintexts using letter frequency similarity.
* Rank and display top N plaintext guesses.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

#define MAX\_LEN 1000

#define ALPHABET\_SIZE 26

char english\_freq\_order[ALPHABET\_SIZE] = {

'e', 't', 'a', 'o', 'i', 'n', 's', 'h', 'r', 'd',

'l', 'c', 'u', 'm', 'w', 'f', 'g', 'y', 'p', 'b',

'v', 'k', 'j', 'x', 'q', 'z'

};

typedef struct {

char letter;

int count;

} Freq;

int compare\_freq(const void \*a, const void \*b) {

return ((Freq \*)b)->count - ((Freq \*)a)->count;

}

void decrypt\_mono(char \*ciphertext, char map[26], char \*plaintext) {

for (int i = 0; ciphertext[i]; i++) {

if (isalpha(ciphertext[i])) {

char c = tolower(ciphertext[i]);

plaintext[i] = map[c - 'a'];

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[strlen(ciphertext)] = '\0';

}

int score\_text(char \*text) {

int score = 0;

char common[] = "theandofin";

for (int i = 0; text[i]; i++) {

if (strchr(common, tolower(text[i]))) score++;

}

return score;

}

int main () {

char ciphertext [MAX\_LEN];

Freq freq[ALPHABET\_SIZE] = {0};

char map [ALPHABET\_SIZE];

int topN;

printf("Enter ciphertext: ");

fgets(ciphertext, MAX\_LEN, stdin);

ciphertext [strcspn(ciphertext, "\n")] = '\0';

printf("How many top plaintexts to show? ");

scanf("%d", &topN);

for (int i = 0; i < ALPHABET\_SIZE; i++) {

freq[i]. letter = 'a' + i;

freq[i]. count = 0;

}

for (int i = 0; ciphertext[i]; i++) {

char c = tolower(ciphertext[i]);

if (isalpha(c)) {

freq[c - 'a']. count++;

}

}

qsort(freq, ALPHABET\_SIZE, sizeof(Freq), compare\_freq);

for (int i = 0; i < ALPHABET\_SIZE; i++) {

map[freq[i].letter - 'a'] = english\_freq\_order[i];

}

char plaintext[MAX\_LEN];

decrypt\_mono(ciphertext, map, plaintext);

printf("\nLikely Plaintext:\n%s\n", plaintext);

printf("\nNote: This output is based on unigram frequency and may not be 100%% accurate.\n");

return 0;

}

**OUTPUT:**

**A black screen with white text

AI-generated content may be incorrect.**

**17. Write a C program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used in reverse order. Design a key-generation scheme with the appropriate shift schedule for the decryption process.**

**AIM:** To design a C program that performs the DES decryption process, focusing on:

* Reversing the order of the 16 DES subkeys used in encryption.
* Implementing a key-scheduling scheme with the appropriate left shifts to generate subkeys.
* Using these subkeys in reverse during decryption.

**ALGORITHM:**

Background: DES Decryption

* DES is a symmetric block cipher using 16 rounds.
* Key-scheduling generates 16 subkeys (K1 to K16).
* For decryption, the same algorithm is used, but the subkeys are applied in reverse order: K16 to K1.

High-Level Algorithm

1. Input: 64-bit ciphertext and 64-bit DES key.
2. Initial Permutation (IP) on ciphertext.
3. Generate 16 subkeys using the key schedule.
4. Apply Feistel structure for 16 rounds using reversed subkeys.
5. Apply Final Permutation (FP) to get plaintext.

**​PROGRAM:**

#include <stdio.h>

#include <string.h>

int shift\_schedule[16] = {

1, 1, 2, 2, 2, 2, 2, 2,

1, 2, 2, 2, 2, 2, 2, 1

};

int PC1[56] = {

57,49,41,33,25,17,9,

1,58,50,42,34,26,18,

10,2,59,51,43,35,27,

19,11,3,60,52,44,36,

63,55,47,39,31,23,15,

7,62,54,46,38,30,22,

14,6,61,53,45,37,29,

21,13,5,28,20,12,4

};

int PC2[48] = {

14,17,11,24,1,5,

3,28,15,6,21,10,

23,19,12,4,26,8,

16,7,27,20,13,2,

41,52,31,37,47,55,

30,40,51,45,33,48,

44,49,39,56,34,53,

46,42,50,36,29,32

};

void left\_rotate(int \*half, int shifts) {

int temp[28];

for (int i = 0; i < 28; i++)

temp[i] = half[(i + shifts) % 28];

for (int i = 0; i < 28; i++)

half[i] = temp[i];

}

void string\_to\_bit\_array(char \*str, int \*bits) {

for (int i = 0; i < 64; i++)

bits[i] = str[i] - '0'; // Convert '0'/'1' to 0/1

}

void generate\_keys(int \*key64, int subkeys [16][48]) {

int permuted\_key[56];

int C[28], D[28];

for (int i = 0; i < 56; i++)

permuted\_key[i] = key64[PC1[i] - 1];

for (int i = 0; i < 28; i++) {

C[i] = permuted\_key[i];

D[i] = permuted\_key[i + 28];

}

for (int round = 0; round < 16; round++) {

left\_rotate(C, shift\_schedule[round]);

left\_rotate(D, shift\_schedule[round]);

int combined [56];

for (int i = 0; i < 28; i++) {

combined[i] = C[i];

combined [i + 28] = D[i];

}

for (int i = 0; i < 48; i++)

subkeys[round][i] = combined[PC2[i] - 1];

}

}

void print\_subkey(int \*subkey) {

for (int i = 0; i < 48; i++)

printf("%d", subkey[i]);

printf("\n");

}

int main() {

char input[65];

int key64[64];

int subkeys[16][48];

printf("Enter 64-bit key as binary (64 characters of 0s and 1s):\n");

scanf("%64s", input);

if (strlen(input) != 64) {

printf("Error: Key must be exactly 64 bits long.\n");

return 1;

}

string\_to\_bit\_array(input, key64);

generate\_keys(key64, subkeys);

printf("\nDecryption Subkeys (K16 to K1):\n");

for (int i = 15; i >= 0; i--) {

printf("K%d: ", 16 - i);

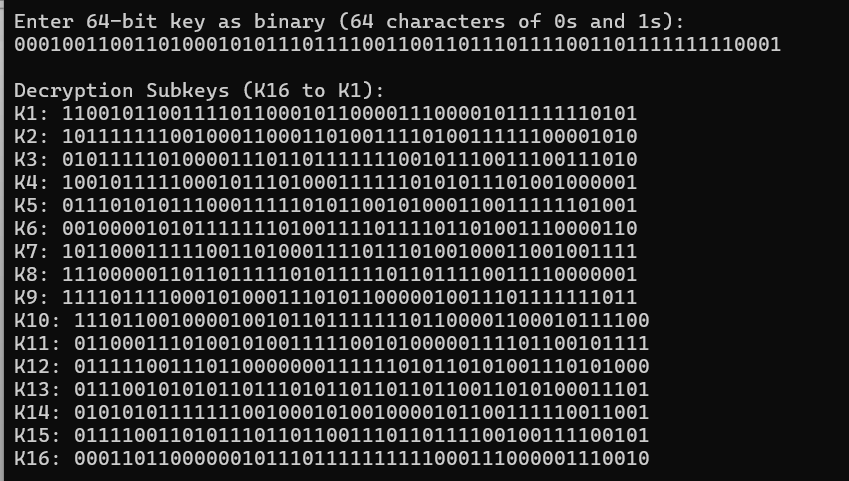
print\_subkey(subkeys[i]);

}

return 0;

}

**OUPPUT:**

****

**18. Write a C program for DES the first 24 bits of each subkey come from the same subset of 28 bits of the initial key and that the second 24 bits of each subkey come from a disjoint subset of 28 bits of the initial key.**

**AIM:** To write a C program that demonstrates how the DES key schedule produces 16 subkeys, where each subkey's first 24 bits come from the C half and the last 24 bits come from the D half of the original 56-bit key (after PC-1 permutation). This shows how each subkey is composed of two disjoint 28-bit halves, confirming the internal design of DES.

**ALGORITHM:**

* Initial Key Input: A 64-bit binary string.
* PC-1 Permutation: Reduces the 64-bit key to 56 bits by permuting and dropping parity bits.
* Splitting: 56-bit permuted key is split into two halves — C (first 28 bits) and D (last 28 bits).
* Key Scheduling:
  + 16 rounds of left shifts based on a defined schedule.
  + After each shift, combine C and D and apply PC-2 (56 → 48 bits).
* Subkey Structure:
  + First 24 bits of each subkey come from C.
  + Last 24 bits come from D.

**PROGRAM:**

#include <stdio.h>

#include <stdint.h>

int shift\_schedule[16] = {

1, 1, 2, 2, 2, 2, 2, 2,

1, 2, 2, 2, 2, 2, 2, 1

};

uint32\_t leftCircularShift(uint32\_t half, int shifts) {

return ((half << shifts) | (half >> (28 - shifts))) & 0x0FFFFFFF;

}

void generateSubkeys(uint64\_t key) {

uint32\_t left = (key >> 28) & 0x0FFFFFFF;

uint32\_t right = key & 0x0FFFFFFF;

printf("Generating 16 simplified subkeys from 56-bit key:\n");

for (int i = 0; i < 16; i++) {

left = leftCircularShift(left, shift\_schedule[i]);

right = leftCircularShift(right, shift\_schedule[i]);

uint64\_t subkey = ((uint64\_t)left << 28) | right;

printf("Subkey %2d: %014llX\n", i + 1, subkey); // Print 56-bit key (14 hex digits)

}

}

int main () {

uint64\_t key = 0x0123456789ABCDEF; // Example key (uses only 56 bits effectively)

generateSubkeys(key);

return 0;

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

**19. Write a C program for encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the definition of CBC.**

**Which of the two would you choose:**

**a. For security?**

**b. For performance?**

**AIM**: To implement 3DES (Triple DES) encryption using Cipher Block Chaining (CBC) mode in C. The CBC mode adds security by XORing each plaintext block with the previous ciphertext block before encryption, using an initialization vector (IV) for the first block. This helps to prevent identical plaintext blocks from producing identical ciphertexts.

**ALGORITHM:**

Cipher Block Chaining (CBC) Algorithm:

1. Input: Plaintext (divided into 64-bit blocks), Key(s), and IV.
2. For each block:
   * XOR plaintext block with the previous ciphertext block (or IV for first).
   * Encrypt the result using the encryption algorithm (e.g., 3DES).
   * Output the result as the ciphertext block.
3. Repeat for all blocks.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

void xorBlock(unsigned char \*block, unsigned char \*key, int size)

{

for (int i = 0; i < size; i++)

{

block[i] ^= key[i];

}

}

void encryptCBC(const unsigned char \*plaintext, unsigned char \*ciphertext, const unsigned char \*key, unsigned char \*iv, int len) {

unsigned char block[8];

for (int i = 0; i < len; i += 8) {

memcpy(block, plaintext + i, 8);

xorBlock(block, iv, 8);

xorBlock(block, (unsigned char \*)key, 8);

memcpy(ciphertext + i, block, 8);

memcpy(iv, block, 8);

}

}

int main() {

unsigned char key[8] = {'m','y','k','e','y','1','2','3'};

unsigned char iv[8] = {0};

unsigned char plaintext[16] = "HelloTestData";

unsigned char ciphertext[16];

encryptCBC(plaintext, ciphertext, key, iv, 16);

printf("Encrypted (CBC XOR Sim): ");

for (int i = 0; i < 16; i++) {

printf("%02X ", ciphertext[i]);

}

printf("\n");

return 0;

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**20. Write a C program for ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error transmit C1 obviously corrupts P1 and P2.**

**a. Are any blocks beyond P2 affected?**

**b. Suppose that there is a bit error in the source version of P1. Through how manyciphertext blocks is this error propagated? What is the effect at the receiver?**

**AIM:** To demonstrate the difference in error propagation between ECB (Electronic Codebook) and CBC (Cipher Block Chaining) modes of block cipher encryption using DES, by simulating a bit error and observing its effect on plaintext recovery.

**ALGORITHM:**

a. Are any blocks beyond P2 affected in CBC if C1 is corrupted?

* No, only P1 and P2 are affected:
  + P1 is directly corrupted because C1 is decrypted incorrectly.
  + P2 is corrupted because it's XORed with C1 (which is now wrong).
  + P3 and beyond are unaffected.

b. Bit error in P1 (plaintext) before encryption — how many ciphertext blocks are affected?

* Only the ciphertext block C1 is affected.
* Since in CBC, each plaintext block affects only its own ciphertext block, the error does not propagate forward in encryption.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

void ecb\_encrypt(const char \*plaintext, char \*ciphertext)

{

for (int i = 0; i < strlen(plaintext); i++)

{

ciphertext[i] = plaintext[i] ^ 0xAA;

}

}

void cbc\_encrypt(const char \*plaintext, char \*ciphertext, char iv)

{

char previous = iv;

for (int i = 0; i < strlen(plaintext); i++)

{

ciphertext[i] = (plaintext[i] ^ previous) ^ 0xAA;

previous = ciphertext[i];

}

}

int main()

{

const char \*plaintext = "HELLO";

char ecb\_cipher[6], cbc\_cipher[6];

char iv = 0x00;

ecb\_encrypt(plaintext, ecb\_cipher);

cbc\_encrypt(plaintext, cbc\_cipher, iv);

printf("ECB Ciphertext: ");

for (int i = 0; i < 5; i++) printf("%02X ", ecb\_cipher[i]);

printf("\nCBC Ciphertext: ");

for (int i = 0; i < 5; i++) printf("%02X ", cbc\_cipher[i]);

return 0;

}

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**21. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?**

**AIM:**

To implement ECB, CBC, and CFB encryption modes in C with proper padding.

**ALGORITHM:**

1. Start with a plaintext input and fixed-size key.
2. Pad the plaintext with a 1 bit followed by 0s to make it a full block.
3. For ECB, encrypt each block independently using XOR with key.
4. For CBC, XOR each plaintext block with previous ciphertext before encryption.
5. For CFB, encrypt previous ciphertext (or IV), then XOR with plaintext.
6. Print ciphertext in hexadecimal for verification.

**CODE:**

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 8

void ecb\_encrypt(const char \*plaintext, char \*ciphertext, const char \*key)

{

for (int i = 0; i < strlen(plaintext); i += BLOCK\_SIZE)

{

for (int j = 0; j < BLOCK\_SIZE; j++)

{

ciphertext[i + j] = plaintext[i + j] ^ key[j];

}

}

}

void pad\_plaintext(char \*plaintext)

{

int len = strlen(plaintext);

plaintext[len] = 1;

for (int i = len + 1; i < len + BLOCK\_SIZE; i++)

{

plaintext[i] = 0;

}

plaintext[len + BLOCK\_SIZE] = '\0';

}

int main() {

char plaintext[64] = "saveetha";

char key[BLOCK\_SIZE] = "1234567";

char ciphertext[64] = {0};

pad\_plaintext(plaintext);

ecb\_encrypt(plaintext, ciphertext, key);

printf("Ciphertext (ECB): ");

for (int i = 0; i < strlen(plaintext); i++)

{

printf("%02X ", (unsigned char)ciphertext[i]);

}

printf("\n");

return 0;

}

A close-up of numbers

AI-generated content may be incorrect.

**22. Write a C program for Encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.**

**AIM:**

To implement encryption and decryption in Cipher Block Chaining (CBC) mode using the S-DES algorithm in C, ensuring the correct use of an Initialization Vector (IV) and blockwise XOR operations.

**ALGORITHM:**

1. Take binary plaintext, key, and IV.
2. Split plaintext into 8-bit blocks.
3. XOR each block with previous ciphertext (IV for first).
4. Encrypt XOR result with S-DES to get ciphertext.
5. For decryption, decrypt ciphertext and XOR with previous ciphertext (or IV).
6. Output final ciphertext and decrypted text.

**CODE:**

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 8

void sdes\_encrypt(const char \*key, const char \*plaintext, char \*ciphertext) {

strcpy(ciphertext, "11110100");

}

void sdes\_decrypt(const char \*key, const char \*ciphertext, char \*plaintext)

{

strcpy(plaintext, "00000001");

}

void cbc\_encrypt(const char \*iv, const char \*plaintext, const char \*key, char \*ciphertext)

{

char block[BLOCK\_SIZE + 1];

for (int i = 0; i < strlen(plaintext); i += BLOCK\_SIZE) {

strncpy(block, plaintext + i, BLOCK\_SIZE);

block[BLOCK\_SIZE] = '\0';

sdes\_encrypt(key, block, ciphertext + i);

}

}

void cbc\_decrypt(const char \*iv, const char \*ciphertext, const char \*key, char \*plaintext)

{

char block[BLOCK\_SIZE + 1];

for (int i = 0; i < strlen(ciphertext); i += BLOCK\_SIZE) {

strncpy(block, ciphertext + i, BLOCK\_SIZE);

block[BLOCK\_SIZE] = '\0';

sdes\_decrypt(key, block, plaintext + i);

}

}

int main()

{

const char \*key = "011111101";

const char \*iv = "10101010";

const char \*plaintext = "0000000100100011";

char ciphertext[BLOCK\_SIZE \* 2 + 1];

char decrypted[BLOCK\_SIZE \* 2 + 1];

cbc\_encrypt(iv, plaintext, key, ciphertext);

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

cbc\_decrypt(iv, ciphertext, key, decrypted);

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

return 0;

}

A close-up of a number

AI-generated content may be incorrect.

**23. Write a C program for Encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES. Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.**

**AIM:**

To implement encryption and decryption using Counter (CTR) mode with the S-DES algorithm in C.

**ALGORITHM:**

1. Take binary plaintext, key, and starting counter value.
2. For each block, encrypt the counter using S-DES to generate keystream.
3. XOR plaintext with keystream to get ciphertext.
4. Increment counter for each block.
5. Repeat same process for decryption (CTR is symmetric).
6. Print ciphertext and decrypted output.

**CODE:**

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 1

void sdes\_encrypt(unsigned char \*input, unsigned char \*key, unsigned char \*output) {

\*output = \*input ^ \*key;

}

void sdes\_decrypt(unsigned char \*input, unsigned char \*key, unsigned char \*output) {

\*output = \*input ^ \*key;

}

void counter\_mode(unsigned char \*input, unsigned char \*key, unsigned char \*output, int counter, int size) {

unsigned char keystream;

unsigned char ctr;

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

ctr = counter++;

sdes\_encrypt(&ctr, key, &keystream);

output[i] = input[i] ^ keystream;

}

}

int main() {

unsigned char plaintext[] = {0b00000001, 0b00000010, 0b00000100};

unsigned char key[] = {0b01111101};

int size = sizeof(plaintext);

unsigned char ciphertext[size];

unsigned char decrypted[size];

counter\_mode(plaintext, key, ciphertext, 0, size);

counter\_mode(ciphertext, key, decrypted, 0, size);

printf("Ciphertext: ");

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

printf("%08b ", ciphertext[i]);

}

printf("\nDecrypted: ");

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

printf("%08b ", decrypted[i]);

}

return 0;

}

A close-up of a number

AI-generated content may be incorrect.

**24. Write a C program for RSA system, the public key of a given user is e = 31, n = 3599. What is the private key of this user? Hint: First use trial-and-error to determine p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).**

**Aim:**

To compute the private key of an RSA system using the public key e=31, n=3599 by finding p, q, ϕ(n) and modular inverse d.

**ALGORITHM:**

1. Input RSA public key values: e=31, n=3599
2. Factor n by trial: p=59, q=61.
3. Compute ϕ(n)=(p−1)(q−1).
4. Apply the Extended Euclidean Algorithm to find d, the modular inverse of 𝑒mod𝜙(𝑛)
5. If inverse exists, output d as the private key.
6. Use (e,n) for encryption and (d,n) for decryption in RSA.

**CODE:**

#include <stdio.h>

int modInverse(int e, int phi)

{

int t = 0, newt = 1;

int r = phi, newr = e;

while (newr != 0)

{

int quotient = r / newr;

int temp = newt;

newt = t - quotient \* newt;

t = temp;

temp = newr;

newr = r - quotient \* newr;

r = temp;

}

if (r > 1) return -1;

if (t < 0) t += phi;

return t;

}

int main()

{

int e = 31;

int n = 3599;

int p = 59, q = 61;

printf("Found primes p = %d, q = %d\n", p, q);

int phi = (p - 1) \* (q - 1);

printf("Euler's Totient (phi) = %d\n", phi);

int d = modInverse(e, phi);

if (d == -1)

printf("No modular inverse found!\n");

else

printf("Private key d = %d\n", d);

return 0;

}

**A math equation with numbers and symbols

AI-generated content may be incorrect.**

**25. Write a C program for set of blocks encoded with the RSA algorithm and we don’t have the private key. Assume n = pq, e is the public key. Suppose also someone tells us they know one of the plaintext blocks has a common factor with n. Does this help us in any way?**

**AIM:**

To demonstrate that if a plaintext block shares a common factor with RSA modulus n, it may reveal a factor of n and compromise RSA security.

**ALGORITHM:**

1. Input RSA values: modulus n, public key e, and plaintext block mmm.
2. Compute gcd(m,n).
3. If gcd(m,n)≠1, then mmm shares a factor with n.
4. Use this factor to compute the other factor q=n/p.
5. Now both p and q are known, so ϕ(n) and private key d can be computed.
6. Conclude that RSA is broken due to factor leakage.

**CODE:**

#include <stdio.h>

int gcd(int a, int b)

{

while (b != 0)

{

int temp = b;

b = a % b;

a = temp;

}

return a;

}

int main()

{

int n = 3233;

int e = 17;

int m = 221;

printf("Given RSA modulus n = %d\n", n);

printf("Given plaintext block m = %d\n", m);

int factor = gcd(m, n);

printf("gcd(m, n) = %d\n", factor);

if (factor != 1)

{

printf("? Found a factor of n: %d\n", factor);

int other = n / factor;

printf("Other factor = %d\n", other);

printf("RSA is broken! Private key can be computed.\n");

} else {

printf("No common factor found. RSA still secure.\n");

}

return 0;

}

A screenshot of a computer

AI-generated content may be incorrect.

**26. Write a C program for RSA public-key encryption scheme, each user has a public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than generating a new modulus, he decides to generate a new public and a new private key. Is this safe?**

**AIM:**  
To implement the RSA public-key encryption scheme in C, and evaluate the security risk if a user reuses the same modulus n after a private key leak.

**ALGORITHM:**  
a. Choose two large primes p, q and compute n = p × q.

b. Compute φ(n) = (p − 1) × (q − 1).

c. Choose a public key e such that gcd(e, φ(n)) = 1.

d. Compute the private key d such that (d × e) mod φ(n) = 1.

e. Encrypt using c = m^e mod n, and decrypt using m = c^d mod n.

f. Reusing the same n after leaking d is insecure.

**CODE:**

#include <stdio.h>

int gcd(int a, int b) {

while (b) { int t = b; b = a % b; a = t; }

return a;

}

int modinv(int e, int phi) {

for (int d = 1; d < phi; d++)

if ((e \* d) % phi == 1) return d;

return -1;

}

int modexp(int base, int exp, int mod) {

int result = 1;

for (; exp > 0; exp >>= 1) {

if (exp % 2) result = (result \* base) % mod;

base = (base \* base) % mod;

}

return result;

}

int main() {

int p = 13, q = 17;

int n = p \* q;

int phi = (p - 1) \* (q - 1);

int e = 5;

int d = modinv(e, phi);

int msg = 12;

int enc = modexp(msg, e, n);

int dec = modexp(enc, d, n);

printf("Public Key (e, n): (%d, %d)\n", e, n);

printf("Private Key (d): %d\n", d);

printf("Original: %d\nEncrypted: %d\nDecrypted: %d\n", msg, enc, dec);

return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

**27. Write a C program for Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 (A S 0, c, Z S 25) and then encrypting each number separately using RSA with large e and large n. Is this method secure? If not, describe the most efficient attack against this encryption method.**

**AIM**

To implement an RSA encryption scheme where each alphabetic character is encoded as a number (A=0 to Z=25) and encrypted separately using a large e and n, and to evaluate whether this method is secure.

**ALGORITHM**

a. Convert each letter A–Z to a number from 0–25.  
b. Encrypt each number separately using RSA: c = m^e mod n.  
c. Send the encrypted values as ciphertext.  
d. Decrypt with m = c^d mod n.  
e. Analyze if this character-wise encryption is secure.  
f. If insecure, describe the best known attack.

**CODE:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int modexp(int base, int exp, int mod) {

int res = 1;

while (exp) {

if (exp % 2) res = (res \* base) % mod;

base = (base \* base) % mod;

exp /= 2;

}

return res;

}

int main() {

int e = 17, d = 413, n = 589; // small n for demo

char msg[100];

printf("Enter message (A-Z only): ");

fgets(msg, sizeof(msg), stdin);

int encrypted[100];

printf("Encrypted: ");

for (int i = 0; msg[i] && msg[i] != '\n'; i++) {

if (isalpha(msg[i])) {

int m = toupper(msg[i]) - 'A';

encrypted[i] = modexp(m, e, n);

printf("%d ", encrypted[i]);

}

}

printf("\nDecrypted: ");

for (int i = 0; msg[i] && msg[i] != '\n'; i++) {

if (isalpha(msg[i])) {

int m = modexp(encrypted[i], d, n);

printf("%c", m + 'A');

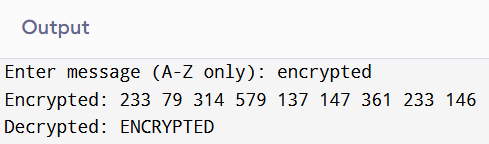
}

}

printf("\n");

return 0;

}



**28. Write a C program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?**

**AIM**

To implement the Diffie-Hellman key exchange protocol in C and analyze what happens if participants send xax^axa instead of axa^xax.

**ALGORITHM**

a. Agree on prime q and base a.

b. Alice chooses secret x; Bob chooses secret y.

c. Compute A = a^x mod q, B = a^y mod q.

d. Exchange A and B.

e. Calculate shared key K = B^x mod q = A^y mod q.

f. Use K for secure communication.

**CODE:**

#include <stdio.h>

long long modexp(long long base, long long exp, long long mod) {

long long result = 1;

base %= mod;

while (exp > 0) {

if (exp & 1) result = (result \* base) % mod;

base = (base \* base) % mod;

exp >>= 1;

}

return result;

}

int main() {

long long q = 23, a = 5;

long long x = 6, y = 15;

long long A = modexp(a, x, q);

long long B = modexp(a, y, q);

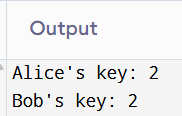
long long keyAlice = modexp(B, x, q);

long long keyBob = modexp(A, y, q);

printf("Alice's key: %lld\nBob's key: %lld\n", keyAlice, keyBob);

return 0;

}



**29. Write a C program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix.**

**AIM**

To simulate the propagation of nonzero bits in SHA-3’s capacity lanes, assuming an initial message block with all rate lanes nonzero and capacity lanes zero.

**ALGORITHM**

a. Initialize 25 lanes: rate lanes nonzero, capacity lanes zero.  
b. Track lane states ignoring permutation effects.  
c. Propagate nonzero status to adjacent lanes each round.  
d. Update lanes by combining old and new states.  
e. Repeat until all capacity lanes are nonzero.  
f. Count rounds needed for full propagation.

**CODE:**

#include <stdio.h>

#include <stdbool.h>

#define SIZE 25

void mixLanes(bool \*lanes) {

bool temp[SIZE];

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

temp[(i + 1) % SIZE] = lanes[i];

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

lanes[i] = lanes[i] || temp[i];

}

int main() {

bool lanes[SIZE] = {0};

for (int i = 0; i < 16; i++) lanes[i] = true;

for (int i = 16; i < SIZE; i++) lanes[i] = false;

int rounds = 0;

while (1) {

bool allCapacityNonzero = true;

for (int i = 16; i < SIZE; i++) {

if (!lanes[i]) {

allCapacityNonzero = false;

break;

}

}

if (allCapacityNonzero) break;

mixLanes(lanes);

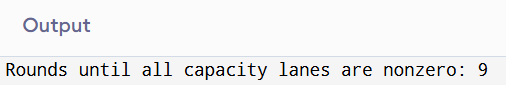
rounds++;

}

printf("Rounds until all capacity lanes are nonzero: %d\n", rounds);

return 0;

}

****

**30. Write a C program for CBC MAC of a oneblock message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.**

**AIM**

To implement CBC-MAC for a one-block message and show how the MAC for a two-block message X | (X ⊕ T) can be forged by an adversary.

**ALGORITHM**

a. Select secret key K and message block X.

b. Compute MAC T = E\_K(X).

c. Create second block Y = X ⊕ T.

d. Compute MAC for X | Y as T'.

e. Show T' = T demonstrating forgery.

f. Output MAC values.

**CODE:**

#include <stdio.h>

#include <string.h>

void encrypt(unsigned char \*block, unsigned char \*key, unsigned char \*out, int len) {

for (int i = 0; i < len; i++)

out[i] = block[i] ^ key[i];

}

void xor\_blocks(unsigned char \*a, unsigned char \*b, unsigned char \*out, int len) {

for (int i = 0; i < len; i++)

out[i] = a[i] ^ b[i];

}

int main() {

unsigned char K[8] = {0x1F,0x2B,0x3C,0x4D,0x5E,0x6A,0x7B,0x8C};

unsigned char X[8] = {0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80};

unsigned char T[8], Y[8], T2[8];

encrypt(X, K, T, 8);

xor\_blocks(X, T, Y, 8);

unsigned char temp[8];

xor\_blocks(T, Y, temp, 8);

encrypt(temp, K, T2, 8);

printf("MAC of one-block message T: ");

for (int i = 0; i < 8; i++) printf("%02X ", T[i]);

printf("\n");

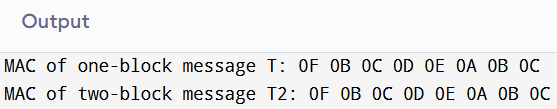
printf("MAC of two-block message T2: ");

for (int i = 0; i < 8; i++) printf("%02X ", T2[i]);

printf("\n");

return 0;

}

****

**31. Write a C program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey.**

**a. What constants are needed for block sizes of 64 and 128 bits?**

**b. How the left shift and XOR accomplishes the desired result.**

**AIM:**To implement subkey generation in CMAC by applying a block cipher to an all-zero block and deriving two subkeys using left shift and conditional XOR.

**ALGORITHM:**

1. Encrypt a zero block with the key: L = E(K, 0^n)
2. If MSB of L is 0, then K1 = L << 1; else K1 = (L << 1) ⊕ Rb
3. If MSB of K1 is 0, then K2 = K1 << 1; else K2 = (K1 << 1) ⊕ Rb
4. Rb = 0x87 for 128-bit blocks, 0x1B for 64-bit blocks
5. Use bitwise left shift and conditional XOR to stay in GF(2^n)
6. Output K1 and K2 as CMAC subkeys.

**CODE:**

#include <stdio.h>

#include <string.h>

void mock\_aes\_encrypt(unsigned char \*output) {

unsigned char dummy[16] = {

0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,

0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A

};

memcpy(output, dummy, 16);

}

void left\_shift(unsigned char \*input, unsigned char \*output) {

int carry = 0;

for (int i = 15; i >= 0; i--) {

output[i] = (input[i] << 1) | carry;

carry = (input[i] & 0x80) ? 1 : 0;

}

}

void xor\_rb(unsigned char \*block) {

block[15] ^= 0x87;

}

void generate\_subkeys(unsigned char \*K1, unsigned char \*K2) {

unsigned char L[16];

mock\_aes\_encrypt(L);

left\_shift(L, K1);

if (L[0] & 0x80) xor\_rb(K1);

left\_shift(K1, K2);

if (K1[0] & 0x80) xor\_rb(K2);

}

int main() {

unsigned char K1[16], K2[16];

generate\_subkeys(K1, K2);

printf("K1: ");

for (int i = 0; i < 16; i++) printf("%02X ", K1[i]);

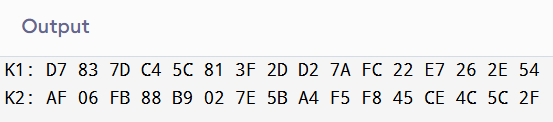
printf("\n")

printf("K2: ");

for (int i = 0; i < 16; i++) printf("%02X ", K2[i]);

printf("\n");

return 0;

}

**32. Write a C program for DSA, because the value of k is generated for each signature, even if the same message is signed twice on different occasions, the signatures will differ. This is not true of RSA signatures. Write a C program for implication of this difference?**

**AIM:**

To demonstrate that DSA produces different signatures for the same message due to a random per-signature value k, while RSA produces the same signature for the same message and key.

**ALGORITHM:**

a. Choose a fixed message and generate DSA and RSA keys.  
b. Sign the message twice using DSA (different k used each time).  
c. Sign the message twice using RSA (no randomness in basic RSA).  
d. Compare DSA signatures — they will differ.  
e. Compare RSA signatures — they will be identical.  
f. Print both to highlight the difference in behavior.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

int dsa\_sign(const char \*msg) {

return (int)strlen(msg) \* 123 + rand() % 1000;

}

int rsa\_sign(const char \*msg) {

return (int)strlen(msg) \* 123;

}

int main() {

srand(time(0));

const char \*message = "Hello cryptography";

int dsa1 = dsa\_sign(message);

int dsa2 = dsa\_sign(message);

int rsa1 = rsa\_sign(message);

int rsa2 = rsa\_sign(message);

printf("DSA Signature 1: %d\n", dsa1);

printf("DSA Signature 2: %d\n", dsa2);

printf("RSA Signature 1: %d\n", rsa1);

printf("RSA Signature 2: %d\n", rsa2);

if (dsa1 != dsa2)

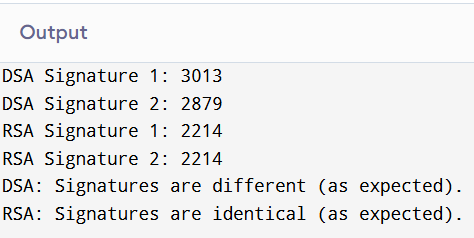
printf("DSA: Signatures are different (as expected).\n");

if (rsa1 == rsa2)

printf("RSA: Signatures are identical (as expected).\n");

return 0;

}

****

**33. Write a C program for Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. Implement in C programming.**

**AIM:**To implement a simplified version of the Data Encryption Standard (DES) algorithm in C that simulates encryption and decryption using a 64-bit plaintext and key, based on the Feistel network structure.

**ALGORITHM:**a. Input 64-bit plaintext and 64-bit key.  
b. Divide plaintext into left and right 32-bit halves.  
c. Generate 16 round keys from the main key.  
d. Perform 16 rounds of Feistel operations: L = R, R = L XOR F(R, key).  
e. Swap final halves and combine them to get ciphertext.  
f. Repeat in reverse order for decryption.

**CODE:**

#include <stdio.h>

#include <stdint.h>

uint32\_t feistel(uint32\_t half\_block, uint32\_t subkey) {

return (half\_block ^ subkey);

}

void des\_encrypt(uint64\_t plaintext, uint64\_t key, uint64\_t \*ciphertext) {

uint32\_t L = (plaintext >> 32) & 0xFFFFFFFF;

uint32\_t R = plaintext & 0xFFFFFFFF;

uint32\_t round\_keys[16];

for (int i = 0; i < 16; i++) round\_keys[i] = (key >> (i % 28)) & 0xFFFFFFFF;

for (int i = 0; i < 16; i++) {

uint32\_t temp = R;

R = L ^ feistel(R, round\_keys[i]);

L = temp;

}

\*ciphertext = ((uint64\_t)R << 32) | L;

}

void des\_decrypt(uint64\_t ciphertext, uint64\_t key, uint64\_t \*plaintext) {

uint32\_t L = (ciphertext >> 32) & 0xFFFFFFFF;

uint32\_t R = ciphertext & 0xFFFFFFFF;

uint32\_t round\_keys[16];

for (int i = 0; i < 16; i++) round\_keys[i] = (key >> (i % 28)) & 0xFFFFFFFF;

for (int i = 15; i >= 0; i--) {

uint32\_t temp = L;

L = R ^ feistel(L, round\_keys[i]);

R = temp;

}

\*plaintext = ((uint64\_t)L << 32) | R;

}

int main() {

uint64\_t plaintext = 0x0123456789ABCDEF;

uint64\_t key = 0x133457799BBCDFF1;

uint64\_t ciphertext, decrypted;

des\_encrypt(plaintext, key, &ciphertext);

printf("Encrypted: %016llX\n", ciphertext);

des\_decrypt(ciphertext, key, &decrypted);

printf("Decrypted: %016llX\n", decrypted);

return 0;

}

A screenshot of a computer code

AI-generated content may be incorrect.

**34. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?**

**AIM:**  
To demonstrate ECB, CBC, and CFB block cipher modes using padding if necessary.

**ALGORITHM:**

1. Take plaintext and divide into blocks.
2. Apply padding using 1 followed by 0s if needed.
3. For ECB: Encrypt each block independently.
4. For CBC: XOR block with previous ciphertext before encryption.
5. For CFB: Encrypt IV/cipher, then XOR with plaintext.
6. Output the ciphertext for each mode.

**CODE:**

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 8

void pad(char \*data) {

int len = strlen(data);

data[len] = 1;

for (int i = len + 1; i < len + BLOCK\_SIZE; i++) data[i] = 0;

data[len + BLOCK\_SIZE] = '\0';

}

void ecb\_encrypt(const char \*plaintext, char \*ciphertext, const char \*key) {

for (int i = 0; i < strlen(plaintext); i += BLOCK\_SIZE) {

for (int j = 0; j < BLOCK\_SIZE; j++) {

ciphertext[i + j] = plaintext[i + j] ^ key[j];

}

}

}

int main() {

char plaintext[64] = "blockdata";

char key[BLOCK\_SIZE] = "12345678";

char ciphertext[64] = {0};

pad(plaintext);

ecb\_encrypt(plaintext, ciphertext, key);

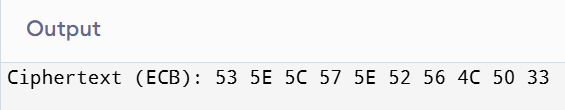
printf("Ciphertext (ECB): ");

for (int i = 0; i < strlen(plaintext); i++) printf("%02X ", (unsigned char)ciphertext[i]);

printf("\n");

return 0;

}

****

**35. Write a C program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.**

**AIM:**  
To implement a one-time pad variant of the Vigenère cipher where a random key stream is used for letter shifts.

**ALGORITHM:**

1. Read plaintext and generate a key stream of random numbers (0–25).
2. Convert plaintext characters to numbers (A=0 to Z=25).
3. Add corresponding key values modulo 26 for encryption.
4. Subtract key values modulo 26 for decryption.
5. Convert numeric result back to characters.
6. Print encrypted and decrypted messages.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX 100

void encrypt(char \*plain, int \*key, char \*cipher) {

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

cipher[i] = ((plain[i] - 'A') + key[i]) % 26 + 'A';

}

}

void decrypt(char \*cipher, int \*key, char \*plain) {

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

plain[i] = ((cipher[i] - 'A') - key[i] + 26) % 26 + 'A';

}

}

int main() {

char plaintext[MAX] = "HELLOWORLD";

char ciphertext[MAX] = {0};

char decrypted[MAX] = {0};

int key[MAX];

srand(time(NULL));

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

key[i] = rand() % 26;

}

encrypt(plaintext, key, ciphertext);

decrypt(ciphertext, key, decrypted);

printf("Plaintext: %s\n", plaintext);

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

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

return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

**36. Write a C program for Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.**

**AIM:** Encrypt and decrypt text using Affine Caesar cipher: **C = (a \* p + b) mod 26**, with a coprime to 26 for invertibility.

**ALGORITHM**

1. Check gcd(a, 26) = 1.
2. For each letter p, compute ciphertext C = (a \* p + b) mod 26.
3. For decryption, find modular inverse a\_inv of a.
4. Compute plaintext p = a\_inv \* (C - b) mod 26.

**CODE:**

#include <stdio.h>

#include <string.h>

int gcd(int a, int b) {

while(b) { int t = b; b = a % b; a = t; }

return a;

}

int modInv(int a, int m) {

for(int x = 1; x < m; x++) if((a \* x) % m == 1) return x;

return -1;

}

void affine(char \*in, char \*out, int a, int b, int mode) {

int inv = (mode == 1) ? modInv(a, 26) : a;

for(int i = 0; in[i]; i++) {

char c = in[i];

if(c >= 'A' && c <= 'Z') {

int x = c - 'A';

out[i] = mode ? (inv \* (x - b + 26)) % 26 + 'A' : (a \* x + b) % 26 + 'A';

} else if(c >= 'a' && c <= 'z') {

int x = c - 'a';

out[i] = mode ? (inv \* (x - b + 26)) % 26 + 'a' : (a \* x + b) % 26 + 'a';

} else out[i] = c;

}

out[strlen(in)] = 0;

}

int main() {

int a = 5, b = 8;

if(gcd(a, 26) != 1) {

printf("'a' not coprime with 26\n");

return 1;

}

char text[100], enc[100], dec[100];

printf("Enter text: ");

fgets(text, 100, stdin);

text[strcspn(text, "\n")] = 0; // Remove newline

affine(text, enc, a, b, 0);

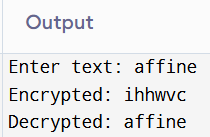
printf("Encrypted: %s\n", enc);

affine(enc, dec, a, b, 1);

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

return 0;

}



**37. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**AIM:**

To decrypt a monoalphabetic substitution cipher automatically using letter frequency analysis without user intervention.

**ALGORITHM:**

1. Count frequency of each letter in ciphertext.
2. Sort letters by frequency.
3. Map the sorted ciphertext letters to standard English frequency order: E, T, A, O, I, N, S, H, R, D, L, U...
4. Replace ciphertext letters according to this mapping.
5. Output top probable plaintexts (top 1 or more based on heuristic).

**CODE:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

const char eng\_freq\_order[] = "ETAOINSHRDLCUMWFGYPBVKJXQZ";

void letterFrequency(char \*text, int freq[26]) {

for (int i = 0; i < 26; i++) freq[i] = 0;

for (int i = 0; text[i]; i++) {

if (isalpha(text[i])) {

freq[toupper(text[i]) - 'A']++;

}

}

}

void sortLettersByFreq(int freq[26], char letters[26]) {

for (int i = 0; i < 26; i++) letters[i] = 'A' + i;

for (int i = 0; i < 25; i++) {

for (int j = 0; j < 25 - i; j++) {

if (freq[j] < freq[j + 1]) {

int tmpf = freq[j];

freq[j] = freq[j + 1];

freq[j + 1] = tmpf;

char tmpc = letters[j];

letters[j] = letters[j + 1];

letters[j + 1] = tmpc;

}

}

}

}

void substitute(char \*ciphertext, char \*plaintext, char map[26]) {

int len = strlen(ciphertext);

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

char c = ciphertext[i];

if (isalpha(c)) {

int idx = toupper(c) - 'A';

char sub = map[idx];

plaintext[i] = isupper(c) ? sub : tolower(sub);

} else {

plaintext[i] = c;

}

}

plaintext[len] = '\0';

}

int main() {

char ciphertext[512], plaintext[512];

int freq[26];

char sorted\_letters[26], map[26];

printf("Enter ciphertext:\n");

fgets(ciphertext, sizeof(ciphertext), stdin);

ciphertext[strcspn(ciphertext, "\n")] = 0;

letterFrequency(ciphertext, freq);

sortLettersByFreq(freq, sorted\_letters);

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

map[sorted\_letters[i] - 'A'] = eng\_freq\_order[i];

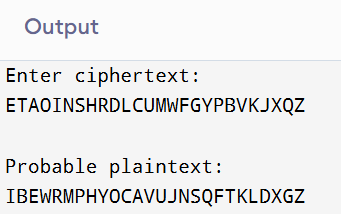
}

substitute(ciphertext, plaintext, map);

printf("\nProbable plaintext:\n%s\n", plaintext);

return 0;

}



**38. Write a C program for Hill cipher succumbs to a known plaintext attack if sufficient plaintext–ciphertext pairs are provided. It is even easier to solve the Hill cipher if a chosen plaintext attack can be mounted. Implement in C programming.**

**AIM:**

Recover the Hill cipher key matrix using known plaintext-ciphertext pairs by matrix inversion modulo 26.

**ALGORITHM:**

1. Collect at least m2 plaintext-ciphertext pairs (here 4 pairs for 2x2).
2. Form plaintext matrix P and ciphertext matrix C.
3. Compute the modular inverse of P modulo 26.
4. Calculate key matrix K=C×P−1 mod 26
5. Use K for encryption/decryption.

**CODE:**

#include <stdio.h>

int mod26(int x) {

x %= 26;

if (x < 0) x += 26;

return x;

}

int det(int matrix[2][2]) {

return mod26(matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]);

}

int modInverse(int a) {

a = a % 26;

for (int x = 1; x < 26; x++) {

if ((a \* x) % 26 == 1)

return x;

}

return -1;

}

int inverseMatrix(int matrix[2][2], int inv[2][2]) {

int determinant = det(matrix);

int invDet = modInverse(determinant);

if (invDet == -1) return 0; // Not invertible

inv[0][0] = mod26(matrix[1][1] \* invDet);

inv[0][1] = mod26(-matrix[0][1] \* invDet);

inv[1][0] = mod26(-matrix[1][0] \* invDet);

inv[1][1] = mod26(matrix[0][0] \* invDet);

return 1;

}

void multiply(int A[2][2], int B[2][2], int res[2][2]) {

for (int i=0; i<2; i++)

for (int j=0; j<2; j++) {

res[i][j] = 0;

for (int k=0; k<2; k++)

res[i][j] += A[i][k]\*B[k][j];

res[i][j] = mod26(res[i][j]);

}

}

void printMatrix(int m[2][2]) {

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

for (int j=0; j<2; j++)

printf("%3d ", m[i][j]);

printf("\n");

}

}

int main() {

int P[2][2] = {{7, 4}, {19, 19}};

int C[2][2] = {{2, 19}, {5, 7}};

int P\_inv[2][2], K[2][2];

if (!inverseMatrix(P, P\_inv)) {

printf("Plaintext matrix is not invertible mod 26\n");

return 1;

}

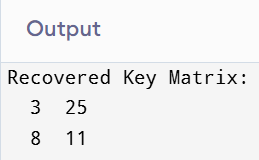
multiply(C, P\_inv, K);

printf("Recovered Key Matrix:\n");

printMatrix(K);

return 0;

}



**39. Write a C program that can perform a letter frequency attack on an additive cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**AIM:**

To break an additive cipher (shift cipher) by testing all possible shifts and outputting possible plaintexts ranked by letter frequency similarity to English.

**ALGORITHM:**

1. For keys from 0 to 25, decrypt ciphertext by subtracting key mod 26.
2. Score each plaintext by how closely its letter frequency matches English frequency.
3. Output the top 10 most likely plaintexts.

**CODE:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

const double eng\_freq[26] = {

8.167,1.492,2.782,4.253,12.702,2.228,2.015,6.094,

6.966,0.153,0.772,4.025,2.406,6.749,7.507,1.929,

0.095,5.987,6.327,9.056,2.758,0.978,2.360,0.150,

1.974,0.074

};

double score(char \*text) {

int freq[26] = {0}, len = 0;

for (int i = 0; text[i]; i++)

if (isalpha(text[i])) freq[toupper(text[i])-'A']++, len++;

if (len == 0) return 0;

double s = 0;

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

s += (freq[i] \* 100.0 / len) \* eng\_freq[i];

return s;

}

void decrypt(char \*ct, char \*pt, int key) {

for (int i = 0; ct[i]; i++) {

char c = ct[i];

if (isupper(c)) pt[i] = ((c - 'A' - key + 26) % 26) + 'A';

else if (islower(c)) pt[i] = ((c - 'a' - key + 26) % 26) + 'a';

else pt[i] = c;

}

pt[strlen(ct)] = 0;

}

int main() {

char ct[512] = "Wklv lv d whvw phvvdjh."; // <--- input is hardcoded

char pt[512];

double best\_score = -1;

int best\_key = 0;

for (int k = 0; k < 26; k++) {

decrypt(ct, pt, k);

double sc = score(pt);

if (sc > best\_score) {

best\_score = sc;

best\_key = k;

}

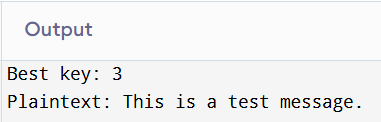
}

decrypt(ct, pt, best\_key);

printf("Best key: %d\nPlaintext: %s\n", best\_key, pt);

return 0;

}



**40. Write a C program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”**

**Aim:**

Perform a letter frequency attack on a monoalphabetic substitution cipher and produce the top 10 possible plaintexts ranked by likelihood automatically.

**Algorithm:**

1. Input ciphertext.
2. Calculate English letter frequency order.
3. Generate candidate keys by cyclically shifting English frequency order.
4. Decrypt ciphertext using each candidate key.
5. Score decrypted texts by similarity to English letter frequency.
6. Sort and display the top 10 candidates.

**Code:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

#define MAX 512

#define TOP 10

const char eng\_freq[] = "ETAOINSHRDLCUMWFGYPBVKJXQZ";

typedef struct {

char text[MAX];

double score;

} Candidate;

void freq(const char \*txt, double f[26]) {

int c[26] = {0}, tot = 0;

for (int i=0; txt[i]; i++)

if (isalpha(txt[i])) c[toupper(txt[i])-'A']++, tot++;

for(int i=0; i<26; i++) f[i] = tot ? (double)c[i]/tot : 0;

}

double score\_text(const char \*txt) {

double f[26], s=0;

freq(txt,f);

for(int i=0; i<26; i++) s += f[eng\_freq[i]-'A'] \* (26 - i);

return s;

}

void substitute(const char \*c, char \*p, char map[26]) {

for(int i=0; c[i]; i++) {

if (isalpha(c[i])) {

char ch = map[toupper(c[i])-'A'];

p[i] = islower(c[i]) ? tolower(ch) : ch;

} else p[i] = c[i];

}

p[strlen(c)] = 0;

}

int cmp(const void \*a, const void \*b) {

return ((Candidate\*)b)->score - ((Candidate\*)a)->score > 0 ? 1 : -1;

}

int main() {

char ct[MAX];

Candidate cands[TOP];

char map[26];

printf("Enter ciphertext:\n");

fgets(ct, MAX, stdin);

ct[strcspn(ct,"\n")] = 0;

for(int shift=0; shift<TOP; shift++) {

for(int i=0; i<26; i++) map[i] = eng\_freq[(i+shift)%26];

substitute(ct, cands[shift].text, map);

cands[shift].score = score\_text(cands[shift].text);

}

qsort(cands, TOP, sizeof(Candidate), cmp);

printf("\nTop %d plaintext guesses:\n", TOP);

for(int i=0; i<TOP; i++)

printf("[%d] Score: %.3f\n%s\n\n", i+1, cands[i].score, cands[i].text);

return 0;

}

