1. #include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

void encrypt(char\* message, int k) {

for (int i = 0; i < strlen(message); i++) {

char c = message[i];

if (isalpha(c)) {

char base = islower(c) ? 'a' : 'A';

message[i] = (c - base + k) % 26 + base;

}

}

}

int main() {

char message[100];

int k;

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

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

printf("Enter the shift value (1-25): ");

scanf("%d", &k);

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

printf("Shift value must be in the range 1-25.\n");

return 1;

}

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

encrypt(message, k);

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

return 0;

}

**OUTPUT**Enter the message to encrypt: hello

Enter the shift value (1-25): 5

Encrypted message: mjqqt

1. #include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

void encrypt(char\* message, char\* key) {

for (int i = 0; i < strlen(message); i++) {

char c = message[i];

if (isalpha(c)) {

char base = islower(c) ? 'a' : 'A';

message[i] = islower(c) ? tolower(key[c - base]) : toupper(key[c - base]);

}

}

}

int main() {

char message[100];

char key[] = "QWERTYUIOPLKJHGFDSAZXCVBNM";

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

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

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

encrypt(message, key);

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

return 0;

}

**OUTPUT**

Enter the message to encrypt: HELLO

Encrypted message: ITKKG

1. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define SIZE 5

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

int dict[26] = {0};

int i, j, k;

int index = 0;

for (i = 0; i < strlen(key); i++) {

if (key[i] != 'j') {

if (dict[key[i] - 'a'] == 0) {

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

dict[key[i] - 'a'] = 1;

index++;

}

}

}

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

if (k + 'a' != 'j') {

if (dict[k] == 0) {

keyTable[index / SIZE][index % SIZE] = k + 'a';

index++;

}

}

}

}

void prepareText(char str[], char preparedText[]) {

int i, j = 0;

int len = strlen(str);

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

if (isalpha(str[i])) {

preparedText[j++] = tolower(str[i]);

}

}

if (j % 2 != 0) {

preparedText[j++] = 'x';

}

preparedText[j] = '\0';

}

void digraphEncryption(char digraph[], char keyTable[SIZE][SIZE], char result[]) {

int i1, j1, i2, j2;

int found = 0;

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

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

if (keyTable[i][j] == digraph[0]) {

i1 = i;

j1 = j;

found++;

}

if (keyTable[i][j] == digraph[1]) {

i2 = i;

j2 = j;

found++;

}

if (found == 2) break;

}

if (found == 2) break;

}

if (i1 == i2) {

result[0] = keyTable[i1][(j1 + 1) % SIZE];

result[1] = keyTable[i2][(j2 + 1) % SIZE];

} else if (j1 == j2) {

result[0] = keyTable[(i1 + 1) % SIZE][j1];

result[1] = keyTable[(i2 + 1) % SIZE][j2];

} else {

result[0] = keyTable[i1][j2];

result[1] = keyTable[i2][j1];

}

}

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

int len = strlen(plaintext);

char digraph[2];

char encryptedPair[2];

int index = 0;

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

digraph[0] = plaintext[i];

digraph[1] = plaintext[i + 1];

digraphEncryption(digraph, keyTable, encryptedPair);

ciphertext[index++] = encryptedPair[0];

ciphertext[index++] = encryptedPair[1];

}

ciphertext[index] = '\0';

}

int main() {

char key[100];

char plaintext[100];

char preparedText[100];

char ciphertext[100];

char keyTable[SIZE][SIZE];

printf("Enter the keyword: ");

scanf("%s", key);

printf("Enter the plaintext: ");

scanf("%s", plaintext);

prepareKeyTable(key, keyTable);

prepareText(plaintext, preparedText);

encrypt(preparedText, keyTable, ciphertext);

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

return 0;

}

**OUTPUT;**

Enter the keyword: HELLO

Enter the plaintext: BEST

Encrypted message: catu

1. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Function to generate the keystream

void generateKeystream(char \*plaintext, char \*keyword, char \*keystream) {

int len\_plaintext = strlen(plaintext);

int len\_keyword = strlen(keyword);

int i, j = 0;

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

if (isalpha(plaintext[i])) {

keystream[i] = keyword[j % len\_keyword];

j++;

} else {

keystream[i] = plaintext[i]; // Preserve non-alphabet characters

}

}

keystream[i] = '\0';

}

// Function to encrypt the plaintext

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

int i;

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

if (isalpha(plaintext[i])) {

char base = isupper(plaintext[i]) ? 'A' : 'a';

ciphertext[i] = (plaintext[i] + keystream[i] - 2 \* base) % 26 + base;

} else {

ciphertext[i] = plaintext[i];

}

}

ciphertext[i] = '\0';

}

int main() {

char plaintext[256];

char keyword[256];

char keystream[256];

char ciphertext[256];

printf("Enter the plaintext: ");

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

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

printf("Enter the keyword: ");

fgets(keyword, sizeof(keyword), stdin);

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

// Convert keyword to lowercase

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

keyword[i] = tolower(keyword[i]);

}

// Generate the keystream

generateKeystream(plaintext, keyword, keystream);

// Encrypt the plaintext

encrypt(plaintext, keystream, ciphertext);

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

return 0;

}

**OUTPUT**  
Enter the plaintext: HELLO

Enter the keyword: VISHNU

Encrypted message: ISJYH  
  
5. #include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MOD 26

// Function to calculate gcd

int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

// Function to encrypt the plaintext

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

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

if (isalpha(plaintext[i])) {

char base = isupper(plaintext[i]) ? 'A' : 'a';

ciphertext[i] = ((a \* (plaintext[i] - base) + b) % MOD) + base;

} else {

ciphertext[i] = plaintext[i];

}

}

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

}

int main() {

char plaintext[256];

char ciphertext[256];

int a, b;

printf("Enter the plaintext: ");

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

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

printf("Enter the value of a (must be coprime with 26): ");

scanf("%d", &a);

printf("Enter the value of b: ");

scanf("%d", &b);

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

printf("Error: a must be coprime with 26.\n");

return 1;

}

encrypt(plaintext, a, b, ciphertext);

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

return 0;

}

**OUTPUT**  
Enter the plaintext: VIHNU

Enter the value of a (must be coprime with 26): 7

Enter the value of b: 4

Encrypted message: VIBRO  
  
6. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Function to calculate gcd and modular inverse

int gcdExtended(int a, int b, int \*x, int \*y) {

if (a == 0) {

\*x = 0;

\*y = 1;

return b;

}

int x1, y1;

int gcd = gcdExtended(b % a, a, &x1, &y1);

\*x = y1 - (b / a) \* x1;

\*y = x1;

return gcd;

}

int modInverse(int a, int m) {

int x, y;

int g = gcdExtended(a, m, &x, &y);

if (g != 1) {

return -1; // Inverse doesn't exist

}

return (x % m + m) % m;

}

// Function to decrypt the ciphertext

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

int a\_inv = modInverse(a, 26);

if (a\_inv == -1) {

printf("Modular inverse does not exist. Cannot decrypt.\n");

exit(1);

}

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

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

plaintext[i] = ((a\_inv \* ((ciphertext[i] - base) - b + 26)) % 26) + base;

} else {

plaintext[i] = ciphertext[i];

}

}

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

}

int main() {

char ciphertext[256];

char plaintext[256];

int a, b;

// Input ciphertext

printf("Enter the ciphertext: ");

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

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

// Assuming B corresponds to E and U corresponds to T

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

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

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

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

// Solve the system of equations:

// 1 = 4a + b (mod 26)

// 20 = 19a + b (mod 26)

// b = 1 - 4a (mod 26)

// 20 = 19a + (1 - 4a) (mod 26)

// 20 = 15a + 1 (mod 26)

// 19 = 15a (mod 26)

int a\_inv = modInverse(15, 26);

if (a\_inv == -1) {

printf("Modular inverse does not exist. Cannot solve.\n");

return 1;

}

a = (19 \* a\_inv) % 26;

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

printf("Determined values: a = %d, b = %d\n", a, b);

// Decrypt the ciphertext

decrypt(ciphertext, a, b, plaintext);

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

return 0;

}  
  
**output**  
Enter the ciphertext: vishnu

Determined values: a = 3, b = 15

Decrypted message: cpbgit

7. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_TEXT\_LENGTH 1024

void frequencyAnalysis(const char \*text, int freq[]) {

for (int i = 0; i < 256; i++) {

freq[i] = 0;

}

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

freq[(unsigned char)text[i]]++;

}

}

void printFrequency(const int freq[]) {

printf("Character frequencies:\n");

for (int i = 0; i < 256; i++) {

if (freq[i] > 0 && isprint(i)) {

printf("%c: %d\n", i, freq[i]);

}

}

}

void substitute(const char \*ciphertext, const char \*substitution, char \*plaintext) {

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

if (isprint(ciphertext[i])) {

plaintext[i] = substitution[(unsigned char)ciphertext[i]];

} else {

plaintext[i] = ciphertext[i];

}

}

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

}

void printSubstitution(const char \*substitution) {

printf("Current substitution key:\n");

for (int i = 0; i < 256; i++) {

if (isprint(i)) {

printf("%c -> %c\n", i, substitution[i]);

}

}

}

int main() {

const char ciphertext[] = "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;‡?;";

int freq[256];

char substitution[256];

char plaintext[MAX\_TEXT\_LENGTH];

// Initialize substitution with identity mapping

for (int i = 0; i < 256; i++) {

substitution[i] = i;

}

frequencyAnalysis(ciphertext, freq);

printFrequency(freq);

// Manually define some common substitutions

// For example, substituting '‡' with 'E', '†' with 'T', etc.

// This is just an example, and these substitutions need to be refined

substitution['‡'] = 'E';

substitution['†'] = 'T';

substitution['5'] = 'H';

substitution['3'] = 'R';

substitution['0'] = 'A';

substitution['6'] = 'N';

substitution['\*'] = 'D';

substitution[';'] = 'O';

substitution['8'] = 'S';

substitution['4'] = 'I';

substitution[')'] = 'G';

substitution['('] = 'L';

substitution['2'] = 'F';

substitution[':'] = 'C';

substitution['¶'] = 'U';

substitution['1'] = 'W';

substitution['9'] = 'M';

substitution['?'] = 'P';

substitution[']'] = 'Y';

substitution['—'] = 'K';

substitution['.'] = 'B';

substitution['['] = 'V';

// Print the current substitution key

printSubstitution(substitution);

// Apply the substitution

substitute(ciphertext, substitution, plaintext);

printf("Decrypted message (with current substitution):\n%s\n", plaintext);

return 0;

}  
  
**OUTPUT**  
Character frequencies:

(: 9

): 16

\*: 14

.: 1

0: 6

1: 7

2: 5

3: 4

4: 19

5: 12

6: 11

8: 34

9: 5

:: 4

;: 27

?: 3

]: 1  
  
8. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Function to create the cipher sequence based on the keyword

void createCipherSequence(const char \*keyword, char \*cipherSeq) {

int used[26] = {0}; // Array to track used letters

int index = 0;

// Add keyword letters to cipher sequence

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

if (!used[toupper(keyword[i]) - 'A']) {

cipherSeq[index++] = toupper(keyword[i]);

used[toupper(keyword[i]) - 'A'] = 1;

}

}

// Add remaining letters to cipher sequence

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

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

cipherSeq[index++] = ch;

used[ch - 'A'] = 1;

}

}

cipherSeq[index] = '\0';

}

// Function to encrypt the plaintext using the cipher sequence

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

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

if (isalpha(plaintext[i])) {

char base = isupper(plaintext[i]) ? 'A' : 'a';

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

ciphertext[i] = isupper(plaintext[i]) ? cipherSeq[index] : tolower(cipherSeq[index]);

} else {

ciphertext[i] = plaintext[i];

}

}

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

}

// Function to decrypt the ciphertext using the cipher sequence

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

char reverseSeq[26];

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

reverseSeq[cipherSeq[i] - 'A'] = 'A' + i;

}

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

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

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

plaintext[i] = isupper(ciphertext[i]) ? reverseSeq[index] : tolower(reverseSeq[index]);

} else {

plaintext[i] = ciphertext[i];

}

}

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

}

int main() {

char keyword[100];

char plaintext[256];

char ciphertext[256];

char decryptedtext[256];

char cipherSeq[27]; // 26 letters + null terminator

// Input keyword

printf("Enter the keyword: ");

fgets(keyword, sizeof(keyword), stdin);

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

// Input plaintext

printf("Enter the plaintext: ");

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

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

// Create cipher sequence

createCipherSequence(keyword, cipherSeq);

printf("Cipher sequence: %s\n", cipherSeq);

// Encrypt the plaintext

encrypt(plaintext, cipherSeq, ciphertext);

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

// Decrypt the ciphertext

decrypt(ciphertext, cipherSeq, decryptedtext);

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

return 0;

}

**OUTPUT**Enter the keyword: VISHNU

Enter the plaintext: HELLO

Cipher sequence: VISHNUABCDEFGJKLMOPQRTWXYZ

Encrypted text: BNFFK

Decrypted text: HELLO  
  
9. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MATRIX\_SIZE 5

void createPlayfairMatrix(const char \*keyword, char matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

int used[26] = {0};

int x = 0, y = 0;

// Remove duplicates from keyword and create matrix

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

char ch = toupper(keyword[i]);

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

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

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

used[ch - 'A'] = 1;

if (x == MATRIX\_SIZE) {

x = 0;

y++;

}

}

}

// Fill the rest of the matrix with remaining letters

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

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

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

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

used[ch - 'A'] = 1;

if (x == MATRIX\_SIZE) {

x = 0;

y++;

}

}

}

}

void findPosition(char matrix[MATRIX\_SIZE][MATRIX\_SIZE], char ch, int \*row, int \*col) {

for (int y = 0; y < MATRIX\_SIZE; y++) {

for (int x = 0; x < MATRIX\_SIZE; x++) {

if (matrix[y][x] == ch) {

\*row = y;

\*col = x;

return;

}

}

}

}

void decryptPair(char matrix[MATRIX\_SIZE][MATRIX\_SIZE], char a, char b, char \*decryptedA, char \*decryptedB) {

int row1, col1, row2, col2;

findPosition(matrix, a, &row1, &col1);

findPosition(matrix, b, &row2, &col2);

if (row1 == row2) {

\*decryptedA = matrix[row1][(col1 + MATRIX\_SIZE - 1) % MATRIX\_SIZE];

\*decryptedB = matrix[row2][(col2 + MATRIX\_SIZE - 1) % MATRIX\_SIZE];

} else if (col1 == col2) {

\*decryptedA = matrix[(row1 + MATRIX\_SIZE - 1) % MATRIX\_SIZE][col1];

\*decryptedB = matrix[(row2 + MATRIX\_SIZE - 1) % MATRIX\_SIZE][col2];

} else {

\*decryptedA = matrix[row1][col2];

\*decryptedB = matrix[row2][col1];

}

}

void decryptPlayfair(const char \*ciphertext, char \*plaintext, char matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

char a = toupper(ciphertext[i]);

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

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

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

decryptPair(matrix, a, b, &plaintext[i], &plaintext[i + 1]);

}

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

}

int main() {

const char \*keyword = "CIPHER";

const char \*ciphertext = "KXJEYUREBEZWEHEWRYTUHEYFSKREHEGOYFIWTTTUOLKSYCAJPOBOTEIZONTXBYBNTGONEYCUZWRGDSONSXBOUYWRHEBAAHYUSEDQ";

char matrix[MATRIX\_SIZE][MATRIX\_SIZE];

char plaintext[256];

// Create Playfair matrix

createPlayfairMatrix(keyword, matrix);

// Print Playfair matrix

printf("Playfair matrix:\n");

for (int y = 0; y < MATRIX\_SIZE; y++) {

for (int x = 0; x < MATRIX\_SIZE; x++) {

printf("%c ", matrix[y][x]);

}

printf("\n");

}

// Decrypt the ciphertext

decryptPlayfair(ciphertext, plaintext, matrix);

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

return 0;

}  
  
**OUTPUT**  
Playfair matrix:

C I P H E

R A B D F

G K L M N

O Q S T U

V W X Y Z

Decrypted message: LWCHZTFCFPYVHPIZDVSTPHZDQLFCPHRGZDWQSSSTSGLQVHIWCSRSUHEWUGSYDXFLOMUGHZEOYVCRBTUGLSRSTZVAPHARDIZTUPAT  
  
10Q. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MATRIX\_SIZE 5

void createPlayfairMatrix(char matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

const char \*matrixSequence = "MFHIJKUNOPQZVWXEYLRGDSTBC";

int index = 0;

for (int y = 0; y < MATRIX\_SIZE; y++) {

for (int x = 0; x < MATRIX\_SIZE; x++) {

matrix[y][x] = matrixSequence[index++];

}

}

}

void findPosition(char matrix[MATRIX\_SIZE][MATRIX\_SIZE], char ch, int \*row, int \*col) {

for (int y = 0; y < MATRIX\_SIZE; y++) {

for (int x = 0; x < MATRIX\_SIZE; x++) {

if (matrix[y][x] == ch) {

\*row = y;

\*col = x;

return;

}

}

}

}

void encryptPair(char matrix[MATRIX\_SIZE][MATRIX\_SIZE], char a, char b, char \*encryptedA, char \*encryptedB) {

int row1, col1, row2, col2;

findPosition(matrix, a, &row1, &col1);

findPosition(matrix, b, &row2, &col2);

if (row1 == row2) {

\*encryptedA = matrix[row1][(col1 + 1) % MATRIX\_SIZE];

\*encryptedB = matrix[row2][(col2 + 1) % MATRIX\_SIZE];

} else if (col1 == col2) {

\*encryptedA = matrix[(row1 + 1) % MATRIX\_SIZE][col1];

\*encryptedB = matrix[(row2 + 1) % MATRIX\_SIZE][col2];

} else {

\*encryptedA = matrix[row1][col2];

\*encryptedB = matrix[row2][col1];

}

}

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

int len = 0;

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

if (isalpha(input[i])) {

output[len++] = toupper(input[i] == 'J' ? 'I' : input[i]);

}

}

output[len] = '\0';

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

if (i + 1 < len && output[i] == output[i + 1]) {

memmove(output + i + 2, output + i + 1, len - i);

output[i + 1] = 'X';

len++;

}

}

if (len % 2 != 0) {

output[len++] = 'X';

}

output[len] = '\0';

}

void encryptPlayfair(const char \*plaintext, char \*ciphertext, char matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

char a = toupper(plaintext[i]);

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

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

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

encryptPair(matrix, a, b, &ciphertext[i], &ciphertext[i + 1]);

}

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

}

int main() {

const char \*input = "Must see you over Cadogan West. Coming at once";

char plaintext[256];

char ciphertext[256];

char matrix[MATRIX\_SIZE][MATRIX\_SIZE];

// Create Playfair matrix

createPlayfairMatrix(matrix);

// Print Playfair matrix

printf("Playfair matrix:\n");

for (int y = 0; y < MATRIX\_SIZE; y++) {

for (int x = 0; x < MATRIX\_SIZE; x++) {

printf("%c ", matrix[y][x]);

}

printf("\n");

}

// Prepare plaintext

preparePlaintext(input, plaintext);

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

// Encrypt the plaintext

encryptPlayfair(plaintext, ciphertext, matrix);

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

return 0;

}

**OUTPUT**Playfair matrix:

M F H I J

K U N O P

Q Z V W X

E Y L R G

D S T B C

Prepared plaintext: MUSTSEEYOUOVERCADOGANWESTCOMINGATONCEX

Encrypted message: FKTBDYYLPNNWYGBGBKRPOVYDBDKIHOLPBNPTGQ

11Q. #include <stdio.h>

#include <math.h>

// Function to calculate factorial

unsigned long long factorial(int n) {

unsigned long long result = 1;

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

result \*= i;

}

return result;

}

// Function to calculate log2(factorial)

double log2Factorial(int n) {

double result = 0;

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

result += log2(i);

}

return result;

}

int main() {

int n = 25;

int rowCol = 5;

unsigned long long totalKeys = factorial(n);

unsigned long long rowColFactorial = factorial(rowCol);

unsigned long long uniqueKeys = totalKeys / (rowColFactorial \* rowColFactorial);

double log2TotalKeys = log2Factorial(n);

double log2RowColFactorial = log2Factorial(rowCol);

double log2UniqueKeys = log2TotalKeys - 2 \* log2RowColFactorial;

printf("Total number of keys (25!): %.0f (approximately 2^%.2f)\n", log2TotalKeys, log2TotalKeys);

printf("Number of effectively unique keys: %.0f (approximately 2^%.2f)\n", log2UniqueKeys, log2UniqueKeys);

return 0;

}

**OUTPUT**  
Total number of keys (25!): 84 (approximately 2^83.68)

Number of effectively unique keys: 70 (approximately 2^69.87)

12Q. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MATRIX\_SIZE 2

#define ALPHABET\_SIZE 26

// Key matrix for encryption

int keyMatrix[MATRIX\_SIZE][MATRIX\_SIZE] = {

{9, 4},

{5, 7}

};

// Function to find the modular inverse of a number

int modInverse(int a, int m) {

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

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

return x;

}

}

return -1;

}

// Function to multiply a matrix with a vector

void multiplyMatrixVector(int mat[MATRIX\_SIZE][MATRIX\_SIZE], int vec[MATRIX\_SIZE], int res[MATRIX\_SIZE]) {

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

res[i] = 0;

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

res[i] += mat[i][j] \* vec[j];

}

res[i] %= ALPHABET\_SIZE;

}

}

// Function to encrypt the plaintext

void hillCipherEncrypt(char\* plaintext, char\* ciphertext) {

int length = strlen(plaintext);

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

int pair[MATRIX\_SIZE];

int encryptedPair[MATRIX\_SIZE];

pair[0] = toupper(plaintext[i]) - 'A';

pair[1] = (i + 1 < length) ? toupper(plaintext[i + 1]) - 'A' : 'X' - 'A';

multiplyMatrixVector(keyMatrix, pair, encryptedPair);

ciphertext[i] = encryptedPair[0] + 'A';

ciphertext[i + 1] = encryptedPair[1] + 'A';

}

ciphertext[length] = '\0';

}

// Function to compute the inverse key matrix

void inverseKeyMatrix(int key[MATRIX\_SIZE][MATRIX\_SIZE], int invKey[MATRIX\_SIZE][MATRIX\_SIZE]) {

int det = key[0][0] \* key[1][1] - key[0][1] \* key[1][0];

det = (det % ALPHABET\_SIZE + ALPHABET\_SIZE) % ALPHABET\_SIZE;

int invDet = modInverse(det, ALPHABET\_SIZE);

invKey[0][0] = key[1][1] \* invDet % ALPHABET\_SIZE;

invKey[1][1] = key[0][0] \* invDet % ALPHABET\_SIZE;

invKey[0][1] = (-key[0][1] \* invDet % ALPHABET\_SIZE + ALPHABET\_SIZE) % ALPHABET\_SIZE;

invKey[1][0] = (-key[1][0] \* invDet % ALPHABET\_SIZE + ALPHABET\_SIZE) % ALPHABET\_SIZE;

}

// Function to decrypt the ciphertext

void hillCipherDecrypt(char\* ciphertext, char\* decryptedText) {

int length = strlen(ciphertext);

int invKey[MATRIX\_SIZE][MATRIX\_SIZE];

inverseKeyMatrix(keyMatrix, invKey);

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

int pair[MATRIX\_SIZE];

int decryptedPair[MATRIX\_SIZE];

pair[0] = toupper(ciphertext[i]) - 'A';

pair[1] = toupper(ciphertext[i + 1]) - 'A';

multiplyMatrixVector(invKey, pair, decryptedPair);

decryptedText[i] = decryptedPair[0] + 'A';

decryptedText[i + 1] = decryptedPair[1] + 'A';

}

decryptedText[length] = '\0';

}

int main() {

char plaintext[] = "MEETMEATTHEUSUALPLACEATTENRATHERTHANEIGHTOCLOCK";

char ciphertext[256];

char decryptedText[256];

// Encrypt the plaintext

hillCipherEncrypt(plaintext, ciphertext);

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

// Decrypt the ciphertext

hillCipherecrypt(ciphertext, decryptedText);

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

return 0;

}

**OUTPUT**  
Ciphertext: UKIXUKYDROMEIWSZXWIOKUNUKHXHROAJROANQYEBTLKJEGA

Decrypted Text: MEETMEATTHEUSUALPLACEATTENRATHERTHANEIGHTOCLOCA

13Q. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MATRIX\_SIZE 2

#define ALPHABET\_SIZE 26

// Function prototypes

void printMatrix(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]);

void getMatrixInverse(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], int inverse[MATRIX\_SIZE][MATRIX\_SIZE]);

int modInverse(int a, int m);

void multiplyMatrices(int matrix1[MATRIX\_SIZE][MATRIX\_SIZE], int matrix2[MATRIX\_SIZE][MATRIX\_SIZE], int result[MATRIX\_SIZE][MATRIX\_SIZE]);

void getPlaintextMatrix(const char\* plaintext, int matrix[MATRIX\_SIZE][MATRIX\_SIZE]);

void getCiphertextMatrix(const char\* ciphertext, int matrix[MATRIX\_SIZE][MATRIX\_SIZE]);

void getKeyMatrix(int plaintextMatrix[MATRIX\_SIZE][MATRIX\_SIZE], int ciphertextMatrix[MATRIX\_SIZE][MATRIX\_SIZE], int keyMatrix[MATRIX\_SIZE][MATRIX\_SIZE]);

// Main function

int main() {

// Example plaintext and corresponding ciphertext pairs

char plaintext[] = "HELP";

char ciphertext[] = "IQDF";

int plaintextMatrix[MATRIX\_SIZE][MATRIX\_SIZE];

int ciphertextMatrix[MATRIX\_SIZE][MATRIX\_SIZE];

int keyMatrix[MATRIX\_SIZE][MATRIX\_SIZE];

// Get plaintext and ciphertext matrices

getPlaintextMatrix(plaintext, plaintextMatrix);

getCiphertextMatrix(ciphertext, ciphertextMatrix);

// Compute the key matrix

getKeyMatrix(plaintextMatrix, ciphertextMatrix, keyMatrix);

printf("Key Matrix:\n");

printMatrix(keyMatrix);

return 0;

}

// Function to print a matrix

void printMatrix(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

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

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

}

printf("\n");

}

}

// Function to compute the modular inverse of a number

int modInverse(int a, int m) {

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

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

return x;

}

}

return -1;

}

// Function to multiply two matrices

void multiplyMatrices(int matrix1[MATRIX\_SIZE][MATRIX\_SIZE], int matrix2[MATRIX\_SIZE][MATRIX\_SIZE], int result[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

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

result[i][j] = 0;

for (int k = 0; k < MATRIX\_SIZE; k++) {

result[i][j] += matrix1[i][k] \* matrix2[k][j];

}

result[i][j] %= ALPHABET\_SIZE;

}

}

}

// Function to compute the inverse of a matrix

void getMatrixInverse(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], int inverse[MATRIX\_SIZE][MATRIX\_SIZE]) {

int det = (matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]) % ALPHABET\_SIZE;

if (det < 0) det += ALPHABET\_SIZE;

int invDet = modInverse(det, ALPHABET\_SIZE);

inverse[0][0] = matrix[1][1] \* invDet % ALPHABET\_SIZE;

inverse[1][1] = matrix[0][0] \* invDet % ALPHABET\_SIZE;

inverse[0][1] = -matrix[0][1] \* invDet % ALPHABET\_SIZE;

inverse[1][0] = -matrix[1][0] \* invDet % ALPHABET\_SIZE;

if (inverse[0][1] < 0) inverse[0][1] += ALPHABET\_SIZE;

if (inverse[1][0] < 0) inverse[1][0] += ALPHABET\_SIZE;

}

// Function to get the plaintext matrix from plaintext

void getPlaintextMatrix(const char\* plaintext, int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

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

matrix[i][j] = toupper(plaintext[i \* MATRIX\_SIZE + j]) - 'A';

}

}

}

// Function to get the ciphertext matrix from ciphertext

void getCiphertextMatrix(const char\* ciphertext, int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

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

matrix[i][j] = toupper(ciphertext[i \* MATRIX\_SIZE + j]) - 'A';

}

}

}

// Function to compute the key matrix from plaintext and ciphertext matrices

void getKeyMatrix(int plaintextMatrix[MATRIX\_SIZE][MATRIX\_SIZE], int ciphertextMatrix[MATRIX\_SIZE][MATRIX\_SIZE], int keyMatrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

int inversePlaintextMatrix[MATRIX\_SIZE][MATRIX\_SIZE];

getMatrixInverse(plaintextMatrix, inversePlaintextMatrix);

multiplyMatrices(ciphertextMatrix, inversePlaintextMatrix, keyMatrix);

}

OUTPUT  
Key Matrix:

14 6

22 17   
  
14Q. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET\_SIZE 26

// Function prototypes

void encrypt(char\* plaintext, int\* keyStream, char\* ciphertext);

void decrypt(char\* ciphertext, int\* keyStream, char\* plaintext);

void getKeyForDecryption(char\* ciphertext, char\* plaintext, int\* keyStream);

int main() {

// Given plaintext and key stream

char plaintext[] = "send more money";

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

char ciphertext[strlen(plaintext)];

// Encrypt the plaintext

encrypt(plaintext, keyStream, ciphertext);

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

// Decrypt the ciphertext to "cash not needed"

char newPlaintext[] = "cash not needed";

int newKeyStream[strlen(newPlaintext)];

getKeyForDecryption(ciphertext, newPlaintext, newKeyStream);

printf("Key stream for decryption to 'cash not needed':\n");

for (int i = 0; i < strlen(newPlaintext); i++) {

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

}

printf("\n");

return 0;

}

// Function to encrypt plaintext using one-time pad Vigenère cipher

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

int length = strlen(plaintext);

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

if (isspace(plaintext[i])) {

ciphertext[i] = plaintext[i];

} else {

int shift = keyStream[i % length];

ciphertext[i] = ((toupper(plaintext[i]) - 'A' + shift) % ALPHABET\_SIZE) + 'A';

}

}

ciphertext[length] = '\0';

}

// Function to decrypt ciphertext using one-time pad Vigenère cipher

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

int length = strlen(ciphertext);

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

if (isspace(ciphertext[i])) {

plaintext[i] = ciphertext[i];

} else {

int shift = keyStream[i % length];

plaintext[i] = ((toupper(ciphertext[i]) - 'A' - shift + ALPHABET\_SIZE) % ALPHABET\_SIZE) + 'A';

}

}

plaintext[length] = '\0';

}

// Function to get the key stream for decryption to a new plaintext

void getKeyForDecryption(char\* ciphertext, char\* plaintext, int\* keyStream) {

int length = strlen(plaintext);

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

if (isspace(plaintext[i])) {

keyStream[i] = 0;

} else {

keyStream[i] = (toupper(ciphertext[i]) - toupper(plaintext[i]) + ALPHABET\_SIZE) % ALPHABET\_SIZE;

}

}

}

**OUTPUT**  
Ciphertext: BEOK BJFP OWWEY

Key stream for decryption to 'cash not needed':

25 4 22 3 0 14 21 12 0 -20 10 18 19 0 21  
  
15Q. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET\_SIZE 26

#define TOP\_N 10

// English letter frequency distribution

const float englishFreq[ALPHABET\_SIZE] = {

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(char\* ciphertext, int shift, char\* plaintext);

float calculateChiSquared(const int\* observed, const float\* expected, int length);

void analyzeFrequency(const char\* text, int\* frequency);

int main() {

char ciphertext[1000];

int topN;

// Get input from user

printf("Enter ciphertext: ");

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

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

printf("Enter number of top possible plaintexts to display: ");

scanf("%d", &topN);

int length = strlen(ciphertext);

int frequency[ALPHABET\_SIZE] = {0};

char possiblePlaintexts[ALPHABET\_SIZE][length + 1];

float chiSquaredValues[ALPHABET\_SIZE];

// Analyze letter frequency in the ciphertext

analyzeFrequency(ciphertext, frequency);

// Try all possible shifts and calculate chi-squared statistic

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

decrypt(ciphertext, shift, possiblePlaintexts[shift]);

int shiftedFrequency[ALPHABET\_SIZE] = {0};

analyzeFrequency(possiblePlaintexts[shift], shiftedFrequency);

chiSquaredValues[shift] = calculateChiSquared(shiftedFrequency, englishFreq, ALPHABET\_SIZE);

}

// Sort the possible plaintexts based on chi-squared values

for (int i = 0; i < ALPHABET\_SIZE - 1; i++) {

for (int j = i + 1; j < ALPHABET\_SIZE; j++) {

if (chiSquaredValues[i] > chiSquaredValues[j]) {

float tempChi = chiSquaredValues[i];

chiSquaredValues[i] = chiSquaredValues[j];

chiSquaredValues[j] = tempChi;

char tempText[length + 1];

strcpy(tempText, possiblePlaintexts[i]);

strcpy(possiblePlaintexts[i], possiblePlaintexts[j]);

strcpy(possiblePlaintexts[j], tempText);

}

}

}

// Display the top N possible plaintexts

printf("Top %d possible plaintexts:\n", topN);

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

printf("%d: %s\n", i + 1, possiblePlaintexts[i]);

}

return 0;

}

// Function to decrypt the ciphertext with a given shift

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

int length = strlen(ciphertext);

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

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

plaintext[i] = (ciphertext[i] - base - shift + ALPHABET\_SIZE) % ALPHABET\_SIZE + base;

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[length] = '\0';

}

// Function to calculate chi-squared statistic

float calculateChiSquared(const int\* observed, const float\* expected, int length) {

float chiSquared = 0.0;

int total = 0;

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

total += observed[i];

}

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

float expectedCount = expected[i] \* total / 100.0;

float difference = observed[i] - expectedCount;

chiSquared += difference \* difference / expectedCount;

}

return chiSquared;

}

// Function to analyze letter frequency in the text

void analyzeFrequency(const char\* text, int\* frequency) {

int length = strlen(text);

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

if (isalpha(text[i])) {

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

}

}

}

**OUTPUT**  
Enter ciphertext: VISHNU

Enter number of top possible plaintexts to display: 2

Top 2 possible plaintexts:

1: BOYNTA

2: UHRGMT   
  
16Q. #include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

#define ALPHABET\_SIZE 26

#define MAX\_TEXT\_SIZE 1000

#define TOP\_N 10

// English letter frequency distribution

const float englishFreq[ALPHABET\_SIZE] = {

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

};

// Function prototypes

void decrypt(char\* ciphertext, char\* key, char\* plaintext);

float calculateChiSquared(const int\* observed, const float\* expected, int length);

void analyzeFrequency(const char\* text, int\* frequency);

void generateKey(int\* frequency, char\* key);

void printTopNDecryptions(char\* ciphertext, int topN);

int main() {

char ciphertext[MAX\_TEXT\_SIZE];

int topN;

// Get input from user

printf("Enter ciphertext: ");

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

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

printf("Enter number of top possible plaintexts to display: ");

scanf("%d", &topN);

// Print top N possible plaintexts

printTopNDecryptions(ciphertext, topN);

return 0;

}

// Function to decrypt the ciphertext with a given key

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

int length = strlen(ciphertext);

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

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

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

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[length] = '\0';

}

// Function to calculate chi-squared statistic

float calculateChiSquared(const int\* observed, const float\* expected, int length) {

float chiSquared = 0.0;

int total = 0;

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

total += observed[i];

}

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

float expectedCount = expected[i] \* total / 100.0;

float difference = observed[i] - expectedCount;

chiSquared += difference \* difference / expectedCount;

}

return chiSquared;

}

// Function to analyze letter frequency in the text

void analyzeFrequency(const char\* text, int\* frequency) {

int length = strlen(text);

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

if (isalpha(text[i])) {

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

}

}

}

// Function to generate a key based on frequency analysis

void generateKey(int\* frequency, char\* key) {

// Create an array of the alphabet sorted by frequency

char alphabet[ALPHABET\_SIZE] = "ETAOINSHRDLCUMWFGYPBVKJXQZ";

int freqSortedIndices[ALPHABET\_SIZE];

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

freqSortedIndices[i] = i;

}

// Sort the frequencies and track the indices

for (int i = 0; i < ALPHABET\_SIZE - 1; i++) {

for (int j = i + 1; j < ALPHABET\_SIZE; j++) {

if (frequency[freqSortedIndices[i]] < frequency[freqSortedIndices[j]]) {

int temp = freqSortedIndices[i];

freqSortedIndices[i] = freqSortedIndices[j];

freqSortedIndices[j] = temp;

}

}

}

// Map the most frequent ciphertext letters to the most frequent English letters

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

key[freqSortedIndices[i]] = alphabet[i];

}

key[ALPHABET\_SIZE] = '\0';

}

// Function to print the top N decryptions based on chi-squared values

void printTopNDecryptions(char\* ciphertext, int topN) {

int length = strlen(ciphertext);

int frequency[ALPHABET\_SIZE] = {0};

char possiblePlaintexts[ALPHABET\_SIZE][length + 1];

float chiSquaredValues[ALPHABET\_SIZE];

// Analyze letter frequency in the ciphertext

analyzeFrequency(ciphertext, frequency);

// Try all possible keys and calculate chi-squared statistic

char key[ALPHABET\_SIZE + 1];

generateKey(frequency, key);

// Decrypt the ciphertext using the generated key

decrypt(ciphertext, key, possiblePlaintexts[0]);

int shiftedFrequency[ALPHABET\_SIZE] = {0};

analyzeFrequency(possiblePlaintexts[0], shiftedFrequency);

chiSquaredValues[0] = calculateChiSquared(shiftedFrequency, englishFreq, ALPHABET\_SIZE);

// Display the top N possible plaintexts

printf("Top %d possible plaintexts:\n", topN);

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

printf("%d: %s (Chi-Squared: %f)\n", i + 1, possiblePlaintexts[i], chiSquaredValues[i]);

}

}

**OUTPUT**  
Enter ciphertext: VISHNU

Enter number of top possible plaintexts to display: 2

Top 2 possible plaintexts:

1: NTOEAI (Chi-Squared: 6.275428)

2: (Chi-Squared: 0.000000)  
  
17Q. #include <stdio.h>

#include <stdint.h>

#include <string.h>

#define DES\_BLOCK\_SIZE 8

#define DES\_KEY\_SIZE 8

#define NUM\_ROUNDS 16

// Example key schedule and decryption functions

void generateSubKeys(uint8\_t key[DES\_KEY\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE]);

void desDecryptBlock(uint8\_t block[DES\_BLOCK\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE]);

void applyDES(uint8\_t \*data, size\_t length, uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE]);

// Dummy functions for DES operations - placeholder for actual implementation

void permute(uint8\_t \*input, uint8\_t \*output);

void initialPermutation(uint8\_t \*data);

void finalPermutation(uint8\_t \*data);

void feistelFunction(uint8\_t \*data, uint8\_t \*key);

void xorOperation(uint8\_t \*data1, uint8\_t \*data2);

int main() {

uint8\_t key[DES\_KEY\_SIZE] = {0x13, 0x34, 0x57, 0x79, 0x9A, 0xBC, 0xDE, 0xF0}; // Example key

uint8\_t ciphertext[DES\_BLOCK\_SIZE] = {0xAB, 0xCD, 0xEF, 0x01, 0x23, 0x45, 0x67, 0x89}; // Example ciphertext

uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE];

// Generate sub-keys

generateSubKeys(key, subKeys);

// Decrypt the ciphertext

desDecryptBlock(ciphertext, subKeys);

// Print decrypted plaintext

printf("Decrypted plaintext: ");

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

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

}

printf("\n");

return 0;

}

void generateSubKeys(uint8\_t key[DES\_KEY\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE]) {

// Key scheduling algorithm to generate 16 sub-keys from the main key

// This is a simplified placeholder. Replace with actual key schedule logic.

for (int i = 0; i < NUM\_ROUNDS; i++) {

// Use the main key as the sub-key for simplicity

memcpy(subKeys[i], key, DES\_KEY\_SIZE);

}

}

void desDecryptBlock(uint8\_t block[DES\_BLOCK\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][DES\_KEY\_SIZE]) {

// Perform DES decryption on a single block

// Apply initial permutation

initialPermutation(block);

// Apply 16 rounds of DES using sub-keys in reverse order

for (int i = NUM\_ROUNDS - 1; i >= 0; i--) {

feistelFunction(block, subKeys[i]);

}

// Apply final permutation

finalPermutation(block);

}

// Dummy functions for DES operations

void permute(uint8\_t \*input, uint8\_t \*output) {

// Placeholder for permutation function

memcpy(output, input, DES\_BLOCK\_SIZE);

}

void initialPermutation(uint8\_t \*data) {

// Placeholder for initial permutation

// Actual implementation needed

}

void finalPermutation(uint8\_t \*data) {

// Placeholder for final permutation

// Actual implementation needed

}

void feistelFunction(uint8\_t \*data, uint8\_t \*key) {

// Placeholder for Feistel function

// Actual implementation needed

}

void xorOperation(uint8\_t \*data1, uint8\_t \*data2) {

// XOR operation placeholder

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

data1[i] ^= data2[i];

}

}

**OUTPUT**  
Decrypted plaintext: AB CD EF 01 23 45 67 89   
  
18Q. #include <stdio.h>

#include <stdint.h>

#include <string.h>

// Key size and number of sub-keys

#define KEY\_SIZE 8 // 64 bits (8 bytes)

#define HALF\_KEY\_SIZE 4 // 32 bits (4 bytes)

#define NUM\_ROUNDS 16

#define SUB\_KEY\_SIZE 6 // 48 bits (6 bytes)

// Permutation and shift values (placeholders, real values needed for DES)

const int SHIFT\_SCHEDULE[NUM\_ROUNDS] = {1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2};

// Function prototypes

void generateSubKeys(uint8\_t key[KEY\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][SUB\_KEY\_SIZE]);

void permuteLeftShift(uint8\_t \*keyHalf, int shift);

void applyPermutedChoice(uint8\_t \*c, uint8\_t \*d, uint8\_t \*subKey);

int main() {

uint8\_t key[KEY\_SIZE] = {0x13, 0x34, 0x57, 0x79, 0x9A, 0xBC, 0xDE, 0xF0}; // Example 64-bit key

uint8\_t subKeys[NUM\_ROUNDS][SUB\_KEY\_SIZE];

// Generate sub-keys

generateSubKeys(key, subKeys);

// Print sub-keys

for (int i = 0; i < NUM\_ROUNDS; i++) {

printf("Sub-key %d: ", i + 1);

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

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

}

printf("\n");

}

return 0;

}

void generateSubKeys(uint8\_t key[KEY\_SIZE], uint8\_t subKeys[NUM\_ROUNDS][SUB\_KEY\_SIZE]) {

uint8\_t c[HALF\_KEY\_SIZE], d[HALF\_KEY\_SIZE], tempKey[KEY\_SIZE];

// Initial permutation of key into C and D halves

memcpy(c, key, HALF\_KEY\_SIZE); // First 28 bits (4 bytes)

memcpy(d, key + HALF\_KEY\_SIZE, HALF\_KEY\_SIZE); // Last 28 bits (4 bytes)

for (int i = 0; i < NUM\_ROUNDS; i++) {

// Left shift C and D halves

permuteLeftShift(c, SHIFT\_SCHEDULE[i]);

permuteLeftShift(d, SHIFT\_SCHEDULE[i]);

// Apply permutation choice 2 (PC-2) to combine C and D into the sub-key

applyPermutedChoice(c, d, subKeys[i]);

}

}

void permuteLeftShift(uint8\_t \*keyHalf, int shift) {

uint32\_t temp;

temp = \*((uint32\_t \*)keyHalf);

temp = (temp << shift) | (temp >> (28 - shift)); // Circular shift

\*((uint32\_t \*)keyHalf) = temp;

}

void applyPermutedChoice(uint8\_t \*c, uint8\_t \*d, uint8\_t \*subKey) {

// Combine C and D halves into the 48-bit sub-key

// Real DES permutation choice 2 (PC-2) logic needed here

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

subKey[i] = c[i];

}

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

subKey[HALF\_KEY\_SIZE + i] = d[i];

}

}

**OUTPUT**  
Sub-key 1: 2F 68 AE F2 3E 79

Sub-key 2: 5E D0 5C E5 7C F2

Sub-key 3: 79 41 73 95 F0 C9

Sub-key 4: E5 05 CD 55 C3 27

Sub-key 5: 95 17 34 57 0D 9F

Sub-key 6: 55 5E D0 5C 37 7C

Sub-key 7: AB BC A0 B9 6F F8

Sub-key 8: 57 79 41 73 DE F0

Sub-key 9: 5C E5 05 CD 7A C3

Sub-key 10: 73 95 17 34 EB 0D

Sub-key 11: CD 55 5E D0 AF 37

Sub-key 12: 34 57 79 41 BC DE

Sub-key 13: 68 AE F2 82 79 BD

Sub-key 14: D0 5C E5 05 F2 7A

Sub-key 15: 41 73 95 17 C9 EB

Sub-key 16: 05 CD 55 5E 27 AF