Caesar cipher

```
#include <stdio.h>
#include <ctype.h>

// Function to encrypt the text using Caesar cipher
void encrypt(char text[], int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        char c = text[i];
        if (isalpha(c)) {
            char offset = isupper(c) ? 'A' : 'a';
            c = (c - offset + shift) % 26 + offset;
        }
        text[i] = c;
    }
}

// Function to decrypt the text using Caesar cipher
void decrypt(char text[], int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        char c = text[i];
        if (isalpha(c)) {
            char offset = isupper(c) ? 'A' : 'a';
            c = (c - offset - shift + 26) % 26 + offset;
        }
        text[i] = c;
    }
```

```
}

int main() {
    char text[100];
    int shift;
    int choice;

    printf("Enter the text: ");
    fgets(text, sizeof(text), stdin);

    printf("Enter the shift value: ");
    scanf("%d", &shift);

    printf("Choose an option:\n1. Encrypt\n2. Decrypt\n");
    scanf("%d", &choice);

    if (choice == 1) {
        encrypt(text, shift);
        printf("Encrypted text: %s\n", text);
    } else if (choice == 2) {
        decrypt(text, shift);
        printf("Decrypted text: %s\n", text);
    } else {
        printf("Invalid choice.\n");
    }
```

```
    return 0;
}
```

monoalphabetic substitution cipher

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Function to encrypt the text using a monoalphabetic substitution cipher
void encrypt(char text[], const char key[]) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            if (islower(text[i])) {
                text[i] = key[text[i] - 'a'];
            } else if (isupper(text[i])) {
                text[i] = toupper(key[text[i] - 'A']);
            }
        }
    }
}

// Function to decrypt the text using a monoalphabetic substitution cipher
void decrypt(char text[], const char key[]) {
    char reverseKey[26];
    for (int i = 0; i < 26; i++) {
        reverseKey[key[i] - 'a'] = 'a' + i;
    }
```

```
    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            if (islower(text[i])) {
                text[i] = reverseKey[text[i] - 'a'];
            } else if (isupper(text[i])) {
                text[i] = toupper(reverseKey[text[i] - 'A']);
            }
        }
    }
}

int main() {
    char text[100];
    char key[27] = "phqgiumeaylnofdxjkrcvstzwb"; // example key

    printf("Enter text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // remove newline character

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

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

    decrypt(text, key);
```

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

    return 0;
}
```

C program for Playfair algorithm

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 5

// Function to create the Playfair matrix
void createMatrix(char key[], char matrix[SIZE][SIZE]) {
    int alphabet[26] = {0};
    int len = strlen(key);
    int x = 0, y = 0;

    for (int i = 0; i < len; i++) {
        if (key[i] == 'j') key[i] = 'i';
        if (alphabet[key[i] - 'a'] == 0) {
            matrix[x][y] = key[i];
            alphabet[key[i] - 'a'] = 1;
            y++;
            if (y == SIZE) {
                y = 0;
                x++;
```

```
            }
        }
    }

    for (char c = 'a'; c <= 'z'; c++) {
        if (c == 'j') continue;
        if (alphabet[c - 'a'] == 0) {
            matrix[x][y] = c;
            alphabet[c - 'a'] = 1;
            y++;
            if (y == SIZE) {
                y = 0;
                x++;
            }
        }
    }
}

// Function to print the Playfair matrix
void printMatrix(char matrix[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%c ", matrix[i][j]);
        }
        printf("\n");
    }
```

```
}

// Function to find the position of a letter in the matrix
void findPosition(char matrix[SIZE][SIZE], char letter, int *row, int *col) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (matrix[i][j] == letter) {
                *row = i;
                *col = j;
                return;
            }
        }
    }
}

// Function to prepare the plaintext (inserts 'x' between repeating letters and
at the end if necessary)
void prepareText(char text[]) {
    int len = strlen(text);
    for (int i = 0; i < len; i++) {
        if (text[i] == 'j') text[i] = 'i';
    }

    for (int i = 0; i < len - 1; i++) {
        if (text[i] == text[i + 1]) {
            for (int j = len; j > i + 1; j--) {
                text[j] = text[j - 1];
```

```
            }
            text[i + 1] = 'x';
            len++;
        }
    }
    if (len % 2 != 0) {
        text[len] = 'x';
        text[len + 1] = '\0';
    }
}

// Function to encrypt the plaintext using the Playfair cipher
void encrypt(char text[], char matrix[SIZE][SIZE]) {
    int len = strlen(text);
    for (int i = 0; i < len; i += 2) {
        int row1, col1, row2, col2;
        findPosition(matrix, text[i], &row1, &col1);
        findPosition(matrix, text[i + 1], &row2, &col2);

        if (row1 == row2) {
            text[i] = matrix[row1][(col1 + 1) % SIZE];
            text[i + 1] = matrix[row2][(col2 + 1) % SIZE];
        } else if (col1 == col2) {
            text[i] = matrix[(row1 + 1) % SIZE][col1];
            text[i + 1] = matrix[(row2 + 1) % SIZE][col2];
        } else {
```

```
            text[i] = matrix[row1][col2];
            text[i + 1] = matrix[row2][col1];
        }
    }
}

// Function to decrypt the ciphertext using the Playfair cipher
void decrypt(char text[], char matrix[SIZE][SIZE]) {
    int len = strlen(text);
    for (int i = 0; i < len; i += 2) {
        int row1, col1, row2, col2;
        findPosition(matrix, text[i], &row1, &col1);
        findPosition(matrix, text[i + 1], &row2, &col2);

        if (row1 == row2) {
            text[i] = matrix[row1][(col1 - 1 + SIZE) % SIZE];
            text[i + 1] = matrix[row2][(col2 - 1 + SIZE) % SIZE];
        } else if (col1 == col2) {
            text[i] = matrix[(row1 - 1 + SIZE) % SIZE][col1];
            text[i + 1] = matrix[(row2 - 1 + SIZE) % SIZE][col2];
        } else {
            text[i] = matrix[row1][col2];
            text[i + 1] = matrix[row2][col1];
        }
    }
}
```

```
int main() {
    char key[] = "playfair example";
    char text[100];
    char matrix[SIZE][SIZE];

    // Remove spaces and prepare the key
    int key_len = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        if (isalpha(key[i])) {
            key[key_len++] = tolower(key[i]);
        }
    }
    key[key_len] = '\0';

    createMatrix(key, matrix);

    printf("Playfair Matrix:\n");
    printMatrix(matrix);

    printf("Enter text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // remove newline character

    prepareText(text);
```

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

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

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

    return 0;
}
```

Hill cipher

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 2

// Function to multiply two matrices
void multiplyMatrix(int a[SIZE][SIZE], int b[SIZE], int result[SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        result[i] = 0;
        for (int j = 0; j < SIZE; j++) {
            result[i] += a[i][j] * b[j];
        }
        result[i] %= 26;
    }
```

```
}

// Function to find the modular inverse of a number
int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 0) {
            return x;
        }
    }
    return -1;
}

// Function to find the determinant of a 2x2 matrix
int determinant(int matrix[SIZE][SIZE]) {
    return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]) % 26;
}

// Function to find the inverse of a 2x2 matrix
void inverseMatrix(int matrix[SIZE][SIZE], int inverse[SIZE][SIZE]) {
    int det = determinant(matrix);
    int invDet = modInverse(det, 26);
    if (invDet == -1) {
        printf("Matrix is not invertible.\n");
        return;
    }
```

```
    inverse[0][0] = (matrix[1][1] * invDet) % 26;
    inverse[0][1] = (-matrix[0][1] * invDet) % 26;
    inverse[1][0] = (-matrix[1][0] * invDet) % 26;
    inverse[1][1] = (matrix[0][0] * invDet) % 26;

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (inverse[i][j] < 0) {
                inverse[i][j] += 26;
            }
        }
    }
}

// Function to encrypt the plaintext using the Hill cipher
void encrypt(char text[], int key[SIZE][SIZE]) {
    int len = strlen(text);
    for (int i = 0; i < len; i += SIZE) {
        int vector[SIZE];
        int result[SIZE];
        for (int j = 0; j < SIZE; j++) {
            vector[j] = text[i + j] - 'a';
        }
        multiplyMatrix(key, vector, result);
        for (int j = 0; j < SIZE; j++) {
```

```
            text[i + j] = result[j] + 'a';
        }
    }
}

// Function to decrypt the ciphertext using the Hill cipher
void decrypt(char text[], int key[SIZE][SIZE]) {
    int inverse[SIZE][SIZE];
    inverseMatrix(key, inverse);
    int len = strlen(text);
    for (int i = 0; i < len; i += SIZE) {
        int vector[SIZE];
        int result[SIZE];
        for (int j = 0; j < SIZE; j++) {
            vector[j] = text[i + j] - 'a';
        }
        multiplyMatrix(inverse, vector, result);
        for (int j = 0; j < SIZE; j++) {
            text[i + j] = result[j] + 'a';
        }
    }
}

int main() {
    int key[SIZE][SIZE] = {{5, 17}, {4, 15}}; // example key
    char text[100];
```

```
    printf("Enter text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // remove newline character

    // Ensure text length is a multiple of SIZE by padding with 'x'
    int len = strlen(text);
    if (len % SIZE != 0) {
        for (int i = len; i < len + SIZE - (len % SIZE); i++) {
            text[i] = 'x';
        }
        text[len + SIZE - (len % SIZE)] = '\0';
    }

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

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

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

    return 0;
}
```

C program that can perform a letter frequency attack

```
#include <stdio.h>
```

```
#include <string.h>
#include <ctype.h>

#define ALPHABET_SIZE 26

// Function to analyze letter frequency in the ciphertext
void analyzeFrequency(char text[], int freq[]) {
    for (int i = 0; i < strlen(text); i++) {
        if (isalpha(text[i])) {
            freq[tolower(text[i]) - 'a']++;
        }
    }
}

// Function to print letter frequencies
void printFrequencies(int freq[]) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        printf("%c: %d\n", 'a' + i, freq[i]);
    }
}

// Function to sort frequencies and map them to the most common English letters
void mapFrequencies(int freq[], char mapping[]) {
    // Frequency order of letters in English (from most frequent to least frequent)
    char frequencyOrder[] = "etaoinshrdlcumwfgypbvkjxqz";
```

```
// Create an array of letter-frequency pairs
int letter_freq[ALPHABET_SIZE][2];
for (int i = 0; i < ALPHABET_SIZE; i++) {
   letter_freq[i][0] = i;
   letter_freq[i][1] = freq[i];
}

// Sort the array by frequency
for (int i = 0; i < ALPHABET_SIZE - 1; i++) {
   for (int j = i + 1; j < ALPHABET_SIZE; j++) {
      if (letter_freq[i][1] < letter_freq[j][1]) {
         int temp0 = letter_freq[i][0];
         int temp1 = letter_freq[i][1];
         letter_freq[i][0] = letter_freq[j][0];
         letter_freq[i][1] = letter_freq[j][1];
         letter_freq[j][0] = temp0;
         letter_freq[j][1] = temp1;
      }
   }
}

// Map the sorted frequencies to the most common English letters
for (int i = 0; i < ALPHABET_SIZE; i++) {
   mapping[letter_freq[i][0]] = frequencyOrder[i];
}
```

```
}

// Function to decrypt the ciphertext using the frequency mapping
void decrypt(char text[], char mapping[]) {
    for (int i = 0; i < strlen(text); i++) {
        if (isalpha(text[i])) {
            char mapped_char = mapping[tolower(text[i]) - 'a'];
            if (isupper(text[i])) {
                text[i] = toupper(mapped_char);
            } else {
                text[i] = mapped_char;
            }
        }
    }
}

int main() {
    char text[1000];
    int freq[ALPHABET_SIZE] = {0};
    char mapping[ALPHABET_SIZE] = {0};

    printf("Enter ciphertext: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline character

    analyzeFrequency(text, freq);
```

```
    printf("\nLetter Frequencies:\n");
    printFrequencies(freq);

    mapFrequencies(freq, mapping);

    printf("\nFrequency Mapping:\n");
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        printf("%c -> %c\n", 'a' + i, mapping[i]);
    }

    decrypt(text, mapping);
    printf("\nDecrypted text: %s\n", text);

    return 0;
}
```

C program for DES algorithm for decryption

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DES_BLOCK_SIZE 8

// Initial permutation table
int IP[] = {
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
```

```
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17,  9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
};

// Final permutation table
int FP[] = {
    40,  8, 48, 16, 56, 24, 64, 32,
    39,  7, 47, 15, 55, 23, 63, 31,
    38,  6, 46, 14, 54, 22, 62, 30,
    37,  5, 45, 13, 53, 21, 61, 29,
    36,  4, 44, 12, 52, 20, 60, 28,
    35,  3, 43, 11, 51, 19, 59, 27,
    34,  2, 42, 10, 50, 18, 58, 26,
    33,  1, 41,  9, 49, 17, 57, 25
};

// Example key schedule for simplicity
int key_schedule[16][48];

// Expansion table
int E[] = {
    32,  1,  2,  3,  4,  5,
```

```
     4,  5,  6,  7,  8,  9,
     8,  9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32,  1
};

// S-boxes
int S[8][4][16] = { ... /* S-box values */ ... };

// P permutation table
int P[] = {
    16,  7, 20, 21,
    29, 12, 28, 17,
     1, 15, 23, 26,
     5, 18, 31, 10,
     2,  8, 24, 14,
    32, 27,  3,  9,
    19, 13, 30,  6,
    22, 11,  4, 25
};

// Function to apply a permutation
void permute(unsigned char *input, unsigned char *output, int *table, int n) {
```

```
    for (int i = 0; i < n; i++) {
        int pos = table[i] - 1;
        output[i / 8] |= ((input[pos / 8] >> (7 - (pos % 8))) & 0x01) << (7 - (i % 8));
    }
}

// Function to perform the DES Feistel function
void feistel(unsigned char *right, int *subkey, unsigned char *output) {
    unsigned char expanded[6] = {0};
    unsigned char sbox_input[6] = {0};
    unsigned char sbox_output[4] = {0};
    unsigned char permuted[4] = {0};

    // Expansion
    permute(right, expanded, E, 48);

    // XOR with subkey
    for (int i = 0; i < 6; i++) {
        sbox_input[i] = expanded[i] ^ subkey[i];
    }

    // S-box substitution
    for (int i = 0; i < 8; i++) {
        int row = ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 0))) & 0x01) << 1 |
                  ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 5))) & 0x01);
        int col = ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 1))) & 0x01) << 3 |
```

```
            ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 2))) & 0x01) << 2 |
            ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 3))) & 0x01) << 1 |
            ((sbox_input[i / 6] >> (7 - (i * 6 % 48 + 4))) & 0x01);
        sbox_output[i / 2] |= S[i][row][col] << (4 * (1 - (i % 2)));
    }

    // P permutation
    permute(sbox_output, permuted, P, 32);

    // Output
    memcpy(output, permuted, 4);
}

// Function to perform DES decryption on a single block
void des_decrypt_block(unsigned char *input, unsigned char *output) {
    unsigned char ip[8] = {0};
    unsigned char fp[8] = {0};
    unsigned char left[4] = {0};
    unsigned char right[4] = {0};
    unsigned char temp[4] = {0};

    // Initial permutation
    permute(input, ip, IP, 64);

    // Split into left and right halves
    memcpy(left, ip, 4);
```

```
    memcpy(right, ip + 4, 4);

    // 16 rounds of processing
    for (int i = 15; i >= 0; i--) {
        memcpy(temp, right, 4);
        feistel(right, key_schedule[i], right);
        for (int j = 0; j < 4; j++) {
            right[j] ^= left[j];
        }
        memcpy(left, temp, 4);
    }

    // Combine halves
    memcpy(fp, right, 4);
    memcpy(fp + 4, left, 4);

    // Final permutation
    permute(fp, output, FP, 64);
}

int main() {
    // Example ciphertext block (64 bits / 8 bytes)
    unsigned char ciphertext[DES_BLOCK_SIZE] = {0x85, 0xE8, 0x13, 0x54, 0x0F,
0x0A, 0xB4, 0x05};
    unsigned char plaintext[DES_BLOCK_SIZE] = {0};
```