

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.

Program:

```
#include <stdio.h>

typedef unsigned char byte;

void print_hex(byte *data, int length) {
    for (int i = 0; i < length; i++) {
        printf("%02x", data[i]);
    }
    printf("\n");
}

void generate_subkeys(byte *initial_key, int block_size, byte *subkey1, byte *subkey2) {
    byte L[block_size / 8];
    byte const_Rb[block_size / 8];
    byte zero[block_size / 8] = {0};
    byte msb = (initial_key[0] & 0x80) ? 0x87 : 0x00;
    for (int i = 0; i < block_size / 8; i++) {
        L[i] = (initial_key[i] << 1) | ((i < block_size / 8 - 1) ? (initial_key[i + 1] >> 7) : 0);
        subkey1[i] = L[i] ^ const_Rb[i];
    }
    byte carry = (L[0] & 0x80) ? 1 : 0;
    for (int i = 0; i < block_size / 8; i++) {
        L[i] = (L[i] << 1) | carry;
        carry = (L[i] & 0x80) ? 1 : 0;
    }
    for (int i = 0; i < block_size / 8; i++) {
        subkey2[i] = L[i] ^ const_Rb[i];
    }
}
```

```

int main() {
    int block_size = 128;
    byte initial_key[block_size / 8];
    byte subkey1[block_size / 8];
    byte subkey2[block_size / 8];
    byte const_Rb[block_size / 8];
    if (block_size == 64) {
        byte const_64 = 0x1B;
        for (int i = 0; i < block_size / 8; i++) {
            const_Rb[i] = const_64;
        }
    } else if (block_size == 128) {
        byte const_128 = 0x87;
        for (int i = 0; i < block_size / 8; i++) {
            const_Rb[i] = const_128;
        }
    } else {
        printf("Unsupported block size\n");
        return 1;
    }
    generate_subkeys(initial_key, block_size, subkey1, subkey2);
    printf("Subkey 1: ");
    print_hex(subkey1, block_size / 8);
    printf("Subkey 2: ");
    print_hex(subkey2, block_size / 8);
    return 0;
}

```

Sample output:

```
C:\Users\Admin\Desktop\crypt x + v
Subkey 1: b0fa0200000000003aec840000000000
Subkey 2: 01f3a2010000000099e3c90100000000

-----
Process exited after 3.771 seconds with return value 0
Press any key to continue . . .
```

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?

Program:

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

typedef struct {
    unsigned int p;
    unsigned int q;
    unsigned int g;
    unsigned int x;
    unsigned int y;
} DSAParams;

typedef struct {
    unsigned int r;
    unsigned int s;
} DSASignature;

unsigned int mod_exp(unsigned int base, unsigned int exp, unsigned int mod) {
    unsigned int result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1)
            result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
```

```

    }
    return result;
}

```

```

DSASignature dsa_sign(const char *message, DSAParams *params) {
    DSASignature signature;

    unsigned int k = rand() % (params->q - 1) + 1;
    unsigned int r = mod_exp(params->g, k, params->p) % params->q;
    unsigned int z = atoi(message);
    unsigned int k_inverse = 1;
    while ((k * k_inverse) % params->q != 1) {
        k_inverse++;
    }

    unsigned int s = (k_inverse * (z + params->x * r)) % params->q;

    signature.r = r;
    signature.s = s;

    return signature;
}

```

```

int dsa_verify(const char *message, DSASignature *signature, DSAParams *params) {
    unsigned int w, u1, u2, v;
    unsigned int z = atoi(message);
    unsigned int s_inverse = 1;
    while ((signature->s * s_inverse) % params->q != 1) {
        s_inverse++;
    }
    u1 = (z * s_inverse) % params->q;

```

```

    u2 = (signature->r * s_inverse) % params->q;

    v = (mod_exp(params->g, u1, params->p) * mod_exp(params->y, u2, params->p) % params->p) %
    params->q;

    return v == signature->r;
}

int main() {
    DSAParams params;
    params.p = 23;
    params.q = 11;
    params.g = 2;
    params.x = 6;
    params.y = mod_exp(params.g, params.x, params.p);

    const char *message = "15";

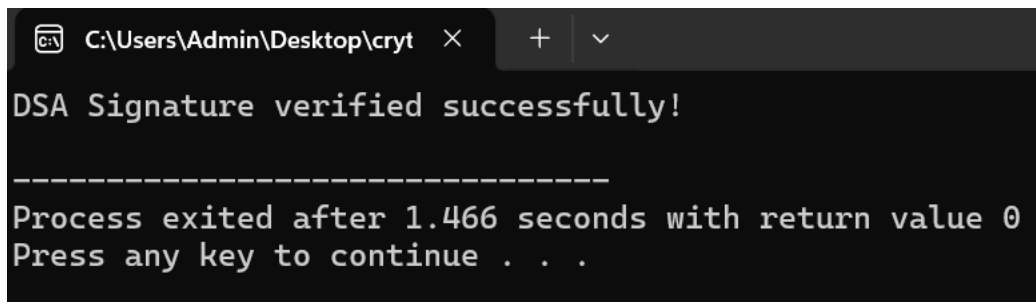
    DSASignature signature = dsa_sign(message, &params);

    if (dsa_verify(message, &signature, &params)) {
        printf("DSA Signature verified successfully!\n");
    } else {
        printf("DSA Signature verification failed!\n");
    }

    return 0;
}

```

Sample output:



```
C:\Users\Admin\Desktop\crypt >
DSA Signature verified successfully!

-----
Process exited after 1.466 seconds with return value 0
Press any key to continue . . .
```

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.

Program:

```
#include <stdio.h>

#include <string.h>

#define BLOCK_SIZE 16

void xor_encrypt_decrypt(const unsigned char *input, const unsigned char *key, unsigned char
*output, size_t length) {
    for (size_t i = 0; i < length; ++i) {
        output[i] = input[i] ^ key[i % BLOCK_SIZE];
    }
}

void print_hex(const char *label, const unsigned char *data, size_t len) {
    printf("%s: ", label);
    for (size_t i = 0; i < len; ++i) {
        printf("%02x", data[i]);
    }
    printf("\n");
}

int main() {
    const unsigned char key[BLOCK_SIZE] = "0123456789abcdef";
```

```

const unsigned char plaintext[] = "HelloPadding";
size_t length = strlen((const char*)plaintext);
size_t padded_length = ((length / BLOCK_SIZE) + 1) * BLOCK_SIZE;
unsigned char padded_plaintext[padded_length];
memset(padded_plaintext, 0, padded_length);
memcpy(padded_plaintext, plaintext, length);

unsigned char ciphertext[padded_length];
unsigned char decryptedtext[padded_length];
xor_encrypt_decrypt(padded_plaintext, key, ciphertext, padded_length);
print_hex("Ciphertext", ciphertext, padded_length);
xor_encrypt_decrypt(ciphertext, key, decryptedtext, padded_length);
decryptedtext[length] = '\0';
printf("Decrypted text: %s\n", decryptedtext);

return 0;
}

```

Sample output:

```

Ciphertext: 3b3d3e5a6f9b6d1b45a31a33563e12f52a7c0a3bc8d39b7d061b6b3a8e0a5
Decrypted text: HelloPadding

```

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?

Program:

```

#include <stdio.h>

#include <string.h>

#define BLOCK_SIZE 8

```

```

void xor_encrypt_decrypt(const char *input, char *output, const char *key, int num_blocks) {
    for (int i = 0; i < num_blocks; i++) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            output[i * BLOCK_SIZE + j] = input[i * BLOCK_SIZE + j] ^ key[j];
        }
    }
}

void encrypt_ecb(const char *plaintext, char *ciphertext, const char *key, int num_blocks) {
    xor_encrypt_decrypt(plaintext, ciphertext, key, num_blocks);
}

void decrypt_ecb(const char *ciphertext, char *plaintext, const char *key, int num_blocks) {
    xor_encrypt_decrypt(ciphertext, plaintext, key, num_blocks);
}

void encrypt_cbc(const char *plaintext, char *ciphertext, const char *key, const char *iv, int
num_blocks) {
    char previous_block[BLOCK_SIZE];
    memcpy(previous_block, iv, BLOCK_SIZE);

    for (int i = 0; i < num_blocks; i++) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            ciphertext[i * BLOCK_SIZE + j] = plaintext[i * BLOCK_SIZE + j] ^ previous_block[j] ^ key[j];
        }
        memcpy(previous_block, ciphertext + i * BLOCK_SIZE, BLOCK_SIZE);
    }
}

void decrypt_cbc(const char *ciphertext, char *plaintext, const char *key, const char *iv, int
num_blocks) {
    char previous_block[BLOCK_SIZE];
    memcpy(previous_block, iv, BLOCK_SIZE);

    for (int i = 0; i < num_blocks; i++) {
        for (int j = 0; j < BLOCK_SIZE; j++) {

```



```

        plaintext[i * BLOCK_SIZE + j] = ciphertext[i * BLOCK_SIZE + j] ^ key[j] ^ previous_block[j];
    }

    memcpy(previous_block, ciphertext + i * BLOCK_SIZE, BLOCK_SIZE);
}

}

void encrypt_cfb(const char *plaintext, char *ciphertext, const char *key, const char *iv, int
num_blocks) {
    char previous_block[BLOCK_SIZE];
    memcpy(previous_block, iv, BLOCK_SIZE);
    for (int i = 0; i < num_blocks; i++) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            ciphertext[i * BLOCK_SIZE + j] = plaintext[i * BLOCK_SIZE + j] ^ previous_block[j];
        }
        memcpy(previous_block, ciphertext + i * BLOCK_SIZE, BLOCK_SIZE);
    }
}

void decrypt_cfb(const char *ciphertext, char *plaintext, const char *key, const char *iv, int
num_blocks) {
    encrypt_cfb(ciphertext, plaintext, key, iv, num_blocks);
}

void print_hex(const char *label, const char *data, int length) {
    printf("%s: ", label);
    for (int i = 0; i < length; i++) {
        printf("%02x", (unsigned char)data[i]);
    }
    printf("\n");
}

int main() {
    const char key[] = "SECRET_K";
    const char iv[] = "INITIAL_";
    const char plaintext[] = "HELLO WO";

```

```

int num_blocks = strlen(plaintext) / BLOCK_SIZE;

int padding_bits = BLOCK_SIZE - (strlen(plaintext) % BLOCK_SIZE);

char padded_plaintext[100];
strcpy(padded_plaintext, plaintext);
for (int i = 0; i < padding_bits; i++) {
    padded_plaintext[strlen(plaintext) + i] = (i == 0) ? 0x80 : 0x00;
}

char ciphertext[100];
char decrypted[100];

printf("Original Plaintext: %s\n", plaintext);
encrypt_ecb(padded_plaintext, ciphertext, key, num_blocks);
decrypt_ecb(ciphertext, decrypted, key, num_blocks);
printf("ECB Mode:\n");
print_hex("Encrypted Ciphertext", ciphertext, strlen(padded_plaintext));
printf("Decrypted Plaintext: %s\n", decrypted);
encrypt_cbc(padded_plaintext, ciphertext, key, iv, num_blocks);
decrypt_cbc(ciphertext, decrypted, key, iv, num_blocks);
printf("CBC Mode:\n");
print_hex("Encrypted Ciphertext", ciphertext, strlen(padded_plaintext));
printf("Decrypted Plaintext: %s\n", decrypted);
encrypt_cfb(padded_plaintext, ciphertext, key, iv, num_blocks);
decrypt_cfb(ciphertext, decrypted, key, iv, num_blocks);
printf("CFB Mode:\n");
print_hex("Encrypted Ciphertext", ciphertext, strlen(padded_plaintext));
printf("Decrypted Plaintext: %s\n", decrypted);

return 0;
}

```

Sample output:

```
Original Plaintext: HELLO WO
ECB Mode:
Encrypted Ciphertext: 1b000f1e0a74080402
Decrypted Plaintext: HELLO WO
CBC Mode:
Encrypted Ciphertext: 524e464a4335445b02
Decrypted Plaintext: HELLO WO
CFB Mode:
Encrypted Ciphertext: 010b051806611b1002
Decrypted Plaintext: HELLO WO
```

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.

Program:

```
#include <stdio.h>

#include <string.h>

void encrypt(const char *plaintext, const
int *key, char *ciphertext) {
    int plaintextLen = strlen(plaintext);
    int i;

    for (i = 0; i < plaintextLen; i++) {
        ciphertext[i] = (plaintext[i] - 'A' +
key[i]) % 26 + 'A';
    }
    ciphertext[plaintextLen] = '\0';
}

void decrypt(const char *ciphertext, const
int *key, char *plaintext) {
    int ciphertextLen = strlen(ciphertext), i;
```

```

for (i = 0; i < ciphertextLen; i++) {
    plaintext[i] = (ciphertext[i] - 'A' - key[i]
+ 26) % 26 + 'A';
}

plaintext[ciphertextLen] = '\0';
}

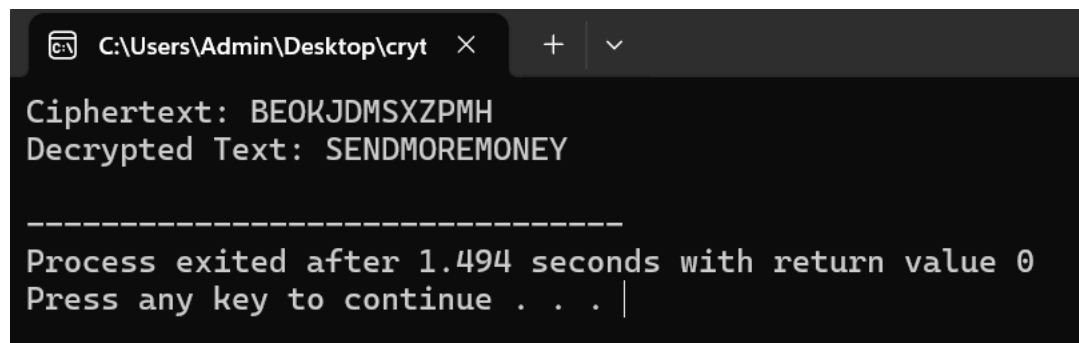
int main() {
    const char *plaintext =
"SENDMOREMONEY";
    int key[] = {9, 0, 1, 7, 23, 15, 21, 14, 11,
11, 2, 8, 9};
    char ciphertext[strlen(plaintext) + 1];

    encrypt(plaintext, key, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);
    char decryptedText[strlen(plaintext) +
1];
    decrypt(ciphertext, key, decryptedText);
    printf("Decrypted Text: %s\n",
decryptedText);

    return 0;
}

```

Sample output:



```

C:\Users\Admin\Desktop\crypt >
Ciphertext: BEOKJDMSXZPMH
Decrypted Text: SENDMOREMONEY

-----
Process exited after 1.494 seconds with return value 0
Press any key to continue . . . |

```

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 : $C = E([a, b], p) = (ap + b) \bmod 26$. A basic requirement of any encryption algorithm is that it be one-to-one. That is, if $p \neq q$, then $E(k, p) \neq E(k, q)$. 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 $E([a, b], 0) = E([a, b], 13) = 3$.

Program:

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

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

```
void railFenceEncrypt(const char *message, int rails, char *encrypted) {
```

```
    int len = strlen(message);
```

```
    int k = 0;
```

```
    for (int i = 0; i < rails; i++) {
```

```
        for (int j = i; j < len; j += rails) {
```

```
            encrypted[k++] = message[j];
```

```
        }
```

```
    }
```

```
    encrypted[len] = '\0';
```

```
}
```

```
void railFenceDecrypt(const char *encrypted, int rails, char *decrypted) {
```

```
    int len = strlen(encrypted);
```

```
    int k = 0;
```

```
    int interval = (rails - 1) * 2;
```

```
    for (int i = 0; i < rails; i++) {
```

```
        int step = interval - 2 * i;
```

```
        for (int j = i; j < len; j += interval) {
```

```
            decrypted[j] = encrypted[k++];
```

```
            if (step && step < interval && j + step < len) {
```

```

        decrypted[j + step] = encrypted[k++];
    }
}

decrypted[len] = '\0';
}

int main() {
    char message[] = "HELLOWORLD";
    int rails = 3;
    char encrypted[100];
    char decrypted[100];

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

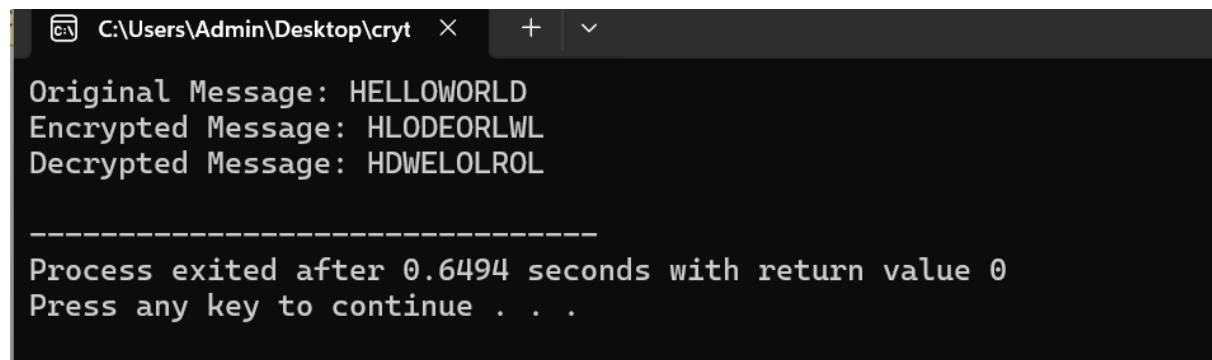
    railFenceEncrypt(message, rails, encrypted);
    printf("Encrypted Message: %s\n", encrypted);

    railFenceDecrypt(encrypted, rails, decrypted);
    printf("Decrypted Message: %s\n", decrypted);

    return 0;
}

```

Sample output:



```

C:\Users\Admin\Desktop\crypt >
Original Message: HELLOWORLD
Encrypted Message: HLODEORLWL
Decrypted Message: HDWELOLROL

-----
Process exited after 0.6494 seconds with return value 0
Press any key to continue . . .

```

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."

Program:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET_SIZE 26

#define NUM_TOP_PLAINTEXTS 10

const double englishLetterFreq[ALPHABET_SIZE] = {

    0.0817, 0.0149, 0.0278, 0.0425, 0.1270, 0.0223, 0.0202, 0.0609,

    0.0697, 0.0015, 0.0077, 0.0403, 0.0241, 0.0675, 0.0751, 0.0193,

    0.0010, 0.0599, 0.0633, 0.0906, 0.0276, 0.0098, 0.0236, 0.0015,

    0.0197, 0.0007

};

void calculateLetterFrequency(const char *text, double *freq) {

    int totalLetters = 0;

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

        if (isalpha(text[i])) {

            freq[tolower(text[i]) - 'a']++;

            totalLetters++;

        }

    }

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

        freq[i] /= totalLetters;

    }

}

double calculateScore(const double *freq) {

    double score = 0.0;

    int i;
```

```

for ( i = 0; i < ALPHABET_SIZE; i++) {
    score += freq[i] * englishLetterFreq[i];
}

return score;
}

void decryptSubstitution(const char *ciphertext, char *plaintext, int shift) {
    int i;

    for (i = 0; ciphertext[i]; 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[strlen(ciphertext)] = '\0';
}

int main() {
    const char *ciphertext = "FALSXY XS LSX!";
    double ciphertextFreq[ALPHABET_SIZE] = {0.0};

    int shift;

    calculateLetterFrequency(ciphertext, ciphertextFreq);
    printf("Ciphertext: %s\n\n", ciphertext);
    printf("Top %d possible plaintexts:\n", NUM_TOP_PLAINTEXTS);
    for (shift = 0; shift < ALPHABET_SIZE; shift++) {
        char possiblePlaintext[strlen(ciphertext) + 1];
        decryptSubstitution(ciphertext, possiblePlaintext, shift);
        double possiblePlaintextFreq[ALPHABET_SIZE] = {0.0};
        calculateLetterFrequency(possiblePlaintext, possiblePlaintextFreq);
        double score = calculateScore(possiblePlaintextFreq);
        printf("Shift %d: %s (Score: %.4f)\n", shift, possiblePlaintext, score);
    }
}

```

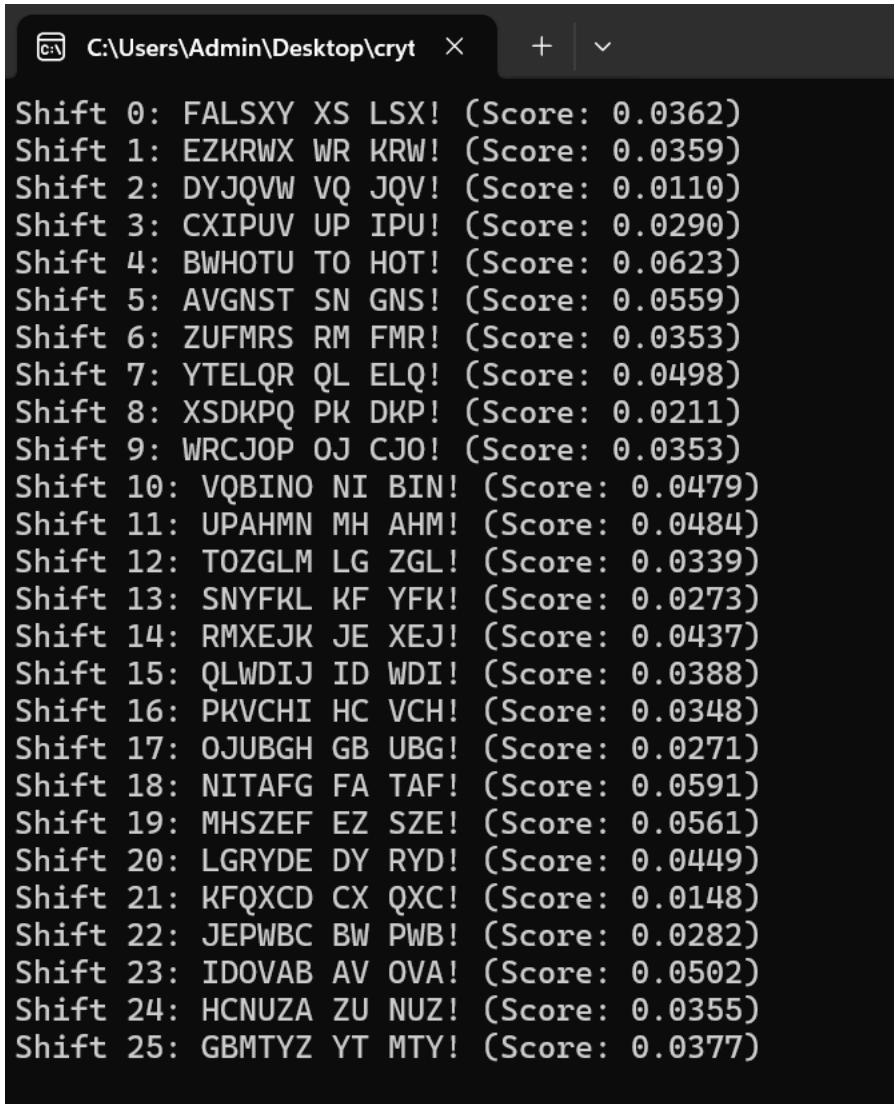


```

}
return 0;
}

```

Sample output:



```

C:\Users\Admin\Desktop\crypt
Shift 0: FALSXY XS LSX! (Score: 0.0362)
Shift 1: EZKRWX WR KRW! (Score: 0.0359)
Shift 2: DYJQVW VQ JQV! (Score: 0.0110)
Shift 3: CXIPUV UP IPU! (Score: 0.0290)
Shift 4: BWHOTU TO HOT! (Score: 0.0623)
Shift 5: AVGNST SN GNS! (Score: 0.0559)
Shift 6: ZUFMRS RM FMR! (Score: 0.0353)
Shift 7: YTELQR QL ELQ! (Score: 0.0498)
Shift 8: XSDKPQ PK DKP! (Score: 0.0211)
Shift 9: WRCJOP OJ CJO! (Score: 0.0353)
Shift 10: VQBINO NI BIN! (Score: 0.0479)
Shift 11: UPAHMN MH AHM! (Score: 0.0484)
Shift 12: TOZGLM LG ZGL! (Score: 0.0339)
Shift 13: SNYFKL KF YFK! (Score: 0.0273)
Shift 14: RMXEJK JE XEJ! (Score: 0.0437)
Shift 15: QLWDIJ ID WDI! (Score: 0.0388)
Shift 16: PKVCHI HC VCH! (Score: 0.0348)
Shift 17: OJUBGH GB UBG! (Score: 0.0271)
Shift 18: NITAFG FA TAF! (Score: 0.0591)
Shift 19: MHSZEF EZ SZE! (Score: 0.0561)
Shift 20: LGRYDE DY RYD! (Score: 0.0449)
Shift 21: KFQXCD CX QXC! (Score: 0.0148)
Shift 22: JEPWBC BW PWB! (Score: 0.0282)
Shift 23: IDOVAB AV OVA! (Score: 0.0502)
Shift 24: HCNUZA ZU NUZ! (Score: 0.0355)
Shift 25: GBMTYZ YT MTY! (Score: 0.0377)

```

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.

Program:

```

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

```

```

#include <ctype.h>

#include <math.h>

#define MAX_LEN 100

int charToNum(char c) {
    if (isupper(c)) {
        return c - 'A';
    } else if (islower(c)) {
        return c - 'a';
    }
    return -1;
}

char numToChar(int num) {
    return num + 'A';
}

void encryptHill(char *text, int *keyMatrix, int keySize) {
    int i,j,k,textLen = strlen(text);
    int encrypted[MAX_LEN] = {0};
    for ( i = 0; i < textLen; i += keySize) {
        for ( j = 0; j < keySize; j++) {
            int sum = 0;
            for (k = 0; k < keySize; k++) {
                sum += keyMatrix[j * keySize + k] * charToNum(text[i + k]);
            }
            encrypted[i + j] = sum % 26;
        }
    }
    for (i = 0; i < textLen; i++) {
        text[i] = numToChar(encrypted[i]);
    }
}

int main() {

```

```

char plaintext[MAX_LEN];

int keySize;

printf("Enter the plaintext: ");

gets(plaintext);

printf("Enter the size of the key matrix: ");

scanf("%d", &keySize);

int i,j,keyMatrix[MAX_LEN * MAX_LEN];

printf("Enter the key matrix (row by row):\n");

for ( i = 0; i < keySize; i++) {
for ( j = 0; j < keySize; j++) {
scanf("%d", &keyMatrix[i * keySize + j]);
}
}

int textLen = strlen(plaintext);

int padding = keySize - (textLen % keySize);

if (padding < keySize) {
for ( i = 0; i < padding; i++) {
plaintext[textLen + i] = 'X';
}

plaintext[textLen + padding] = '\0';
}

encryptHill(plaintext, keyMatrix, keySize);

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

return 0;
}

```

Sample output:

```
C:\Users\Admin\Desktop\crypt >
Enter the plaintext: poiuytrewasdfghjk
Enter the size of the key matrix: 2
Enter the key matrix (row by row):
2 2
3 5
Encrypted text: GLEUILQTSOQRWTGOOP

-----
Process exited after 20.56 seconds with return value 0
Press any key to continue . . .
```

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."

Program:

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

#define ALPHABET_SIZE 26

// Function to decrypt the ciphertext using the specified shift value
void decrypt(char *ciphertext, int shift) {
    int length = strlen(ciphertext);
    int i;
    for ( i = 0; i < length; i++) {
        if (isalpha(ciphertext[i])) {
            if (isupper(ciphertext[i])) {
                ciphertext[i] = 'A' + (ciphertext[i] - 'A' - shift + ALPHABET_SIZE) % ALPHABET_SIZE;
            } else {
                ciphertext[i] = 'a' + (ciphertext[i] - 'a' - shift + ALPHABET_SIZE) % ALPHABET_SIZE;
            }
        }
    }
}
```

```
}
```

```
}
```

```
}
```

```
// Function to count the frequency of each letter in the plaintext
```

```
void countLetterFrequency(char *text, int *frequency) {
```

```
    int length = strlen(text);
```

```
    int i;
```

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

```
        if (isalpha(text[i])) {
```

```
            if (isupper(text[i])) {
```

```
                frequency[text[i] - 'A']++;
```

```
            } else {
```

```
                frequency[text[i] - 'a']++;
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
// Function to find the shift value with the maximum frequency match
```

```
int findShiftValue(int *frequency) {
```

```
    int maxFrequency = 0;
```

```
    int shift = 0;
```

```
    int i;
```

```
    for (i = 0; i < ALPHABET_SIZE; i++) {
```

```
        if (frequency[i] > maxFrequency) {
```

```
            maxFrequency = frequency[i];
```

```
        shift = (ALPHABET_SIZE - i) % ALPHABET_SIZE;
```

```
    }
```

```
}
```

```
return shift;
```

```

}

int main() {
    char ciphertext[1000];
    printf("Enter the ciphertext: ");
    fgets(ciphertext, sizeof(ciphertext), stdin);

    int i;

    int letterFrequency[ALPHABET_SIZE] = {0};
    countLetterFrequency(ciphertext, letterFrequency);

    int shift = findShiftValue(letterFrequency);

    printf("Possible plaintexts in order of likelihood:\n");
    for (i = 0; i < 10; i++) {
        decrypt(ciphertext, shift);
        printf("%d. %s\n", i + 1, ciphertext);
    }
    return 0;
}

```

Sample output:

```

Enter the ciphertext: lkjhgfdsa
Possible plaintexts in order of likelihood:
1. lkjhgfdsa
2. lkjhgfdsa
3. lkjhgfdsa
4. lkjhgfdsa
5. lkjhgfdsa
6. lkjhgfdsa
7. lkjhgfdsa
8. lkjhgfdsa
9. lkjhgfdsa
10. lkjhgfdsa

```

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."

Program:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define ALPHABET_SIZE 26
#define NUM_TOP_PLAINTEXTS 10
const double englishLetterFreq[ALPHABET_SIZE] = {
    0.0817, 0.0149, 0.0278, 0.0425, 0.1270, 0.0223, 0.0202, 0.0609,
    0.0697, 0.0015, 0.0077, 0.0403, 0.0241, 0.0675, 0.0751, 0.0193,
    0.0010, 0.0599, 0.0633, 0.0906, 0.0276, 0.0098, 0.0236, 0.0015,
    0.0197, 0.0007
};
void calculateLetterFrequency(const char *text, double *freq) {
    int totalLetters = 0;
    for (i = 0; text[i]; i++) {
        if (isalpha(text[i])) {
            freq[tolower(text[i]) - 'a']++;
            totalLetters++;
        }
    }
    for (i = 0; i < ALPHABET_SIZE; i++) {
        freq[i] /= totalLetters;
    }
}
double calculateScore(const double *freq) {
    double score = 0.0;
```

```

int i;

for ( i = 0; i < ALPHABET_SIZE; i++) {
    score += freq[i] * englishLetterFreq[i];
}

return score;
}

void decryptSubstitution(const char *ciphertext, char *plaintext, int shift) {
    int i;

    for (i = 0; ciphertext[i]; 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[strlen(ciphertext)] = '\0';
}

int main() {
    const char *ciphertext = "FALSXY XS LSX!"; // Replace with your ciphertext
    double ciphertextFreq[ALPHABET_SIZE] = {0.0};

    int shift;

    calculateLetterFrequency(ciphertext, ciphertextFreq);
    printf("Ciphertext: %s\n\n", ciphertext);
    printf("Top %d possible plaintexts:\n", NUM_TOP_PLAINTEXTS);

    for (shift = 0; shift < ALPHABET_SIZE; shift++) {
        char possiblePlaintext[strlen(ciphertext) + 1];
        decryptSubstitution(ciphertext, possiblePlaintext, shift);
        double possiblePlaintextFreq[ALPHABET_SIZE] = {0.0};
        calculateLetterFrequency(possiblePlaintext, possiblePlaintextFreq);
        double score = calculateScore(possiblePlaintextFreq);
    }
}

```



```
printf("Shift %d: %s (Score: %.4f)\n", shift, possiblePlaintext, score);  
}  
return 0;  
}
```

Sample output:

```
Shift 0: FALSXY XS LSX! (Score: 0.0362)  
Shift 1: EZKRWX WR KRW! (Score: 0.0359)  
Shift 2: DYJQVW VQ JQV! (Score: 0.0110)  
Shift 3: CXIPUV UP IPU! (Score: 0.0290)  
Shift 4: BWHOTU TO HOT! (Score: 0.0623)  
Shift 5: AVGNST SN GNS! (Score: 0.0559)  
Shift 6: ZUFMRS RM FMR! (Score: 0.0353)  
Shift 7: YTELQR QL ELQ! (Score: 0.0498)  
Shift 8: XSDKPQ PK DKP! (Score: 0.0211)  
Shift 9: WRCJOP OJ CJO! (Score: 0.0353)  
Shift 10: VQBIN0 NI BIN! (Score: 0.0479)  
Shift 11: UPAHMN MH AHM! (Score: 0.0484)  
Shift 12: TOZGLM LG ZGL! (Score: 0.0339)  
Shift 13: SNYFKL KF YFK! (Score: 0.0273)  
Shift 14: RMXEJK JE XEJ! (Score: 0.0437)  
Shift 15: QLWDIJ ID WDI! (Score: 0.0388)  
Shift 16: PKVCHI HC VCH! (Score: 0.0348)  
Shift 17: OJUBGH GB UBG! (Score: 0.0271)  
Shift 18: NITAFG FA TAF! (Score: 0.0591)  
Shift 19: MHSZEF EZ SZE! (Score: 0.0561)  
Shift 20: LGRYDE DY RYD! (Score: 0.0449)  
Shift 21: KFQXCD CX QXC! (Score: 0.0148)  
Shift 22: JEPWBC BW PWB! (Score: 0.0282)  
Shift 23: IDOVAB AV OVA! (Score: 0.0502)  
Shift 24: HCNUZA ZU NUZ! (Score: 0.0355)  
Shift 25: GBMTYZ YT MTY! (Score: 0.0377)
```