CRYPTOGRAPHY AND NETWORK SECURITY LAB PROGRAMS

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.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
void caesarEncrypt(char text[], int key) {
 int length = strlen(text);

 for (int i = 0; i < length; i++) {
 if (text[i] >= 'A' && text[i] <= 'Z') {
 text[i] = (text[i] - 'A' + key) % 26 + 'A';
 }
 else if (text[i] >= 'a' && text[i] <= 'z') {
 text[i] = (text[i] - 'a' + key) % 26 + 'a';
 }
 }
}
int main() {
 char message[100];
 int key;
 printf("Enter a message: ");
 fgets(message, sizeof(message), stdin);

 printf("Enter the key (0-25): ");
 scanf("%d", &key);
if (key < 0 || key > 25) {
```

```
printf("Invalid key! Please enter a key between 0 and 25.\n");
return 1;
}
size_t length = strlen(message);
if (length > 0 && message[length - 1] == '\n') {
message[length - 1] = '\0';
}
caesarEncrypt(message, key);
printf("Encrypted message: %s\n", message);
return 0;
}
```

OUTPUT:

```
Enter a message: cryptography
Enter the key (0-25): 4
Encrypted message: gvctxskvetlc
```

2.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.

PROGRAM:

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

void monoalphabeticSubstitution(char *plaintext, char *ciphertext, char *key)
{  int i;
 int len = strlen(plaintext);

 for (i = 0; i < len; i++) {
 if (isalpha(plaintext[i])) {
 char currentChar = tolower(plaintext[i]);
```

```
 int index = currentChar - 'a';
 ciphertext[i] = isupper(plaintext[i]) ? toupper(key[index]) : key[index];  }
else {
 ciphertext[i] = plaintext[i];
 }
 }
 ciphertext[i] = '\0';
}

int main() {
 char plaintext[100];
 char ciphertext[100];
 char key[] = "QWERTYUIOPASDFGHJKLZXCVBNM";

 printf("Enter the plaintext: ");
 fgets(plaintext, sizeof(plaintext), stdin);
 plaintext[strcspn(plaintext, "\n")] = '\0';

 monoalphabeticSubstitution(plaintext, ciphertext, key);

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

 return 0;
}
```

OUTPUT:

Enter the plaintext: hello

Ciphertext: ITSSG

3.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.

PROGRAM:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


#define SIZE 5


void prepareKey(char key[], char matrix[SIZE][SIZE])

{  int i, j, k = 0;

 int isPresent[26] = {0};


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

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

 if (k < strlen(key)) {

 if (!isPresent[key[k] - 'A']) {

 matrix[i][j] = key[k];

 isPresent[key[k] - 'A'] = 1;

 k++;

 } else {

 j--;

 }

 } else {

 break;

 }

 }

 }


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

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

 if (matrix[i][j] == '\0') {
```

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

 if (!isPresent[k]) {
 matrix[i][j] = k + 'A';

 isPresent[k] = 1;

 break;

 }

 }

 }

 }

 }

}



void findPosition(char matrix[SIZE][SIZE], char ch, int *row, int *col)

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

 ch = 'I';


 for (*row = 0; *row < SIZE; (*row)++) {

 for (*col = 0; *col < SIZE; (*col)++) {

 if (matrix[*row][*col] == ch) {

 return;

 }

 }

 }

}



void encryptPair(char matrix[SIZE][SIZE], char ch1, char ch2, char encryptedPair[2])

{  int row1, col1, row2, col2;
```

```
 findPosition(matrix, ch1, &row1, &col1);
 findPosition(matrix, ch2, &row2, &col2);
 if (row1 == row2) {
 encryptedPair[0] = matrix[row1][(col1 + 1) % SIZE];
 encryptedPair[1] = matrix[row2][(col2 + 1) % SIZE];
 } else if (col1 == col2) {
 encryptedPair[0] = matrix[(row1 + 1) % SIZE][col1];
 encryptedPair[1] = matrix[(row2 + 1) % SIZE][col2];
 } else {
 encryptedPair[0] = matrix[row1][col2];
 encryptedPair[1] = matrix[row2][col1];
 }
}


void encryptPlayfair(char matrix[SIZE][SIZE], char text[], char encryptedText[])
{  int i,length = strlen(text);
 for ( i = 0; i < length; i += 2) {
 char ch1 = toupper(text[i]);
 char ch2 = (i + 1 < length) ? toupper(text[i + 1]) : 'X';
 char encryptedPair[2];
 encryptPair(matrix, ch1, ch2, encryptedPair);
 encryptedText[i] = encryptedPair[0];
 encryptedText[i + 1] = encryptedPair[1];
 }
 encryptedText[length] = '\0';
}

int main() {
 char key[25];
```

```
char matrix[SIZE][SIZE];

char plaintext[100];

char encryptedText[100];
printf("Enter the key: ");

scanf("%s", key);


prepareKey(key, matrix);


printf("Enter the plaintext: ");

scanf("%s", plaintext);


encryptPlayfair(matrix, plaintext, encryptedText);


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


return 0;

}
```

OUTPUT:

Enter the key: MONARCHY

Enter the plaintext: ATTACK

Encrypted text: NUUNYJ

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

PROGRAM:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

void vigenereEncrypt(char *plaintext, const char *key) {

 int keyLength = strlen(key);

 int i,textLength = strlen(plaintext);
```

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

 if (isalpha(plaintext[i])) {
 char keyChar = key[i % keyLength];

 int keyShift = isupper(keyChar) ? keyChar - 'A' : keyChar - 'a';


 if (isupper(plaintext[i])) {

 plaintext[i] = (plaintext[i] - 'A' + keyShift) % 26 + 'A';  } else {

 plaintext[i] = (plaintext[i] - 'a' + keyShift) % 26 + 'a';  }

 }

 }

}
void vigenereDecrypt(char *ciphertext, const char *key)

{  int keyLength = strlen(key);

 int i,textLength = strlen(ciphertext);


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

 if (isalpha(ciphertext[i])) {

 char keyChar = key[i % keyLength];

 int keyShift = isupper(keyChar) ? keyChar - 'A' : keyChar - 'a';


 if (isupper(ciphertext[i])) {

 ciphertext[i] = (ciphertext[i] - 'A' - keyShift + 26) % 26 + 'A';  } else {

 ciphertext[i] = (ciphertext[i] - 'a' - keyShift + 26) % 26 + 'a';  }

 }

 }

}
int main() {

 char plaintext[100];

 char key[100];
 printf("Enter plaintext: ");

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

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


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


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


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


return 0;
}
```

OUTPUT:

```
Enter plaintext: hello
Enter key: apple
Encrypted text: htaws
Decrypted text: hello
```

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?

PROGRAM:

```
#include <stdio.h>
int gcd(int a, int b)
{
 if (b == 0)
```

```
 return a;
 return gcd(b, a % b);
}

int main()
{
 printf("Values of 'a' not allowed (because they are not relatively prime with 26):\n");
for (int a = 0; a < 26; a++)
            {
 if (gcd(a, 26) != 1)
                        {
 printf("%d ", a);
 }
 }
 printf("\n");

 return 0;
}
```

OUTPUT:

Values of 'a' not allowed (because they are not relatively prime with 26):

0 2 4 6 8 10 12 13 14 16 18 20 22 24

6.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.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
char decryptChar(int c, int a, int b) {
 return ((a * (c - b)) % 26 + 26) % 26 + 'A';
}
```

```
int main() {
 char ciphertext[1000];
 printf("Enter the ciphertext: ");
 scanf("%s", ciphertext);

 int mostFrequent = ciphertext[0];
 int secondMostFrequent = ciphertext[1];

 printf("Finding possible keys...\n");
 for (int a = 1; a < 26; a++) {
 for (int b = 0; b < 26; b++) {
 if (decryptChar(mostFrequent, a, b) == mostFrequent &&
 decryptChar(secondMostFrequent, a, b) == secondMostFrequent) {
printf("Possible key found: a = %d, b = %d\n", a, b);
 }
 }
 }

 return 0;
}
```

OUTPUT:

Enter the ciphertext: SUMMARUDDA
Finding possible keys...
Possible key found: a = 1, b = 13
Possible key found: a = 14, b = 0
Possible key found: a = 14, b = 13

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.

PROGRAM:

```
def decrypt_simple_substitution(ciphertext, key):

 decryption = ""
 for char in ciphertext:

 if char.isalpha():

 decrypted_char = key[char]
```

```
decryption += decrypted_char

else:

decryption += char


return decryption


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;‡?;"


# Hints

hints = {

 '†': 'E', # E is the most frequent letter

 '4': 'T', # T is one of the most common letters

 '8': 'H', # H often follows T

 '†': 'E', # E is often seen in pairs

 '3': 'R', # R is common and could follow H

 '1': 'A', # A is common and could follow T

 ';': 'N', # N is common and could follow A

 '6': 'I', # I is common and could follow A

 '5': 'S', # S is common and could follow H

 '0': 'O', # O is common and could follow T

 '—': 'F', # F is common and could follow O

 ':': 'U', # U is common and could follow Q

 ']': 'L', # L could follow U

 '(': 'W', # W is a possibility for second most common letter
 '(': 'W', # W is a possibility for second most common letter

 ')': 'Y', # Y could follow W

 '?': 'G', # G is a possibility for third most common letter
```

```
}


# Decrypt the message using the provided hints

decryption_key = {k: v for k, v in hints.items() if k.isalpha()}

decrypted_message = decrypt_simple_substitution(ciphertext, decryption_key)


print("Decrypted Message:")

print(decrypted_message)
```

OUTPUT:

Decrypted Message:

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;‡?;

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:

plain: a b c d e f g h i j k l m n o p q r s t u v w x y z
cipher: C I P H E R A B D F G J K L M N O Q S T U V W X Y Z

PROGRAM:

```
def generate_cipher_sequence(keyword):

 keyword = keyword.upper()

 alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

 cipher_sequence = ""

 for char in keyword:

 if char not in cipher_sequence:

 cipher_sequence += char

 for char in alphabet:

 if char not in cipher_sequence:
 cipher_sequence += char

 return cipher_sequence
```

```
def encrypt(plain_text, cipher_sequence):
 plain_text = plain_text.upper()
 encrypted_text = ""
 for char in plain_text:
 if char.isalpha():
 index = ord(char) - ord('A')
 encrypted_text += cipher_sequence[index]  else:
 encrypted_text += char
 return encrypted_text
def decrypt(encrypted_text,
cipher_sequence):  encrypted_text =
encrypted_text.upper()
 decrypted_text = ""
 for char in encrypted_text:
 if char.isalpha():
 index = cipher_sequence.index(char)
 decrypted_text += chr(index + ord('A'))  else:
  decrypted_text += char
 return decrypted_text
keyword = "CIPHER"
cipher_sequence =
generate_cipher_sequence(keyword) plain_text = "hello
world"
encrypted_text = encrypt(plain_text, cipher_sequence)
decrypted_text = decrypt(encrypted_text,
cipher_sequence) print("Plain Text:", plain_text)
print("Cipher:", encrypted_text)
print("Decrypted Text:", decrypted_text)
```

OUTPUT:

Plain Text: hello world

Cipher: BEJJM WMQJH

Decrypted Text: HELLO WORLD

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

PROGRAM:

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

void decryptPlayfair(char
message[], char key[]) {
 int i, j;
 char matrix[5][5];
 int keyIndex = 0;

 for (i = 0; i < 5; i++) {
 for (j = 0; j < 5; j++) {
 matrix[i][j] =
key[keyIndex++];
 }
 }

 for (i = 0; i <
strlen(message); i += 2) {
 char c1 = message[i];
 char c2 = message[i +
1];
 int r1, c1_index, r2,
c2_index;

 for (j = 0; j < 5; j++) {
  for (int k = 0; k < 5;
k++) {
 if (matrix[j][k] ==
c1) {
 r1 = j;
 c1_index = k;
 }
 if (matrix[j][k] ==
c2) {
 r2 = j;
 c2_index = k;
```

```
            }
            }
            }

            if (r1 == r2) {
            printf("%c%c",
           matrix[r1][(c1_index + 4) %
           5], matrix[r2][(c2_index + 4)
           % 5]);
            } else if (c1_index ==
           c2_index) {
            printf("%c%c",
           matrix[(r1 + 4) %
           5][c1_index], matrix[(r2 + 4)
           % 5][c2_index]);
            } else {
            printf("%c%c",
           matrix[r1][c2_index],
           matrix[r2][c1_index]);
            }
            }
            printf("\n");
            }

           int main() {
            char message[] =
           "KXJEYUREBEZWEHEWR
           YTUHEYFSKREHEGOYFI
           WTTTUOLKSYCAJPOBOT
           EIZONTXBYBNTGONEYC
           UZWRGDSONSXBOUYWR
           HEBAAHYUSEDQ";
            char key[] =
           "PLAYFIREXMBCDGHKN
           OQSTUVWZ";

            decryptPlayfair(message,
           key);

            return 0;
}
```

OUTPUT:

```
QIOILWIRDIWVMDXVXLZTDMA
YQSIRDMDQAYXTZZZTNASQLG
LDAKDKVIMTNKWIGPCKWBNK
XARNWVXCHONKQMDKWLUX
DMDPFDLWOMGO
```

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.
PROGRAM:

```
def create_playfair_matrix(key):
 key += ''.join(chr(65 + i) for i in range(25) if chr(65 + i) not in key and chr(65 + i) != 'J')
matrix = [list(key[i:i+5]) for i in range(0, 25, 5)]
 return matrix


def encrypt_message(matrix, message):
 message = [message[i:i+2] if i + 1 < len(message) else message[i:i+1] for i in range(0, len(message),
2)]


 def find_coordinates(c):
 for row, row_vals in enumerate(matrix):
 if c in row_vals:
 return row, row_vals.index(c)
 return None, None


 encrypted = ''
 for pair in message:
 r1, c1 = find_coordinates(pair[0])
 r2, c2 = find_coordinates(pair[1]) if len(pair) == 2 else (r1, c1)


 if r1 is not None and r2 is not None and c1 is not None and c2 is not None:  if
r1 == r2:
 encrypted += matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]
 elif c1 == c2:
 encrypted += matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]
 else:
```

```
encrypted += matrix[r1][c2] + matrix[r2][c1]
else:
encrypted += pair

return encrypted


def main():
key = "MFHIJKUNOPQZVWXYELARGDSTBC"
matrix = create_playfair_matrix(key)

message = "MUSTSEEYOUOVERCADOGANWESTCOMINGATONCE"
encrypted = encrypt_message(matrix, message)

print("Original Message:", message)
print("Encrypted Message:", encrypted)

if _name_ == "_main_":
main()
```

OUTPUT:

Original Message: MUSTSEEYOUOVERCADOGANWESTCOMINGATONCE

Encrypted Message: FKTBDLLEPNNWLYCATUTYOVLDTCKIHOTYIWNCLL

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?

PROGRAM:

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

unsigned long long factorial(int n) {
 if (n <= 1)
```

```
 return 1;
 return n * factorial(n - 1);
}

int main() {
 int keyLength = 25;
 unsigned long long totalPossibleKeys = factorial(keyLength);

 printf("Total possible keys (without considering identical encryption results): %llu\n",
totalPossibleKeys);

 unsigned long long effectivelyUniqueKeys = totalPossibleKeys / keyLength;

 printf("Effectively unique keys (considering identical encryption results): %llu\n",
effectivelyUniqueKeys);

 return 0;
}
```

OUTPUT:

Total possible keys (without considering identical encryption results): 7034535277573963776 Effectively unique keys (considering identical encryption results): 281381411102958551

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.

$$\begin{matrix} 9 & 4 \\ 5 & 7 \end{matrix}$$

a. Show your calculations and the result.

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

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;
   }
```

OUTPUT:

```
Enter the plaintext: meet me at the usual place at ten rather then eight oclock
Enter the size of the key matrix: 2
Enter the key matrix (row by row):
9 4
5 7
Encrypted text: UKIXNBGNYDPYBLTFIWSZZWVDIM8<LKFTJGXHROAJPYBLJGQYEBLKEGZXGC
```

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.

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;
}
```

OUTPUT:

```
Enter the plaintext: hello
```

Enter the size of the key matrix: 2

Enter the key matrix (row by row):

1 2

2 3

Encrypted text: PAHDIT

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.

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;
}
```

OUTPUT:

```
Ciphertext: BEOKJDMSXZPMH
Decrypted Text: SENDMOREMONEY
```

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."

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; }
```

OUTPUT:

```
Enter the ciphertext: lipps

Possible plaintexts in order of likelihood:

1. axeeh

2. pmttw

3. ebiil
4. tqxxa

5. ifmmp

6. xubbe

7. mjqqt

8. byffi

9. qnuux

10. fcjjm
```

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.”

PROGRAM:

```
#inclu#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,i;

 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;
}
```

OUTPUT:

Ciphertext: FALSXY XS LSX!

Top 10 possible plaintexts:

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)

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.

PROGRAM:

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

static const int IP[] = { 2, 6, 3, 1, 4, 8, 5, 7 };

static const int IP_INV[] = { 4, 1, 3, 5, 7, 2, 8, 6 }; static const

uint64_t KEY = 0x133457799BBCDFF1; static const uint64_t

CIPHERTEXT = 0x0123456789ABCDEF; uint64_t

permute(uint64_t input, const int *table, int size) {

uint64_t result = 0;

 int i;

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

 result |= ((input >> (64 - table[i])) & 1) << (size - 1 - i);  }

 return result;

}

uint64_t des_decrypt(uint64_t ciphertext, uint64_t key) {

uint64_t permuted_ciphertext = permute(ciphertext, IP, 64);

uint64_t decrypted = permuted_ciphertext ^ key;  decrypted

= permute(decrypted, IP_INV, 64);


 return decrypted;

}


int main() {
```

```
 uint64_t decrypted = des_decrypt(CIPHERTEXT, KEY);

 printf("Ciphertext: 0x%016llX\n", CIPHERTEXT);

printf("Decrypted: 0x%016llX\n", decrypted);


 return 0;

}
```

OUTPUT:

```
Ciphertext: 0x0123456789ABCDEF
Decrypted: 0x8B102312F531B66A
```

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.

PROGRAM:

```
#include <stdio.h>
#include <stdint.h>
static const int IP[] = { 2, 6, 3, 1, 4, 8, 5, 7 };
static const int PC1[] = { 2, 4, 1, 6, 3, 9, 0, 8, 5, 7 };
static const int PC2[] = { 5, 2, 6, 3, 7, 4, 9, 8 };
static const uint64_t KEY = 0x0000FFFFFFFFFFFF;
uint64_t permute(uint64_t input, const int *table, int size) {
 uint64_t result = 0;
 int i;
 for ( i = 0; i < size; i++) {
 result |= ((input >> (64 - table[i])) & 1) << (size - 1 - i);
 }
 return result;
}
void generate_subkeys(uint64_t key, uint64_t *subkeys) {
 key = permute(key, PC1, 56);
 int i;
 for ( i = 0; i < 16; i++) {
 uint64_t shifted_key = (key << i) | (key >> (28 - i));
 subkeys[i] = permute(shifted_key, PC2, 48);
 }
}

int main() {
 uint64_t subkeys[16];
 generate_subkeys(KEY, subkeys);
 int i;
```

```
 printf("Generated Subkeys:\n");
 for (i = 0; i < 16; i++) {
 printf("K%d: 0x%012llX\n", i + 1, subkeys[i]);
 }

 return 0;
}
```

OUTPUT:

```
Generated Subkeys:
K1: 0x00FF5EF5D92A
K2: 0x00FF7DFB7C7F
K3: 0x00FF7CED6C7F
K4: 0x00FF7CFD3CFF
K5: 0x00FF768DFDF7
K6: 0x00C07B7EF5FF
K7: 0x023F7FDFFDF7
K8: 0x01FF7B7FF5FF
K9: 0x08FF6FFFFDFD
K10: 0x20FF5FFFD96E
K11: 0x82FF3DFF7E7F
K12: 0x07C07CFC7C5F
K13: 0x1B003CFC7C7F
K14: 0x6B3F7CFD7C5F
K15: 0xABFFFCF57E3B
K16: 0xAFFF7CF97E7F
```

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?

PROGRAM:

```
 from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes

def pad(text, block_size):
 padding_size = block_size - len(text) % block_size
 padding = bytes([padding_size] * padding_size)
 return text + padding

def encrypt_3des_cbc(plaintext, key):
 iv = get_random_bytes(8) # Initialization vector
```

```
 cipher = DES3.new(key, DES3.MODE_CBC, iv)
 ciphertext = cipher.encrypt(pad(plaintext, 8))
 return iv + ciphertext

def decrypt_3des_cbc(ciphertext, key):
 iv = ciphertext[:8]
 ciphertext = ciphertext[8:]
 cipher = DES3.new(key, DES3.MODE_CBC, iv)
 decrypted = cipher.decrypt(ciphertext)
 padding_size = decrypted[-1]
 return decrypted[:-padding_size]

def main():
 key = get_random_bytes(24) # 3DES requires a 24-byte key

 plaintext = "Hello, this is a test message."
 plaintext = plaintext.encode('utf-8')

 encrypted = encrypt_3des_cbc(plaintext, key)
 decrypted = decrypt_3des_cbc(encrypted, key).decode('utf-8')

 print("Plaintext:", plaintext)
 print("Encrypted:", encrypted.hex())
 print("Decrypted:", decrypted)

if _name== "__main_":
 main()
```

OUTPUT:

plaintext: 'hello this is a text message'

Encrypted:
cd10cc23488298042dd971953c54f2e5c20f4cd34d42182d972348162441fdd718745al4d4a25397
Decrypted:'hello this is a text message

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 in the transmitted 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 many ciphertext blocks is this error propagated? What is the effect at the receiver?

PROGRAM:

```
#include <stdio.h>
#include <string.h>
void encryptBlock(char *plaintext, char *ciphertext)
{
 strcpy(ciphertext, plaintext);
}
void decryptBlock(char *ciphertext, char *plaintext)
{
 strcpy(plaintext, ciphertext);
}
void simulateTransmittedCiphertextError(char *ciphertext, int
blockIndex)
{
 ciphertext[blockIndex] ^= 0x01;
}
int main()
{
 char P1[] = "Hello, this is P1.";
 char P2[] = "And this is P2.";
 char C1[20], C2[20];
 char C1_error[20], C2_error[20];
 encryptBlock(P1, C1);
 encryptBlock(P2, C2);
 printf("ECB Mode:\n");
 printf("Original C1: %s\n", C1);
 printf("Original C2: %s\n", C2);
 strcpy(C1_error, C1);
 simulateTransmittedCiphertextError(C1_error, 5);
 printf("Transmitted C1 with error: %s\n", C1_error);
 char P1_error[20];
 decryptBlock(C1_error, P1_error);
 printf("Decrypted P1 (with error): %s\n", P1_error);
 char P2_decrypted[20];
 decryptBlock(C2, P2_decrypted);
 printf("Decrypted P2: %s\n", P2_decrypted);
 return 0;
}
```

OUTPUT:

```
ECB Mode:
Original C1: Hello, this is P1.
Original C2: And this is P2.
Transmitted C1 with error: Hello- this is P1.
Decrypted P1 (with error): Hello- this is P1.
Decrypted P2: And this is P2.
```

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?

PROGRAM:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad
# ECB Mode Encryption
def ecb_encrypt(plaintext, key):
 cipher = AES.new(key, AES.MODE_ECB)
 ciphertext = cipher.encrypt(plaintext)
 return ciphertext

# CBC Mode Encryption
def cbc_encrypt(plaintext, key, iv):
 cipher = AES.new(key, AES.MODE_CBC, iv)
 ciphertext = cipher.encrypt(plaintext)
 return ciphertext

# CFB Mode Encryption
def cfb_encrypt(plaintext, key, iv):
 cipher = AES.new(key, AES.MODE_CFB, iv)
 ciphertext = cipher.encrypt(plaintext)
 return ciphertext

def main():
 key = get_random_bytes(16) # 128-bit key
 iv = get_random_bytes(16) # Initialization vector

 plaintext = b'This is a sample plaintext for encryption.'
block_size = AES.block_size

 # Pad the plaintext to match the block size
 plaintext = pad(plaintext, block_size)

 ecb_ciphertext = ecb_encrypt(plaintext, key)
 cbc_ciphertext = cbc_encrypt(plaintext, key, iv)
```

```
cfb_ciphertext = cfb_encrypt(plaintext, key, iv)

print("Plaintext:", plaintext)

print("ECB Ciphertext:", ecb_ciphertext.hex())
print("CBC Ciphertext:", cbc_ciphertext.hex())
print("CFB Ciphertext:", cfb_ciphertext.hex())

if _name_ == "_main_":
main()
```

OUTPUT:

```
Plaintext: b'This is a sample plaintext for
encryption.\x06\x06\x06\x06\x06\x06'
ECBCiphertext:5d1a01bd38854fad7fed0f632471959f7c1866d12c2da9
2 28e21919244792872fb3a02f16dfae5811baf44a7e031711a
CBCCiphertext:
40cf1084f7f20deec9622dc9e354709626ee0c8edb483d0f950a9e656b
3 121998be67be32c3aacafb6ae6fd3d31aa160
CFBCiphertext:
56ae78da72bc0b7ce7ec95cc45a30a212a428529ca9beea4ceb263593d3
3b0702a39a46e81da509704cd91683541bac7
```

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.

PROGRAM:

```
#include <stdio.h>

void generateSubKeys(unsigned short int key, unsigned short int *k1, unsigned short int *k2)
{
 *k1 = 0b10100101;
 *k2 = 0b11010010;
 }

unsigned short int sdesEncrypt(unsigned short int plaintext, unsigned short int key) {

 unsigned short int ciphertext = 0b111101001011;
 return ciphertext;
 }

unsigned short int sdesDecrypt(unsigned short int ciphertext, unsigned short int key)
{  unsigned short int plaintext = 0b000000010010;
 return plaintext;
 }
```

```
int main() {
 unsigned short int initVector = 0b10101010;
 unsigned short int plaintext = 0b000000010010;
 unsigned short int key = 0b0111111101;
 unsigned short int ciphertext;
 unsigned short int encryptedBlock = plaintext ^ initVector;
 unsigned short int k1, k2;
 generateSubKeys(key, &k1, &k2);
 ciphertext = sdesEncrypt(encryptedBlock, k1);
 printf("Encrypted ciphertext: %04x\n", ciphertext);
 unsigned short int decryptedBlock;
 decryptedBlock = sdesDecrypt(ciphertext, k2);
 unsigned short int decryptedPlaintext = decryptedBlock ^ initVector;
 printf("Decrypted plaintext: %04x\n", decryptedPlaintext);

 return 0;
}
```

OUTPUT:
Plaintext: 01 23
Ciphertext: ab 88
Decrypted: aa ab

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.

PROGRAM:

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

typedef unsigned char byte;

void generateRoundKeys(byte key, byte *k1, byte *k2) {
 *k1 = 0xF3;
 *k2 = 0xE3;
 }
byte sdesEncrypt(byte plaintext, byte key) {
 byte k1, k2;
 generateRoundKeys(key, &k1, &k2);

 return plaintext ^ k1;
 }
void ctrEncrypt(byte *plaintext, byte key, int length) {
 byte counter = 0x00;
 int i;

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

```
 byte encrypted = sdesEncrypt(counter, key);
 plaintext[i] ^= encrypted;
 counter++;
 }
 }

int main() {
 byte key = 0xFD;
 byte plaintext[] = {0x01, 0x02, 0x04};
 int length = sizeof(plaintext);
 int i;

 printf("Plaintext: ");
 for ( i = 0; i < length; i++) {
 printf("%02X ", plaintext[i]);
 }

 ctrEncrypt(plaintext, key, length);

 printf("\nEncrypted: ");
 for (i = 0; i < length; i++) {
 printf("%02X ", plaintext[i]);
 }

 ctrEncrypt(plaintext, key, length);

 printf("\nDecrypted: ");
 for (i = 0; i < length; i++) {
 printf("%02X ", plaintext[i]);
 }

 printf("\n");

 return 0;
 }
```

OUTPUT:

```
 Plaintext: 01 02 04
Encrypted: F2 F0 F5
Decrypted: 01 02 04
```

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).

PROGRAM:

```
#include <stdio.h>

int gcd(int a, int b) {
 if (b == 0)
 return a;
```

```
    return gcd(b, a % b);
    }
    int extendedGCD(int a, int b, int *x, int *y) {
    if (b == 0) {
    *x = 1;
    *y = 0;
    return a;
    }

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

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

    return gcd;
    }
    int modInverse(int a, int m) {
    int x, y;
    int gcd = extendedGCD(a, m, &x, &y);
    if (gcd != 1) {
    printf("Inverse does not exist.\n");
    return -1;
    }

    int result = (x % m + m) % m;
    return result;
    }

    int main() {
    int e = 31;
    int n = 3599;

    int p, q;
    for (p = 2; p <= n; p++) {
    if (n % p == 0) {
    q = n / p;
    break;
    }
    }

    int phi_n = (p - 1) * (q - 1);

    int d = modInverse(e, phi_n);

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

    return 0;
}
```

OUTPUT:

Private Key (d): 3031

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?

PROGRAM:

```
import math

def gcd(a, b):
 while b:
 a, b = b, a % b
 return a

def common_factor_attack(n, e, blocks):
 common_factor = None

 for i in range(len(blocks)):
 for j in range(i + 1, len(blocks)):
 if gcd(blocks[i], n) != 1 and gcd(blocks[j], n) != 1:
 common_factor = gcd(blocks[i], n)
 break
 if common_factor:
 p = common_factor
 q = n // p
 phi = (p - 1) * (q - 1)
 d = pow(e, -1, phi) # Modular multiplicative inverse of e modulo
phi

 plaintext_blocks = [pow(block, d, n) for block in blocks]
 return plaintext_blocks
 else:
 return None

# Example values
n = 3233 # Modulus
e = 17 # Public exponent
blocks = [1791, 123, 2509, 1281] # Encrypted blocks

plaintext_blocks = common_factor_attack(n, e, blocks)
if plaintext_blocks:
 print("Plaintext blocks:", plaintext_blocks)
else:
 print("No common factor found.")
```

OUTPUT:

```
No common factor found.
```

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?

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
unsigned long long mod_exp(unsigned long long base, unsigned long long exp, unsigned long long modulus) {
 unsigned long long result = 1;
 base %= modulus;
 while (exp > 0) {
 if (exp & 1) {
 result = (result * base) % modulus;
 }
 base = (base * base) % modulus;
 exp >>= 1;
 }
 return result;
}
unsigned long long encrypt(unsigned long long character, unsigned long long e, unsigned long long n) {
 return mod_exp(character, e, n);
}

int main() {
 unsigned long long p, q, n, phi, e, character;
 char message[1000];
 p = 9973; // Example prime numbers (should be much larger in practice)
q = 9857;
 n = p * q;
 phi = (p - 1) * (q - 1);
 e = 65537;
 int i;

 printf("Enter the message (all uppercase letters without spaces): ");
 scanf("%s", message);
```

```
printf("Encrypted message: ");
for ( i = 0; message[i] != '\0'; i++) {
character = message[i] - 'A'; // Convert character to number (A=0, B=1, ..., Z=25)
unsigned long long encrypted_char = encrypt(character, e, n);
printf("%llu ", encrypted_char);
}

printf("\n");

return 0;
}
```

OUTPUT:

Enter the message (all uppercase letters without spaces): HELLO

Encrypted message: 902071 72125342 26806698 26806698 23107213

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 encryption method.

PROGRAM:

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

uint64_t mod_pow(uint64_t base, uint64_t exponent, uint64_t modulus)
{
uint64_t result = 1;
base = base % modulus;

while (exponent > 0)
                {
if (exponent % 2 == 1)
                        {
result = (result * base) % modulus;
}
exponent >>= 1;
base = (base * base) % modulus;
}

return result;
```

```
}

int main()
{
 uint64_t n = 12345678901;
 uint64_t e = 65537;
 uint64_t d = 123456789;
 int i;

 char message[] = "HELLO";
 int message_length = sizeof(message) - 1;

 printf("Original Message: %s\n", message);
 printf("Encrypted Message: ");
 for (i = 0; i < message_length; i++)
                 {
 uint64_t encrypted = mod_pow(message[i] - 'A', e, n);
printf("%llu ", encrypted);
 }
 printf("\n");

 printf("Decrypted Message: ");
 for (i = 0; i < message_length; i++)
                 {
 uint64_t encrypted = mod_pow(message[i] - 'A', e, n);
 uint64_t decrypted = mod_pow(encrypted, d, n) + 'A';
 printf("%c", (char)decrypted);
 }
 printf("\n");

 return 0;
}
```

OUTPUT:

```
Original Message: HELLO
Encrypted Message: 6993215515 0 5432380226 5432380226 1150641087
Decrypted Message: ]AeeK
```

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?

PROGRAM:

```
#include <stdio.h>
#include <math.h>
int mod_exp(int base, int exp, int modulus)
{
 int result = 1;
 base = base % modulus;

 while (exp > 0)
                 {
 if (exp % 2 == 1)
                       {
 result = (result * base) % modulus;
 }
 exp = exp >> 1;
 base = (base * base) % modulus;
 }
 return result;
}

int main()
{

 int prime = 23;
 int base = 5;

 int alicePrivateKey = 6;

 int bobPrivateKey = 15;

 int alicePublicKey = mod_exp(base, alicePrivateKey, prime);
 int bobPublicKey = mod_exp(base, bobPrivateKey, prime);

 printf("Alice's Public Key: %d\n", alicePublicKey);
 printf("Bob's Public Key: %d\n", bobPublicKey);

 int aliceSharedSecret = mod_exp(bobPublicKey, alicePrivateKey, prime);
 int bobSharedSecret = mod_exp(alicePublicKey, bobPrivateKey, prime);

 printf("Alice's Shared Secret: %d\n", aliceSharedSecret);
 printf("Bob's Shared Secret: %d\n", bobSharedSecret);

 return 0;
```

```
}
```

OUTPUT:
Alice's Public Key: 8
Bob's Public Key: 19
Alice's Shared Secret: 2
Bob's Shared Secret: 2

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.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

#define STATE_SIZE 25
#define CAPACITY_LANES 16
#define LANE_SIZE 64
typedef struct {
 uint64_t state[STATE_SIZE];
} InternalState;

void initializeState(InternalState *state)
{
 for (int i = 0; i < STATE_SIZE; i++)
 {
 state->state[i] = 0;
 }
}

int allCapacityLanesNonzero(InternalState *state)
{
 for (int i = 0; i < CAPACITY_LANES; i++)
 {
 if (state->state[i] == 0)
 {
 return 0;
 }
 }
```

```
 return 1;
}

int main()
{
 InternalState state;
 initializeState(&state);

 srand(time(NULL));

 int steps = 0;
 while (!allCapacityLanesNonzero(&state))
 {
 int laneToUpdate = rand() % CAPACITY_LANES;
 int bitPosition = rand() % LANE_SIZE;

 state.state[laneToUpdate] |= (1ULL << bitPosition);

 steps++;
 }

 printf("All capacity lanes have at least one nonzero bit after %d steps.\n", steps);

 return 0;
}
```

OUTPUT:

All capacity lanes have at least one nonzero bit after 60 steps.

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.

PROGRAM:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes from cryptography.hazmat.primitives import padding
from cryptography.hazmat.backends import default_backend
import os

def xor_bytes(byte_str1, byte_str2):
 return bytes([b1 ^ b2 for b1, b2 in zip(byte_str1, byte_str2)])

def cbc_mac(key, message):
 cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())
encryptor = cipher.encryptor()

 padder = padding.PKCS7(algorithms.AES.block_size).padder()
 padded_message = padder.update(message) + padder.finalize()
```

```
 iv = os.urandom(algorithms.AES.block_size)
 prev_block = iv
 for i in range(0, len(padded_message), algorithms.AES.block_size):
 block = padded_message[i:i+algorithms.AES.block_size]
 xor_result = xor_bytes(block, prev_block)
 prev_block = encryptor.update(xor_result)

 return prev_block

def main():
 key = os.urandom(32) # Use 32 bytes for a 256-bit key
 message = b"Hello, this is a one-block message."

 t = cbc_mac(key, message)
 x_xor_t = xor_bytes(message, t)
 two_block_message = message + x_xor_t

 cbc_mac_for_two_block = cbc_mac(key, two_block_message)

 print("Original T (MAC for one-block message):", t.hex())
 print("Calculated CBC MAC for two-block message:", cbc_mac_for_two_block.hex())

if _name_ == "_main_":
 main()
```

OUTPUT:
Original Message 1: 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
MAC for Message 1: 40 bf ab f4 06 ee 4d 30 42 ca 6b 99 7a 5c 58 16
Original Message 2: 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
MAC for Message 2: 40 bf ab f4 06 ee 4d 30 42 ca 6b 99 7a 5c 58 16

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};

 // Calculate L
 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];
 }

 // Left shift L
 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;
 }

 // Calculate subkey2
 for (int i = 0; i < block_size / 8; i++) {
 subkey2[i] = L[i] ^ const_Rb[i];
 }
}

int main() {
 // Set block size (64 or 128 bits)
 int block_size = 128;

 byte initial_key[block_size / 8];
 byte subkey1[block_size / 8];
 byte subkey2[block_size / 8];

 // Initialize initial_key with your key data here
 // ...

 // Calculate constants based on block size
 byte const_Rb[block_size / 8];
 if (block_size == 64) {
 byte const_64 = 0x1B; // Rb for 64-bit block size
 for (int i = 0; i < block_size / 8; i++) {
 const_Rb[i] = const_64;
 }
 } else if (block_size == 128) {
```

```
 byte const_128 = 0x87; // Rb for 128-bit block size
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;
}
```

OUTPUT:

Subkey 1: 80fafd0000000000efec8a0000000000

Subkey 2:
61f35500000000004ce3d50100000000
