Cryptography and Network Security Lab
Assignment 2

AES

Consider a sender and receiver who need to exchange data confidentially using symmetric encryption. Write a C/C++/Java program that implements AES encryption and decryption using a 64/128/256 bits key size and 64 bit block size.

```
int[] box = {0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,

        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};

    int[] invBox = {

        0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,

        0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

        0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,

        0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

        0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,

        0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

        0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,

        0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
        0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,

        0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,

        0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,

        0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,

        0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,

        0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,

        0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,

        0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};
```

Concept:
The concept involves implementing the Advanced Encryption Standard (AES) algorithm for symmetric encryption and decryption. AES operates on blocks of data, with block sizes of 128 bits being the most common. It uses a key of varying lengths (128, 192, or 256 bits) to encrypt and decrypt data. The algorithm consists of several rounds, with the number of rounds depending on the key size.

Algorithm:

1. Key Expansion:
   - Generate round keys from the initial key provided by the user. This involves applying key expansion algorithms specific to AES for generating round keys for each round of encryption and decryption.

2. Encryption:
   - Input the plaintext data to be encrypted.
   - Add the initial round key to the plaintext.
   - Perform a specified number of rounds (10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys), each consisting of four stages:
     - SubBytes: Substitute each byte of the state matrix with a corresponding byte from the S-box.
     - ShiftRows: Shift the rows of the state matrix cyclically to the left.
     - MixColumns: Mix the columns of the state matrix using a predefined matrix multiplication operation.
     - AddRoundKey: XOR the state matrix with the round key derived from the expanded key.
   - After the final round, output the encrypted data.

3. Decryption:
   - Input the ciphertext data to be decrypted.
   - Add the final round key to the ciphertext.
   - Perform a specified number of rounds in reverse order, each consisting of four stages:
     - InvShiftRows: Reverse the row shifts performed during encryption.
     - InvSubBytes: Reverse the byte substitution performed during encryption using the inverse S-box.
     - AddRoundKey: XOR the state matrix with the round key derived from the expanded key.
     - InvMixColumns: Reverse the column mixing operation using a predefined matrix multiplication operation.
   - After the final round, output the decrypted data.

4. Key Expansion:
   - Generate the round keys in reverse order for decryption using the same key expansion algorithm used for encryption.

Programming:
Implement the described algorithm in a programming language such as C/C++/Java, utilizing the provided S-box and inverse S-box lookup tables for SubBytes and InvSubBytes operations, respectively. Ensure proper handling of input data, key expansion, and round operations. Test the implementation with various key sizes (128, 192, 256 bits) and input data to verify correctness and security of the AES encryption and decryption.

CODE

```
#include <stdio.h>  // for printf
#include <stdlib.h> // for malloc, free

enum errorCode
{
   SUCCESS = 0,
   ERROR_AES_UNKNOWN_KEYSIZE,
   ERROR_MEMORY_ALLOCATION_FAILED,
};

// Implementation: S-Box

unsigned char sbox[256] = {
   //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,  // 0
   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,  // 1
   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,  // 2
   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,  // 3
   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,  // 4
   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,  // 5
   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,  // 6
   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,  // 7
   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,  // 8
   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,  // 9
   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,  // A
   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,  // B
   0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,  // C
   0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,  // D
   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,  // E
   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16}; // F

unsigned char rsbox[256] =
   {0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82,
0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee,
0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d,
0x8b, 0xd1, 0x25, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, 0x6c, 0x70,
0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc,
0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd,
0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d,
0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb,
0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec,
0x5f, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d,
0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};

unsigned char getSBoxValue(unsigned char num);
unsigned char getSBoxInvert(unsigned char num);

// Implementation: Rotate
void rotate(unsigned char *word);

// Implementation: Rcon
unsigned char Rcon[255] = {
   0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
   0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
   0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
   0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
   0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
   0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
   0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
   0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
   0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
   0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
   0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
   0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
   0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
   0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
   0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
   0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
   0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
   0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
   0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
   0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb};

unsigned char getRconValue(unsigned char num);

// Implementation: Key Schedule Core
void core(unsigned char *word, int iteration);

// Implementation: Key Expansion

enum keySize
{
   SIZE_16 = 16,
   SIZE_24 = 24,
   SIZE_32 = 32
};

void expandKey(unsigned char *expandedKey, unsigned char *key, enum keySize, size_t expandedKeySize);

// Implementation: AES Encryption

// Implementation: subBytes
void subBytes(unsigned char *state);
// Implementation: shiftRows
void shiftRows(unsigned char *state);
void shiftRow(unsigned char *state, unsigned char nbr);
// Implementation: addRoundKey
void addRoundKey(unsigned char *state, unsigned char *roundKey);
// Implementation: mixColumns
unsigned char galois_multiplication(unsigned char a, unsigned char b);
void mixColumns(unsigned char *state);
void mixColumn(unsigned char *column);
// Implementation: AES round
void aes_round(unsigned char *state, unsigned char *roundKey);
// Implementation: the main AES body
void createRoundKey(unsigned char *expandedKey, unsigned char *roundKey);
void aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds);
// Implementation: AES encryption
char aes_encrypt(unsigned char *input, unsigned char *output, unsigned char *key, enum keySize size);
// AES Decryption
void invSubBytes(unsigned char *state);
void invShiftRows(unsigned char *state);
void invShiftRow(unsigned char *state, unsigned char nbr);
void invMixColumns(unsigned char *state);
void invMixColumn(unsigned char *column);
void aes_invRound(unsigned char *state, unsigned char *roundKey);
void aes_invMain(unsigned char *state, unsigned char *expandedKey, int nbrRounds);
char aes_decrypt(unsigned char *input, unsigned char *output, unsigned char *key, enum keySize size);

int main(int argc, char *argv[])
{
   // the expanded keySize
   int expandedKeySize = 176;
```

```
// the expanded key
unsigned char expandedKey[expandedKeySize];

// the cipher key
unsigned char key[16] = {'k', 'k', 'k', 'k', 'e', 'e', 'e', 'e', 'y', 'y', 'y', 'y', '.', '.', '.', '.'};

// the cipher key size
enum keySize size = SIZE_16;

// the plaintext
unsigned char plaintext[16] = {'a', 'b', 'c', 'd', 'e', 'f', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0'};

// the ciphertext
unsigned char ciphertext[16];

// the decrypted text
unsigned char decryptedtext[16];

int i;

printf("**************************************************\n");
printf("*   Basic implementation of AES algorithm in C   *\n");
printf("**************************************************\n");

printf("\nCipher Key (HEX format):\n");

for (i = 0; i < 16; i++)
{
    // Print characters in HEX format, 16 chars per line
    printf("%2.2x%c", key[i], ((i + 1) % 16) ? ' ' : '\n');
}

// Test the Key Expansion
expandKey(expandedKey, key, size, expandedKeySize);

printf("\nExpanded Key (HEX format):\n");

for (i = 0; i < expandedKeySize; i++)
{
    printf("%2.2x%c", expandedKey[i], ((i + 1) % 16) ? ' ' : '\n');
}

printf("\nPlaintext (HEX format):\n");

for (i = 0; i < 16; i++)
{
    printf("%2.2x%c", plaintext[i], ((i + 1) % 16) ? ' ' : '\n');
}

// AES Encryption
aes_encrypt(plaintext, ciphertext, key, SIZE_16);

printf("\nCiphertext (HEX format):\n");

for (i = 0; i < 16; i++)
{
    printf("%2.2x%c", ciphertext[i], ((i + 1) % 16) ? ' ' : '\n');
}

// AES Decryption
aes_decrypt(ciphertext, decryptedtext, key, SIZE_16);

printf("\nDecrypted text (HEX format):\n");
```

```
    for (i = 0; i < 16; i++)
    {
        printf("%2.2x%c", decryptedtext[i], ((i + 1) % 16) ? ' ' : '\n');
    }

    return 0;
}

unsigned char getSBoxValue(unsigned char num)
{
    return sbox[num];
}

unsigned char getSBoxInvert(unsigned char num)
{
    return rsbox[num];
}

/* Rijndael's key schedule rotate operation
 * rotate the word eight bits to the left
 *
 * rotate(1d2c3a4f) = 2c3a4f1d
 *
 * word is an char array of size 4 (32 bit)
 */
void rotate(unsigned char *word)
{
    unsigned char c;
    int i;

    c = word[0];
    for (i = 0; i < 3; i++)
        word[i] = word[i + 1];
    word[3] = c;
}

unsigned char getRconValue(unsigned char num)
{
    return Rcon[num];
}

void core(unsigned char *word, int iteration)
{
    int i;

    // rotate the 32-bit word 8 bits to the left
    rotate(word);

    // apply S-Box substitution on all 4 parts of the 32-bit word
    for (i = 0; i < 4; ++i)
    {
        word[i] = getSBoxValue(word[i]);
    }

    // XOR the output of the rcon operation with i to the first part (leftmost) only
    word[0] = word[0] ^ getRconValue(iteration);
}

/* Rijndael's key expansion
 * expands an 128,192,256 key into an 176,208,240 bytes key
 *
 * expandedKey is a pointer to an char array of large enough size
 * key is a pointer to a non-expanded key
 */

void expandKey(unsigned char *expandedKey,
               unsigned char *key,
               enum keySize size,
               size_t expandedKeySize)
{
    // current expanded keySize, in bytes
    int currentSize = 0;
    int rconIteration = 1;
    int i;
    unsigned char t[4] = {0}; // temporary 4-byte variable

    // set the 16,24,32 bytes of the expanded key to the input key
    for (i = 0; i < size; i++)
        expandedKey[i] = key[i];
    currentSize += size;

    while (currentSize < expandedKeySize)
    {
        // assign the previous 4 bytes to the temporary value t
        for (i = 0; i < 4; i++)
        {
            t[i] = expandedKey[(currentSize - 4) + i];
        }

        /* every 16,24,32 bytes we apply the core schedule to t
         * and increment rconIteration afterwards
         */
        if (currentSize % size == 0)
        {
            core(t, rconIteration++);
        }

        // For 256-bit keys, we add an extra sbox to the calculation
        if (size == SIZE_32 && ((currentSize % size) == 16))
        {
            for (i = 0; i < 4; i++)
                t[i] = getSBoxValue(t[i]);
        }

        /* We XOR t with the four-byte block 16,24,32 bytes before the new expanded key.
         * This becomes the next four bytes in the expanded key.
         */
        for (i = 0; i < 4; i++)
        {
            expandedKey[currentSize] = expandedKey[currentSize - size] ^ t[i];
            currentSize++;
        }
    }
}

void subBytes(unsigned char *state)
{
    int i;
    /* substitute all the values from the state with the value in the SBox
     * using the state value as index for the SBox
     */
    for (i = 0; i < 16; i++)
        state[i] = getSBoxValue(state[i]);
}
```

```
void shiftRows(unsigned char *state)
{
    int i;
    // iterate over the 4 rows and call shiftRow() with that row
    for (i = 0; i < 4; i++)
        shiftRow(state + i * 4, i);
}

void shiftRow(unsigned char *state, unsigned char nbr)
{
    int i, j;
    unsigned char tmp;
    // each iteration shifts the row to the left by 1
    for (i = 0; i < nbr; i++)
    {
        tmp = state[0];
        for (j = 0; j < 3; j++)
            state[j] = state[j + 1];
        state[3] = tmp;
    }
}

void addRoundKey(unsigned char *state, unsigned char *roundKey)
{
    int i;
    for (i = 0; i < 16; i++)
        state[i] = state[i] ^ roundKey[i];
}

unsigned char galois_multiplication(unsigned char a, unsigned char b)
{
    unsigned char p = 0;
    unsigned char counter;
    unsigned char hi_bit_set;
    for (counter = 0; counter < 8; counter++)
    {
        if ((b & 1) == 1)
            p ^= a;
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if (hi_bit_set == 0x80)
            a ^= 0x1b;
        b >>= 1;
    }
    return p;
}

void mixColumns(unsigned char *state)
{
    int i, j;
    unsigned char column[4];

    // iterate over the 4 columns
    for (i = 0; i < 4; i++)
    {
        // construct one column by iterating over the 4 rows
        for (j = 0; j < 4; j++)
        {
            column[j] = state[(j * 4) + i];
        }

        // apply the mixColumn on one column
        mixColumn(column);
        // put the values back into the state
        for (j = 0; j < 4; j++)
        {
            state[(j * 4) + i] = column[j];
        }
    }
}

void mixColumn(unsigned char *column)
{
    unsigned char cpy[4];
    int i;
    for (i = 0; i < 4; i++)
    {
        cpy[i] = column[i];
    }
    column[0] = galois_multiplication(cpy[0], 2) ^
                galois_multiplication(cpy[3], 1) ^
                galois_multiplication(cpy[2], 1) ^
                galois_multiplication(cpy[1], 3);

    column[1] = galois_multiplication(cpy[1], 2) ^
                galois_multiplication(cpy[0], 1) ^
                galois_multiplication(cpy[3], 1) ^
                galois_multiplication(cpy[2], 3);

    column[2] = galois_multiplication(cpy[2], 2) ^
                galois_multiplication(cpy[1], 1) ^
                galois_multiplication(cpy[0], 1) ^
                galois_multiplication(cpy[3], 3);

    column[3] = galois_multiplication(cpy[3], 2) ^
                galois_multiplication(cpy[2], 1) ^
                galois_multiplication(cpy[1], 1) ^
                galois_multiplication(cpy[0], 3);
}

void aes_round(unsigned char *state, unsigned char *roundKey)
{
    subBytes(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state, roundKey);
}

void createRoundKey(unsigned char *expandedKey, unsigned char *roundKey)
{
    int i, j;
    // iterate over the columns
    for (i = 0; i < 4; i++)
    {
        // iterate over the rows
        for (j = 0; j < 4; j++)
            roundKey[(i + (j * 4))] = expandedKey[(i * 4) + j];
    }
}

void aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds)
{
    int i = 0;

    unsigned char roundKey[16];
    createRoundKey(expandedKey, roundKey);
    addRoundKey(state, roundKey);

    for (i = 1; i < nbrRounds; i++)
    {
        createRoundKey(expandedKey + 16 * i, roundKey);
        aes_round(state, roundKey);
    }

    createRoundKey(expandedKey + 16 * nbrRounds, roundKey);
    subBytes(state);
    shiftRows(state);
    addRoundKey(state, roundKey);
}

char aes_encrypt(unsigned char *input,
                 unsigned char *output,
                 unsigned char *key,
                 enum keySize size)
{
    // the expanded keySize
    int expandedKeySize;

    // the number of rounds
    int nbrRounds;

    // the expanded key
    unsigned char *expandedKey;

    // the 128 bit block to encode
    unsigned char block[16];

    int i, j;

    // set the number of rounds
    switch (size)
    {
    case SIZE_16:
        nbrRounds = 10;
        break;
    case SIZE_24:
        nbrRounds = 12;
        break;
    case SIZE_32:
        nbrRounds = 14;
        break;
    default:
        return ERROR_AES_UNKNOWN_KEYSIZE;
        break;
    }

    expandedKeySize = (16 * (nbrRounds + 1));

    expandedKey = (unsigned char *)malloc(expandedKeySize * sizeof(unsigned char));

    if (expandedKey == NULL)
    {
        return ERROR_MEMORY_ALLOCATION_FAILED;
    }
    else
    {
        /* Set the block values, for the block:
         * a0,0 a0,1 a0,2 a0,3
         * a1,0 a1,1 a1,2 a1,3
         * a2,0 a2,1 a2,2 a2,3
         * a3,0 a3,1 a3,2 a3,3
         * the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3
         */

        // iterate over the columns
        for (i = 0; i < 4; i++)
        {
            // iterate over the rows
            for (j = 0; j < 4; j++)
                block[(i + (j * 4))] = input[(i * 4) + j];
        }

        // expand the key into an 176, 208, 240 bytes key
        expandKey(expandedKey, key, size, expandedKeySize);

        // encrypt the block using the expandedKey
        aes_main(block, expandedKey, nbrRounds);

        // unmap the block again into the output
        for (i = 0; i < 4; i++)
        {
            // iterate over the rows
            for (j = 0; j < 4; j++)
                output[(i * 4) + j] = block[(i + (j * 4))];
        }

        // de-allocate memory for expandedKey
        free(expandedKey);
        expandedKey = NULL;
    }

    return SUCCESS;
}

void invSubBytes(unsigned char *state)
{
    int i;
    /* substitute all the values from the state with the value in the SBox
     * using the state value as index for the SBox
     */
    for (i = 0; i < 16; i++)
        state[i] = getSBoxInvert(state[i]);
}

void invShiftRows(unsigned char *state)
{
    int i;
    // iterate over the 4 rows and call invShiftRow() with that row
    for (i = 0; i < 4; i++)
        invShiftRow(state + i * 4, i);
}

void invShiftRow(unsigned char *state, unsigned char nbr)
{
    int i, j;
    unsigned char tmp;
    // each iteration shifts the row to the right by 1
    for (i = 0; i < nbr; i++)
    {
        tmp = state[3];
        for (j = 3; j > 0; j--)
            state[j] = state[j - 1];
        state[0] = tmp;
    }
}

void invMixColumns(unsigned char *state)
{
    int i, j;
    unsigned char column[4];

    // iterate over the 4 columns
    for (i = 0; i < 4; i++)
    {
        // construct one column by iterating over the 4 rows
        for (j = 0; j < 4; j++)
        {
            column[j] = state[(j * 4) + i];
        }

        // apply the invMixColumn on one column
        invMixColumn(column);

        // put the values back into the state
        for (j = 0; j < 4; j++)
        {
            state[(j * 4) + i] = column[j];
        }
    }
}

void invMixColumn(unsigned char *column)
{
    unsigned char cpy[4];
    int i;
    for (i = 0; i < 4; i++)
    {
        cpy[i] = column[i];
    }
    column[0] = galois_multiplication(cpy[0], 14) ^
                galois_multiplication(cpy[3], 9) ^
                galois_multiplication(cpy[2], 13) ^
                galois_multiplication(cpy[1], 11);
    column[1] = galois_multiplication(cpy[1], 14) ^
                galois_multiplication(cpy[0], 9) ^
                galois_multiplication(cpy[3], 13) ^
                galois_multiplication(cpy[2], 11);
    column[2] = galois_multiplication(cpy[2], 14) ^
                galois_multiplication(cpy[1], 9) ^
                galois_multiplication(cpy[0], 13) ^
                galois_multiplication(cpy[3], 11);
    column[3] = galois_multiplication(cpy[3], 14) ^
                galois_multiplication(cpy[2], 9) ^
                galois_multiplication(cpy[1], 13) ^
                galois_multiplication(cpy[0], 11);
}

void aes_invRound(unsigned char *state, unsigned char *roundKey)
{

    invShiftRows(state);
    invSubBytes(state);
    addRoundKey(state, roundKey);
    invMixColumns(state);
}

void aes_invMain(unsigned char *state, unsigned char *expandedKey, int nbrRounds)
{
    int i = 0;

    unsigned char roundKey[16];

    createRoundKey(expandedKey + 16 * nbrRounds, roundKey);
    addRoundKey(state, roundKey);

    for (i = nbrRounds - 1; i > 0; i--)
    {
        createRoundKey(expandedKey + 16 * i, roundKey);
        aes_invRound(state, roundKey);
    }

    createRoundKey(expandedKey, roundKey);
    invShiftRows(state);
    invSubBytes(state);
    addRoundKey(state, roundKey);
}

char aes_decrypt(unsigned char *input,
                 unsigned char *output,
                 unsigned char *key,
                 enum keySize size)
{
    // the expanded keySize
    int expandedKeySize;

    // the number of rounds
    int nbrRounds;

    // the expanded key
    unsigned char *expandedKey;

    // the 128 bit block to decode
    unsigned char block[16];

    int i, j;

    // set the number of rounds
    switch (size)
    {
    case SIZE_16:
        nbrRounds = 10;
        break;
    case SIZE_24:
        nbrRounds = 12;
        break;
    case SIZE_32:
        nbrRounds = 14;
        break;
    default:
        return ERROR_AES_UNKNOWN_KEYSIZE;
        break;
    }

    expandedKeySize = (16 * (nbrRounds + 1));

    expandedKey = (unsigned char *)malloc(expandedKeySize * sizeof(unsigned char));
    if (expandedKey == NULL)
    {
        return ERROR_MEMORY_ALLOCATION_FAILED;
    }
    else
    {
        /* Set the block values, for the block:
         * a0,0 a0,1 a0,2 a0,3
         * a1,0 a1,1 a1,2 a1,3
         * a2,0 a2,1 a2,2 a2,3
         * a3,0 a3,1 a3,2 a3,3
         * the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3
         */

        // iterate over the columns
        for (i = 0; i < 4; i++)
        {
            // iterate over the rows
            for (j = 0; j < 4; j++)
                block[(i + (j * 4))] = input[(i * 4) + j];
        }

        // expand the key into an 176, 208, 240 bytes key
        expandKey(expandedKey, key, size, expandedKeySize);

        // decrypt the block using the expandedKey
        aes_invMain(block, expandedKey, nbrRounds);

        // unmap the block again into the output
        for (i = 0; i < 4; i++)
        {
            // iterate over the rows
            for (j = 0; j < 4; j++)
                output[(i * 4) + j] = block[(i + (j * 4))];
        }

        // de-allocate memory for expandedKey
        free(expandedKey);
        expandedKey = NULL;
    }

    return SUCCESS;
}
```

SAMPLE INPUT OUTPUT

```
**************************************************
*   Basic implementation of AES algorithm in C   *
**************************************************

Cipher Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e

Expanded Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e
5b 5a 5a 5a 3e 3f 3f 3f 47 46 46 46 69 68 68 68
1c 1f 1f a3 22 20 20 9c 65 66 66 da 0c 0e 0e b2
b3 b4 28 5d 91 94 08 c1 f4 f2 6e 1b f8 fc 60 a9
0b 64 fb 1c 9a f0 f3 dd 6e 02 9d c6 96 fe fd 6f
a0 30 53 8c 3a c0 a0 51 54 c2 3d 97 c2 3c c0 f8
6b 8a 12 a9 51 4a b2 f8 05 88 8f 6f c7 b4 4f 97
```

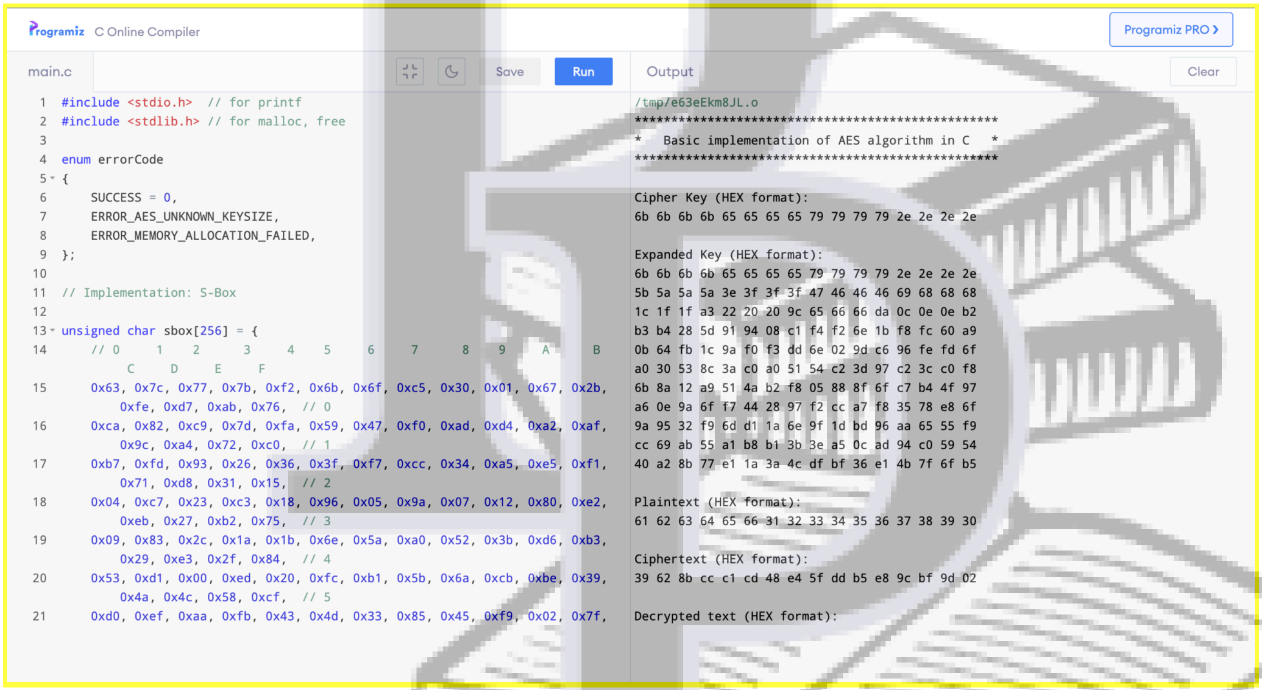
a6 0e 9a 6f f7 44 28 97 f2 cc a7 f8 35 78 e8 6f
9a 95 32 f9 6d d1 1a 6e 9f 1d bd 96 aa 65 55 f9
cc 69 ab 55 a1 b8 b1 3b 3e a5 0c ad 94 c0 59 54
40 a2 8b 77 e1 1a 3a 4c df bf 36 e1 4b 7f 6f b5

Plaintext (HEX format):
61 62 63 64 65 66 31 32 33 34 35 36 37 38 39 30

Ciphertext (HEX format):
39 62 8b cc c1 cd 48 e4 5f dd b5 e8 9c bf 9d 02

Decrypted text (HEX format):
61 62 63 64 65 66 31 32 33 34 35 36 37 38 39 30

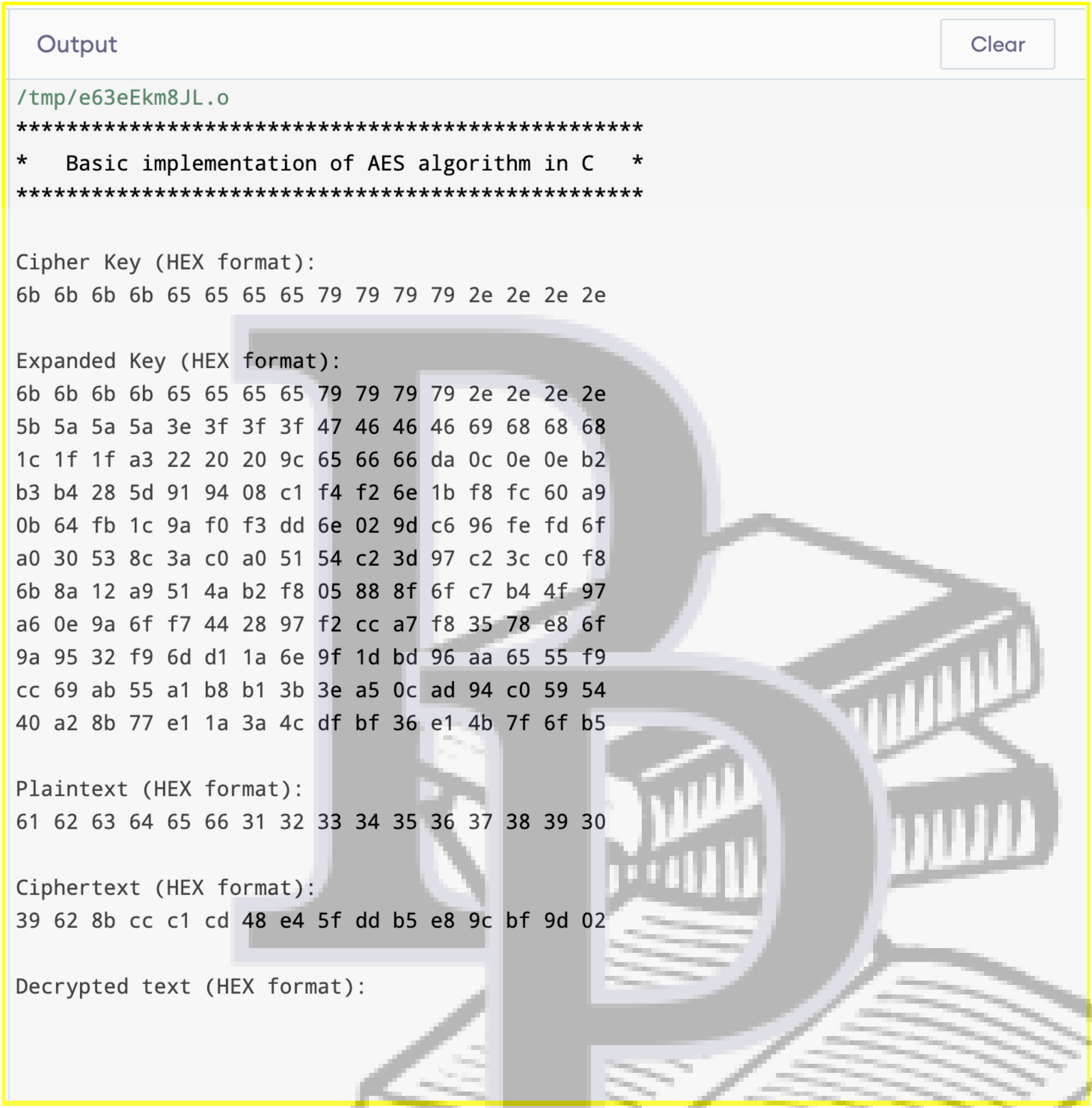
CODE SCREENSHOT:



```c
#include <stdio.h>  // for printf
#include <stdlib.h> // for malloc, free

enum errorCode
{
    SUCCESS = 0,
    ERROR_AES_UNKNOWN_KEYSIZE,
    ERROR_MEMORY_ALLOCATION_FAILED,
};

// Implementation: S-Box

unsigned char sbox[256] = {
    // 0     1    2      3     4    5     6     7      8    9     A      B
        C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
        0xfe, 0xd7, 0xab, 0x76,  // 0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
        0x9c, 0xa4, 0x72, 0xc0,  // 1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
        0x71, 0xd8, 0x31, 0x15,  // 2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
        0xeb, 0x27, 0xb2, 0x75,  // 3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
        0x29, 0xe3, 0x2f, 0x84,  // 4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
        0x4a, 0x4c, 0x58, 0xcf,  // 5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
```

```
/tmp/e63eEkm8JL.o
**************************************************
*   Basic implementation of AES algorithm in C   *
**************************************************

Cipher Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e

Expanded Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e
5b 5a 5a 5a 3e 3f 3f 3f 47 46 46 46 69 68 68 68
1c 1f 1f a3 22 20 20 9c 65 66 66 da 0c 0e 0e b2
b3 b4 28 5d 91 94 08 c1 f4 f2 6e 1b f8 fc 60 a9
0b 64 fb 1c 9a f0 f3 dd 6e 02 9d c6 96 fe fd 6f
a0 30 53 8c 3a c0 a0 51 54 c2 3d 97 c2 3c c0 f8
6b 8a 12 a9 51 4a b2 f8 05 88 8f 6f c7 b4 4f 97
a6 0e 9a 6f f7 44 28 97 f2 cc a7 f8 35 78 e8 6f
9a 95 32 f9 6d d1 1a 6e 9f 1d bd 96 aa 65 55 f9
cc 69 ab 55 a1 b8 b1 3b 3e a5 0c ad 94 c0 59 54
40 a2 8b 77 e1 1a 3a 4c df bf 36 e1 4b 7f 6f b5

Plaintext (HEX format):
61 62 63 64 65 66 31 32 33 34 35 36 37 38 39 30

Ciphertext (HEX format):
39 62 8b cc c1 cd 48 e4 5f dd b5 e8 9c bf 9d 02

Decrypted text (HEX format):
```

OUTPUT SCREENSHOT:


Output Clear

```
/tmp/e63eEkm8JL.o
**************************************************
*   Basic implementation of AES algorithm in C   *
**************************************************

Cipher Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e

Expanded Key (HEX format):
6b 6b 6b 6b 65 65 65 65 79 79 79 79 2e 2e 2e 2e
5b 5a 5a 5a 3e 3f 3f 3f 47 46 46 46 69 68 68 68
1c 1f 1f a3 22 20 20 9c 65 66 66 da 0c 0e 0e b2
b3 b4 28 5d 91 94 08 c1 f4 f2 6e 1b f8 fc 60 a9
0b 64 fb 1c 9a f0 f3 dd 6e 02 9d c6 96 fe fd 6f
a0 30 53 8c 3a c0 a0 51 54 c2 3d 97 c2 3c c0 f8
6b 8a 12 a9 51 4a b2 f8 05 88 8f 6f c7 b4 4f 97
a6 0e 9a 6f f7 44 28 97 f2 cc a7 f8 35 78 e8 6f
9a 95 32 f9 6d d1 1a 6e 9f 1d bd 96 aa 65 55 f9
cc 69 ab 55 a1 b8 b1 3b 3e a5 0c ad 94 c0 59 54
40 a2 8b 77 e1 1a 3a 4c df bf 36 e1 4b 7f 6f b5

Plaintext (HEX format):
61 62 63 64 65 66 31 32 33 34 35 36 37 38 39 30

Ciphertext (HEX format):
39 62 8b cc c1 cd 48 e4 5f dd b5 e8 9c bf 9d 02

Decrypted text (HEX format):
```

2) RSA

CODE:

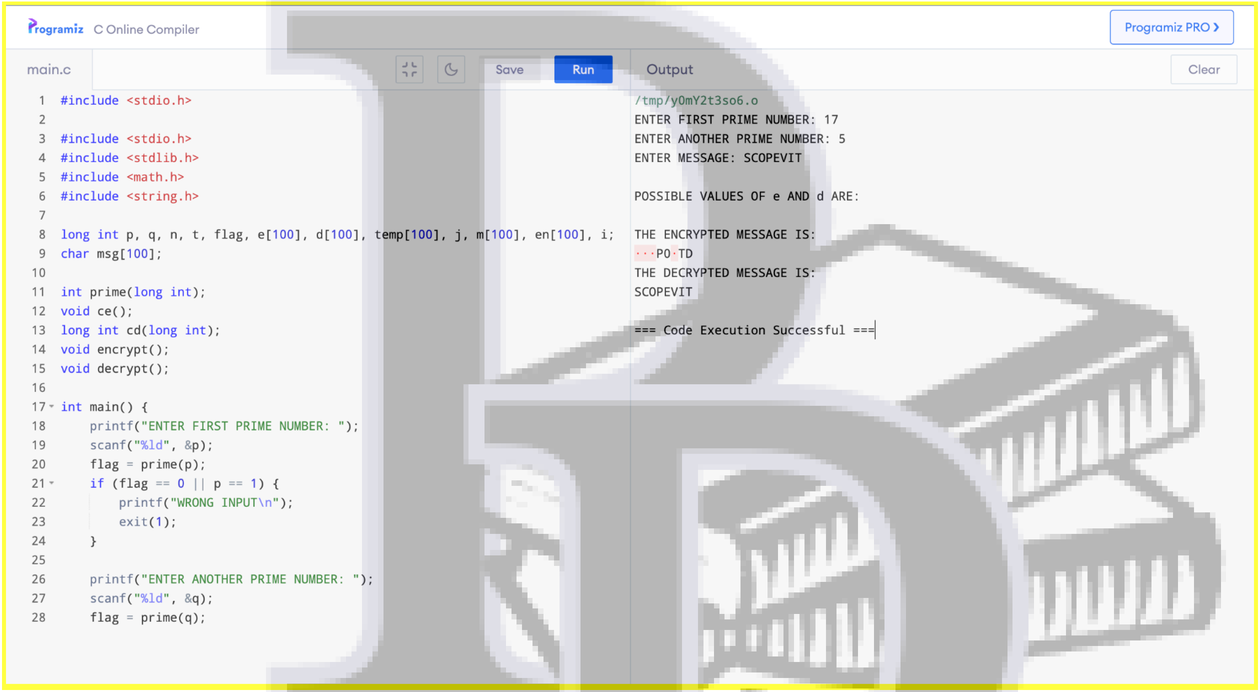
```
#include <stdio.h>

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

long int p, q, n, t, flag, e[100], d[100], temp[100], j, m[100], en[100], i;
char msg[100];

int prime(long int);
void ce();
long int cd(long int);
void encrypt();
void decrypt();
```

```
int main() {
    printf("ENTER FIRST PRIME NUMBER: ");
    scanf("%ld", &p);
    flag = prime(p);
    if (flag == 0 || p == 1) {
        printf("WRONG INPUT\n");
        exit(1);
    }

    printf("ENTER ANOTHER PRIME NUMBER: ");
    scanf("%ld", &q);
    flag = prime(q);
    if (flag == 0 || q == 1 || p == q) {
        printf("WRONG INPUT\n");
        exit(1);
    }

    printf("ENTER MESSAGE: ");
    scanf(" %[^\n]s", msg);
    for (i = 0; i < strlen(msg); i++)
        m[i] = msg[i];
    n = p * q;
    t = (p - 1) * (q - 1);

    ce();

    printf("\nPOSSIBLE VALUES OF e AND d ARE:\n");
    for (i = 0; i < j - 1; i++)
        printf("%ld\t%ld\n", e[i], d[i]);

    encrypt();
    decrypt();

    return 0;
}

int prime(long int pr) {
    int i;
    if (pr == 1)
        return 0;

    for (i = 2; i <= sqrt(pr); i++) {
        if (pr % i == 0)
            return 0;
    }
    return 1;
}

void ce() {
    int k;
    k = 0;
    for (i = 2; i < t; i++) {
        if (t % i == 0)
            continue;
        flag = prime(i);
        if (flag == 1 && i != p && i != q) {
            e[k] = i;
            flag = cd(e[k]);
            if (flag > 0) {
                d[k] = flag;
                k++;
            }
            if (k == 99)
            break;
        }
    }
}

long int cd(long int x) {
    long int k = 1;
    while (1) {
        k = k + t;
        if (k % x == 0)
            return (k / x);
    }
}

void encrypt() {
    long int pt, ct, key = e[0], k, len;
    i = 0;
    len = strlen(msg);
    while (i < len) {
        pt = m[i];
        pt = pt - 96;
        k = 1;
        for (j = 0; j < key; j++) {
            k = k * pt;
            k = k % n;
        }
        temp[i] = k;
        ct = k + 96;
        en[i] = ct;
        i++;
    }
    en[i] = -1;
    printf("\nTHE ENCRYPTED MESSAGE IS:\n");
    for (i = 0; en[i] != -1; i++)
        printf("%c", (char)en[i]);
}

void decrypt() {
    long int pt, ct, key = d[0], k;
    i = 0;
    while (en[i] != -1) {
        ct = temp[i];
        k = 1;
        for (j = 0; j < key; j++) {
            k = k * ct;
            k = k % n;
        }
        pt = k + 96;
        m[i] = pt;
        i++;
    }
    m[i] = -1;
    printf("\nTHE DECRYPTED MESSAGE IS:\n");
    for (i = 0; m[i] != -1; i++)
        printf("%c", (char)m[i]);
}
```

SAMPLE INPUT OUTPUT

```
ENTER FIRST PRIME NUMBER: 17
ENTER ANOTHER PRIME NUMBER: 5
ENTER MESSAGE: SCOPEVIT

POSSIBLE VALUES OF e AND d ARE:

THE ENCRYPTED MESSAGE IS:
P0TD
THE DECRYPTED MESSAGE IS:
SCOPEVIT
```

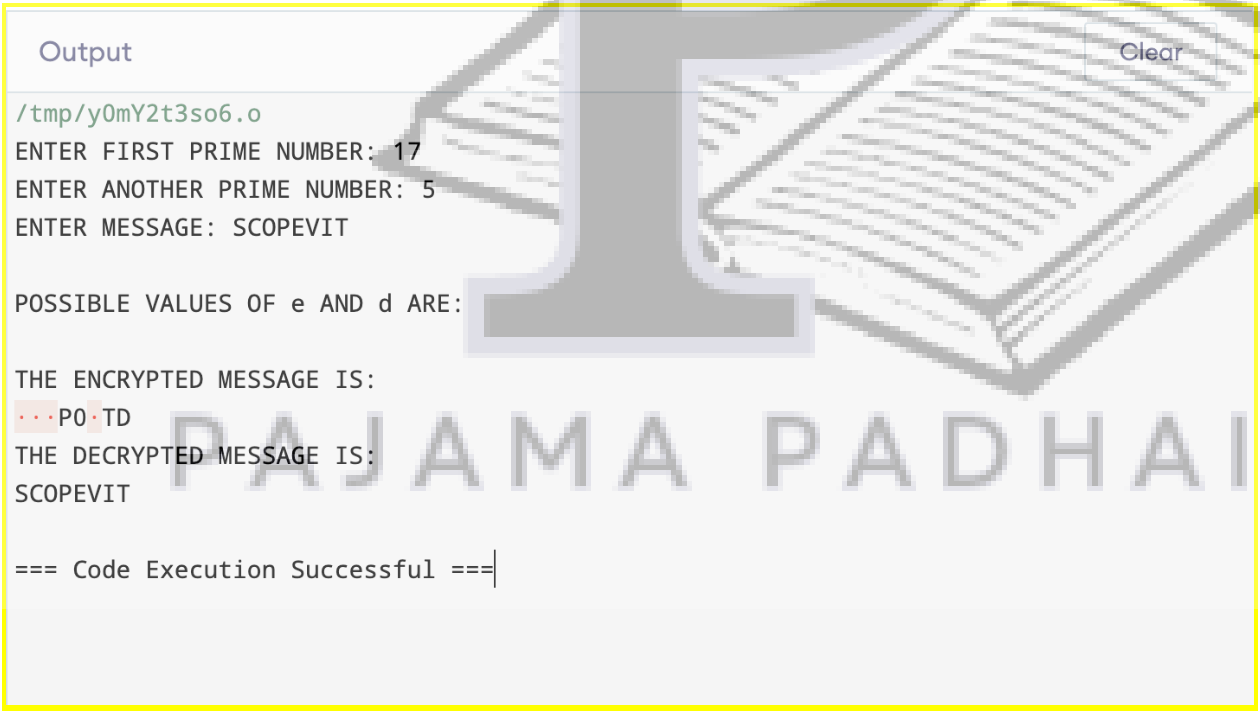
CODE SCREENSHOT:



```c
#include <stdio.h>

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

long int p, q, n, t, flag, e[100], d[100], temp[100], j, m[100], en[100], i;
char msg[100];

int prime(long int);
void ce();
long int cd(long int);
void encrypt();
void decrypt();

int main() {
    printf("ENTER FIRST PRIME NUMBER: ");
    scanf("%ld", &p);
    flag = prime(p);
    if (flag == 0 || p == 1) {
        printf("WRONG INPUT\n");
        exit(1);
    }

    printf("ENTER ANOTHER PRIME NUMBER: ");
    scanf("%ld", &q);
    flag = prime(q);
```

```
/tmp/y0mY2t3so6.o
ENTER FIRST PRIME NUMBER: 17
ENTER ANOTHER PRIME NUMBER: 5
ENTER MESSAGE: SCOPEVIT

POSSIBLE VALUES OF e AND d ARE:

THE ENCRYPTED MESSAGE IS:
···P0·TD
THE DECRYPTED MESSAGE IS:
SCOPEVIT

=== Code Execution Successful ===
```

OUTPUT SCREENSHOT:



```
/tmp/y0mY2t3so6.o
ENTER FIRST PRIME NUMBER: 17
ENTER ANOTHER PRIME NUMBER: 5
ENTER MESSAGE: SCOPEVIT

POSSIBLE VALUES OF e AND d ARE:

THE ENCRYPTED MESSAGE IS:
···P0·TD
THE DECRYPTED MESSAGE IS:
SCOPEVIT

=== Code Execution Successful ===
```