Name: Nikhil Danapgol
Batch: B2
Subject: CNS Lab
PRN: 2019BTECS00036

Assignment 7

Aim : To encrypt given plain text using the DES algorithm.

Theory :
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.

Code:

```
#include <bits/stdc++.h>
using namespace std;

string hexToBin(string s) {
    unordered_map<char, string> mp;
    mp['0'] = "0000";
    mp['1'] = "0001";
    mp['2'] = "0010";
    mp['3'] = "0011";
    mp['4'] = "0100";
    mp['5'] = "0101";
    mp['6'] = "0110";
    mp['7'] = "0111";
    mp['8'] = "1000";
    mp['9'] = "1001";
    mp['A'] = "1010";
    mp['B'] = "1011";
    mp['C'] = "1100";
    mp['D'] = "1101";
    mp['E'] = "1110";
```

```
    mp['F'] = "1111";
    stringstream bin;
    for (int i = 0; i < s.size(); i++) {
        bin << mp[s[i]];
    }
    return bin.str();
}
string binToHex(string s) {
    unordered_map<string, string> mp;
    mp["0000"] = "0";
    mp["0001"] = "1";
    mp["0010"] = "2";
    mp["0011"] = "3";
    mp["0100"] = "4";
    mp["0101"] = "5";
    mp["0110"] = "6";
    mp["0111"] = "7";
    mp["1000"] = "8";
    mp["1001"] = "9";
    mp["1010"] = "A";
    mp["1011"] = "B";
    mp["1100"] = "C";
    mp["1101"] = "D";
    mp["1110"] = "E";
    mp["1111"] = "F";
    stringstream hex;
    for (int i = 0; i < s.length(); i += 4) {
        string ch = s.substr(i, 4);
        hex << mp[ch];
    }
    return hex.str();
}

string permute(string k, int *arr, int n) {
    stringstream per;
    for (int i = 0; i < n; i++) {
        per << k[arr[i] - 1];
    }
```

```
    return per.str();
}

string shiftLeft(string k, int shifts) {
    string s = "";
    for (int i = 0; i < shifts; i++) {
        for (int j = 1; j < 28; j++) {
            s += k[j];
        }
        s += k[0];
        k = s;
        s = "";
    }
    return k;
}

string XOR(string a, string b) {
    stringstream ans;
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == b[i]) {
            ans << "0";
        } else {
            ans << "1";
        }
    }
    return ans.str();
}

string encrypt(string plain, vector<string> rkb, vector<string> rk) {
    // Hexadecimal to binary
    plain = hexToBin(plain);

    // Initial Permutation Table
    int initial_perm[64] = {58, 50, 42, 34, 26, 18, 10, 2,
                            60, 52, 44, 36, 28, 20, 12, 4,
                            62, 54, 46, 38, 30, 22, 14, 6,
                            64, 56, 48, 40, 32, 24, 16, 8,
                            57, 49, 41, 33, 25, 17, 9, 1,
```

```
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7
               };
// Initial Permutation
plain = permute(plain, initial_perm, 64);
cout << "After initial permutation: " << binToHex(plain) << endl;

// Splitting
string left = plain.substr(0, 32);
string right = plain.substr(32, 32);
cout << "After splitting: L0=" << binToHex(left)
    << " R0=" << binToHex(right) << endl;

// Expansion D-box Table
int exp_d[48] = {32, 1, 2, 3, 4, 5, 4, 5,
           6, 7, 8, 9, 8, 9, 10, 11,
           12, 13, 12, 13, 14, 15, 16, 17,
           16, 17, 18, 19, 20, 21, 20, 21,
           22, 23, 24, 25, 24, 25, 26, 27,
           28, 29, 28, 29, 30, 31, 32, 1
          };

// S-box Table
int s[8][4][16] = {{
     14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
     0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
     4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
     15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
  },
  {  15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
     3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
     0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
     13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9
  },

  {  10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
     13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
```

```
        13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
        1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12
    },
    {   7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
        13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
        10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
        3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
    },
    {   2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
        14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
        4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
        11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
    },
    {   12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
        10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
        9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
        4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
    },
    {   4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
        13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
        1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
        6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
    },
    {   13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
        1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
        7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
        2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
    }
};

// Straight Permutation Table
int per[32] = {16, 7, 20, 21,
            29, 12, 28, 17,
            1, 15, 23, 26,
            5, 18, 31, 10,
            2, 8, 24, 14,
            32, 27, 3, 9,
            19, 13, 30, 6,
```

```
               22, 11, 4, 25
              };

   cout << endl;
   for (int i = 0; i < 16; i++) {
      // Expansion D-box
      string right_expanded = permute(right, exp_d, 48);

      // XOR RoundKey[i] and right_expanded
      string x = XOR(rkb[i], right_expanded);

      // S-boxes
      string op = "";
      for (int i = 0; i < 8; i++) {
         int row = 2 * int(x[i * 6] - '0') + int(x[i * 6 + 5] - '0');
         int col = 8 * int(x[i * 6 + 1] - '0') + 4 * int(x[i * 6 + 2] - '0') + 2 * int(x[i * 6 + 3] - '0')
+ int(x[i * 6 + 4] - '0');
         int val = s[i][row][col];
         op += char(val / 8 + '0');
         val = val % 8;
         op += char(val / 4 + '0');
         val = val % 4;
         op += char(val / 2 + '0');
         val = val % 2;
         op += char(val + '0');
      }
      // Straight D-box
      op = permute(op, per, 32);

      // XOR left and op
      x = XOR(op, left);

      left = x;

      // Swapper
      if (i != 15) {
         swap(left, right);
      }
```

```
    cout << "Round " << i + 1 << " " << binToHex(left) << " "
         << binToHex(right) << " " << rk[i] << endl;
  }

  // Combination
  string combine = left + right;

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

  // Final Permutation
  string cipher = binToHex(permute(combine, final_perm, 64));
  return cipher;
}
int main() {
  string plain, key;

  // plain = "This is a test text";
  // key = "this is a test";
  // Key Generation

  cout << "Enter the plain text: ";
  getline(cin, plain);
  cout << "Enter the key: ";
  getline(cin, key);

  // Hex to binary
  key = hexToBin(key);

  // Parity bit drop table
```

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

// getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56); // key without parity

// Number of bit shifts
int shift_table[16] = {1, 1, 2, 2,
                       2, 2, 2, 2,
                       1, 2, 2, 2,
                       2, 2, 2, 1
                      };

// Key- Compression Table
int key_comp[48] = {14, 17, 11, 24, 1, 5,
                    3, 28, 15, 6, 21, 10,
                    23, 19, 12, 4, 26, 8,
                    16, 7, 27, 20, 13, 2,
                    41, 52, 31, 37, 47, 55,
                    30, 40, 51, 45, 33, 48,
                    44, 49, 39, 56, 34, 53,
                    46, 42, 50, 36, 29, 32
                   };

// Splitting
string left = key.substr(0, 28);
string right = key.substr(28, 28);

vector<string> rkb; // rkb for RoundKeys in binary
vector<string> rk;  // rk for RoundKeys in hexadecimal
for (int i = 0; i < 16; i++) {
```

```
        // Shifting
        left = shiftLeft(left, shift_table[i]);
        right = shiftLeft(right, shift_table[i]);

        // Combining
        string combine = left + right;

        // Key Compression
        string RoundKey = permute(combine, key_comp, 48);

        rkb.push_back(RoundKey);
        rk.push_back(binToHex(RoundKey));
    }

    cout << "\nEncryption:\n\n";
    string cipher = encrypt(plain, rkb, rk);
    cout << "\nCipher Text: " << cipher << endl;

    cout << "\nDecryption\n\n";
    reverse(rkb.begin(), rkb.end());
    reverse(rk.begin(), rk.end());
    string text = encrypt(cipher, rkb, rk);
    cout << "\nPlain Text: " << text << endl;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe
Enter the plain text: Nikhil
Enter the key: Wce

Encryption:

After initial permutation:
After splitting: L0= R0=

Round 1  FFFFFFFF
Round 2 FFFFFFFF FFFFFFFF
Round 3 FFFFFFFF C7240634
Round 4 C7240634 C7240634
Round 5 C7240634 FFFFFFFF
Round 6 FFFFFFFF FFFFFFFF
Round 7 FFFFFFFF C7240634
Round 8 C7240634 C7240634
Round 9 C7240634 FFFFFFFF
Round 10 FFFFFFFF FFFFFFFF
Round 11 FFFFFFFF C7240634
Round 12 C7240634 C7240634
Round 13 C7240634 FFFFFFFF
Round 14 FFFFFFFF FFFFFFFF
Round 15 FFFFFFFF C7240634
Round 16 C7240634 C7240634

Cipher Text: C0CCFF000333C0C0
```

```
Decryption

After initial permutation: C7240634C7240634
After splitting: L0=C7240634 R0=C7240634

Round 1 C7240634 FFFFFFFF
Round 2 FFFFFFFF FFFFFFFF
Round 3 FFFFFFFF C7240634
Round 4 C7240634 C7240634
Round 5 C7240634 FFFFFFFF
Round 6 FFFFFFFF FFFFFFFF
Round 7 FFFFFFFF C7240634
Round 8 C7240634 C7240634
Round 9 C7240634 FFFFFFFF
Round 10 FFFFFFFF FFFFFFFF
Round 11 FFFFFFFF C7240634
Round 12 C7240634 C7240634
Round 13 C7240634 FFFFFFFF
Round 14 FFFFFFFF FFFFFFFF
Round 15 FFFFFFFF C7240634
Round 16 C7240634 C7240634

Plain Text: C0CCFF000333C0C0

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment 8

Aim: To encrypt given plain text using AES algorithm.

Theory:

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely used today as it is a much stronger than DES and triple DES despite being harder to implement.

Code:

```
#include <bits/stdc++.h>
using namespace std;

unsigned char s[256] =
{
    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
};

// Encryption: Multiply by 2 for MixColumns
unsigned char mul2[] =
{
    0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10, 0x12, 0x14,
0x16, 0x18, 0x1a, 0x1c, 0x1e,
    0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0x30, 0x32, 0x34,
0x36, 0x38, 0x3a, 0x3c, 0x3e,
    0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0x50, 0x52, 0x54,
0x56, 0x58, 0x5a, 0x5c, 0x5e,
    0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0x70, 0x72, 0x74,
0x76, 0x78, 0x7a, 0x7c, 0x7e,
    0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0x90, 0x92, 0x94,
0x96, 0x98, 0x9a, 0x9c, 0x9e,
    0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0xb0, 0xb2, 0xb4,
0xb6, 0xb8, 0xba, 0xbc, 0xbe,
    0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0xd0, 0xd2, 0xd4,
0xd6, 0xd8, 0xda, 0xdc, 0xde,
    0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0xf0, 0xf2, 0xf4,
0xf6, 0xf8, 0xfa, 0xfc, 0xfe,
    0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0x0b, 0x09, 0x0f,
0x0d, 0x03, 0x01, 0x07, 0x05,
    0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0x2b, 0x29, 0x2f,
0x2d, 0x23, 0x21, 0x27, 0x25,
    0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0x4b, 0x49, 0x4f,
0x4d, 0x43, 0x41, 0x47, 0x45,
    0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0x6b, 0x69, 0x6f,
0x6d, 0x63, 0x61, 0x67, 0x65,
    0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0x8b, 0x89, 0x8f,
0x8d, 0x83, 0x81, 0x87, 0x85,
    0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0xab, 0xa9, 0xaf,
0xad, 0xa3, 0xa1, 0xa7, 0xa5,
    0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0xcb, 0xc9, 0xcf,
0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
    0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0xeb, 0xe9, 0xef,
0xed, 0xe3, 0xe1, 0xe7, 0xe5
```

```
};

// Encryption: Multiply by 3 for MixColumns
unsigned char mul3[] =
{
    0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09, 0x18, 0x1b, 0x1e,
0x1d, 0x14, 0x17, 0x12, 0x11,
    0x30, 0x33, 0x36, 0x35, 0x3c, 0x3f, 0x3a, 0x39, 0x28, 0x2b, 0x2e,
0x2d, 0x24, 0x27, 0x22, 0x21,
    0x60, 0x63, 0x66, 0x65, 0x6c, 0x6f, 0x6a, 0x69, 0x78, 0x7b, 0x7e,
0x7d, 0x74, 0x77, 0x72, 0x71,
    0x50, 0x53, 0x56, 0x55, 0x5c, 0x5f, 0x5a, 0x59, 0x48, 0x4b, 0x4e,
0x4d, 0x44, 0x47, 0x42, 0x41,
    0xc0, 0xc3, 0xc6, 0xc5, 0xcc, 0xcf, 0xca, 0xc9, 0xd8, 0xdb, 0xde,
0xdd, 0xd4, 0xd7, 0xd2, 0xd1,
    0xf0, 0xf3, 0xf6, 0xf5, 0xfc, 0xff, 0xfa, 0xf9, 0xe8, 0xeb, 0xee,
0xed, 0xe4, 0xe7, 0xe2, 0xe1,
    0xa0, 0xa3, 0xa6, 0xa5, 0xac, 0xaf, 0xaa, 0xa9, 0xb8, 0xbb, 0xbe,
0xbd, 0xb4, 0xb7, 0xb2, 0xb1,
    0x90, 0x93, 0x96, 0x95, 0x9c, 0x9f, 0x9a, 0x99, 0x88, 0x8b, 0x8e,
0x8d, 0x84, 0x87, 0x82, 0x81,
    0x9b, 0x98, 0x9d, 0x9e, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85,
0x86, 0x8f, 0x8c, 0x89, 0x8a,
    0xab, 0xa8, 0xad, 0xae, 0xa7, 0xa4, 0xa1, 0xa2, 0xb3, 0xb0, 0xb5,
0xb6, 0xbf, 0xbc, 0xb9, 0xba,
    0xfb, 0xf8, 0xfd, 0xfe, 0xf7, 0xf4, 0xf1, 0xf2, 0xe3, 0xe0, 0xe5,
0xe6, 0xef, 0xec, 0xe9, 0xea,
    0xcb, 0xc8, 0xcd, 0xce, 0xc7, 0xc4, 0xc1, 0xc2, 0xd3, 0xd0, 0xd5,
0xd6, 0xdf, 0xdc, 0xd9, 0xda,
    0x5b, 0x58, 0x5d, 0x5e, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45,
0x46, 0x4f, 0x4c, 0x49, 0x4a,
    0x6b, 0x68, 0x6d, 0x6e, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75,
0x76, 0x7f, 0x7c, 0x79, 0x7a,
    0x3b, 0x38, 0x3d, 0x3e, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25,
0x26, 0x2f, 0x2c, 0x29, 0x2a,
    0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15,
0x16, 0x1f, 0x1c, 0x19, 0x1a
};

// Used in KeyExpansion
unsigned char rcon[256] = {
    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, 0x8d
};

// Decryption: Inverse Rijndael S-box
unsigned char inv_s[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
};

// Decryption: Multiply by 9 for InverseMixColumns
unsigned char mul9[256] =
{
    0x00, 0x09, 0x12, 0x1b, 0x24, 0x2d, 0x36, 0x3f, 0x48, 0x41, 0x5a,
0x53, 0x6c, 0x65, 0x7e, 0x77,
    0x90, 0x99, 0x82, 0x8b, 0xb4, 0xbd, 0xa6, 0xaf, 0xd8, 0xd1, 0xca,
0xc3, 0xfc, 0xf5, 0xee, 0xe7,
    0x3b, 0x32, 0x29, 0x20, 0x1f, 0x16, 0x0d, 0x04, 0x73, 0x7a, 0x61,
0x68, 0x57, 0x5e, 0x45, 0x4c,
    0xab, 0xa2, 0xb9, 0xb0, 0x8f, 0x86, 0x9d, 0x94, 0xe3, 0xea, 0xf1,
0xf8, 0xc7, 0xce, 0xd5, 0xdc,
    0x76, 0x7f, 0x64, 0x6d, 0x52, 0x5b, 0x40, 0x49, 0x3e, 0x37, 0x2c,
0x25, 0x1a, 0x13, 0x08, 0x01,
    0xe6, 0xef, 0xf4, 0xfd, 0xc2, 0xcb, 0xd0, 0xd9, 0xae, 0xa7, 0xbc,
0xb5, 0x8a, 0x83, 0x98, 0x91,
    0x4d, 0x44, 0x5f, 0x56, 0x69, 0x60, 0x7b, 0x72, 0x05, 0x0c, 0x17,
0x1e, 0x21, 0x28, 0x33, 0x3a,
```

```
    0xdd, 0xd4, 0xcf, 0xc6, 0xf9, 0xf0, 0xeb, 0xe2, 0x95, 0x9c, 0x87,
0x8e, 0xb1, 0xb8, 0xa3, 0xaa,
    0xec, 0xe5, 0xfe, 0xf7, 0xc8, 0xc1, 0xda, 0xd3, 0xa4, 0xad, 0xb6,
0xbf, 0x80, 0x89, 0x92, 0x9b,
    0x7c, 0x75, 0x6e, 0x67, 0x58, 0x51, 0x4a, 0x43, 0x34, 0x3d, 0x26,
0x2f, 0x10, 0x19, 0x02, 0x0b,
    0xd7, 0xde, 0xc5, 0xcc, 0xf3, 0xfa, 0xe1, 0xe8, 0x9f, 0x96, 0x8d,
0x84, 0xbb, 0xb2, 0xa9, 0xa0,
    0x47, 0x4e, 0x55, 0x5c, 0x63, 0x6a, 0x71, 0x78, 0x0f, 0x06, 0x1d,
0x14, 0x2b, 0x22, 0x39, 0x30,
    0x9a, 0x93, 0x88, 0x81, 0xbe, 0xb7, 0xac, 0xa5, 0xd2, 0xdb, 0xc0,
0xc9, 0xf6, 0xff, 0xe4, 0xed,
    0x0a, 0x03, 0x18, 0x11, 0x2e, 0x27, 0x3c, 0x35, 0x42, 0x4b, 0x50,
0x59, 0x66, 0x6f, 0x74, 0x7d,
    0xa1, 0xa8, 0xb3, 0xba, 0x85, 0x8c, 0x97, 0x9e, 0xe9, 0xe0, 0xfb,
0xf2, 0xcd, 0xc4, 0xdf, 0xd6,
    0x31, 0x38, 0x23, 0x2a, 0x15, 0x1c, 0x07, 0x0e, 0x79, 0x70, 0x6b,
0x62, 0x5d, 0x54, 0x4f, 0x46
};

// Decryption: Multiply by 11 for InverseMixColumns
unsigned char mul11[256] =
{
    0x00, 0x0b, 0x16, 0x1d, 0x2c, 0x27, 0x3a, 0x31, 0x58, 0x53, 0x4e,
0x45, 0x74, 0x7f, 0x62, 0x69,
    0xb0, 0xbb, 0xa6, 0xad, 0x9c, 0x97, 0x8a, 0x81, 0xe8, 0xe3, 0xfe,
0xf5, 0xc4, 0xcf, 0xd2, 0xd9,
    0x7b, 0x70, 0x6d, 0x66, 0x57, 0x5c, 0x41, 0x4a, 0x23, 0x28, 0x35,
0x3e, 0x0f, 0x04, 0x19, 0x12,
    0xcb, 0xc0, 0xdd, 0xd6, 0xe7, 0xec, 0xf1, 0xfa, 0x93, 0x98, 0x85,
0x8e, 0xbf, 0xb4, 0xa9, 0xa2,
    0xf6, 0xfd, 0xe0, 0xeb, 0xda, 0xd1, 0xcc, 0xc7, 0xae, 0xa5, 0xb8,
0xb3, 0x82, 0x89, 0x94, 0x9f,
    0x46, 0x4d, 0x50, 0x5b, 0x6a, 0x61, 0x7c, 0x77, 0x1e, 0x15, 0x08,
0x03, 0x32, 0x39, 0x24, 0x2f,
    0x8d, 0x86, 0x9b, 0x90, 0xa1, 0xaa, 0xb7, 0xbc, 0xd5, 0xde, 0xc3,
0xc8, 0xf9, 0xf2, 0xef, 0xe4,
    0x3d, 0x36, 0x2b, 0x20, 0x11, 0x1a, 0x07, 0x0c, 0x65, 0x6e, 0x73,
0x78, 0x49, 0x42, 0x5f, 0x54,
    0xf7, 0xfc, 0xe1, 0xea, 0xdb, 0xd0, 0xcd, 0xc6, 0xaf, 0xa4, 0xb9,
0xb2, 0x83, 0x88, 0x95, 0x9e,
    0x47, 0x4c, 0x51, 0x5a, 0x6b, 0x60, 0x7d, 0x76, 0x1f, 0x14, 0x09,
0x02, 0x33, 0x38, 0x25, 0x2e,
```

```
    0x8c, 0x87, 0x9a, 0x91, 0xa0, 0xab, 0xb6, 0xbd, 0xd4, 0xdf, 0xc2,
0xc9, 0xf8, 0xf3, 0xee, 0xe5,
    0x3c, 0x37, 0x2a, 0x21, 0x10, 0x1b, 0x06, 0x0d, 0x64, 0x6f, 0x72,
0x79, 0x48, 0x43, 0x5e, 0x55,
    0x01, 0x0a, 0x17, 0x1c, 0x2d, 0x26, 0x3b, 0x30, 0x59, 0x52, 0x4f,
0x44, 0x75, 0x7e, 0x63, 0x68,
    0xb1, 0xba, 0xa7, 0xac, 0x9d, 0x96, 0x8b, 0x80, 0xe9, 0xe2, 0xff,
0xf4, 0xc5, 0xce, 0xd3, 0xd8,
    0x7a, 0x71, 0x6c, 0x67, 0x56, 0x5d, 0x40, 0x4b, 0x22, 0x29, 0x34,
0x3f, 0x0e, 0x05, 0x18, 0x13,
    0xca, 0xc1, 0xdc, 0xd7, 0xe6, 0xed, 0xf0, 0xfb, 0x92, 0x99, 0x84,
0x8f, 0xbe, 0xb5, 0xa8, 0xa3
};

// Decryption: Multiply by 13 for InverseMixColumns
unsigned char mul13[256] =
{
    0x00, 0x0d, 0x1a, 0x17, 0x34, 0x39, 0x2e, 0x23, 0x68, 0x65, 0x72,
0x7f, 0x5c, 0x51, 0x46, 0x4b,
    0xd0, 0xdd, 0xca, 0xc7, 0xe4, 0xe9, 0xfe, 0xf3, 0xb8, 0xb5, 0xa2,
0xaf, 0x8c, 0x81, 0x96, 0x9b,
    0xbb, 0xb6, 0xa1, 0xac, 0x8f, 0x82, 0x95, 0x98, 0xd3, 0xde, 0xc9,
0xc4, 0xe7, 0xea, 0xfd, 0xf0,
    0x6b, 0x66, 0x71, 0x7c, 0x5f, 0x52, 0x45, 0x48, 0x03, 0x0e, 0x19,
0x14, 0x37, 0x3a, 0x2d, 0x20,
    0x6d, 0x60, 0x77, 0x7a, 0x59, 0x54, 0x43, 0x4e, 0x05, 0x08, 0x1f,
0x12, 0x31, 0x3c, 0x2b, 0x26,
    0xbd, 0xb0, 0xa7, 0xaa, 0x89, 0x84, 0x93, 0x9e, 0xd5, 0xd8, 0xcf,
0xc2, 0xe1, 0xec, 0xfb, 0xf6,
    0xd6, 0xdb, 0xcc, 0xc1, 0xe2, 0xef, 0xf8, 0xf5, 0xbe, 0xb3, 0xa4,
0xa9, 0x8a, 0x87, 0x90, 0x9d,
    0x06, 0x0b, 0x1c, 0x11, 0x32, 0x3f, 0x28, 0x25, 0x6e, 0x63, 0x74,
0x79, 0x5a, 0x57, 0x40, 0x4d,
    0xda, 0xd7, 0xc0, 0xcd, 0xee, 0xe3, 0xf4, 0xf9, 0xb2, 0xbf, 0xa8,
0xa5, 0x86, 0x8b, 0x9c, 0x91,
    0x0a, 0x07, 0x10, 0x1d, 0x3e, 0x33, 0x24, 0x29, 0x62, 0x6f, 0x78,
0x75, 0x56, 0x5b, 0x4c, 0x41,
    0x61, 0x6c, 0x7b, 0x76, 0x55, 0x58, 0x4f, 0x42, 0x09, 0x04, 0x13,
0x1e, 0x3d, 0x30, 0x27, 0x2a,
    0xb1, 0xbc, 0xab, 0xa6, 0x85, 0x88, 0x9f, 0x92, 0xd9, 0xd4, 0xc3,
0xce, 0xed, 0xe0, 0xf7, 0xfa,
    0xb7, 0xba, 0xad, 0xa0, 0x83, 0x8e, 0x99, 0x94, 0xdf, 0xd2, 0xc5,
0xc8, 0xeb, 0xe6, 0xf1, 0xfc,
```

```
    0x67, 0x6a, 0x7d, 0x70, 0x53, 0x5e, 0x49, 0x44, 0x0f, 0x02, 0x15, 0x18, 0x3b, 0x36, 0x21, 0x2c,
    0x0c, 0x01, 0x16, 0x1b, 0x38, 0x35, 0x22, 0x2f, 0x64, 0x69, 0x7e, 0x73, 0x50, 0x5d, 0x4a, 0x47,
    0xdc, 0xd1, 0xc6, 0xcb, 0xe8, 0xe5, 0xf2, 0xff, 0xb4, 0xb9, 0xae, 0xa3, 0x80, 0x8d, 0x9a, 0x97
};

// Decryption: Multiply by 14 for InverseMixColumns
unsigned char mul14[256] =
{
    0x00, 0x0e, 0x1c, 0x12, 0x38, 0x36, 0x24, 0x2a, 0x70, 0x7e, 0x6c, 0x62, 0x48, 0x46, 0x54, 0x5a,
    0xe0, 0xee, 0xfc, 0xf2, 0xd8, 0xd6, 0xc4, 0xca, 0x90, 0x9e, 0x8c, 0x82, 0xa8, 0xa6, 0xb4, 0xba,
    0xdb, 0xd5, 0xc7, 0xc9, 0xe3, 0xed, 0xff, 0xf1, 0xab, 0xa5, 0xb7, 0xb9, 0x93, 0x9d, 0x8f, 0x81,
    0x3b, 0x35, 0x27, 0x29, 0x03, 0x0d, 0x1f, 0x11, 0x4b, 0x45, 0x57, 0x59, 0x73, 0x7d, 0x6f, 0x61,
    0xad, 0xa3, 0xb1, 0xbf, 0x95, 0x9b, 0x89, 0x87, 0xdd, 0xd3, 0xc1, 0xcf, 0xe5, 0xeb, 0xf9, 0xf7,
    0x4d, 0x43, 0x51, 0x5f, 0x75, 0x7b, 0x69, 0x67, 0x3d, 0x33, 0x21, 0x2f, 0x05, 0x0b, 0x19, 0x17,
    0x76, 0x78, 0x6a, 0x64, 0x4e, 0x40, 0x52, 0x5c, 0x06, 0x08, 0x1a, 0x14, 0x3e, 0x30, 0x22, 0x2c,
    0x96, 0x98, 0x8a, 0x84, 0xae, 0xa0, 0xb2, 0xbc, 0xe6, 0xe8, 0xfa, 0xf4, 0xde, 0xd0, 0xc2, 0xcc,
    0x41, 0x4f, 0x5d, 0x53, 0x79, 0x77, 0x65, 0x6b, 0x31, 0x3f, 0x2d, 0x23, 0x09, 0x07, 0x15, 0x1b,
    0xa1, 0xaf, 0xbd, 0xb3, 0x99, 0x97, 0x85, 0x8b, 0xd1, 0xdf, 0xcd, 0xc3, 0xe9, 0xe7, 0xf5, 0xfb,
    0x9a, 0x94, 0x86, 0x88, 0xa2, 0xac, 0xbe, 0xb0, 0xea, 0xe4, 0xf6, 0xf8, 0xd2, 0xdc, 0xce, 0xc0,
    0x7a, 0x74, 0x66, 0x68, 0x42, 0x4c, 0x5e, 0x50, 0x0a, 0x04, 0x16, 0x18, 0x32, 0x3c, 0x2e, 0x20,
    0xec, 0xe2, 0xf0, 0xfe, 0xd4, 0xda, 0xc8, 0xc6, 0x9c, 0x92, 0x80, 0x8e, 0xa4, 0xaa, 0xb8, 0xb6,
    0x0c, 0x02, 0x10, 0x1e, 0x34, 0x3a, 0x28, 0x26, 0x7c, 0x72, 0x60, 0x6e, 0x44, 0x4a, 0x58, 0x56,
    0x37, 0x39, 0x2b, 0x25, 0x0f, 0x01, 0x13, 0x1d, 0x47, 0x49, 0x5b, 0x55, 0x7f, 0x71, 0x63, 0x6d,
    0xd7, 0xd9, 0xcb, 0xc5, 0xef, 0xe1, 0xf3, 0xfd, 0xa7, 0xa9, 0xbb, 0xb5, 0x9f, 0x91, 0x83, 0x8d
```

```
};

// Auxiliary function for KeyExpansion
void KeyExpansionCore(unsigned char *in, unsigned char i) {
    // Rotate left by one byte: shift left
    unsigned char t = in[0];
    in[0] = in[1];
    in[1] = in[2];
    in[2] = in[3];
    in[3] = t;

    // S-box 4 bytes
    in[0] = s[in[0]];
    in[1] = s[in[1]];
    in[2] = s[in[2]];
    in[3] = s[in[3]];

    // RCon
    in[0] ^= rcon[i];
}

void KeyExpansion(unsigned char inputKey[16], unsigned char
expandedKeys[176]) {
    // The first 128 bits are the original key
    for (int i = 0; i < 16; i++) {
        expandedKeys[i] = inputKey[i];
    }

    int bytesGenerated = 16;  // Bytes we've generated so far
    int rconIteration = 1;    // Keeps track of rcon value
    unsigned char tmpCore[4]; // Temp storage for core

    while (bytesGenerated < 176) {
        /* Read 4 bytes for the core
         * They are the previously generated 4 bytes
         * Initially, these will be the final 4 bytes of the original
key
         */
        for (int i = 0; i < 4; i++) {
            tmpCore[i] = expandedKeys[i + bytesGenerated - 4];
        }

        // Perform the core once for each 16 byte key
```

```
        if (bytesGenerated % 16 == 0) {
            KeyExpansionCore(tmpCore, rconIteration++);
        }

        for (unsigned char a = 0; a < 4; a++) {
            expandedKeys[bytesGenerated] = expandedKeys[bytesGenerated
- 16] ^ tmpCore[a];
            bytesGenerated++;
        }
    }
}

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

void SubBytesEncrypt(unsigned char *state) {
    for (int i = 0; i < 16; i++) {
        state[i] = s[state[i]];
    }
}

// Shift left, adds diffusion
void ShiftRowsEncrypt(unsigned char *state) {
    unsigned char tmp[16];

    /* Column 1 */
    tmp[0] = state[0];
    tmp[1] = state[5];
    tmp[2] = state[10];
    tmp[3] = state[15];

    /* Column 2 */
    tmp[4] = state[4];
    tmp[5] = state[9];
    tmp[6] = state[14];
    tmp[7] = state[3];

    /* Column 3 */
    tmp[8] = state[8];
```

```
    tmp[9] = state[13];
    tmp[10] = state[2];
    tmp[11] = state[7];

    /* Column 4 */
    tmp[12] = state[12];
    tmp[13] = state[1];
    tmp[14] = state[6];
    tmp[15] = state[11];

    for (int i = 0; i < 16; i++) {
        state[i] = tmp[i];
    }
}

/* MixColumns uses mul2, mul3 look-up tables
 * Source of diffusion
 */
void MixColumns(unsigned char *state) {
    unsigned char tmp[16];

    tmp[0] = (unsigned char)mul2[state[0]] ^ mul3[state[1]] ^ state[2]
^ state[3];
    tmp[1] = (unsigned char)state[0] ^ mul2[state[1]] ^ mul3[state[2]]
^ state[3];
    tmp[2] = (unsigned char)state[0] ^ state[1] ^ mul2[state[2]] ^
mul3[state[3]];
    tmp[3] = (unsigned char)mul3[state[0]] ^ state[1] ^ state[2] ^
mul2[state[3]];

    tmp[4] = (unsigned char)mul2[state[4]] ^ mul3[state[5]] ^ state[6]
^ state[7];
    tmp[5] = (unsigned char)state[4] ^ mul2[state[5]] ^ mul3[state[6]]
^ state[7];
    tmp[6] = (unsigned char)state[4] ^ state[5] ^ mul2[state[6]] ^
mul3[state[7]];
    tmp[7] = (unsigned char)mul3[state[4]] ^ state[5] ^ state[6] ^
mul2[state[7]];

    tmp[8] = (unsigned char)mul2[state[8]] ^ mul3[state[9]] ^ state[10]
^ state[11];
    tmp[9] = (unsigned char)state[8] ^ mul2[state[9]] ^ mul3[state[10]]
^ state[11];
```

```
    tmp[10] = (unsigned char)state[8] ^ state[9] ^ mul2[state[10]] ^ mul3[state[11]];
    tmp[11] = (unsigned char)mul3[state[8]] ^ state[9] ^ state[10] ^ mul2[state[11]];

    tmp[12] = (unsigned char)mul2[state[12]] ^ mul3[state[13]] ^ state[14] ^ state[15];
    tmp[13] = (unsigned char)state[12] ^ mul2[state[13]] ^ mul3[state[14]] ^ state[15];
    tmp[14] = (unsigned char)state[12] ^ state[13] ^ mul2[state[14]] ^ mul3[state[15]];
    tmp[15] = (unsigned char)mul3[state[12]] ^ state[13] ^ state[14] ^ mul2[state[15]];

    for (int i = 0; i < 16; i++) {
        state[i] = tmp[i];
    }
}

/* Each round operates on 128 bits at a time
 * The number of rounds is defined in AESEncrypt()
 */
void RoundEncrypt(unsigned char *state, unsigned char *key) {
    SubBytesEncrypt(state);
    ShiftRowsEncrypt(state);
    MixColumns(state);
    AddRoundKeyEncrypt(state, key);
}

void FinalRoundEncrypt(unsigned char *state, unsigned char *key) {
    SubBytesEncrypt(state);
    ShiftRowsEncrypt(state);
    AddRoundKeyEncrypt(state, key);
}

void AESEncrypt(unsigned char *message, unsigned char *expandedKey, unsigned char *encryptedMessage) {
    unsigned char state[16]; // Stores the first 16 bytes of original message

    for (int i = 0; i < 16; i++) {
        state[i] = message[i];
    }
```

```
    int numberOfRounds = 9;

    AddRoundKeyEncrypt(state, expandedKey); // Initial round

    for (int i = 0; i < numberOfRounds; i++) {
        RoundEncrypt(state, expandedKey + (16 * (i + 1)));
    }

    FinalRoundEncrypt(state, expandedKey + 160);

    // Copy encrypted state to buffer
    for (int i = 0; i < 16; i++) {
        encryptedMessage[i] = state[i];
    }
}

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

/* InverseMixColumns uses mul9, mul11, mul13, mul14 look-up tables
 * Unmixes the columns by reversing the effect of MixColumns in
encryption
 */
void InverseMixColumnsDecrypt(unsigned char *state) {
    unsigned char tmp[16];

    tmp[0] = (unsigned char)mul14[state[0]] ^ mul11[state[1]] ^
mul13[state[2]] ^ mul9[state[3]];
    tmp[1] = (unsigned char)mul9[state[0]] ^ mul14[state[1]] ^
mul11[state[2]] ^ mul13[state[3]];
    tmp[2] = (unsigned char)mul13[state[0]] ^ mul9[state[1]] ^
mul14[state[2]] ^ mul11[state[3]];
    tmp[3] = (unsigned char)mul11[state[0]] ^ mul13[state[1]] ^
mul9[state[2]] ^ mul14[state[3]];

    tmp[4] = (unsigned char)mul14[state[4]] ^ mul11[state[5]] ^
mul13[state[6]] ^ mul9[state[7]];
```

```
    tmp[5] = (unsigned char)mul9[state[4]] ^ mul14[state[5]] ^
mul11[state[6]] ^ mul13[state[7]];
    tmp[6] = (unsigned char)mul13[state[4]] ^ mul9[state[5]] ^
mul14[state[6]] ^ mul11[state[7]];
    tmp[7] = (unsigned char)mul11[state[4]] ^ mul13[state[5]] ^
mul9[state[6]] ^ mul14[state[7]];

    tmp[8] = (unsigned char)mul14[state[8]] ^ mul11[state[9]] ^
mul13[state[10]] ^ mul9[state[11]];
    tmp[9] = (unsigned char)mul9[state[8]] ^ mul14[state[9]] ^
mul11[state[10]] ^ mul13[state[11]];
    tmp[10] = (unsigned char)mul13[state[8]] ^ mul9[state[9]] ^
mul14[state[10]] ^ mul11[state[11]];
    tmp[11] = (unsigned char)mul11[state[8]] ^ mul13[state[9]] ^
mul9[state[10]] ^ mul14[state[11]];

    tmp[12] = (unsigned char)mul14[state[12]] ^ mul11[state[13]] ^
mul13[state[14]] ^ mul9[state[15]];
    tmp[13] = (unsigned char)mul9[state[12]] ^ mul14[state[13]] ^
mul11[state[14]] ^ mul13[state[15]];
    tmp[14] = (unsigned char)mul13[state[12]] ^ mul9[state[13]] ^
mul14[state[14]] ^ mul11[state[15]];
    tmp[15] = (unsigned char)mul11[state[12]] ^ mul13[state[13]] ^
mul9[state[14]] ^ mul14[state[15]];

    for (int i = 0; i < 16; i++) {
        state[i] = tmp[i];
    }
}

// Shifts rows right (rather than left) for decryption
void ShiftRowsDecrypt(unsigned char *state) {
    unsigned char tmp[16];

    /* Column 1 */
    tmp[0] = state[0];
    tmp[1] = state[13];
    tmp[2] = state[10];
    tmp[3] = state[7];

    /* Column 2 */
    tmp[4] = state[4];
    tmp[5] = state[1];
```

```
    tmp[6] = state[14];
    tmp[7] = state[11];

    /* Column 3 */
    tmp[8] = state[8];
    tmp[9] = state[5];
    tmp[10] = state[2];
    tmp[11] = state[15];

    /* Column 4 */
    tmp[12] = state[12];
    tmp[13] = state[9];
    tmp[14] = state[6];
    tmp[15] = state[3];

    for (int i = 0; i < 16; i++) {
        state[i] = tmp[i];
    }
}

/* Perform substitution to each of the 16 bytes
 * Uses inverse S-box as lookup table
 */
void SubBytesDecrypt(unsigned char *state) {
    for (int i = 0; i < 16; i++) { // Perform substitution to each of
the 16 bytes
        state[i] = inv_s[state[i]];
    }
}

/* Each round operates on 128 bits at a time
 * The number of rounds is defined in AESDecrypt()
 * Not surprisingly, the steps are the encryption steps but reversed
 */
void RoundDecrypt(unsigned char *state, unsigned char *key) {
    SubRoundKeyDecrypt(state, key);
    InverseMixColumnsDecrypt(state);
    ShiftRowsDecrypt(state);
    SubBytesDecrypt(state);
}

// Same as RoundDecrypt() but no InverseMixColumns
void InitialRoundDecrypt(unsigned char *state, unsigned char *key) {
```

```
    SubRoundKeyDecrypt(state, key);
    ShiftRowsDecrypt(state);
    SubBytesDecrypt(state);
}

/* The AES decryption function
 * Organizes all the decryption steps into one function
 */
void AESDecrypt(unsigned char *encryptedMessage, unsigned char
*expandedKey, unsigned char *decryptedMessage) {
    unsigned char state[16]; // Stores the first 16 bytes of encrypted
message

    for (int i = 0; i < 16; i++) {
        state[i] = encryptedMessage[i];
    }

    InitialRoundDecrypt(state, expandedKey + 160);

    int numberOfRounds = 9;

    for (int i = 8; i >= 0; i--) {
        RoundDecrypt(state, expandedKey + (16 * (i + 1)));
    }

    SubRoundKeyDecrypt(state, expandedKey); // Final round

    // Copy decrypted state to buffer
    for (int i = 0; i < 16; i++) {
        decryptedMessage[i] = state[i];
    }
}

int main() {

    cout << "AES Algorithm" << endl;
    cout << "Enter 1 for encryption \n 2 for decryption" << endl;
    int choice;
    cin >> choice;
    if (choice == 1) {
        char message[1024];

        cout << "Enter the message to encrypt: ";
```

```
        fflush(stdin);
        cin.getline(message, sizeof(message));
        cout << message << endl;

        // Pad message to 16 bytes
        int originalLen = strlen((const char *)message);

        int paddedMessageLen = originalLen;

        if ((paddedMessageLen % 16) != 0) {
            paddedMessageLen = (paddedMessageLen / 16 + 1) * 16;
        }

        unsigned char *paddedMessage = new unsigned
char[paddedMessageLen];
        for (int i = 0; i < paddedMessageLen; i++) {
            if (i >= originalLen) {
                paddedMessage[i] = 0;
            } else {
                paddedMessage[i] = message[i];
            }
        }

        unsigned char *encryptedMessage = new unsigned
char[paddedMessageLen];

        string str;
        ifstream infile;
        infile.open("keyfile", ios::in | ios::binary);

        if (infile.is_open()) {
            getline(infile, str); // The first line of file should be
the key
            infile.close();
        }

        else
            cout << "Unable to open file";

        istringstream hex_chars_stream(str);
        unsigned char key[16];
        int i = 0;
        unsigned int c;
```

```
        while (hex_chars_stream >> hex >> c) {
            key[i] = c;
            i++;
        }

        unsigned char expandedKey[176];

        KeyExpansion(key, expandedKey);

        for (int i = 0; i < paddedMessageLen; i += 16) {
            AESEncrypt(paddedMessage + i, expandedKey, encryptedMessage
+ i);
        }

        cout << "Encrypted message in hex:" << endl;
        for (int i = 0; i < paddedMessageLen; i++) {
            cout << hex << (int)encryptedMessage[i];
            cout << " ";
        }

        cout << endl;

        // Write the encrypted string out to file "message.aes"
        ofstream outfile;
        outfile.open("message.aes", ios::out | ios::binary);
        if (outfile.is_open()) {
            outfile << encryptedMessage;
            outfile.close();
            cout << "Wrote encrypted message to file message.aes" <<
endl;
        }

        else
            cout << "Unable to open file";

        // Free memory
        delete[] paddedMessage;
        delete[] encryptedMessage;
    } else if (choice == 2) {
        string msgstr;
        ifstream infile;
        infile.open("message.aes", ios::in | ios::binary);
```

```
        if (infile.is_open()) {
            getline(infile, msgstr); // The first line of file is the
message
            cout << "Read in encrypted message from message.aes" <<
endl;
            infile.close();
        }

        else
            cout << "Unable to open file";

        char *msg = new char[msgstr.size() + 1];

        strcpy(msg, msgstr.c_str());

        int n = strlen((const char *)msg);

        unsigned char *encryptedMessage = new unsigned char[n];
        for (int i = 0; i < n; i++) {
            encryptedMessage[i] = (unsigned char)msg[i];
        }

        // Free memory
        delete[] msg;

        // Read in the key
        string keystr;
        ifstream keyfile;
        keyfile.open("keyfile", ios::in | ios::binary);

        if (keyfile.is_open()) {
            getline(keyfile, keystr); // The first line of file should
be the key
            cout << "Read in the 128-bit key from keyfile" << endl;
            keyfile.close();
        }

        else
            cout << "Unable to open file";

        istringstream hex_chars_stream(keystr);
        unsigned char key[16];
        int i = 0;
```

```
        unsigned int c;
        while (hex_chars_stream >> hex >> c) {
            key[i] = c;
            i++;
        }

        unsigned char expandedKey[176];

        KeyExpansion(key, expandedKey);

        int messageLen = strlen((const char *)encryptedMessage);

        unsigned char *decryptedMessage = new unsigned
char[messageLen];

        for (int i = 0; i < messageLen; i += 16) {
            AESDecrypt(encryptedMessage + i, expandedKey,
decryptedMessage + i);
        }

        cout << "Decrypted message in hex:" << endl;
        for (int i = 0; i < messageLen; i++) {
            cout << hex << (int)decryptedMessage[i];
            cout << " ";
        }
        cout << endl;
        cout << "Decrypted message: ";
        for (int i = 0; i < messageLen; i++) {
            cout << decryptedMessage[i];
        }
        cout << endl;
    } else {
        cout << "Invalid choice" << endl;
    }
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>g++ aes.cpp

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe
AES Algorithm
Enter 1 for encryption
 2 for decryption
1
Enter the message to encrypt: Nikhil
Nikhil
Unable to open fileEncrypted message in hex:
ff b4 6d f7 86 78 9e 5b b5 52 e5 4e d2 66 28 20
Wrote encrypted message to file message.aes

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment - 9

Aim : Prime Factorization of large numbers

Theory : It is the method of factorizing the number into exactly 2 prime factors, such that production of this 2 numbers is equal to given number

Code :

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<long long> vl;
#define pll pair<ll, ll>
#define vpll vector<pll>
#define vb vector<bool>
#define PB push_back
#define MP make_pair
#define ln "\n"
#define forn(i,e) for(ll i=0; i<e; i++
#define forsn(i,s,e) for(ll i=s; i<e; i++)
#define rforn(i,e) for(ll i=e; i>=0; i--)
#define rforsn(i,s,e) for(ll i=s; i>=e; i--)
#define vasort(v) sort(v.begin(), v.end())
```

```cpp
#define vdsort(v) sort(v.begin(), v.end(),greater<ll>())
#define arrasort(arr,n) sort(arr,arr+n)
#define arrdsort(arr,n) sort(arr,arr+n,greater<ll>())

#define F first
#define S second

#define out1(x1) cout << x1 << ln
#define out2(x1,x2) cout << x1 << " " << x2 << ln
#define out3(x1,x2,x3) cout << x1 << " " << x2 << " " << x3 << ln
#define out4(x1,x2,x3,x4) cout << x1 << " " << x2 << " " << x3 << " " << x4 << ln
#define out5(x1,x2,x3,x4,x5) cout << x1 << " " << x2 << " " << x3 << " " << x4 << " " << x5 << ln
#define out6(x1,x2,x3,x4,x5,x6) cout << x1 << " " << x2 << " " << x3 << " " << x4 << " " << x5 << " " << x6 << ln

#define in1(x1) cin >> x1
#define in2(x1,x2) cin >> x1 >> x2
#define in3(x1,x2,x3) cin >> x1 >> x2 >> x3
#define in4(x1,x2,x3,x4) cin >> x1 >> x2 >> x3 >> x4
#define in5(x1,x2,x3,x4,x5) cin >> x1 >> x2 >> x3 >> x4 >> x5
#define in6(x1,x2,x3,x4,x5,x6) cin >> x1 >> x2 >> x3 >> x4 >> x5 >> x6

#define mz(a,val)   memset(a,val,sizeof(a))
#define arrin(a,n) forn(i,n) cin >> a[i];
#define arrout(a,n) forn(i,n) {cout << a[i] << " ";} cout << ln;

#define fio ios_base::sync_with_stdio(false);cin.tie(NULL);cout.tie(NULL)
#define mod 1000000007

void file()
{
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
}
```

```
string longDivision(string number, ll divisor)
{
    // As result can be very large store it in string
    string ans;

    // Find prefix of number that is larger
    // than divisor.
    ll idx = 0;
    ll temp = number[idx] - '0';
    while (temp < divisor)
        temp = temp * 10 + (number[++idx] - '0');

    // Repeatedly divide divisor with temp. After
    // every division, update temp to include one
    // more digit.
    while (number.size() > idx) {
        // Store result in answer i.e. temp / divisor
        ans += (temp / divisor) + '0';

        // Take next digit of number
        temp = (temp % divisor) * 10 + number[++idx] - '0';
    }

    // If divisor is greater than number
    if (ans.length() == 0)
        return "0";

    // else return ans
    return ans;
}

string multiply(string num1, string num2)
{
    int len1 = num1.size();
    int len2 = num2.size();
    if (len1 == 0 || len2 == 0)
        return "0";

    // will keep the result number in vector
    // in reverse order
    vector<int> result(len1 + len2, 0);
```

```
// Below two indexes are used to find positions
// in result.
int i_n1 = 0;
int i_n2 = 0;

// Go from right to left in num1
for (int i = len1 - 1; i >= 0; i--)
{
    int carry = 0;
    int n1 = num1[i] - '0';

    // To shift position to left after every
    // multiplication of a digit in num2
    i_n2 = 0;

    // Go from right to left in num2
    for (int j = len2 - 1; j >= 0; j--)
    {
        // Take current digit of second number
        int n2 = num2[j] - '0';

        // Multiply with current digit of first number
        // and add result to previously stored result
        // at current position.
        int sum = n1 * n2 + result[i_n1 + i_n2] + carry;

        // Carry for next iteration
        carry = sum / 10;

        // Store result
        result[i_n1 + i_n2] = sum % 10;

        i_n2++;
    }

    // store carry in next cell
    if (carry > 0)
        result[i_n1 + i_n2] += carry;

    // To shift position to left after every
    // multiplication of a digit in num1.
    i_n1++;
}
```

```
    // ignore '0's from the right
    int i = result.size() - 1;
    while (i >= 0 && result[i] == 0)
        i--;

    // If all were '0's - means either both or
    // one of num1 or num2 were '0'
    if (i == -1)
        return "0";

    // generate the result string
    string s = "";

    while (i >= 0)
        s += std::to_string(result[i--]);

    return s;
}

ll isPrime(ll n)
{
    // Corner case
    if (n <= 1)
        return 0;

    // Check from 2 to square root of n
    for (ll i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            return 0;

    return 1;
}

int main()
{
    ll t = 1;
    //cin >> t;

    while (t--)
    {
        string s;
        cout<<"\n Enter the number : ";
```

```
        cin >> s;

        ll till = 100000;
        for (ll i = 1; i < till; i++)
        {
            //cout << i << endl;
            if (isPrime(i) == 0)
            {
                continue;
            }

            //cout << i << endl;

            ll first = i;

            string fs = to_string(first);

            string x = longDivision(s, i);

            if (multiply(fs, x) != s)
                continue;

            cout << first << endl;
            cout << x << endl;
            cout << endl;

            break;
        }
    }
    return 0;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>g++ primeFact.cpp

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe

 Enter the number : 25
5
5


D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment - 10

Aim : To calculate GCD using Euclidean Algorithm

Theory : In mathematics, the Euclidean algorithm, or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two integers (numbers), the largest number that divides them both without a remainder. It is named after the ancient Greek mathematician Euclid, who first described it in his Elements (c. 300 BC). It is an example of an algorithm, a step-by-step procedure for performing a calculation according to well-defined rules, and is one of the oldest algorithms in common use. It can be used to reduce fractions to their simplest form, and is a part of many other number-theoretic and cryptographic calculations.

Code :

```
#include <bits/stdc++.h>
using namespace std;

int findGCD(int num1, int num2)
{
    if (num1 == 0)
    return num2;
    return findGCD(num2 % num1, num1);
}

int main()
{
    int num1, num2;
    cout << "\n Enter 1st number : ";
    cin >> num1;

    cout << "\n Enter 2nd number : ";
```

```
    cin >> num2;

    int gcd = findGCD(num1, num2);
    cout << "\n GCD is " << gcd << endl;

    return 0;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>g++ Assignment_9_Eucladian.cpp

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe

 Enter 1st number : 55

 Enter 2nd number : 2

 GCD is 1
```

Assignment - 11

Aim : To calculate GCD using Extended Euclidean Algorithm

Theory : Extended Euclidean Algorithm is an extension of the Euclidean Algorithm that computes the greatest common divisor (GCD) of integers a and b. GCD is the largest integer that divides both a and b without any remainder.

Code :

```
#include <bits/stdc++.h>
using namespace std;

int ansS, ansT;

int findGcdExtended(int r1, int r2, int s1, int s2, int t1, int t2)
{
    // Base Case
    if (r2 == 0)
    {
        ansS = s1;
        ansT = t1;
        return r1;
    }

    int q = r1 / r2;
    int r = r1 % r2;

    int s = s1 - q * s2;
    int t = t1 - q * t2;

    cout << q << " " << r1 << " " << r2 << " " << r << " " << s1 << " "
<< s2 << " " << s << " " << t1 << " " << t2 << " " << t << endl;
```

```
    return findGcdExtended(r2, r, s2, s, t2, t);
}

int main()
{
    int num1, num2, s, t;
    cout << "\n Enter 1st number : ";
    cin >> num1;

    cout << "\n Enter 2nd number : ";
    cin >> num2;

    int gcd = findGcdExtended(num1, num2, 1, 0, 0, 1);
    cout << "GCD is " << gcd << endl;
    cout << "Value of s : "<<ansS << " " <<"Value of t : "<<ansT <<
endl;

    return 0;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>g++ assignment10_extendedEucl.cpp

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe

 Enter 1st number : 161

 Enter 2nd number : 28
5 161 28 21 1 0 1 0 1 -5
1 28 21 7 0 1 -1 1 -5 6
3 21 7 0 1 -1 4 -5 6 -23
GCD is 7
Value of s : -1 Value of t : 6

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment - 12

Aim : To Implement RSA algorithm

Theory : RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes, the Public Key is given to everyone and the Private key is kept private. An example of asymmetric cryptography : A client (for example browser) sends its public key to the server and requests some data. The server encrypts the data using the client's public key and sends the encrypted data. The client receives this data and decrypts it.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Function for extended Euclidean Algorithm
int ansS, ansT;
int findGcdExtended(int r1, int r2, int s1, int s2, int t1, int t2)
{
    // Base Case
    if (r2 == 0)
    {
        ansS = s1;
        ansT = t1;
        return r1;
    }

    int q = r1 / r2;
    int r = r1 % r2;

    int s = s1 - q * s2;
    int t = t1 - q * t2;
```

```
    cout << q << " " << r1 << " " << r2 << " " << r << " " << s1 << " "
<< s2 << " " << s << " " << t1 << " " << t2 << " " << t << endl;

    return findGcdExtended(r2, r, s2, s, t2, t);
}


int modInverse(int A, int M)
{
    int x, y;
    int g = findGcdExtended(A, M, 1, 0, 0, 1);
    if (g != 1) {
        cout << "\n Inverse doesn't exist";
        return 0;
    }
    else {

        // m is added to handle negative x

        int res = (ansS % M + M) % M;
        cout << "\n Inverse is" << res << endl;
        return res;
    }
}

long long powM(long long a, long long b, long long n)
{
    if (b == 1)
        return a % n;
    long long x = powM(a, b / 2, n);
    x = (x * x) % n;
    if (b % 2)
        x = (x * a) % n;
    return x;
}

int findGCD(int num1, int num2)
{
    if (num1 == 0)
        return num2;
    return findGCD(num2 % num1, num1);
```

```
}

// Code to demonstrate RSA algorithm
int main()
{

    // Two random prime numbers
    long long p, q, e, msg;
    //17 31 7 2

    cout << "\n Please enter 2 prime number : ";
    cin >> p >> q;

    cout << "\n Enter value of e : ";
    cin >> e;

    cout << "\n Enter message to encrpyt : ";
    cin >> msg;

    cout << "\n 2 random prime numbers selected are " << p << " " << q
<< endl;

    // First part of public key:
    long long n = p * q;
    cout << "\n Product of two prime number n is " << n << endl;

    // Finding other part of public key.
    // e stands for encrypt

    cout << "\n Taken e is " << e << endl;

    long long phi = (p - 1) * (q - 1);
    cout << "\n phi is " << phi << endl;

    while (e < phi) {
        // e must be co-prime to phi and
        // smaller than phi.
        if (findGCD(e, phi) == 1)
            break;
        else
            e++;
    }
```

```
    cout << "\n Final e value is " << e << endl;


    // Private key (d stands for decrypt)

    long long d = modInverse(e, phi);
    cout << "\n d is " << d << endl;

    cout << "\n So now our public key is " << "<" << e << "," << n <<
">" << endl;
    cout << "\n So now our private key is " << "<" << d << "," << n <<
">" << endl << endl;

    // Message to be encrypted

    cout << "\n Message date is " << msg << endl;

    // Encryption c = (msg ^ e) % n
    long long c = powM(msg, e, n);
    cout << "\n Encrypted Message is " << c << endl;

    // Decryption m = (c ^ d) % n
    long long m = powM(c, d, n);
    cout << "\n Original Message is " << m << endl;

    return 0;
}
```

Output :

```
Please enter 2 prime number : 3
11

Enter value of e : 7

Enter message to encrpyt : 50

2 random prime numbers selected are 3 11

Product of two prime number n is 33

Taken e is 7

phi is 20

Final e value is 7
0 7 20 7 1 0 1 0 1 0
2 20 7 6 0 1 -2 1 0 1
1 7 6 1 1 -2 3 0 1 -1
6 6 1 0 -2 3 -20 1 -1 7

Inverse is3

d is 3

So now our public key is <7,33>

So now our private key is <3,33>


Message date is 50

Encrypted Message is 8

Original Message is 17
```

Assignment - 13

Aim : Diffi-helman key exchange Algo

Theory :

The Diffie-Hellman algorithm is one of the most important algorithms used for establishing a shared secret. At the time of exchanging data over a public network, we can use the shared secret for secret communication. We use an elliptic curve for generating points and getting a secret key using the parameters.

1. We will take four variables, i.e., P (prime), G (the primitive root of P), and a and b (private values).
2. The variables P and G both are publicly available. The sender selects a private value, either a or b, for generating a key to exchange publicly. The receiver receives the key, and that generates a secret key, after which the sender and receiver both have the same secret key to encrypt

Code :

```
#include <bits/stdc++.h>
using namespace std;


long long powM(long long a, long long b, long long n)
```

```cpp
{
    if (b == 1)
        return a % n;
    long long x = powM(a, b / 2, n);
    x = (x * x) % n;
    if (b % 2)
        x = (x * a) % n;
    return x;
}


bool checkPrimitiveRoot(long long alpha, long long q)
{
    map<long long, int> m;

    for (long long i = 1; i < q; i++)
    {
        long long x = powM(alpha, i, q);
        //cout << x << endl;
        if (m.find(x) != m.end())
            return 0;
        m[x] = 1;
    }
    return 1;
}


int main()
{
    long long q, alpha;

    q = 7; // A prime number q is taken
    alpha = 5; // A primitive root of q


    if (checkPrimitiveRoot(alpha, q) == 0)
    {
        cout << "alpha is not primitive root of q";
        return 0;
    }
    else
    {
```

```
        cout << alpha << " is private root of " << q << endl;
    }

    long long xa, ya;
    xa = 3; // xa is the chosen private key
    ya = powM(alpha, xa, q); // public key of alice
    cout << "\n private key of alice is " << xa << endl;
    cout << "\n public key of alice is " << ya << endl << endl;

    long long xb, yb;
    xb = 4; // xb is the chosen private key
    yb = powM(alpha, xb, q); // public key of bob
    cout << "\n private key of bob is " << xb << endl;
    cout << "\n public key of bob is " << yb << endl << endl;

    //key generation
    long long k1, k2;
    k1 = powM(yb, xa, q); // Secret key for Alice
    k2 = powM(ya, xb, q); // Secret key for Bob

    cout << "\n generted key by a is " << k1 << endl;
    cout << "\n generted key by b is " << k2 << endl << endl;

    return 0;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>g++ assignment12_diffHell.cpp

D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe
7 is private root of 71

 private key of alice is 4

 public key of alice is 58


 private key of bob is 3

 public key of bob is 59


 generted key by a is 4

 generted key by b is 4


D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment - 14

Aim : CRT Algorithm

Theory :

Let me write the following set of k equations:

x = a1 (mod n1)

.

.

.

x = ak (mod nk)

This is equivalent to saying that x mod ni = ai (for i=1...k). The notation above is common in group theory, where you can define the group of integers modulo some number n and then you state equivalences (or congruence) within that group. So x is the unknown; instead of knowing x, we know the remainder of the division of x by a group of numbers. If the numbers ni are pairwise coprimes (i.e. each one is coprime with all the others) then the equations have exactly one solution. Such solution will be modulo N, with N equal to the product of all the ni.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Function for extended Euclidean Algorithm
int ansS, ansT;
int findGcdExtended(int r1, int r2, int s1, int s2, int t1, int t2)
{
    // Base Case
    if (r2 == 0)
    {
        ansS = s1;
        ansT = t1;
        return r1;
    }

    int q = r1 / r2;
    int r = r1 % r2;

    int s = s1 - q * s2;
    int t = t1 - q * t2;

    cout << q << " " << r1 << " " << r2 << " " << r << " " << s1 << " "
<< s2 << " " << s << " " << t1 << " " << t2 << " " << t << endl;

    return findGcdExtended(r2, r, s2, s, t2, t);
}
```

```cpp
int modInverse(int A, int M)
{
    int x, y;
    int g = findGcdExtended(A, M, 1, 0, 0, 1);
    if (g != 1) {
        cout << "\n Inverse doesn't exist";
        return 0;
    }
    else {

        // m is added to handle negative x

        int res = (ansS % M + M) % M;
        cout << "\n Inverse is " << res << endl;
        return res;
    }
}



int findX(vector<int> num, vector<int> rem, int k)
{
    // Compute product of all numbers
    int prod = 1;
    for (int i = 0; i < k; i++)
        prod *= num[i];

    // Initialize result
    int result = 0;

    // Apply above formula
    for (int i = 0; i < k; i++) {
        int pp = prod / num[i];
        result += rem[i] * modInverse(pp, num[i]) * pp;
    }

    return result % prod;
}
```

```
int main()
{

    // 3
    // 3 4 5
    // 2 3 1

    int k;
    cout << "\n Enter total count of equations : ";
    cin >> k;

    vector<int> num(k), rem(k);
    cout<<"\n Enter divisors : ";
    for (int i = 0; i < k; i++)
        cin >> num[i];

    cout<<"\n Enter remainders : ";
    for (int i = 0; i < k; i++)
        cin >> rem[i];

    int x = findX(num, rem, k);
    cout << "\n x is " << x;


    return 0;
}
```

Output :

```
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>a.exe

 Enter total count of equations : 3

 Enter divisors : 5
7
11

 Enter remainders : 1
1
3
15 77 5 2 1 0 1 0 1 -15
2 5 2 1 0 1 -2 1 -15 31
2 2 1 0 1 -2 5 -15 31 -77

 Inverse is 3
7 55 7 6 1 0 1 0 1 -7
1 7 6 1 0 1 -1 1 -7 8
6 6 1 0 1 -1 7 -7 8 -55

 Inverse is 6
3 35 11 2 1 0 1 0 1 -3
5 11 2 1 0 1 -5 1 -3 16
2 2 1 0 1 -5 11 -3 16 -35

 Inverse is 6

 x is 36
D:\WCE_ENGINEERING\BTECH_SEM1\CNS lab\LA2>
```

Assignment - 15 (SHA - 512)

Code :

```
#include<bits/stdc++.h>

#define ull unsigned long long

#define SHA_512_INPUT_REPRESENTATION_LENGTH 128
#define BLOCK_SIZE 1024

#define BUFFER_COUNT 8
#define WORD_LENGTH 64
#define ROUND_COUNT 80

using namespace std;

void file()
{
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
}

void initialiseBuffersAndConstants(vector<ull>& buffers, vector<ull>&
constants)
{
    buffers = {
        0x6a09e667f3bcc908, 0xbb67ae8584caa73b, 0x3c6ef372fe94f82b,
0xa54ff53a5f1d36f1,
        0x510e527fade682d1, 0x9b05688c2b3e6c1f, 0x1f83d9abfb41bd6b,
0x5be0cd19137e2179
    };

    constants = {
        0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f,
0xe9b5dba58189dbbc, 0x3956c25bf348b538,
        0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118,
0xd807aa98a3030242, 0x12835b0145706fbe,
        0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2, 0x72be5d74f27b896f,
0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
```

```
        0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
        0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4,
0x76f988da831153b5, 0x983e5152ee66dfab,
        0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4,
0xc6e00bf33da88fc2, 0xd5a79147930aa725,
        0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc,
0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,
        0x53380d139d95b3df, 0x650a73548baf63de, 0x766a0abb3c77b2a8,
0x81c2c92e47edaee6, 0x92722c851482353b,
        0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791,
0xc76c51a30654be30, 0xd192e819d6ef5218,
        0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
        0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63,
0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,
        0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc, 0x78a5636f43172f60,
0x84c87814a1f0ab72, 0x8cc702081a6439ec,
        0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915,
0xc67178f2e372532b, 0xca273eceea26619c,
        0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178,
0x06f067aa72176fba, 0x0a637dc5a2c898a6,
        0x113f9804bef90dae, 0x1b710b35131c471b, 0x28db77f523047d84,
0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
        0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
0x5fcb6fab3ad6faec, 0x6c44198c4a475817
    };
}

string sha512Padding(string input)
{
    string finalPlainText = "";

    for (int i = 0 ; i < input.size() ; ++i)
    {
        finalPlainText += bitset<8>((int)input[i]).to_string();
    }

    finalPlainText += '1';

    int plainTextSize = input.size() * 8;
    int numberOfZeros = BLOCK_SIZE - ((plainTextSize +
SHA_512_INPUT_REPRESENTATION_LENGTH + 1) % BLOCK_SIZE);
```

```
    while (numberOfZeros--)
    {
        finalPlainText += '0';
    }

    finalPlainText +=
bitset<SHA_512_INPUT_REPRESENTATION_LENGTH>(plainTextSize).to_string();

    cout << "Plain text length = " << plainTextSize << endl;
    cout << "Plain text length after padding = " <<
finalPlainText.length() << endl << endl;

    return finalPlainText;
}

ull getUllFromString(string str)
{
    bitset<WORD_LENGTH> word(str);
    return word.to_ullong();
}

static inline ull rotr64(ull n, ull c)
{
    const unsigned int mask = (CHAR_BIT * sizeof(n) - 1);
    c &= mask;
    return (n >> c) | (n << ((-c)&mask ));
}

int main()
{
    file();

    vector<ull> buffers(BUFFER_COUNT);
    vector<ull> constants(ROUND_COUNT);

    initialiseBuffersAndConstants(buffers, constants);

    cout << "Enter Text: ";
    string input;
    getline(cin, input);

    cout << "Input: " << input << endl;
```

```
    string paddedInput = sha512Padding(input);

    cout << "Padded Input:" << " " << paddedInput << endl << endl;

    for (int i = 0 ; i < paddedInput.size() ; i += BLOCK_SIZE)
    {
        string currentBlock = paddedInput.substr(i, BLOCK_SIZE);

        vector<ull> w(ROUND_COUNT);
        for (int j = 0 ; j < 16 ; ++j)
        {
            w[j] = getUllFromString(currentBlock.substr(j,
WORD_LENGTH));
        }

        for (int j = 16 ; j < 80 ; ++j)
        {
            ull sigma1 = (rotr64(w[j - 15], 1)) ^ (rotr64(w[j - 15],
8)) ^ (w[j - 15] >> 7);
            ull sigma2 = (rotr64(w[j - 2], 19)) ^ (rotr64(w[j - 2],
61)) ^ (w[j - 2] >> 6);

            w[j] = w[j - 16] + sigma1 + w[j - 7] + sigma2;
        }

        ull a = buffers[0], b = buffers[1], c = buffers[2], d =
buffers[3];
        ull e = buffers[4], f = buffers[5], g = buffers[6], h =
buffers[7];

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

            ull sum0 = (rotr64(a, 28)) ^ (rotr64(a, 34)) ^ (rotr64(a,
39));
            ull sum1 = (rotr64(e, 14)) ^ (rotr64(e, 18)) ^ (rotr64(e,
41));

            ull ch = (e && f) ^ ((!e) && g);
            ull temp1 = h + sum1 + ch + constants[i] + w[i];

            ull majorityFunction = (a && b) ^ (a && c) ^ (b && c);
            ull temp2 = sum0 + majorityFunction;
```

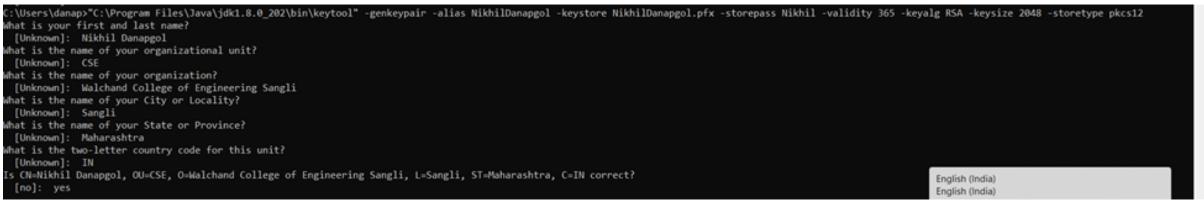
```
            h = g;
            g = f;
            f = e;
            e = d + temp1;
            d = c;
            c = b;
            b = a;
            a = temp1 + temp2;
        }

        buffers[0] += a;
        buffers[1] += b;
        buffers[2] += c;
        buffers[3] += d;
        buffers[4] += e;
        buffers[5] += f;
        buffers[6] += g;
        buffers[7] += h;
    }
    cout << "Output of SHA-512 Algorithm: " << endl;
    for (int i = 0 ; i < BUFFER_COUNT ; ++i)
    {
        cout << setfill('0') << setw(16) << right << hex << buffers[i];
    }

    return 0;
}
```

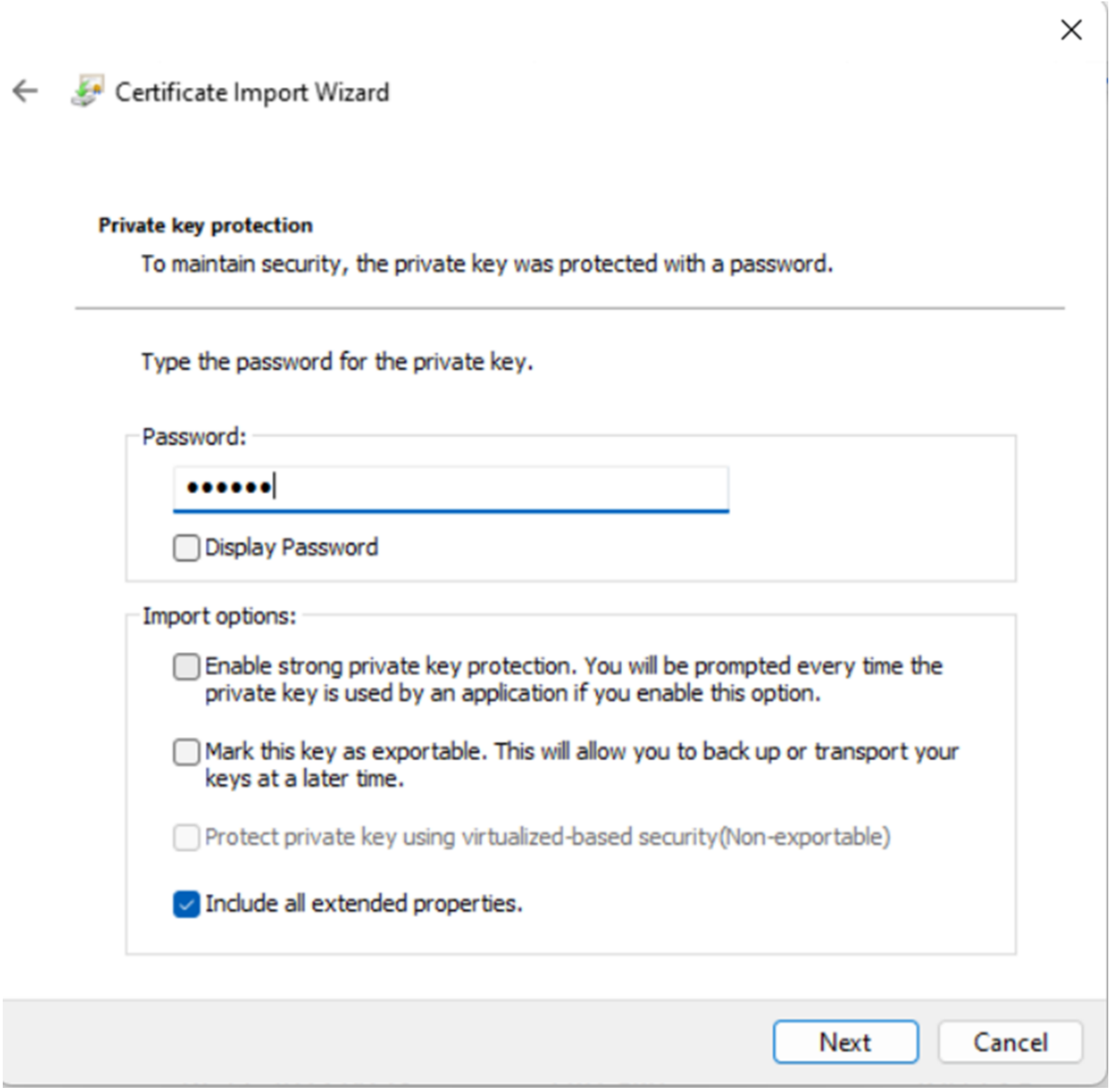
Output :

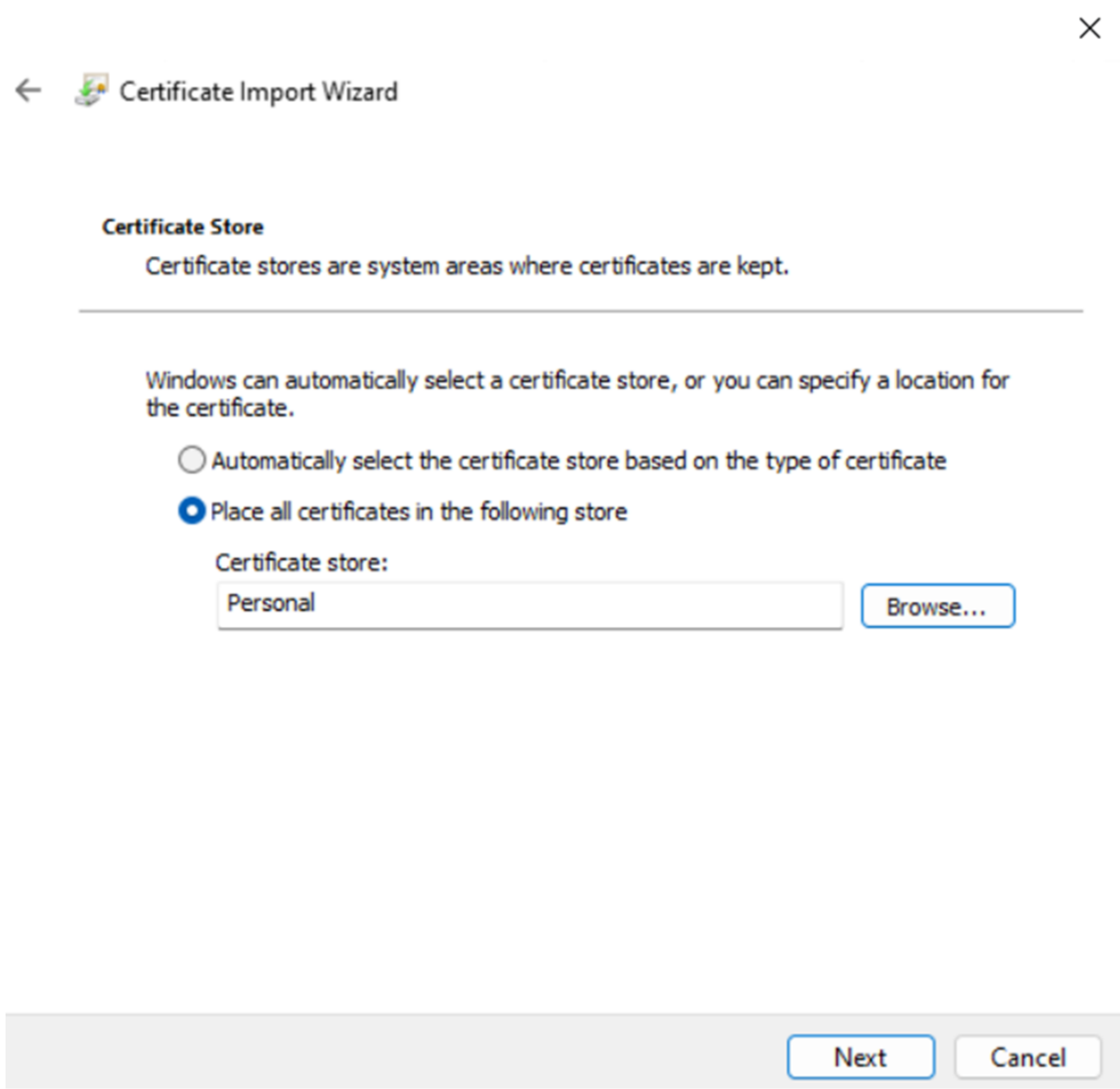
```
input.txt
1    Nikhil
```

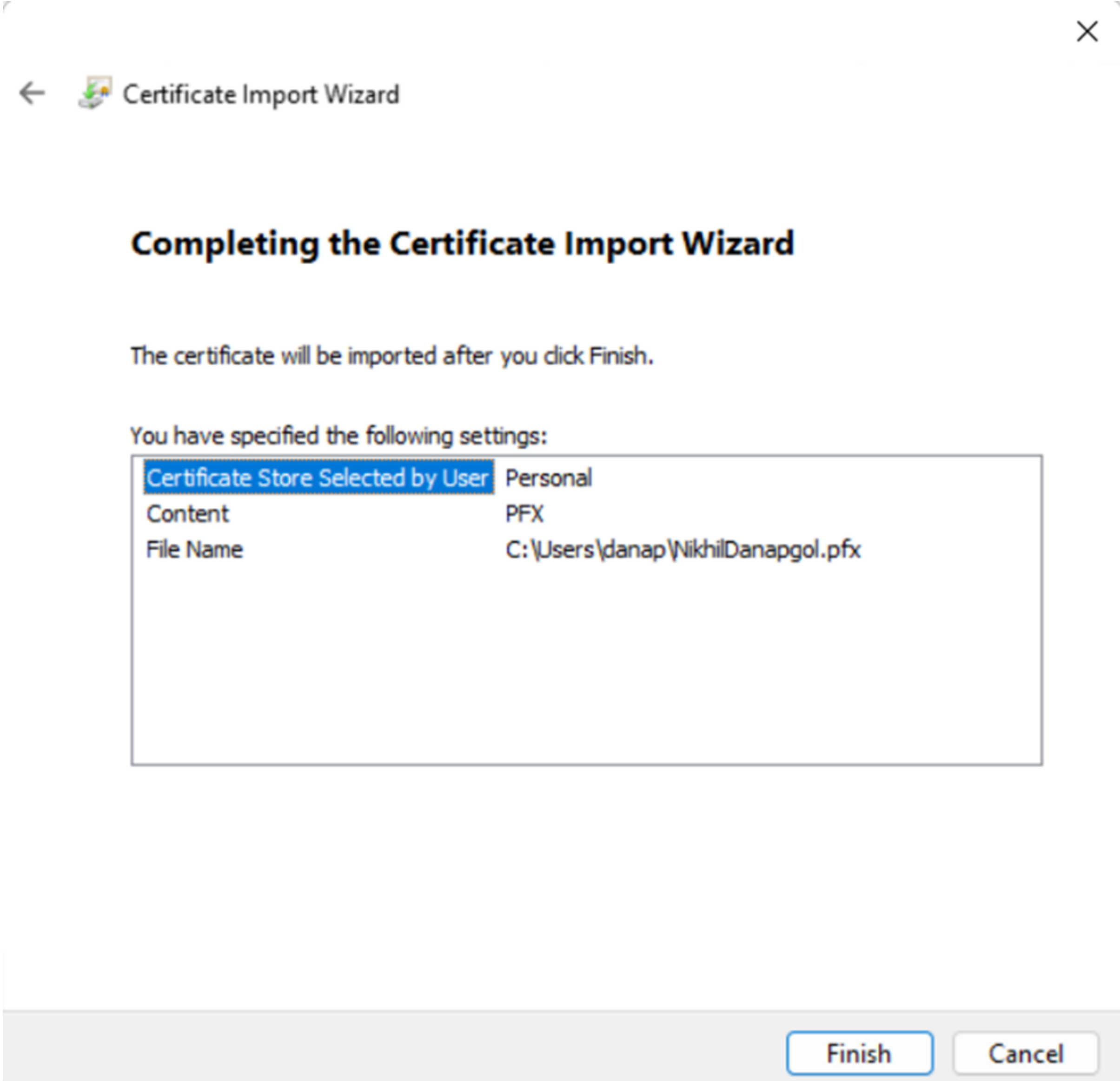
```
Enter Text: Input: Nikhil
Plain text length = 48
Plain text length after padding = 1024

Padded Input: 010011100110100101101011011010000110100101101100100000000000000000000000000000000000000000000000000000000000000000

Output of SHA-512 Algorithm:
966647fde42bb5a45e9df023048b0e8181ae5d4ad44e2ca7a36bf6f3b8e87865cded29805bbc834bcca0eb245a67aab170369be33a2744d6e44f400f581d8643
```

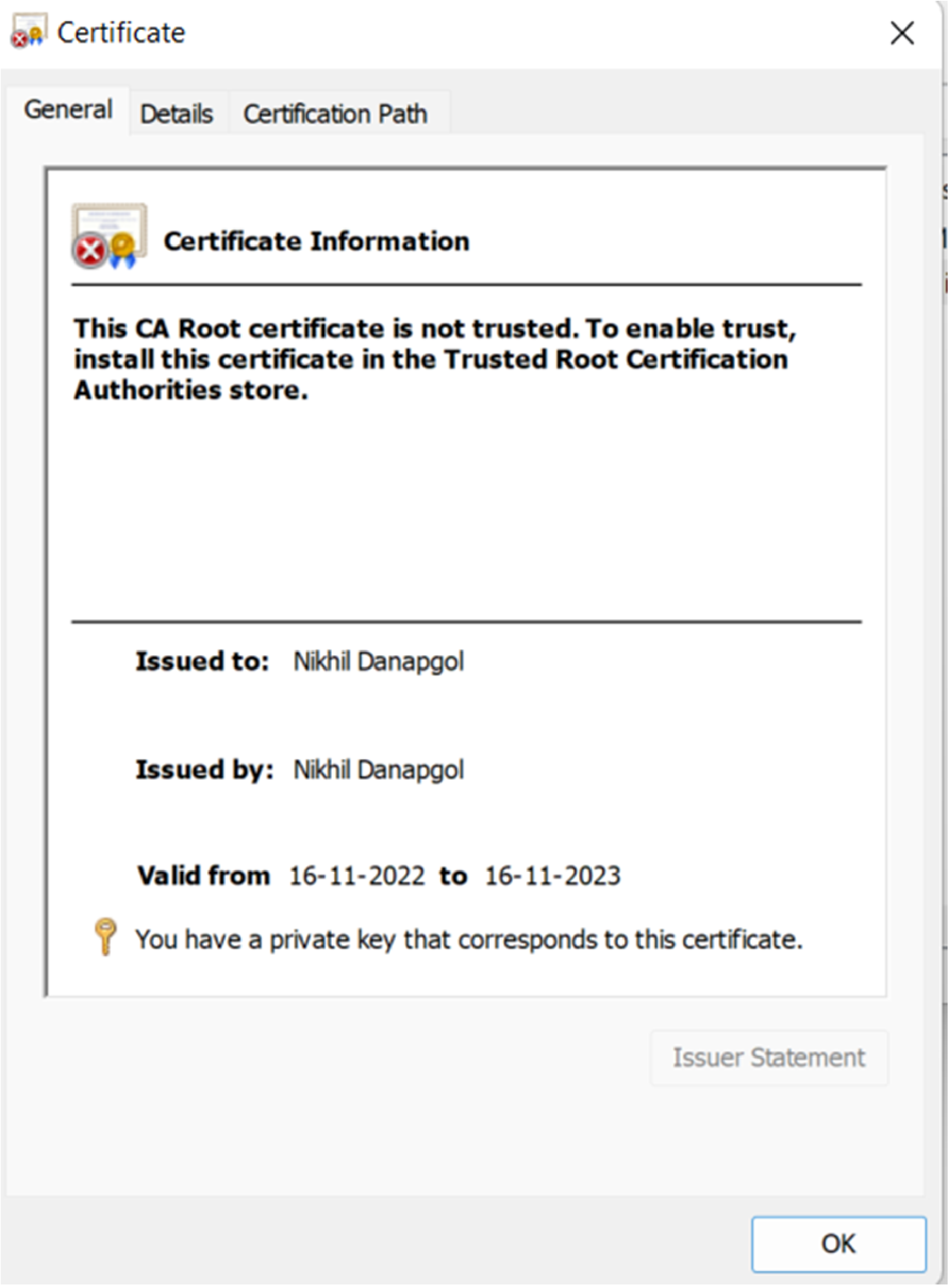
Digital Certificate:

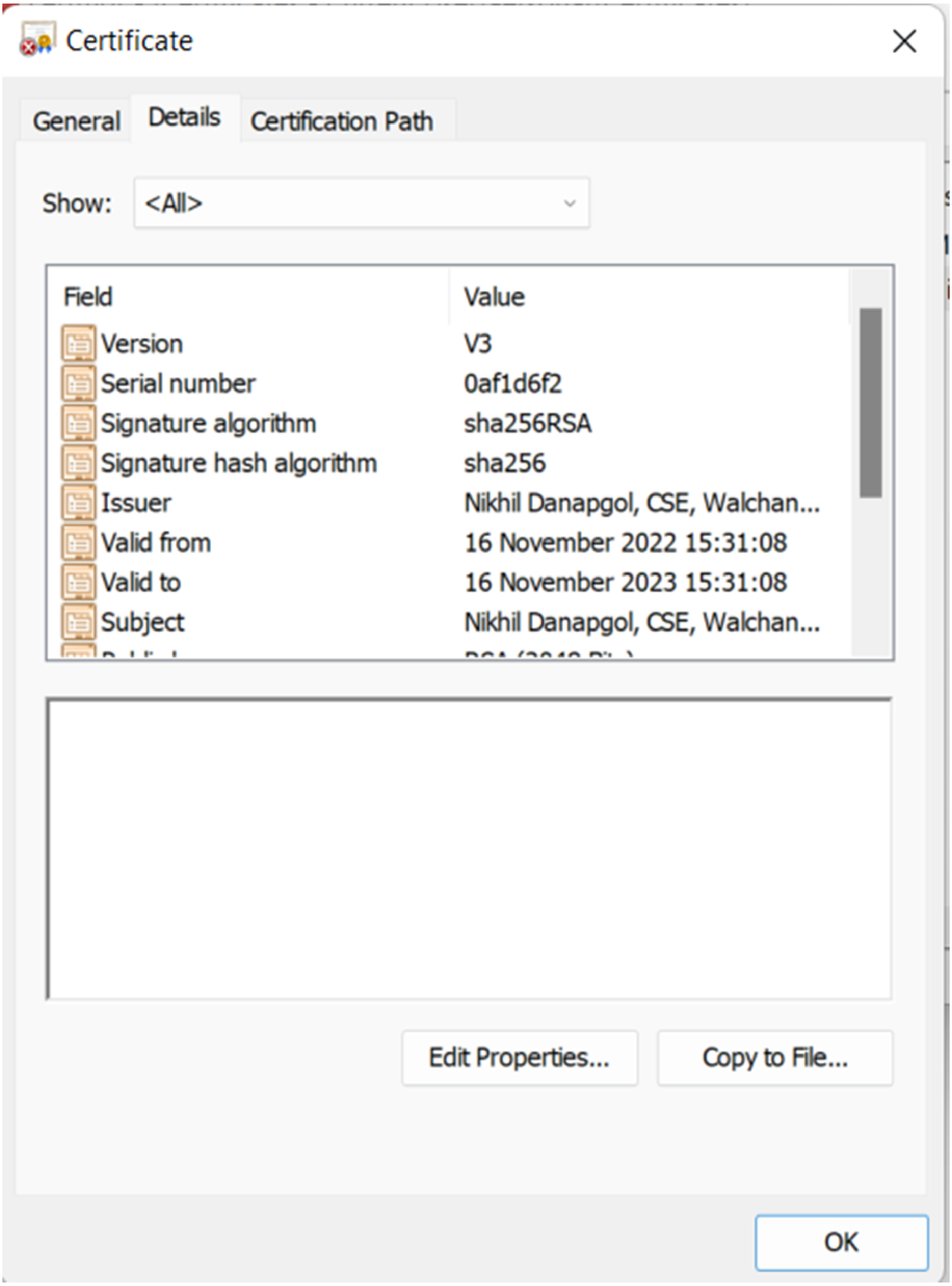

C:\Users\danap>"C:\Program Files\Java\jdk1.8.0_202\bin\keytool" -genkeypair -alias NikhilDanapgol -keystore NikhilDanapgol.pfx -storepass Nikhil -validity 365 -keyalg RSA -keysize 2048 -storetype pkcs12
What is your first and last name?
[Unknown]: Nikhil Danapgol
What is the name of your organizational unit?
[Unknown]: CSE
What is the name of your organization?
[Unknown]: Walchand College of Engineering Sangli
What is the name of your City or Locality?
[Unknown]: Sangli
What is the name of your State or Province?
[Unknown]: Maharashtra
What is the two-letter country code for this unit?
[Unknown]: IN
Is CN=Nikhil Danapgol, OU=CSE, O=Walchand College of Engineering Sangli, L=Sangli, ST=Maharashtra, C=IN correct?
[no]: yes
English (India)
English (India)


Certificate Import Wizard
Private key protection
To maintain security, the private key was protected with a password.
Type the password for the private key.
Password:
Display Password
Import options:
Enable strong private key protection. You will be prompted every time the private key is used by an application if you enable this option.
Mark this key as exportable. This will allow you to back up or transport your keys at a later time.
Protect private key using virtualized-based security(Non-exportable)
Include all extended properties.
Next
Cancel

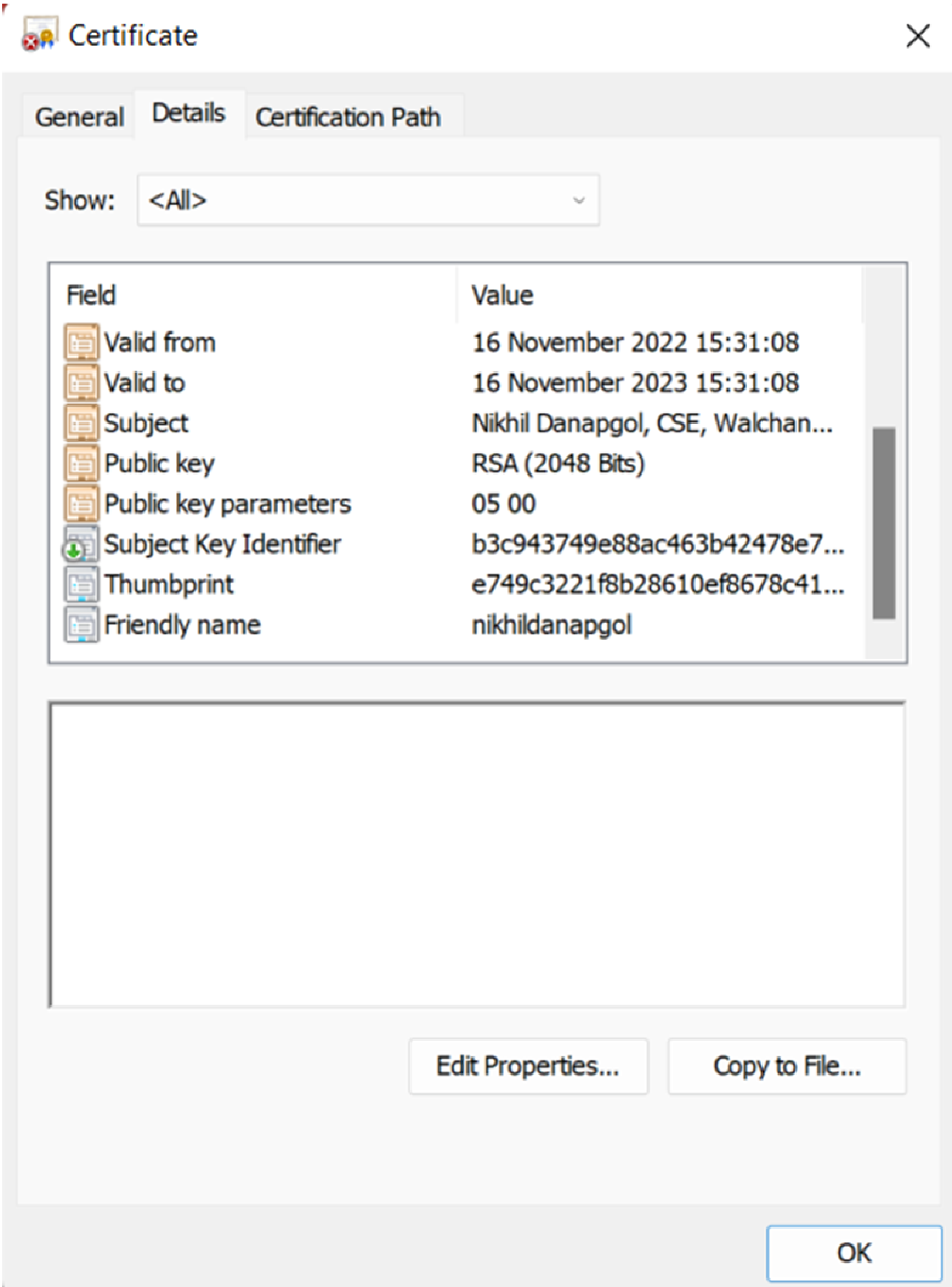

Certificate Import Wizard
Certificate Store
Certificate stores are system areas where certificates are kept.
Windows can automatically select a certificate store, or you can specify a location for the certificate.
Automatically select the certificate store based on the type of certificate
Place all certificates in the following store
Certificate store:
Personal
Browse...
Next
Cancel


Certificate Import Wizard
Completing the Certificate Import Wizard
The certificate will be imported after you click Finish.
You have specified the following settings:
Certificate Store Selected by User Personal
Content PFX
File Name C:\Users\danap\NikhilDanapgol.pfx
Finish
Cancel


Certificate
General
Details
Certification Path
Certificate Information
This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.
Issued to: Nikhil Danapgol
Issued by: Nikhil Danapgol
Valid from 16-11-2022 to 16-11-2023
You have a private key that corresponds to this certificate.
Issuer Statement
OK



Certificate
General
Details
Certification Path
Show:
<All>
Field
Value
Version
V3
Serial number
0af1d6f2
Signature algorithm
sha256RSA
Signature hash algorithm
sha256
Issuer
Nikhil Danapgol, CSE, Walchan...
Valid from
16 November 2022 15:31:08
Valid to
16 November 2023 15:31:08
Subject
Nikhil Danapgol, CSE, Walchan...
Edit Properties...
Copy to File...
OK


Certificate
General
Details
Certification Path
Show:
<All>
Field
Value
Valid from
16 November 2022 15:31:08
Valid to
16 November 2023 15:31:08
Subject
Nikhil Danapgol, CSE, Walchan...
Public key
RSA (2048 Bits)
Public key parameters
05 00
Subject Key Identifier
b3c943749e88ac463b42478e7...
Thumbprint
e749c3221f8b28610ef8678c41...
Friendly name
nikhildanapgol
Edit Properties...
Copy to File...
OK