```
#include
<iostream>
```

```
#include <cstdio>
#include <string>
#include <map>
#include <algorithm>
#include <cmath>
#include <vector>
using namespace std;


map <int, int> PDP;          //parity drop map
map <int, int> KCT;          //key compression map
map <int, int> IP;           //initial permutation map
map <int, int> FP;           //final permutation map
vector <string> leftVal;     //container for left
vector <string> rightVal;    //container for right
vector <string> keyVal;      //container for round key


int parityBitDrop[] = {
   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
};
```

```
int keyCompressionTable[] = {
   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
};


int initialPermutation[] = {
   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
};


int expansionPBoxTable [] = {
   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
```

```
};


int sBox1[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}
};


int sBox2[4][16] = {
   {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}
};


int sBox3[4][16] = {
   {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}
};


int sBox4[4][16] = {
   {  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}
```

```
};


int sBox5[4][16] = {
   { 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}
};


int sBox6[4][16] = {
   {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}
};


int sBox7[4][16] = {
   { 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}
};


int sBox8[4][16] = {
   {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}
```

```
};


int straightPermutation[] = {
   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
};


int finalPermutation[] = {
   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
};


void preCalc(){
   for(int i=0; i<64; i++) {
       IP[initialPermutation[i]] = i+1;
       FP[finalPermutation[i]] = i+1;
       if(i<56)
           PDP[parityBitDrop[i]] = i+1;
       if(i<48)
           KCT[keyCompressionTable[i]] = i+1;
   }
```

```
}


string decimalToBinary(unsigned long long n){
   string as = "";
   while(n>0){
       as += n % 2 + '0';
       n /= 2;
   }
   int len = as.size();
   for(int i=0; i<8-len; i++)
       as += "0";
   reverse(as.begin(), as.end());
   return as;
}


string parityBitDropPermutation(string as){
   map <int, int> parityBit;
   for(int i=0; i<as.size(); i++)
       parityBit[i+1] = as[i];




   int k = 1;
   string bs = "";
   for(int i=0; i<56; i++)
       bs += "0";


   for(int i=0; i<as.size(); i++){
```

```
        bs[PDP[k]-1] = parityBit[i+1];
        k++;
    }

    //clear
    parityBit.clear();

    return bs;
}

string binaryToHex(string as) {
    string bs = "", res = "";
    for(int i=0; i<as.size(); i++){
        if((i+1)%4 == 0) {
            bs += as[i];
            if(bs == "0000")      res += "0";
            else if(bs == "0001") res += "1";
            else if(bs == "0010") res += "2";
            else if(bs == "0011") res += "3";
            else if(bs == "0100") res += "4";
            else if(bs == "0101") res += "5";
            else if(bs == "0110") res += "6";
            else if(bs == "0111") res += "7";
            else if(bs == "1000") res += "8";
            else if(bs == "1001") res += "9";
            else if(bs == "1010") res += "A";
            else if(bs == "1011") res += "B";
            else if(bs == "1100") res += "C";
            else if(bs == "1101") res += "D";
```

```
            else if(bs == "1110") res += "E";
            else if(bs == "1111") res += "F";
            bs.clear();
        } else
            bs += as[i];
    }
    return res;
}


string hexToBinary(string bs) {
    string res = "";
    for(int i=0; i<bs.size(); i++){
        if(bs[i] == '0')      res += "0000";
        else if(bs[i] == '1') res += "0001";
        else if(bs[i] == '2') res += "0010";
        else if(bs[i] == '3') res += "0011";
        else if(bs[i] == '4') res += "0100";
        else if(bs[i] == '5') res += "0101";
        else if(bs[i] == '6') res += "0110";
        else if(bs[i] == '7') res += "0111";
        else if(bs[i] == '8') res += "1000";
        else if(bs[i] == '9') res += "1001";
        else if(bs[i] == 'A') res += "1010";
        else if(bs[i] == 'B') res += "1011";
        else if(bs[i] == 'C') res += "1100";
        else if(bs[i] == 'D') res += "1101";
        else if(bs[i] == 'E') res += "1110";
        else if(bs[i] == 'F') res += "1111";
    }
    return res;
```

```
}


string circularLeftShift(int n, string as){
   string ret;
   for(int i=n; i<as.size(); i++)
       ret += as[i];
   if(n == 1)
       ret+= as[0];
   else {
       ret += as[0];
       ret += as[1];
   }
   return ret;
}


string combineBit(string as, string bs){
   string ret = "";
   ret += as;
   ret += bs;
   return ret;
}


string compressionPermutation(string as){
   map <int, int> KC;
   for(int i=0; i<as.size(); i++)
       KC[i+1] = as[i];
```

```
    int k = 1;
    string bs = "";
    for(int i=0; i<48; i++)
        bs += "0";

    for(int i=0; i<as.size(); i++){
        bs[KCT[k]-1] = KC[i+1];
        k++;
    }

    //clear
    KC.clear();

    return bs;
}

string initPermutation(string as){
    int k = 1;
    string bs = "";
    for(int i=0; i<64; i++)
        bs += "0";

    for(int i=0; i<as.size(); i++){
        bs[IP[k]-1] = as[i];
        k++;
    }
```

```
    return bs;
}


string ExpansionPBT(string as){
    int k = 0;
    string bs = "";
    for(int i=0; i<48; i++)
        bs += "0";
    for(int i=0; i<48; i++){
        bs[i] = as[expansionPBoxTable[k]-1];
        k++;
    }
    return bs;
}


string xorOperation(string as, string bs){
    string ret = "";
    for(int i=0; i<as.size(); i++){
        if(as[i] == bs[i]) ret += "0";
        else  ret += "1";
    }
    return ret;
}


int binaryToDecimal(string as) {
    int res, k;
    res = k = 0;
    for(int i=as.size()-1; i>=0; i--)
        res += ((as[i] - '0') * pow(2,k++));
```

```
    return res;
}


int sboxPermutation(int sbox, string bs){
    int row, column;
    string r = "", c = "";

    //collecting row bit
    r += bs[0];
    r += bs[5];

    //collecting column bit
    for(int k=1; k<5; k++)
        c += bs[k];

    //row number binary to deciaml conversion
    row = binaryToDecimal(r);

    //column number binary to decimal conversion
    column = binaryToDecimal(c);

    int x; //x is result variable

    switch(sbox){
        case 1:
            x = sBox1[row][column];
```

```
            break;
        case 2:
            x = sBox2[row][column];
            break;
        case 3:
            x = sBox3[row][column];
            break;
        case 4:
            x = sBox4[row][column];
            break;
        case 5:
            x = sBox5[row][column];
            break;
        case 6:
            x = sBox6[row][column];
            break;
        case 7:
            x = sBox7[row][column];
            break;
        case 8:
            x = sBox8[row][column];
            break;
    }
    return x;
}


string decimalToBinary_4_bit(int n){
    if(n == 0) return "0000";
    string as = "";
    while(n>0){
```

```
        as += n % 2 + '0';
        n /= 2;
    }
    int len = as.size();
    for(int i=0; i<4-len; i++)
        as += "0";
    reverse(as.begin(), as.end());
    return as;
}


string straightP(string as){
    int k = 0;
    string bs = "";
    for(int i=0; i<32; i++)
        bs += "0";
    for(int i=0; i<32; i++){
        bs[i] = as[straightPermutation[k]-1];
        k++;
    }
    return bs;
}


string fiPermutation(string as){
    int k = 1;
    string bs = "";
    for(int i=0; i<64; i++)
        bs += "0";


    for(int i=0; i<as.size(); i++){
```

```
        bs[FP[k]-1] = as[i];
        k++;
    }
    return bs;
}


void clear(){
    leftVal.clear();
    rightVal.clear();
    keyVal.clear();
}


int main()
{
    string key, plainText;
    preCalc();
    freopen("input.txt", "r", stdin);
    //freopen("output.txt", "w", stdout);
    while(cin>>plainText>>key){
        //print the plainText and key
        cout<<"Plain Text: "<<plainText<<endl;
        cout<<"key: "<<key<<endl;


        //convert Hex plain Text into binary
        plainText = hexToBinary(plainText);


        //initial permutation
        plainText = initPermutation(plainText);
```

```
string plainText_left, plainText_right;
int flag = 0;
for(int i=0; i<plainText.size(); i++){
    if(i == 32) flag = 1;
    if(flag == 0) plainText_left += plainText[i];
    else plainText_right += plainText[i];
}

//Parity Drop
key = hexToBinary(key);
key = parityBitDropPermutation(key);

int Round = 1; //initialize the round number
string original_key, original_plaintext_right;
original_plaintext_right = plainText_right;

while(true) {
    if(Round>16)
        break;
    if(Round > 1) {
        key = original_key;
        plainText_right = plainText_left;
        plainText_left = original_plaintext_right;
        original_plaintext_right = plainText_right;
    }
```

```
//pushing left and right plaintext
if(Round>1){
    leftVal.push_back(binaryToHex(plainText_left));
    rightVal.push_back(binaryToHex(plainText_right));
}


//divide the key 28 bit parts
string _28_key_left, _28_key_right;
int flag = 0;
for(int i=0; i<key.size(); i++) {
    if(i == 28) flag = 1;
    if(flag == 0) _28_key_left += key[i];
    else _28_key_right += key[i];
}



//Circular Left shift
if(Round == 1 || Round == 2 ||Round == 9 ||Round == 16) {
    _28_key_left = circularLeftShift(1, _28_key_left);
    _28_key_right = circularLeftShift(1, _28_key_right);
} else {
    _28_key_left = circularLeftShift(2, _28_key_left);
    _28_key_right = circularLeftShift(2, _28_key_right);
}



//combine the keys;
```

```
key = combineBit(_28_key_left, _28_key_right);
original_key = key;


//Compression Permutation
key = compressionPermutation(key);


//Expansion P-box or make right plain text into 32 to 48 bits
plainText_right = ExpansionPBT(plainText_right);


//Whitener or XOR operation
plainText_right = xorOperation(plainText_right, key);


//S - box
string bs = "", ret = "";
int sbox = 1;
for(int i=0; i<plainText_right.size(); i++){
    if((i+1)%6 == 0){
        bs += plainText_right[i];
        int x = sboxPermutation(sbox, bs);
        ret += decimalToBinary_4_bit(x);
        sbox++;
        bs.clear();
    } else
        bs += plainText_right[i];
}
plainText_right = ret;
```

```
            //Straight permutaion
            plainText_right = straightP(plainText_right);

            //Again palintext_left xor plaintext_right
            plainText_left = xorOperation(plainText_left, plainText_right);

            //pushing key
            keyVal.push_back(binaryToHex(key));

            //Increment Round
            Round++;
        }
        leftVal.push_back(binaryToHex(plainText_left));
        rightVal.push_back(binaryToHex(original_plaintext_right));

        //combine left and right
        string cipherText = combineBit(plainText_left,
original_plaintext_right);

        //Do the final permutation
        cipherText = fiPermutation(cipherText);

        //Printing result
        puts("------------------------------------------------------------");
        printf("%s%16s%16s%21s\n","Round","Left", "Right", "Round key");
        puts("------------------------------------------------------------");
        for(int i=0; i<16; i++)
```

```
            printf("%s%.2d%15s%15s%20s\n","Round: ", i+1, leftVal[i].c_str(),
rightVal[i].c_str(), keyVal[i].c_str());
        puts("------------------------------------------------------------");



        //printing the Ciphertext in Hex
        cout<<"Cipher Text: "<<binaryToHex(cipherText)<<endl;



        //clear
        clear();
    }
    return 0;
}
```