DES 算法设计

学号：15331087

姓名：高俊杰

一、DES算法原理概述

DES算法是一种对称加密算法，这种特性使得，利用明文和密钥，我们加密一次能得到密文，然后利用密文和密钥再次经过DES加密，得到原文。

简要介绍对称加密算法。一个对称加密由明文(原始信息或数据，作为算法的输入)、 加密算法(对明文进行各种替换和转换)、 密钥(算法的另一个输入，决定算法进行的具体替换和转换)、 密文(已被打乱的消息输出)、 解密算法(加密算法的反向执行)5部分组成。

算法首先需要两样数据，一个是64位bits的明文Plain_Text，另一个是64位bits的密钥Key_64。在选定密钥的情况下，我们就能通过DES算法加密明文了。下面详细介绍DES的加密步骤。

1.第一步：IP置换。我们已有一个公开确定的IP矩阵，它是一个大小为64，里面的数据分别是1到64的矩阵，用来初步混淆明文。原理是让明文的每一位对应上IP矩阵的每一位，取其中的数据作为新的下标，重新排列得到一串64位长的数据M0。这里提醒，整个算法过程用到的矩阵的里面存的下标都是从一开始的，并不是我们熟悉的从0开始。

2.第二步：T迭代。这是整套算法里最关键的部分，里面会有多种处理手法。首先将上一步得到的M0断开成两个32位长的串L和R。然后将L和R经过16轮的迭代。其中每轮的L会重新赋值为上一轮的R，而R则会赋值为上一轮的L 异或上 Feistel函数返回值的值。

3.接下来顺势详细谈谈Feistel函数。它接收两个参数，一个是上一轮迭代而来的32位长的R，一个是后面会提到的48位长的子密钥Key_Sub。先要扩展32位的R到48位的E_R，用于与Key_Sub异或得到一个命名为result的48位长变量。然后result每连续6位就经过一次S_Box映射，每次都从6位压缩成4位。这样的映射会循环8次，那么最后映射结果连起来，又得到一个32位长的串F_S。将F_S经过一次P置换得到F_P。最后它会返回32位长的F_P用于赋值给下一轮的R。

4.上面提到的扩展32位的R到48位的E_R，用的是大小为48的矩阵E，里面存着对应R的下标，类似前面的置换过程。这也是一个公开确定的矩阵。

5.上面提到的S_Box映射，用的是8个给定的大小为4*16的矩阵S_Box，里面存的每一个元素都是能用4位二进制表示的数。这是一系列公开确定的矩阵，这也是整个算法里最让人摸不清的地方。但是我们先这样用吧。我们将上面的result每连续6位取出，分别表示为b1b2b3b4b5b6，然后令行号row=b1b6，列号col=b2b3b4b5，这样刚好能从一个S_Box中确定出一个4位长的数据。这样的过程会循环8次，分别从不同S_Box中得到一个4位数据，连接起来就得到了32位长的F_S。

6.上面提到的P置换，用的是大小为32的矩阵P，同样存着新的下标，同样类似前面的置换过程。这也是一个公开确定的矩阵。

7.T迭代的最后，我们再将最终的32位的L和R换个位置，存进一个64位长的变量RL里并返回RL，才结束这一步。

8.第三步，也就是加密过程的最后一步，IP逆置换。同样，我们给定了IP的逆矩阵，然后模仿第一步再做一次一样的映射，我们就得到密文Cipher_Text了。这一步看似简陋不必要，但其实是为了维护DES算法的对称性。只有保持了对称性，我们才能利用密文和密钥经过又一轮加密从而解密得到原文。

9.然后就要谈谈16个48位长的Key_Sub的产生了。64位密钥的先要经过一轮PC_1置换，得到56位的Key_56，然后将其分成左右各28位的C和D。对C和D按规则进行16轮的循环左移，每次得到的结果拼接回Key_56，然后进行PC_2置换得到48位的串，即一个Key_Sub。这16个Key_Sub在加密过程中分别用于上述的16次的T迭代，而在解密过程中，顺序需要反过来才能正确解密。

10.上面提到的PC_1置换和PC_2置换也是类似上面的映射，只不过PC_1置换是64位压缩为56位，PC_2置换是56位再映射成48位。PC_1矩阵和PC_2矩阵也已给出。

二、总体结构

下面开始介绍我对这个算法的实现。实现用的是C++，原因有二。一是我打算用面向对象将DES算法封装成一个类，用一个密钥创建一个DES对象，并只对外提供加密和解密两个函数接口。其余的实现细节（如用到的矩阵、T迭代、Feistel函数和子密钥生成等）对用户隐藏，较好地保证安全性。二是C++有bitset这个标准库，能直观地操作一个个bit，比C语言方便实现。

类DES是这次需要实现的类，我需要实现的功能很简单，给定一串64位的密钥，用密钥创建一个DES对象，提供64位的明文串，加密返回一串64位的密文，然后解密密文，使得解密结果和原文完全一致。

三、模块分解

类DES的声明放在文件"DES.h"中，并包含了静态常量的初始化。

实现的函数则放在文件"DES.cpp"中，测试用的主函数在"main.cpp"中。

首先DES的构造函数需要传入64位的bitset类型数据，赋值给私有变量Key_64，即我们后来用到的密钥。

然后DES的加密函数Encrypt和解密函数Decrypt也需要传入64位的bitset类型数据，这是分别我们需要加密的明文和需要解密的密文。

IP_Substitute函数是加密第一步的IP置换。传入的参数是明文，返回初步混淆后的64位密文。

IP_Inv_Substitute函数是加密最后一步的IP逆置换。传入T迭代后的密文，返回最终的64位密文。

T_Iteration函数是加密第一步的T迭代，传入第一步IP置换后的密文以及子密钥的引用，返回迭代后的64位密文。

Feistel函数是T迭代里用到的Feistel函数，用于将密文和密钥混合加密。

E_Expand函数是将32位的R扩展成48位的E_R。

P_Substitute函数是将32位的F_S映射为32位的F_P。

Key_64_To_56和Key_56_To_48以及Key_28_Shift_Left都是对上面所说的密钥处理拆分的过程，而函数Key_Gen调用这些函数从而得到16个48位的子密钥。

Key_Inv函数反转子密钥序列，用在解密函数当中。

四、数据结构

bitset是主要用到的数据结构，用来表示给定位数的bit串。过程中用到的矩阵如S_BOX，IP, PC_1等，用数组储存，作为DES类的静态私有常量。另外在测试用的主函数里，用到string类型的字符串来初始化bitset类型数据如明文和密钥。

类DES中我尽量不留变量，这样的话只靠函数计算并返回数值就比较安全，不容易泄露对象内的信息。但是初始化所需的密钥不能没有，所以用一个64位的bitset存着。另外子密钥序列的生成和解密所需的反转顺序的子密钥序列都能通过函数参数传引用的方法而避免用私有变量保存。故只需有一个私有变量即可。

五、C++算法过程

DES.h

```
#include <bitset>

using namespace std;

class DES {
public:
```

```
    DES(bitset<64> Key);

    bitset<64> Encrypt(bitset<64> Text);

    bitset<64> Decrypt(bitset<64> Text);

private:
    static int IP[];
    static int IP_Inv[];
    static int E[];
    static int S_BOX[8][4][16];
    static int P[];
    static int PC_1[];
    static int PC_2[];

    bitset<64> IP_Substitute(bitset<64> M);

    bitset<64> IP_Inv_Substitute(bitset<64> RL);

    bitset<32> Feistel(bitset<32> R, bitset<48> ki);

    bitset<48> E_Expand(bitset<32> R);

    bitset<32> P_Substitute(bitset<32> F_S);

    bitset<64> T_Iteration(bitset<64> M0, bitset<48>* &Key_Sub);

    bitset<56> Key_64_To_56(bitset<64> Key);

    bitset<28> Key_28_Shift_Left(bitset<28> Key_56_Half, int num);

    bitset<48> Key_56_To_48(bitset<56> Key_56);

    void Key_Gen(bitset<64> Key_64, bitset<48>* &Key_Sub);

    void Key_Inv(bitset<48>* &Key_Sub);

    bitset<64> Key_64;
};


int DES::IP[] = {
        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 DES::IP_Inv[] = {
        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
};

int DES::E[] = {
        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 DES::S_BOX[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}
    }
};

int DES::P[] = {
        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 DES::PC_1[] = {
        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 DES::PC_2[] = {
        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
};
```

DES.cpp

```
#include <iostream>
#include <bitset>
#include "DES.h"

using namespace std;

DES::DES(bitset<64> Key) {
    Key_64 = Key;
}

bitset<64> DES::Encrypt(bitset<64> Text) {
    bitset<64> M0 = IP_Substitute(Text);
    bitset<48> * Key_Sub = new bitset<48>[16];
    Key_Gen(Key_64, Key_Sub);
    bitset<64> RL = T_Iteration(M0, Key_Sub);
    bitset<64> C = IP_Inv_Substitute(RL);
    return C;
}

bitset<64> DES::Decrypt(bitset<64> Text) {
    bitset<64> C0 = IP_Substitute(Text);
    bitset<48> * Key_Sub = new bitset<48>[16];
    Key_Gen(Key_64, Key_Sub);
    Key_Inv(Key_Sub);
    bitset<64> RL = T_Iteration(C0, Key_Sub);
    bitset<64> M = IP_Inv_Substitute(RL);
    return M;
}

void DES::Key_Gen(bitset<64> Key_64, bitset<48>* &Key_Sub) {
    bitset<56> Key_56 = Key_64_To_56(Key_64);
    bitset<28> C, D;
    for (int i = 0; i < 28; i++) {
        C[i] = Key_56[i];
        D[i] = Key_56[i + 28];
    }
    for (int i = 0; i < 16; i++) {
        if (i == 0 || i == 1 || i == 8 || i == 15) {
            C = Key_28_Shift_Left(C, 1);
            D = Key_28_Shift_Left(D, 1);
        } else {
            C = Key_28_Shift_Left(C, 2);
            D = Key_28_Shift_Left(D, 2);
        }
        for (int j = 0; j < 28; j++) {
            Key_56[j] = C[j];
            Key_56[j + 28] = D[j];
        }
        Key_Sub[i] = Key_56_To_48(Key_56);
        cout << "Key_Sub: " << Key_Sub[i] << endl;
    }
}

void DES::Key_Inv(bitset<48>* &Key_Sub) {
    for (int i = 0; i < 8; i++) {
```

```
        bitset<48> tmp = Key_Sub[i];
        Key_Sub[i] = Key_Sub[15 - i];
        Key_Sub[15 - i] = tmp;
    }
}

bitset<64> DES::IP_Substitute(bitset<64> M) {
    cout << "IP_Substitute  M: " << M << endl;
    bitset<64> M0;
    for (int i = 0; i < 64; i++) {
        M0[i] = M[IP[i] - 1];
    }
    cout << "IP_Substitute M0: " << M0 << endl;
    return M0;
}

bitset<32> DES::Feistel(bitset<32> R, bitset<48> ki) {
    bitset<48> E_R = E_Expand(R);
    bitset<48> result = E_R ^ ki;
    cout << "Feistel result: " << result << endl;

    bitset<32> F_S;
    for (int i = 0; i < 8; i++) {
        int row = result[6 * i] * 2 + result[6 * i + 5];
        int col = result[6 * i + 1] * 8 + result[6 * i + 2] * 4 + result[6 * i + 3] * 2 + result[6 * i + 4];
        bitset<4> tmp(S_BOX[i][row][col]);
        for (int k = 0; k < 4; k++) {
            F_S[i * 4 + k] = tmp[k];
        }
    }

    cout << "Feistel    F_S: " << F_S << endl;
    bitset<32> F_P = P_Substitute(F_S);
    cout << "Feistel    F_P: " << F_P << endl;
    return F_P;
}

bitset<48> DES::E_Expand(bitset<32> R) {
    bitset<48> E_R;
    for (int i = 0; i < 48; i++) {
        E_R[i] = R[E[i] - 1];
    }
    return E_R;
}

bitset<32> DES::P_Substitute(bitset<32> F_S) {
    bitset<32> F_P;
    for(int i = 0; i < 32; i++) {
        F_P[i] = F_S[P[i] - 1];
    }
    return F_P;
}

bitset<64> DES::T_Iteration(bitset<64> M0, bitset<48>* &Key_Sub) {
    bitset<32> L, R;
    for (int i = 0; i < 32; i++) {
        L[i] = M0[i];
        R[i] = M0[i + 32];
    }
    for (int i = 0; i < 16; i++) {
        bitset<32> L_Next(R);
        R = L ^ Feistel(R, Key_Sub[i]);
        L = L_Next;
        cout << "L: " << L << " R: " << R << endl;
    }
    bitset<64> RL;
    for (int i = 0; i < 32; i++) {
        RL[i] = R[i];
        RL[i + 32] = L[i];
    }
```

```
    cout << "T_Iteration RL: " << RL << endl;
    return RL;
}

bitset<64> DES::IP_Inv_Substitute(bitset<64> RL) {
    cout << "IP_Inv_Substitute RL: " << RL << endl;
    bitset<64> C;
    for (int i = 0; i < 64; i++) {
        C[i] = RL[IP_Inv[i] - 1];
    }
    cout << "IP_Inv_Substitute  C: " << C << endl;
    return C;
}

bitset<56> DES::Key_64_To_56(bitset<64> Key_64) {
    bitset<56> Key_56;
    for (int i = 0; i < 56; i++) {
        Key_56[i] = Key_64[PC_1[i] - 1];
    }
    cout << "Key_64_To_56  Key_56: " << Key_56 << endl;
    return Key_56;
}

bitset<28> DES::Key_28_Shift_Left(bitset<28> Key_56_Half, int num) {
    bitset<28> Key_28;
    Key_28 = (Key_56_Half << num) | (Key_56_Half >> (28 - num));
    cout << "Key_28_Shift_Left  Key_28: " << Key_28 << endl;
    return Key_28;
}

bitset<48> DES::Key_56_To_48(bitset<56> Key_56) {
    bitset<48> Key_48;
    for (int i = 0; i < 48; i++) {
        Key_48[i] = Key_56[PC_2[i] - 1];
    }
    cout << "Key_56_To_48  Key_48: " << Key_48 << endl;
    return Key_48;
}
```

main.cpp

```
#include <iostream>
#include "DES.cpp"

using namespace std;

int main() {
    string Text_Str = "0101000101010111010001010101001001010100010110010101010101001001";
    bitset<64> text(Text_Str);
    cout << "Text: " << text << endl;
    string Key_Str = "0101101001011000010000110101011001000010010011100100110101001100";
    bitset<64> key(Key_Str);
    cout << " Key: " << key << endl;
    DES des(key);
    bitset<64> C = des.Encrypt(text);
    cout << "Cipher_Text: " << C << endl;
    bitset<64> M = des.Decrypt(C);
    cout << "Plain_Text: " << M << endl;
    cout << "Original Text = Decoded Text? " << ((text == M)?"Yes.": "No.") << endl;
}
```

六、算法运行情况

下图是运行的情况的截图以及打印的文本.

可以看到最下面两行的显示是"Plain_Text: 0101000101010111010001010101001001010100010110010101010101001001 Original Text = Decoded Text? Yes."。这说明解密后的密文和原文匹配上了，说明DES算法成功实现了。

> Text: 0101000101010111010001010101001001010100010110010101010101001001 Key: 0101101001011000010000110101011001000010010011100100110101001100 IP_Substitute M: 0101000101010111010001010101001001010100010110010101010101001001 IP_Substitute M0: 1111111101111011010101101110011001110000000000000000010100000000001010 Key_64_To_56 Key_56: 1110000010110000000001111111100111110100000111101010000100 Key_Sub: 1110010110100100011111000101001111011101000010001 Key_Sub: 1111000110001100010111000101010100000001101011101 Key_Sub:

10110001011001001100111011011010000000110011110 Key_Sub:
10101001010001011000101111100001101000011101001O Key_Sub:
00101011110100001000100110100000101001111110011O Key_Sub:
00101100110100101110100111001010101101100010101O Key_Sub:
11001100010110101101000100001011101111110000100O Key_Sub:
10011101001110101001010100010011010111101000011O Key_Sub:
10011100001111110001000101011001010010001001110O Key_Sub:
00010111001110110001101011100001010000011101110l Key_Sub:
00110110011010110011101011000101100000011110011l Key_Sub:
01010010111010010011001111001100101000111010101O Key_Sub:
01010010101000011110011110101011101001110000001O Key_Sub:
11001010100001110110011000100010101111101000011O Key_Sub:
11000011100101100111010011000010011111100000110O Key_Sub:
11100101100011100110010001101001010100000100110l L: 11111111011110110101011011100111 R:
00000010011001001100000110011001 L: 00000010011001001100000110011001 R:
01100100110101010100010000110001 L: 01100100110101010100010000110001 R:
00111011011100100001111010100011 L: 00111011011100100001111010100011 R:
00110011001011101000110111010000 L: 00110011001011101000110111010000 R:
00000111001010101010100111110111 L: 00000111001010101010100111110111 R:
00101011101111111000011100101011 L: 00101011101111111000011100101011 R:
00001101110001011011101010010011 L: 00001101110001011011101010010011 R:
00101011101110110100100110010101 L: 00101011101110110100100110010101 R:
00101001000111011000110110010110 L: 00101001000111011000110110010110 R:
01010001110011011011000100101010 L: 01010001110011011011000100101010 R:
01001111001010100110111001100000 L: 01001111001010100110111001100000 R:
00110111100010111000000100100111 L: 00110111100010111000000100100111 R:
00010101100011110001110110011110 L: 00010101100011110001110110011110 R:
01111000100111101101101001111001 L: 01111000100111101101101001111001 R:
01000111000111110000011100010100 L: 01000111000111110000011100010100 R:
01010001110111001110010010001100 T_Iteration RL:
01000111000111110000011100010100010100011101110011100100100011100 IP_Inv_Substitute RL:
01000111000111110000011100010100010100011101110011100100100011100 IP_Inv_Substitute C:
11010100010101000111111100110010101100010000100011101000000101010 Cipher_Text:
11010100010101000111111100110010101100010000100011101000000101010 IP_Substitute M:
11010100010101000111111100110010101100010000100011101000000101010 IP_Substitute M0:
01000111000111110000011100010100010100011101110011100100100011100 Key_64_To_56 Key_56:
11100000101100000000011111111001111101000001111010100010O Key_Sub:
11100101101001000111110001010011110111010000100l Key_Sub:
11110001100011000101110001010101000000011010110l Key_Sub:
10110001011001001100111011011010000000110011110 Key_Sub:
10101001010001011000101111100001101000011101001O Key_Sub:
00101011110100001000100110100000101001111110011O Key_Sub:
00101100110100101110100111001010101101100010101O Key_Sub:
11001100010110101101000100001011101111110000100O Key_Sub:
10011101001110101001010100010011010111101000011O Key_Sub:
10011100001111110001000101011001010010001001110O Key_Sub:
00010111001110110001101011100001010000011101110l Key_Sub:
00110110011010110011101011000101100000011110011l Key_Sub:
01010010111010010011001111001100101000111010101O Key_Sub:
01010010101000011110011110101011101001110000001O Key_Sub:
11001010100001110110011000100010101111101000011O Key_Sub:
11000011100101100111010011000010011111100000110O Key_Sub:
11100101100011100110010001101001010100000100110l L: 01000111000111110000011100010100 R:

01111000100111101101101001111001 L: 01111000100111101101101001111001 R:
00010101100011110001110110011110 L: 00010101100011110001110110011110 R:
00110111100010111000000100100111 L: 00110111100010111000000100100111 R:
01001111001010100110111001100000 L: 01001111001010100110111001100000 R:
01010001110011011011000100101010 L: 01010001110011011011000100101010 R:
00101001000111011000110110010110 L: 00101001000111011000110110010110 R:
00101011101110110100100110010101 L: 00101011101110110100100110010101 R:
00001101110001011011101010010011 L: 00001101110001011011101010010011 R:
00101011101111111000011100101011 L: 00101011101111111000011100101011 R:
00000111001010101010100111110111 L: 00000111001010101010100111110111 R:
00110011001011101000110111010000 L: 00110011001011101000110111010000 R:
00111011011100100001111010100011 L: 00111011011100100001111010100011 R:
01100100110101010100010000110001 L: 01100100110101010100010000110001 R:
00000010011001001100000110011001 L: 00000010011001001100000110011001 R:
11111111011110110101011011100111 L: 11111111011110110101011011100111 R:
00000000000000001010000000001010 T_Iteration RL:
11111111011110110101011011100111000000000000000001010000000001010 IP_Inv_Substitute RL:
11111111011110110101011011100111000000000000000001010000000001010 IP_Inv_Substitute C:
01010001010101110100010101010010010101000101100101010101010010010 Plain_Text:
01010001010101110100010101010010010101000101100101010101010010010 Original Text = Decoded Text? Yes.