个人信息

学号：1911410

姓名：付文轩

专业：信息安全

实验步骤及实验要求

实验步骤

1. 算法分析

   请参照教材内容，分析MD5算法实现的每一步原理。

2. 算法实现：
   利用Visual C + +语言，自己编写MD5的实现代码，并检验代码实现的正确性。

3. 雪崩效应检验：
   尝试对一个长字符串进行Hash运算，并获得其运算结果。对该字符串进行轻微的改动，比如增加一个空格或标点，比较Hash结果值的改变位数。进行8次这样的测试。

实验要求

1. 自己编写完整的MD5实现代码，并提交程序和程序流程图。
2. 对编好的MD5算法，测试其雪崩效应，要求给出文本改变前和改变后的Hash值，并计算出改变的位数。写出8次测试的结果，并计算出平均改变的位数。

MD5算法

消息填充

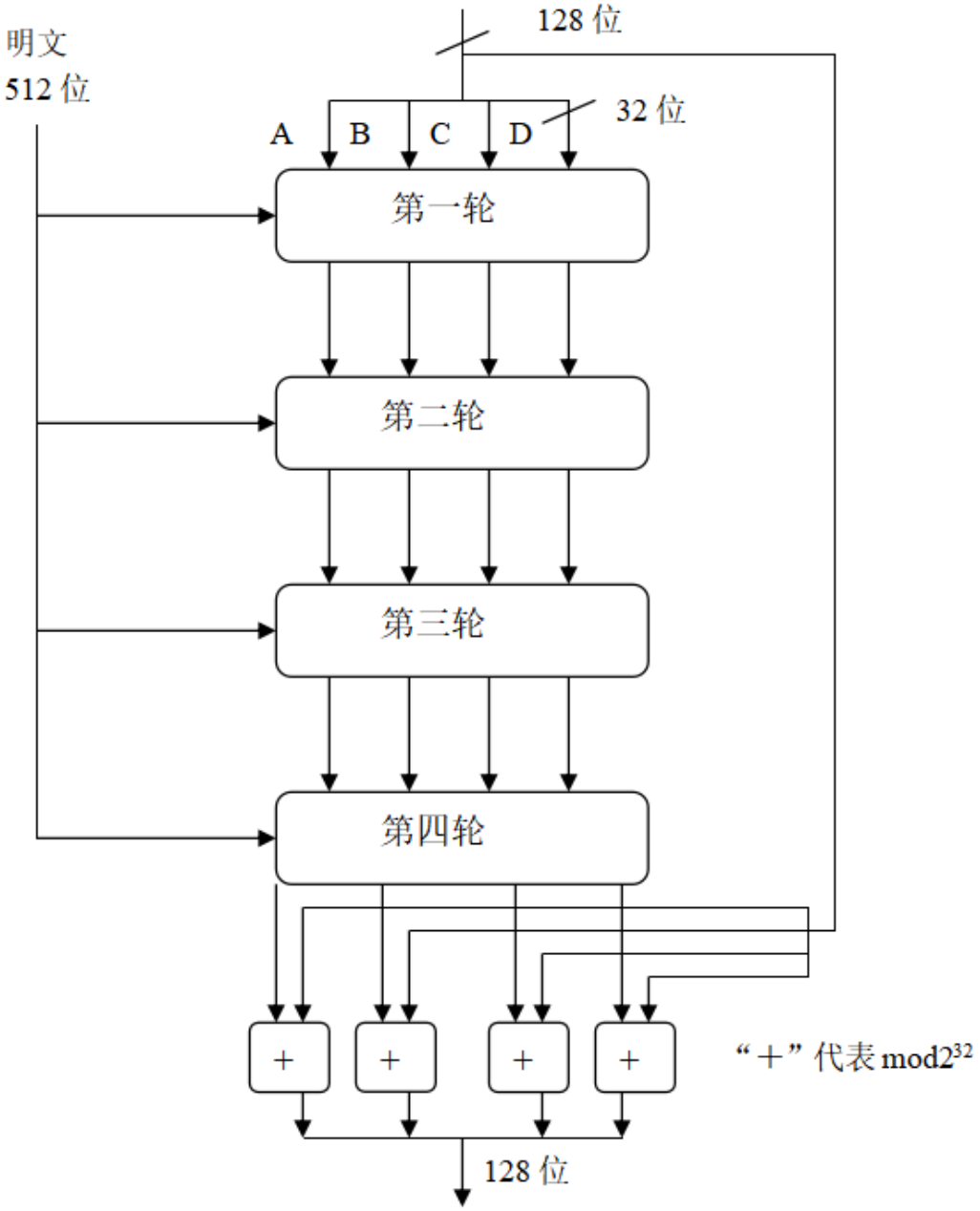
首先需要对信息进行填充，使其字节长度与448模512同余，即信息的字节长度扩展至 $n*512+448$，n为一个正整数。填充的方法如下：在信息的后面填充第一位为1，其余各位均为0，直到满足上面的条件时才停止用0对信息的填充。然后，再在这个结果后面附加一个以64位二进制表示的填充前信息长度。经过这两步的处理，现在的信息字节长度为 $n*512+448+64=(n+1)*512$，即长度恰好是512的整数倍，这样做的目的是为了满足后面处理中对信息长度的要求。

对应代码为：

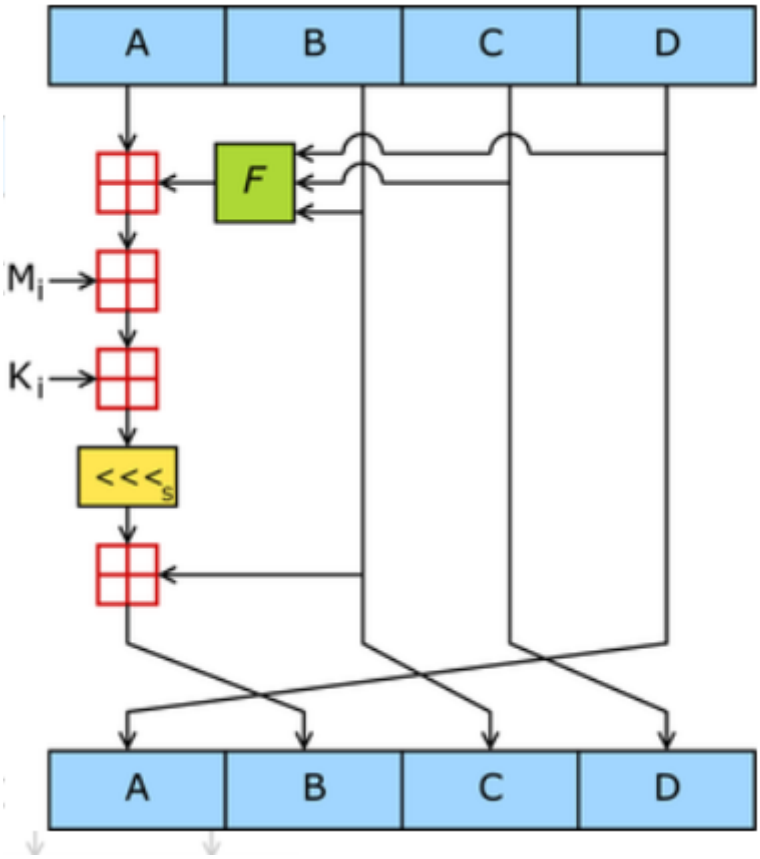
```
// 进行填充
messageByte[lengthInByte >> 2] |= 0x80 << ((lengthInByte % 4) * 8);

// 附加信息
messageByte[groupNum * 16 - 2] = lengthInByte * 8;
```

计算过程



MD5的计算主要分为4轮，其中每一轮都会进行16次操作，也就是说总共会进行4*16次的操作。每次操作对A、B、C、D中的3个做一次非线性函数运算，然后将所得结果加上第四个变量，文本的一个子分组（32位）和一个常数。再将所得结果向左循环移S位，并加上A、B、C、D其中之一。最后用该结果取代A、B、C、D其中之一。其中A、B、C、D的初始值为：A_0 = 0x01234567，B_0 = 0x89abcdef,，C_0=0xfedcba98，D_0=0x76543210



F：一个非线性函数，一个函数运算一次

Mi：表示一个 32-bits 的输入数据

Ki：表示一个 32-bits 常数，用来完成每次不同的计算

逻辑函数

```
//基本逻辑函数
unsigned int F(unsigned int b, unsigned int c, unsigned int d) {
    return ((b & c) | ((~b) & (d)));
}
unsigned int G(unsigned int b, unsigned int c, unsigned int d) {
    return ((b & d) | (c & (~d)));
}
unsigned int H(unsigned int b, unsigned int c, unsigned int d) {
    return (b ^ c ^ d);
}
unsigned int I(unsigned int b, unsigned int  c, unsigned int d) {
    return (c ^ (b | (~d)));
}

//x循环左移n位
unsigned int shift(unsigned int x, unsigned int n) {
    return ((x << n) | (x >> (32 - n)));
}
```

算法主要实现代码

```
/// <summary>
```

```
/// 计算MD5的值
/// </summary>
/// <param name="message">任意长度的待处理消息</param>
/// <returns>128比特MD5字符串</returns>
string myGetMD5(string message) {
    //初始化,, 这里要按照大端字节序的方式进行存储
    unsigned int A = 0x67452301;
    unsigned int B = 0xefcdab89;
    unsigned int C = 0x98badcfe;
    unsigned int D = 0x10325476;

    // 得到消息的长度，单位：字节
    int lengthInByte = message.length();
    // 计算分组数量，同时需要注意：如果一开始就刚好是message=448(mod 512)也需要进行10000填充，所以算出来的直接+1
    int groupNum = ((lengthInByte + 8) / 64) + 1;

    // messageByte数组中的每一个元素，对应消息二进制表示中的32bit（4字节）
    // 每个分组的长度为 16 * 32bit = 512bit
    unsigned int* messageByte = new unsigned int[groupNum * 16];

    // 初始化
    memset(messageByte, 0, sizeof(unsigned int) * groupNum * 16);

    //将string类型变量 保存进 messageByte数组中
    for (int i = 0; i < lengthInByte; i++)
    {
        //每个字中对应存储4个字节的数据
        messageByte[i / 4] |= message[i] << ((i % 4) * 8);
    }

    // 进行填充
    messageByte[lengthInByte >> 2] |= 0x80 << ((lengthInByte % 4) * 8);

    // 附加信息
    messageByte[groupNum * 16 - 2] = lengthInByte * 8;

    // 临时变量，用来保存每轮中每一步计算之后的abcd
    unsigned int a, b, c, d;

    // 循环处理每个分组
    for (int i = 0; i < groupNum; i++)
    {
        a = A;
        b = B;
        c = C;
        d = D;
        unsigned int g;
        int k;

        // 对每个分组进行64轮压缩
        for (int j = 0; j < 64; j++)
        {
            // 根据轮数不同，使用不同的压缩函数
            if (j < 16)
            {
                // 对B C D进行基本逻辑函数运算
                g = F(b, c, d);
```

```
                // 计算k，以从消息分组中取出对应的字
                k = j;
            }
            else if (j >= 16 && j < 32)
            {
                // 对B C D进行基本逻辑函数运算
                g = G(b, c, d);
                // 计算k，以从消息分组中取出对应的字
                k = (1 + 5 * j) % 16;
            }
            else if (j >= 32 && j < 48)
            {
                // 对B C D进行基本逻辑函数运算
                g = H(b, c, d);
                // 计算k，以从消息分组中取出对应的字
                k = (5 + 3 * j) % 16;
            }
            else if (j >= 48 && j < 64)
            {
                // 对B C D进行基本逻辑函数运算
                g = I(b, c, d);
                // 计算k，以从消息分组中取出对应的字
                k = (7 * j) % 16;
            }

            unsigned tempd = d;
            d = c;
            c = b;
            b = b + shift(a + g + messageByte[i * 16 + k] + T[j], s[j]);
            a = tempd;
        }
        // 更新值
        A = a + A;
        B = b + B;
        C = c + C;
        D = d + D;
    }
    //返回16进制形式字符串
    return Int2HexString(A) + Int2HexString(B) + Int2HexString(C) +
Int2HexString(D);
}
```

实验计算结果

输入一个随机的字符串：`1a65dsg489121asdg`

得到结果为：

```
======================MD5计算程序======================
请输入一个字符串(为了进行雪崩效应的测试，请输入的字符串长度>=8)：
1a65dsg489121asdg
经过计算可以得到MD5值为：
07155a6fa59176c2a1a75581a0590dd8
```

使用工具计算得到：

验证得到实验结果正确

雪崩效应测试

计算位数不同的函数

```
/// <summary>
/// 雪崩效应测试时计算不同位数的函数
/// </summary>
/// <param name="NewSumry">改变过后的字符串</param>
/// <param name="OrigiSumry">原始字符串</param>
/// <returns>不同的位数</returns>
int getCmpVal(string newMd5, string originMd5) {
    int indexOfTemp = 0;
    bitset<128> newmd5;
    // 使用最笨的办法，switch case把16进制的字符表示转换成2进制的结果
    for (; indexOfTemp < 32; indexOfTemp++) {
        int startOfEdit = 127 - 4 * indexOfTemp;
        switch (newMd5[indexOfTemp]) {
        case '0':
```

```
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 0;
            break;
        case '1':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 1;
            break;
        case '2':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case '3':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 1;
            break;
        case '4':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 0;
            break;
        case '5':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 1;
            break;
        case '6':
            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case '7':

            newmd5[startOfEdit] = 0;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 1;
            break;
        case '8':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 0;
            break;
        case '9':

            newmd5[startOfEdit] = 1;
```

```
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'A':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'B':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'C':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'D':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'E':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'F':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'a':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'b':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 0;
            newmd5[startOfEdit - 2] = 1;
```

```
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'c':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'd':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 0;
            newmd5[startOfEdit - 3] = 1;
            break;
        case 'e':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 0;
            break;
        case 'f':

            newmd5[startOfEdit] = 1;
            newmd5[startOfEdit - 1] = 1;
            newmd5[startOfEdit - 2] = 1;
            newmd5[startOfEdit - 3] = 1;
            break;
        }
    }

    indexOfTemp = 0;
    bitset<128> origin;
    // 使用最笨的办法，switch case把16进制的字符表示转换成2进制的结果
    for (; indexOfTemp < 32; indexOfTemp++) {
        int startOfEdit = 127 - 4 * indexOfTemp;
        switch (originMd5[indexOfTemp]) {
        case '0':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 0;
            break;
        case '1':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 1;
            break;
        case '2':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case '3':
```

```
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
        case '4':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 0;
            break;
        case '5':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 1;
            break;
        case '6':
            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case '7':

            origin[startOfEdit] = 0;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
        case '8':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 0;
            break;
        case '9':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 1;
            break;
        case 'A':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case 'B':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
```

```
        case 'C':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 0;
            break;
        case 'D':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 1;
            break;
        case 'E':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case 'F':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
        case 'a':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case 'b':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 0;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
        case 'c':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 0;
            break;
        case 'd':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 0;
            origin[startOfEdit - 3] = 1;
            break;
        case 'e':

```

```
            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 0;
            break;
        case 'f':

            origin[startOfEdit] = 1;
            origin[startOfEdit - 1] = 1;
            origin[startOfEdit - 2] = 1;
            origin[startOfEdit - 3] = 1;
            break;
        }
    }

    bitset<128> result;
    result = newmd5 ^ origin;

    return result.count();
}
```

测试雪崩效应的函数

这里我们在改变的时候是直接对原始字符串中的char进行数值上的+1，则通常情况下只会改变1位或者2位，共进行8轮测试，相关代码如下：

```
    for (int i = 0; i < 8; i++) {
        cout << "\n\n=========================第" << i + 1 << "轮雪崩效应测试=========================\n";
        cout << "字符串的第" << i + 1 << "个char的值+1，得到新的字符串为：\n";
        newM = m;
        newM[i] += 1;
        cout << newM << endl;
        // 计算新的md5
        string newMd5M = myGetMD5(newM);
        cout << "经过计算可以得到修改过后的字符串MD5值为：\n" << newMd5M << endl;

        changeBit[i] = getCmpVal(newMd5M, md5M);
        sumBit += changeBit[i];
        cout << "经过比较后发生变化的位数为：";
        cout << changeBit[i] << endl;
    }

    averageBit = sumBit / 8;
    cout << "\n\n经过计算得到平均改变的位数为：" << averageBit << endl;
```

实验结果

得到结果如下：

```
=========================第1轮雪崩效应测试=========================
字符串的第1个char的值+1，得到新的字符串为：
```

```
2a65dsg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
b08fa0001f0e349c338d73f18937033d
经过比较后发生变化的位数为：68


==========第2轮雪崩效应测试==========
字符串的第2个char的值+1，得到新的字符串为：
1b65dsg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
31996c4c30defa03f52cc59ef7378246
经过比较后发生变化的位数为：63


==========第3轮雪崩效应测试==========
字符串的第3个char的值+1，得到新的字符串为：
1a75dsg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
69745a92214692fb803e244676ff4d82
经过比较后发生变化的位数为：60


==========第4轮雪崩效应测试==========
字符串的第4个char的值+1，得到新的字符串为：
1a66dsg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
c3d123fdc560f2d65a90003a27233eae
经过比较后发生变化的位数为：65


==========第5轮雪崩效应测试==========
字符串的第5个char的值+1，得到新的字符串为：
1a65esg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
751af42e8f96fa2035b597d5f3bd9e5a
经过比较后发生变化的位数为：53


==========第6轮雪崩效应测试==========
字符串的第6个char的值+1，得到新的字符串为：
1a65dtg489121asdg
经过计算可以得到修改过后的字符串MD5值为：
be8e5364b5f8822453dc40aef331b59d
经过比较后发生变化的位数为：63


==========第7轮雪崩效应测试==========
字符串的第7个char的值+1，得到新的字符串为：
1a65dsh489121asdg
经过计算可以得到修改过后的字符串MD5值为：
1d8c28dfa85e3bdea94ca9bbe60aadca
经过比较后发生变化的位数为：58


==========第8轮雪崩效应测试==========
字符串的第8个char的值+1，得到新的字符串为：
1a65dsg589121asdg
经过计算可以得到修改过后的字符串MD5值为：
```

```
3bc4aa0ee22700d1c182ae5f8c2a91bc
经过比较后发生变化的位数为：65


经过计算得到平均改变的位数为：61
```