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. 为了加深对RSA算法的了解，根据已知参数：p = 3，q = 11，m = 2，手工计算公钥和私钥，并对明文m进行加密，然后对密文进行解密
2. 编写一个程序，用于生成512比特的素数。
3. 利用2中程序生成的素数，构建一个n的长度为1024比特的RSA算法，利用该算法实现对明文的加密和解密。
4. 在附件中还给出了一个可以进行RSA加密和解密的对话框程序RSATool，运行这个程序加密一段文字，了解RSA算法原理。

实验要求

1. 对实验步骤2，写出生成素数的原理，包括随机数的生成原理和素性检测的内容，并给出程序框图；
2. 对实验步骤3，要求分别实现加密和解密两个功能，并分别给出程序框图。

大整数类

由于C++对于大的整数无法支持运算，所以当整数大到一定程度时，仅适用C++自带的类型没有办法完成相关实验，所以需要自行构建一个大整数类。

这个大整数类的基础功能有：

- 大整数的存储
    - 位数
    - 数值
    - 符号
- 大整数的相关运算
    - 加减乘除
    - 乘方
    - 取模

基于所有的数都是按照位数存储的特点，我们可以使用**int数组**（数组中的每一个元素都对应着大整数的32位）来表示一个大的整数，数组顺序按照从左到右，对应着大整数的高位到低位。

而相关运算都以运算符重载的形式进行，这里仅展示部分运算符的重载，具体内容见`bigInt.h`和`bigInt.cpp`。

加法运算符重载

```
// 计算两个大数的和,采用竖式相加法
BigInt operator+ (const BigInt& a, const BigInt& b)
{
    BigInt result;
    //64位数据,存放每两位数相加的临时和
    unsigned __int64 sum;
    //carry为进位标志,sub为当两数符号相异时,存放每两位数相减的临时差
    unsigned int carry = 0, sub;
    //取a,b中长度较长的长度
    int length = (a.GetLength() >= b.GetLength() ? a.GetLength() :
b.GetLength());

    //当两数符号相同时,进行加法运算
    if (a.sign == b.sign)
    {
        //每一位进行竖式相加
        for (int i = 0; i < length; i++)
        {
            sum = (unsigned __int64)a.data[i] + b.data[i] + carry;
            result.data[i] = (unsigned int)sum;
            //sum的高位为进位
            carry = (sum >> 32);
        }

        result.sign = a.sign;
        return result;
    }
    else
    {
        //两数符号不同时,进行减法运算
        BigInt tempa, tempb;

        //取出a,b中绝对值较大的作为被减数
        if (a < b)
        {
            tempa = b;
            tempb = a;
        }
        else
        {
            tempa = a;
            tempb = b;
        }

        //每一位进行竖式减
        for (int i = 0; i < length; i++)
        {
            sub = tempb.data[i] + carry;
            if (tempa.data[i] >= sub)
            {
                result.data[i] = tempa.data[i] - sub;
                carry = 0;
            }
            else
            {
```

```
                //借位减
                result.data[i] = (unsigned __int64)tempa.data[i] + (1 << 32)
- sub;
                carry = 1;
            }
        }
        result.sign = tempa.sign;
        return result;
    }
}

```

乘法运算符重载

```
// 大数相乘,采用竖式乘
BigInt operator* (const BigInt& a, const BigInt& b)
{
    //last存放竖式上一行的积,temp存放当前行的积
    BigInt result, last, temp;
    //sum存放当前行带进位的积
    unsigned __int64 sum;
    //存放进位
    unsigned int carry;

    //进行竖式乘
    for (int i = 0; i < b.GetLength(); i++)
    {
        carry = 0;
        //B的每一位与A相乘
        for (int j = 0; j < a.GetLength() + 1; j++)
        {
            sum = ((unsigned __int64)a.data[j]) * (b.data[i]) + carry;
            if ((i + j) < thesize)
                temp.data[i + j] = (unsigned int)sum;
            carry = (sum >> 32);
        }
        result = (temp + last);
        last = result;
        temp.Clear();
    }

    //判断积的符号
    if (a.sign == b.sign)
        result.sign = true;
    else
        result.sign = false;

    return result;
}

```

除法运算符重载

```
// 大数除,采用试商除法,采用二分查找法优化
BigInt operator/ (const BigInt& a, const BigInt& b)
```

```
{
    //mul为当前试商,low,high为二分查找试商时所用的标志
    unsigned int mul, low, high;
    //sub为除数与当前试商的积,subsequent为除数与下一试商的积
    //dividend存放临时被除数
    BigInt dividend, quotient, sub, subsequent;
    int lengtha = a.GetLength(), lengthb = b.GetLength();

    //如果被除数小于除数,直接返回0
    if (a < b)
    {
        if (a.sign == b.sign)
            quotient.sign = true;
        else
            quotient.sign = false;
        return quotient;
    }

    //把被除数按除数的长度从高位截位
    int i;
    for (i = 0; i < lengthb; i++)
        dividend.data[i] = a.data[lengtha - lengthb + i];

    for (i = lengtha - lengthb; i >= 0; i--)
    {
        //如果被除数小于除数,再往后补位
        if (dividend < b)
        {
            for (int j = lengthb; j > 0; j--)
                dividend.data[j] = dividend.data[j - 1];
            dividend.data[0] = a.data[i - 1];
            continue;
        }

        low = 0;
        high = 0xffffffff;

        //二分查找法查找试商
        while (low < high)
        {
            mul = (((unsigned __int64)high) + low) / 2;
            sub = (b * mul);
            subsequent = (b * (mul + 1));

            if (((sub < dividend) && (subsequent > dividend)) || (sub ==
dividend))
                break;
            if (subsequent == dividend)
            {
                mul++;
                sub = subsequent;
                break;
            }
            if ((sub < dividend) && (subsequent < dividend))
            {
                low = mul;
                continue;
            }
```

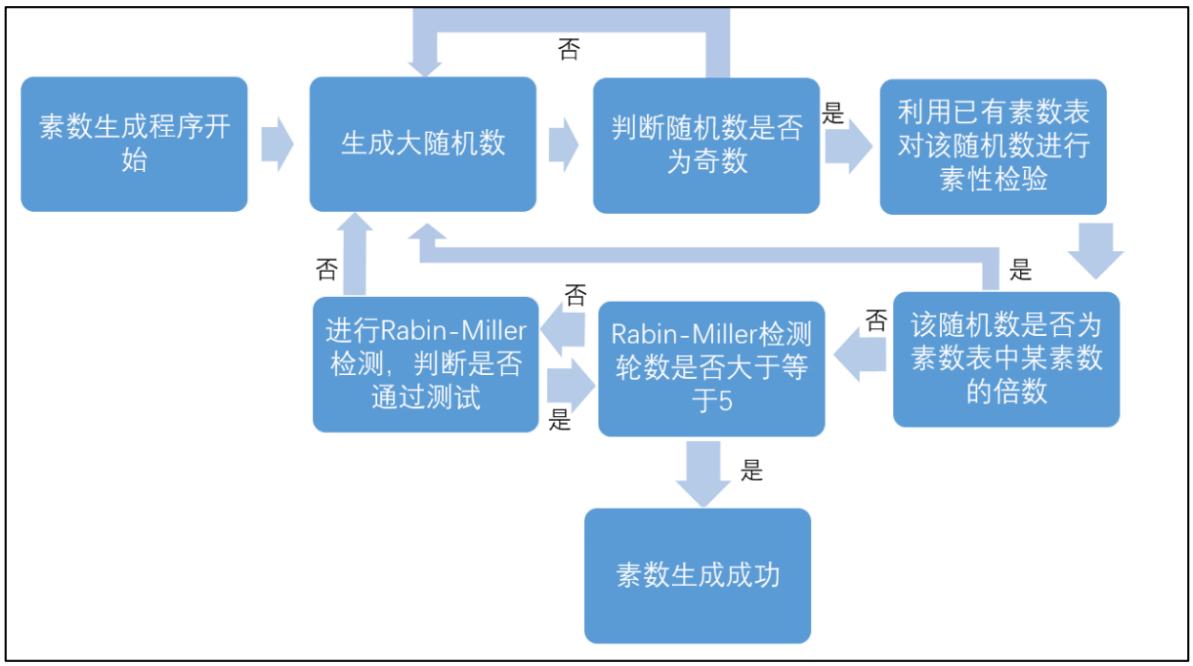
```
            if ((sub > dividend) && (subsequent > dividend))
            {
                high = mul;
                continue;
            }

        }

        //试商结果保存到商中去
        quotient.data[i] = mul;
        //临时被除数变为被除数与试商积的差
        dividend = dividend - sub;

        //临时被除数往后补位
        if ((i - 1) >= 0)
        {
            for (int j = lengthb; j > 0; j--)
                dividend.data[j] = dividend.data[j - 1];
            dividend.data[0] = a.data[i - 1];
        }
    }

    //判断商的符号
    if (a.sign == b.sign)
        quotient.sign = true;
    else
        quotient.sign = false;
    return quotient;
}

```

素数生成

一个素数的生成需要两步：

1. 生成一个随机数
2. 检验这个数是不是奇数，如果是，继续执行步骤3；如果不是，返回步骤1
3. 网上可以找到10000之前的所有素数，可以利用这些素数进行一个初步的检验，看是否是素数，这一步被称为初步检测。如果初步检测通过，执行步骤4；如果不通过，返回步骤1
4. 利用Rabin-Miller检测法进行检验，一共检测5轮。如果5轮检验均通过，则成功生成大素数；如果有任意一轮没有通过，则返回步骤1

相关流程图如下：



随机数生成

实际上在程序中是不存在真正的随机数的，所有的随机数都是基于一定算法算出来的伪随机数，这里使用经典的利用时间进行随机数的生成。

```
void BigInt::Random()
{
    for (int i = 0; i < (thesize / 4); i++) {
        if ((i == 3) || (i == (thesize / 4) - 1)) {
            // 使用之前的两位进行一个加密
            int tempHigh = data[i] - 2;
            int tempLow = data[i] - 1;
            data[i] = encryptForRSA(tempHigh, tempLow);
        }
        else {
            //由于RAND()最大只能产生0X7FFF的数,为了能产生32位的随机数,需要3次RAND()操作
            data[i] = (rand() << 17) + (rand() << 2) + rand() % 4;
        }
    }
}
```

这里我们的大整数类是由64个int组成的数组来进行表示的，在生成随机数的时候，首先使用时间函数进行3次生成，然后将3次生成的内容按照一定规则进行移位后拼接，形成一个32位的int。然后在生成第4位和最后一位的时候会使用之前生成过的结果进行DES加密，并将加密的内容作为一个随机数放进数组中。

DES加密部分内容：

```
int encryptForRSA(int high32, int low32) {
    // 首先使用两个int进行一个处理，让他们随机选择一组key
    int keyNum = (high32 % 4 * low32 / 10 + 1) % 10;

    // 获取key
    for (int countKey = 0; countKey < 8; countKey++) {
```

```
        bitset<8> tempKey(int(cases[keyNum].key[countKey]));
        for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
            key[63 - (countKey * 8 + countOfBit)] = tempKey[7 - countOfBit];
        }
    }

    // 生成子密钥，保存在全局变量的subKey[]中
    generateSubKeys();

    // 对两个输入的int进行处理，使其保存在bitset<64> m中
    bitset<32> tempHigh(high32);
    bitset<32> tempLow(low32);
    for (int i = 0; i < 64; i++) {
        if (i < 32)
            m[i] = tempLow[i];
        else
            m[i] = tempHigh[i - 32];
    }

    // DES加密
    c = encrypt(m);

    // 选取其中的32位，作为一个int进行返回，从第一位不是0开始（保证是奇数，有利于后面的检测）
    bitset<32> resultBitset;
    int indexNotZero = 0;
    for (; indexNotZero < 64; indexNotZero++)
        if (c[indexNotZero])
            break;
    for (int i = 0; i < 32; i++)
        resultBitset[i] = c[indexNotZero++];

    // 转换成int，返回
    int result = int(resultBitset.to_ulong());

    return result;
}
```

其中key是实验1中测试样例的10组key中任选其一，并且在返回int值时，我们通过末尾为1来保证返回的值一定是个奇数，从而使得生成的大数一定是奇数。

素数检验

初步检验

所谓初步检验就是先判断当前的随机数是否是奇数，然后根据已有的一些素数，去查看这个随机数能不能整除之前已有的素数表中的内容。其中素数表保存在`bigint.h`中，具体产生待测素数的相关代码为：

```
// 产生一个待测素数,保证此数为奇数,且不能被素数表中的素数整除
void SortPrime(BigInt& n)
{
    int i = 0;
    BigInt divisor;
    const int length = sizeof(prime) / sizeof(int);
```

```
    while (i != length)
    {
        n.Random();
        while (!n.IsOdd())
            n.Random();

        i = 0;
        for (; i < length; i++)
        {
            divisor = prime[i];
            if ((n % divisor) == 0)
                break;
        }
    }
}
```

Miller-Rabin检测

【算法流程】

(1) 对于偶数和 0，1，2 可以直接判断。

(2) 设要测试的数为 x，我们取一个较小的质数 a，设 s,t，满足 $2^s \cdot t = x-1$（其中 t 是奇数）。

(3) 我们先算出 a^t，然后不断地平方并且进行二次探测（进行 s 次）。

(4) 最后我们根据费马小定律，如果最后 $a^{x-1} \neq 1(mod\ x)$，则说明 x 为合数。

(5) 多次取不同的 a 进行 $Miller-Rabin$ 素数测试，这样可以使正确性更高

【备注】

(1) 我们可以多选择几个 a，如果全部通过，那么 x 大概率是质数。

(2) $Miller-Rabin$ 素数测试中，“大概率”意味着概率非常大，基本上可以放心使用。

(3) 当 a 取遍小等于 30 的所有素数时，可以证明 int 范围内的数不会出错。

(4) 代码中我用的 int 类型，不过实际上 $Miller-Rabin$ 素数测试可以承受更大的范围。

(5) 另外，如果是求一个 $long\ long$ 类型的平方，可能会爆掉，因此有时我们要用“**快速积**”，不能直接乘。

素数生成代码

综合上述流程，本次素数生成的代码为：

```
//产生一个素数
BigInt GeneratePrime()
{
    BigInt n;
    int i = 0;
    // 记录总共产生了多少次素数
    int time = 0;

    //无限次循环，不断产生素数，直到i==5时（通过五轮RabinMiller测试）才会跳出while循环
```

```
    while (i < 5)
    {
        //记录生成了多少次大奇数
        time++;

        //产生一个待测素数
        cout << endl << endl;
        cout << "正在生成第" << time << "个大奇数：" << endl;
        SortPrime(n);
        n.display();

        i = 0;
        //进行五轮RABINMILLER测试,五轮全部通过则素数合格
        for (; i < 5; i++)
        {
            cout << "正在进行第" << i + 1 << "轮RabinMiller测试：";
            if (!RabinMiller(n))
            {
                cout << "……测试失败" << endl << endl;
                break;
            }
            cout << "……测试通过" << endl << endl;
        }
    }
    return n;
}
```

生成结果

```
正在生成第39个大奇数：
D96CC215 DAE2E05D 96D61264 0D48504A 3CF26AFA 4D46F93C 90BA8A7B 81314ACA
ACCEC805 0CF02A25 840E3F8A BF03969B D96CC215 66220086 02FFFBB0 9C5BBAB3
正在进行第1轮RabinMiller测试：……测试通过

正在进行第2轮RabinMiller测试：……测试通过

正在进行第3轮RabinMiller测试：……测试通过

正在进行第4轮RabinMiller测试：……测试通过

正在进行第5轮RabinMiller测试：……测试通过

==================素数p生成成功==================
p：
D96CC215 DAE2E05D 96D61264 0D48504A 3CF26AFA 4D46F93C 90BA8A7B 81314ACA
ACCEC805 0CF02A25 840E3F8A BF03969B D96CC215 66220086 02FFFBB0 9C5BBAB3
```

```
正在生成第24个大奇数：
7FD11AF3 D67D2D7B 1B537271 05BAB0D6 AB9CDCBA 85A9F902 BBDC9F14 D2EE269C
B571232D 2902A0A9 009927B0 761D4EE4 7FD11AF3 348FA9A5 418175EA 0DCB38E5
正在进行第1轮RabinMiller测试：……测试通过

正在进行第2轮RabinMiller测试：……测试通过

正在进行第3轮RabinMiller测试：……测试通过

```

```
正在进行第4轮RabinMiller测试：.....测试通过

正在进行第5轮RabinMiller测试：.....测试通过

==================素数q生成成功==================
q：
7FD11AF3 D67D2D7B 1B537271 05BAB0D6 AB9CDCBA 85A9F902 BBDC9F14 D2EE269C
B571232D 2902A0A9 009927B0 761D4EE4 7FD11AF3 348FA9A5 418175EA 0DCB38E5
```

相关结果可以见`prime.txt`。

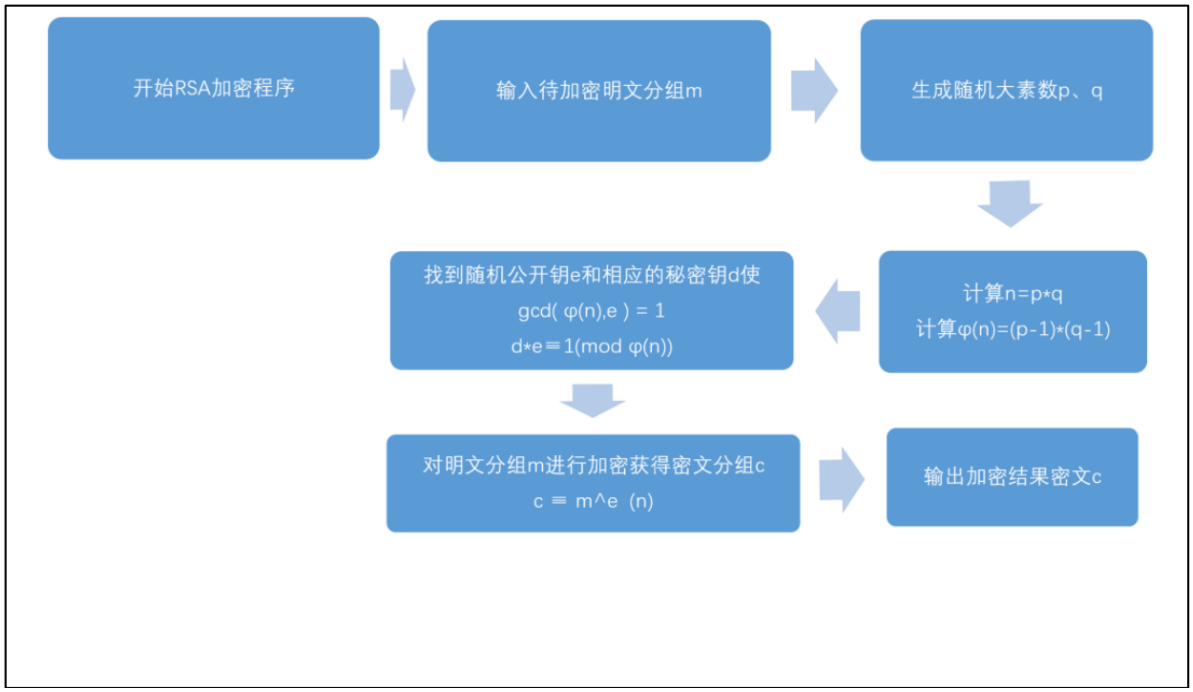
RSA

RSA的加解密运算是比较简单的，当所需要的内容生成结束后，只需要进行模幂运算即可，并且加密和解密的过程相同，只是使用的数据不一样，所以这里只简单放一下流程图和相关代码。

其中两者的模幂运算使用的内容相同，其代码为：

```
// 利用Montgomery算法，
BigInt PowerMode(const BigInt& n, const BigInt& p, const BigInt& m)
{
    BigInt temp = p;
    BigInt base = n % m;
    BigInt result(1);

    //检测指数p的二进制形式的每一位
    while (!(temp <= 1))
    {
        //如果该位为1，则表示该位需要参与模运算
        if (temp.IsOdd())
        {
            result = (result * base) % m;
        }
        base = (base * base) % m;
        temp >> 1;
    }
    return (base * result) % m;
}
```

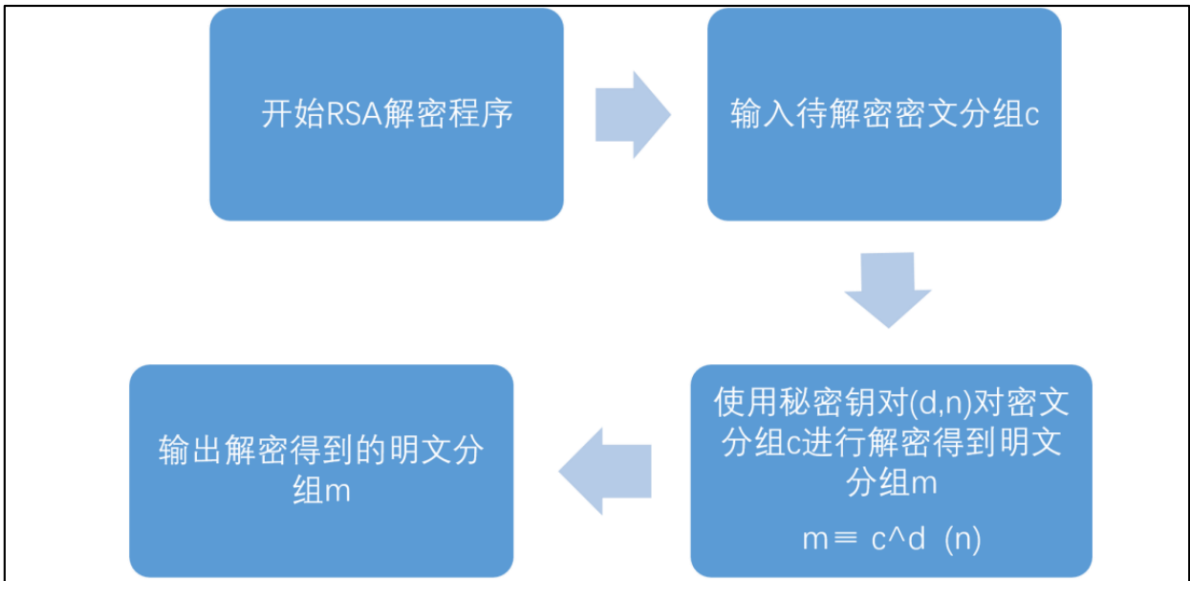
加密



相关代码为：

```
// RSA加密
BigInt RSAEn(BigInt& m, BigInt& e, BigInt& n) {
    // 加密后的结果
    BigInt c;
    c = PowerMode(m, e, n);
    return c;
}
```

解密



相关代码为：

```cpp
// RSA解密
BigInt RSADe(BigInt& c, BigInt& d, BigInt& n) {
    // 解密后的结果
    BigInt m;
    m = PowerMode(c, d, n);
    return m;
}
```

实验结果

本轮测试输出结果：

```
==========================加密准备==========================
经过计算得到公钥(n,e)为：
(6C8E8CF7 6F8BEE7B 209ACA4A 0F0C2BE9 D93FEEA9 4EF22A40 CA8241CF D670AF39
3160289F 77234317 7D8720A8 5F9CB95C 20540853 CA50D5FD 8E3108C9 617D2FD8
9D1C996A F44D491C 3468B530 10D7F102 1B346876 A18F918F 8F2232DF 21EFF884
BD457859 FB904D84 059A3A6B 447F557F CE3C6D5C 68058531 9A66E306 11D62A1F
 ,
1231C8CB 59BD761F C4E8FC83 81E7565B 1CD28B50 6E41C8AA 18AD6F9B 146F41B7
C73D148D 7FAF83E1 79C574D1 0D8AA4F3 1231C8CB ACAD3060 4182DA37 80D24EB1
)
得到私钥(n,d)为：
(6C8E8CF7 6F8BEE7B 209ACA4A 0F0C2BE9 D93FEEA9 4EF22A40 CA8241CF D670AF39
3160289F 77234317 7D8720A8 5F9CB95C 20540853 CA50D5FD 8E3108C9 617D2FD8
9D1C996A F44D491C 3468B530 10D7F102 1B346876 A18F918F 8F2232DF 21EFF884
BD457859 FB904D84 059A3A6B 447F557F CE3C6D5C 68058531 9A66E306 11D62A1F
 ,
0DB89BA3 2F28D4B0 FE187DFD D6A296C0 2A73B2D4 AB7475E9 54440558 0E1E174E
1A818FDC 709591D2 2D1B6E74 28E7F73A 8BFEA6A7 C0FC1672 6FF085DA B13B6D66
0792F3BE 665827D7 F624DF59 9AEE605F BDF0F626 4590DC51 5D46F103 62885305
3AFA13AB 581BC7EB E665DD81 77820D0F D26D3311 FA7EE6C5 99A180CF CAC24989
)




==========================随机生成明文==========================
生成的随机明文为：
C2BEAED1 609F700F F8D49B09 3AD80F51 91CA3C7D D807AD54 15275841 7A718043
6FCC2934 C633E465 2ED04139 41559DEB C2BEAED1 8C7D553A C9522955 A943917F
加密后得到的密文为：
5CB43C00 E9EC4D51 C02D0AE8 FE3D28D3 3220EC9E 4431FCFE 599E5699 FD3D8796
4A594643 FEE2C09E 2BE46C60 799D6ACC 9C40A147 CB8AFE13 40C04049 E900F69A
05C1B91A 2E37DC6C 3E74E919 EE2CB4C2 ED1EC475 F2C272FA 066E0869 649690DB
F736DA96 73E33E74 32A7660C 807DA2F9 BD91F9E5 ED648ADB 670AB65D 592655D0
对密文解密后得到的明文为：
C2BEAED1 609F700F F8D49B09 3AD80F51 91CA3C7D D807AD54 15275841 7A718043
6FCC2934 C633E465 2ED04139 41559DEB C2BEAED1 8C7D553A C9522955 A943917F
RSA相关内容已经写入rsaResult.txt中
```

实验结果的内容保存在了`rsaResult.txt`中，可以进行查看。