RSA 实验报告

崔浩 2018214160

2018-11-10

清华大学软件学院

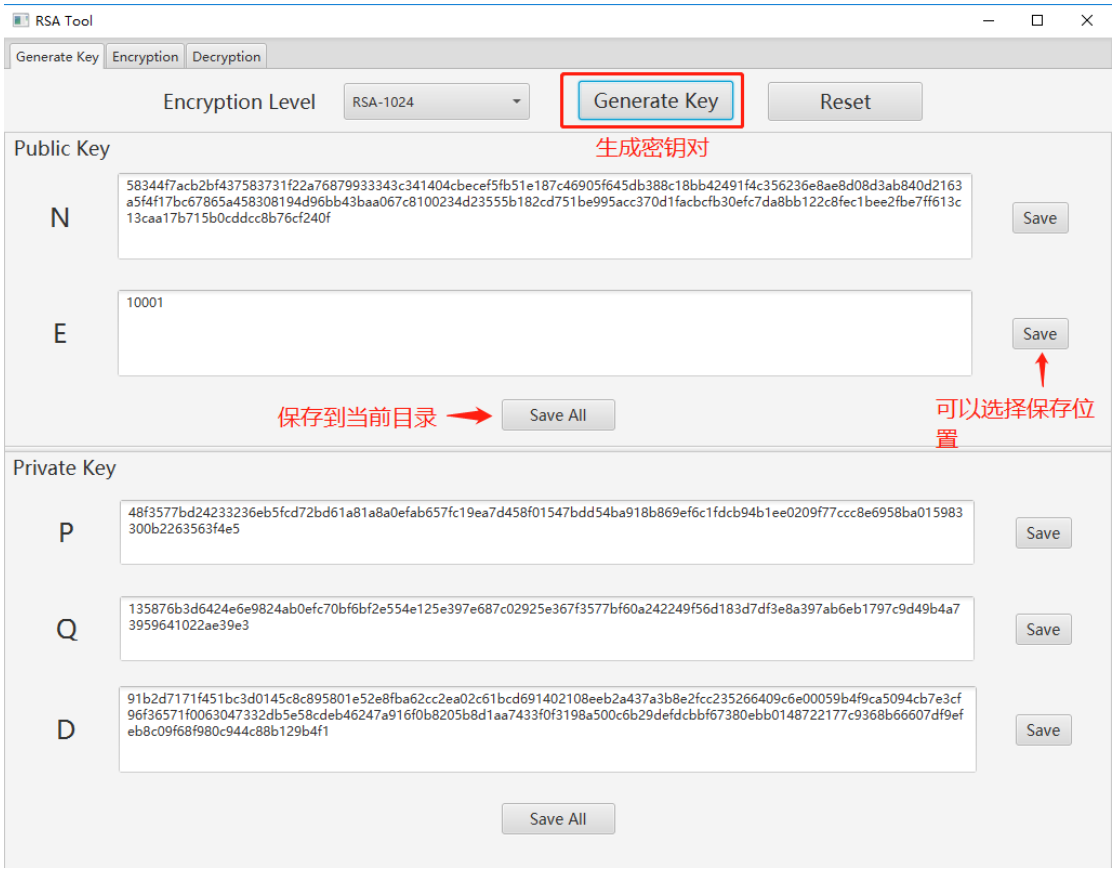
目录

一、实验环境

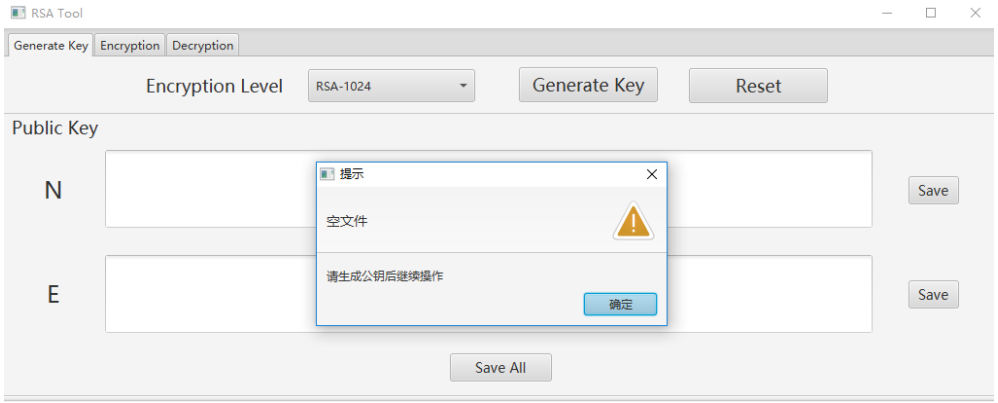
操作系统：Windows10 专业版 64 位
处理器：Intel(R) Core(TM) i7-8700 @3.2GHZ
内存：16G
编程语言： Java 1.8

二、实验截图

1. 生成密钥对

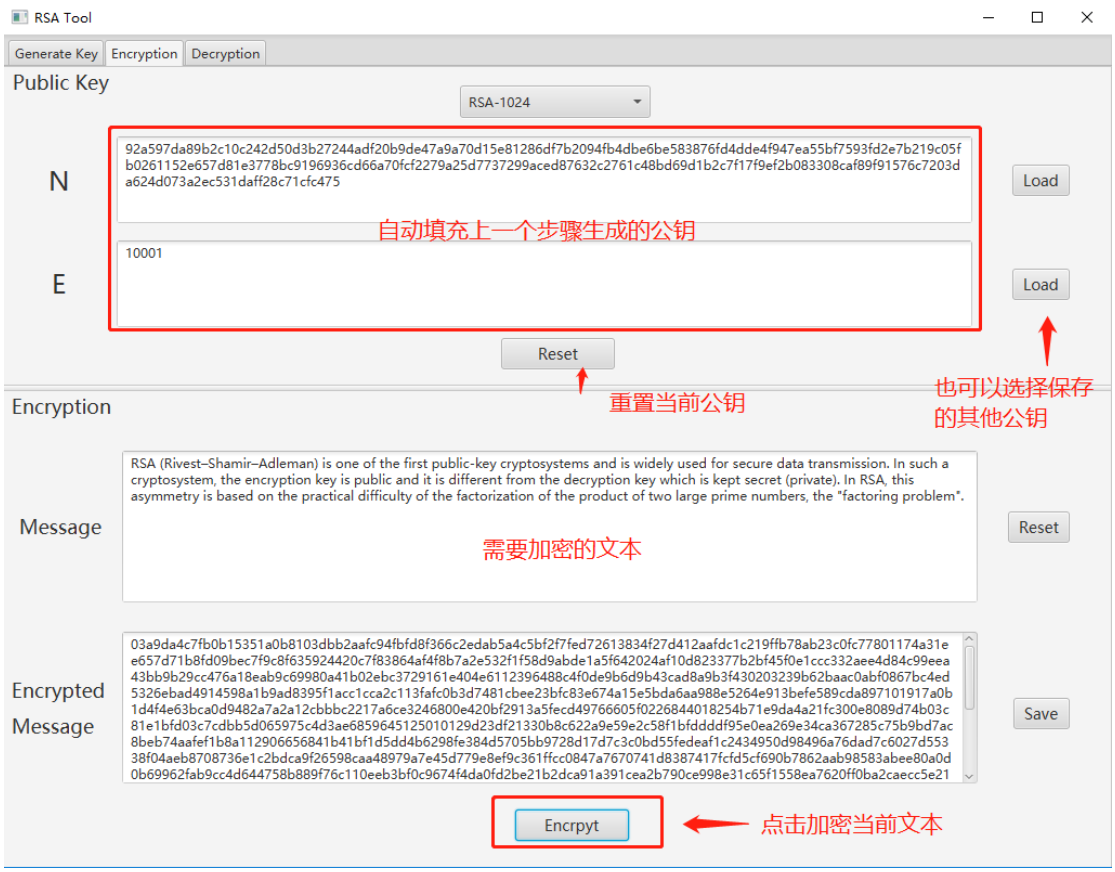


点击 Generate Key 可以生成公钥和私钥对，选择 Encryption Level 可以选择长度，长度包括 256、512、768、1024、2048。点击 Save 可以选择密钥对的保存位置，点击 Save All 可以将公钥对或私钥对保存到程序当前目录下。如果操作错误会有程序提示。

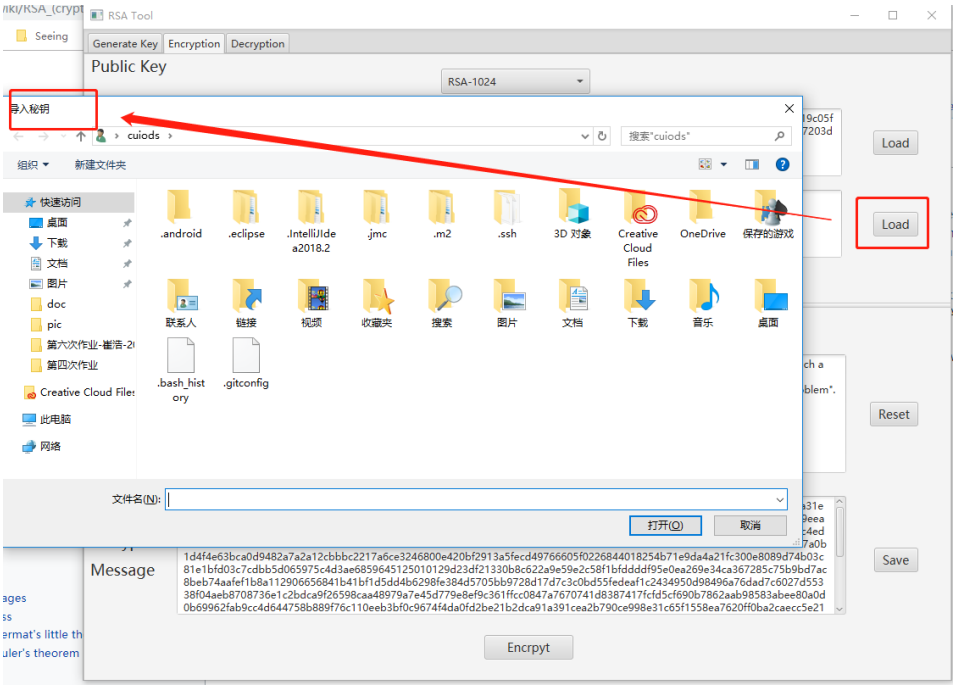

RSA Tool
Generate Key
Encryption
Decryption
Encryption Level
RSA-1024
Generate Key
Reset
Public Key
N
E
Save
Save
Save All
提示
空文件
请生成公钥后继续操作
确定


RSA Tool
Generate Key
Encryption
Decryption
Encryption Level
RSA-1024
Generate Key
Reset
Public Key
N
E
10001
Save
Save
Save All
提示
成功
文件已保存至当前目录
确定

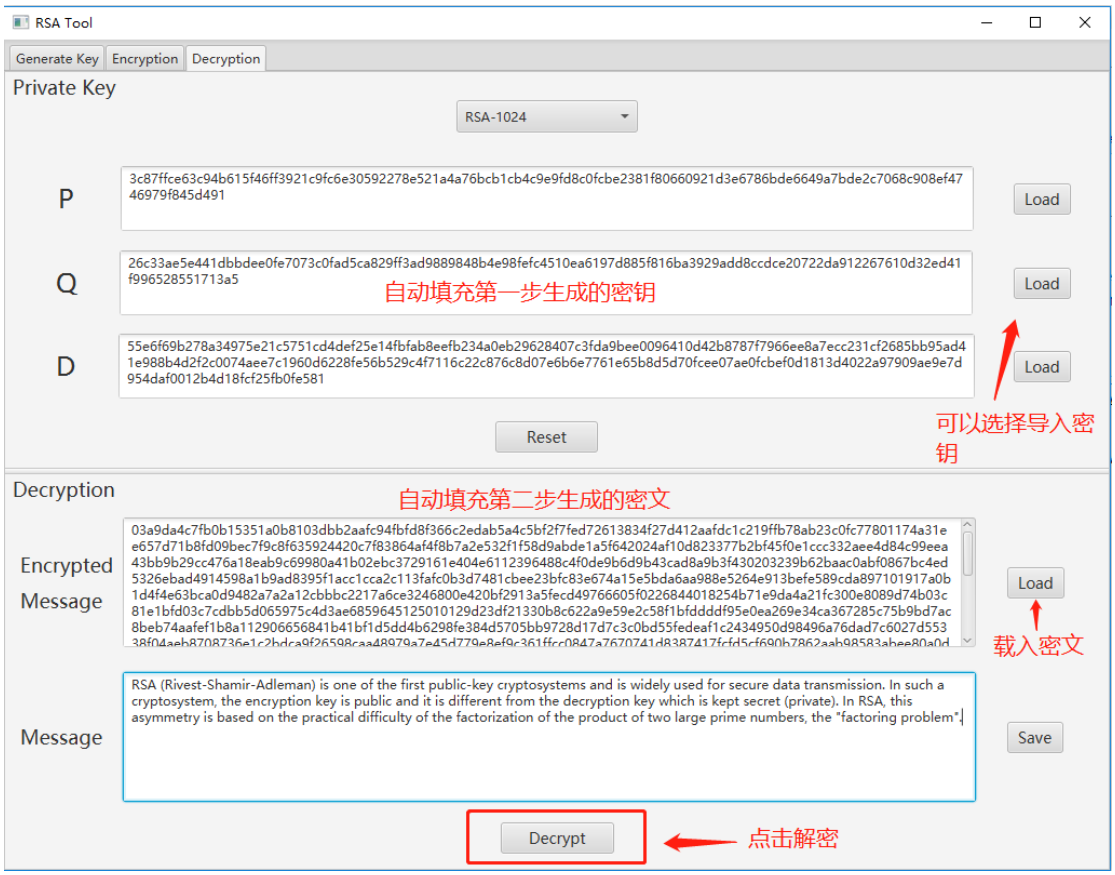
2. 加密


RSA Tool
Generate Key
Encryption
Decryption
Public Key
RSA-1024
N
92a597da89b2c10c242d50d3b27244adf20b9de47a9a70d15e81286df7b2094fb4dbe6be583876fd4dde4f947ea55bf7593fd2e7b219c05f
b0261152e657d81e3778bc9196936cd66a70fcf2279a25d7737299aced87632c2761c48bd69d1b2c7f17f9ef2b083308caf89f91576c7203d
a624d073a2ec531daff28c71cfc475
自动填充上一个步骤生成的公钥
Load
E
10001
Load
Reset
重置当前公钥
也可以选择保存
的其他公钥
Encryption
Message
RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a
cryptosystem, the encryption key is public and it is different from the decryption key which is kept secret (private). In RSA, this
asymmetry is based on the practical difficulty of the factorization of the product of two large prime numbers, the "factoring problem".
需要加密的文本
Reset
Encrypted
Message
03a9da4c7fb0b15351a0b8103dbb2aafc94fbfd8f366c2edab5a4c5bf2f7fed72613834f27d412aafdc1c219ffb78ab23c0fc77801174a31e
e657d71b8fd09bec7f9c8f635924420c7f83864af4f8b7a2e532f1f58d9abde1a5f642024af10d823377b2bf45f0e1ccc332aee4d84c99eea
43bb9b29cc476a18eab9c69980a41b02ebc3729161e404e6112396488c4f0de9b6d9b43cad8a9b3f430203239b62baac0abf0867bc4ed
5326ebad4914598a1b9ad8395f1acc1cca2c113fafc0b3d7481cbee23bfc83e674a15e5bda6aa988e5264e913befe589cda897101917a0b
1d4f4e63bca0d9482a7a2a12cbbbc2217a6ce3246800e420bf2913a5fecd49766605f0226844018254b71e9da4a21fc300e8089d74b03c
81e1bfd03c7cdbb5d065975c4d3ae6859645125010129d23df21330b8c622a9e59e2c58f1bfdddf95e0ea269e34ca367285c75b9bd7ac
8beb74aafef1b8a1129066568 41b41bf1d5dd4b6298fe384d5705bb9728d17d7c3c0bd55fedeaf1c2434950d98496a76dad7c6027d553
38f04aeb8708736e1c2bdca9f26598caa48979a7e45d779e8ef9c361ffcc0847a7670741d8387417fcfd5cf690b7862aab98583abee80a0d
0b69962fab9cc4d644758b889f76c110eeb3bf0c9674f4da0fd2be21b2dca91a391cea2b790ce998e31c65f1558ea7620ff0ba2caecc5e21
Save
Encrpyt
点击加密当前文本

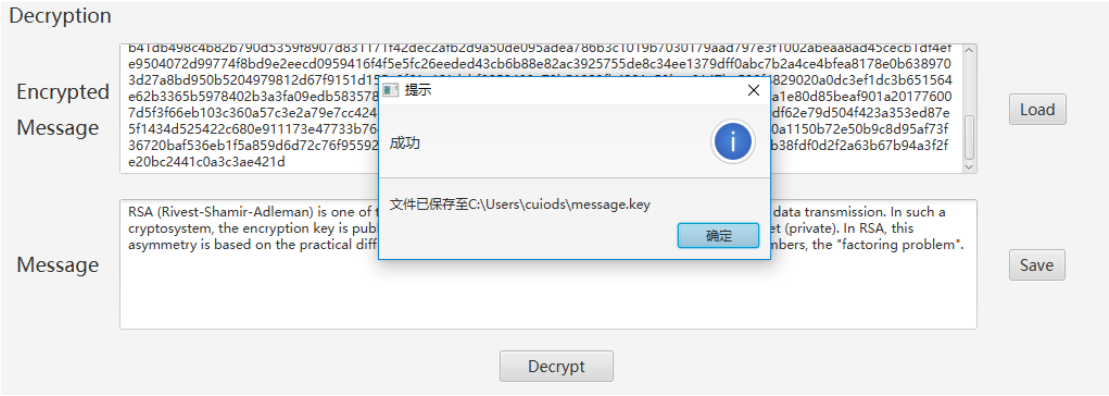
在加密页面，上一步生成的公钥会被自动填充到公钥栏，但是可以点击 load 选择其他文件中保存的公钥，在 message 框内输入需要加密的文本，点击 Encrypt 则可以进行文本加密。



3. 解密



在解密步骤，程序会自动填入第一步中生成的密钥和第二步中生成的密文，点击 Decrypt 可以进行解密，也可以使用其他密文进行解密。点击 save 可以选择保存位置并保存。



三、实验设计

1. 大数的实现

大数的表示

通常的大数表示为 10 进制的数组，例如，数组的第 0 位保存该数 10 进制表示的第一位，数组的第 1 位表述该数十进制表示的第二位……然而，这样的表示浪费了存储空间，一个 int 类型只用来表示 0~9 的数，而大数运算的复杂度通常和大数的数组表示长度相关，这种表示方式由很长的运行时间。

2	1	3	7	8	9

54321	12345	10^9-3	0	10^9-2	10^9-1

本实验使用的是 10^9 进制的表达方式，将大数表示为 10^9 进制，数组的每一位表示该进制表示的一位。Java 的 int 类型表达的整数最大值为 2147483647，约为 $2*10^9$，所以采用这种表达方式，在进行大数的加减时两个位置相加也不会超出整数的表达范围。

实际实现中，使用 sign 表示大数的符号，使用 digits 数组保存每一位的值。

```
// The integer is In 10^9 notation
private int[] digits;
// -1 means less than zero, 1 means more than zero
private int sign;
```

大数的除法运算

大数的除法模拟了手动计算除法，其中有一个步骤为确定商的每一位的值。在大数的实现中通常实现为用一个递增的数 n 来乘以除数得到一个中间值，如果这个中间值小于当前的余数，则将 n 递增，如果刚好大于当前的余数，则说明商的这一位为 n-1。

为了加速寻找每一位 n 的过程，使用二分法来确定 n。算法的伪代码如下：

```
CALCULATE-N(divisor, remainder)
left=0;
right=0;
let temp be an empty integer array
while left < right
      middle = (left + right) / 2
      temp = multiply(divisor, middle)
      if (compareArray(temp, remainder) <= 0)
            left = middle + 1
      else
            right = middle
return left-1
```

其中， compareArray 比较两个大数的数组表示的大小，如果返回值小于等于 0，说明第一个参数小于等于第二个参数数组。

大数的乘法逆元

使用扩展欧几里德算法求大数的逆元。在本次实验中，使用迭代的方法实现欧几里德算法。算法的关键代码为：

```
MyInteger reverse = ZERO;
boolean findReverse = false;
while (true) {
   MyInteger q = r0.divide(r1)[0];
   r = r0.subtract(q.multiply(r1));
   s = s0.subtract(q.multiply(s1));
   t = t0.subtract(q.multiply(t1));
   r0 = r1; r1 = r;
   s0 = s1; s1 = s;
   t0 = t1; t1 = t;
   if (r1.compareAbs(ONE)==0 && !findReverse) {
         reverse = t1;
         findReverse = true;
   }
   if (isZero(r1)) {
```

```
                if (!findReverse) reverse = t1;
                break;
            }
        }
        if (reverse.sign < 0) {
            reverse = reverse.add(integer);
        }
```

2. 素数的生成

生成指定位数的素数

经过多次实验，如果每次都随机生成奇数，进行素数判定，找到素数的过程较为缓慢。这是因为随机生成指定位数的奇数这一过程较为缓慢。为了减少随机的次数，算法进行以下优化：每次生成指定位数的素数时只生成一次指定位数的奇数，如果没有能通过素数判定，则将这个奇数加 2。这样每次只需要进行一次随机运算，算法的详细代码如下：

```
public static MyInteger randomPrime(int bit) {
        //随机生成一个指定位数的奇数
        MyInteger integer = RandomUtil.randomOdd(bit);
        boolean big = ((bit + 1)/2)>=95;
        while (!isPrime(integer, big)) { //进行素数判定
            integer = integer.add(MyInteger.TWO); //如果不是素数，则将这个奇数加 2
        }
        return integer;
}
```

素数判定

素数判定综合使用了数筛法、fermat 测试和 miller-rabin 测试。算法的过程为：

1. 判断给定整数的位数 n，如果 n 小于一个常数，则跳过数筛法测试，否则进行数筛法测试，没有通过返回 false，否则继续下一步。
2. 为了提高效率，进行一次以 2 为底的 Fermat 测试，如果不通过，返回 false，否则继续下一步。
3. 进行 3 次 miller-rabin 测试，如果不同，返回 false，否则返回 true

其中，数筛法选用的是除以已知素数的方法。程序维护了一个 2~100 范围内已知的素数，如果大数能整除其中任何一个素数，则说明该数不是素数。

算法的详细代码如下：

```
public static boolean isPrime(MyInteger integer, boolean big) {
        if (big) { //  如果是大数则进行数筛法测试
            int preTest = preSelection(integer);
            if (preTest > 0)
                return true;
            else if (preTest < 0)
                return false;
        }
        if (!fermatTest(integer, 1)) { //进行一次 fermat 测试
            return false;
        }
        return millerRabinTest(integer, 3); //进行三次 miller-rabin 测试
    }
```

3. mod 运算的优化

分治法优化 num^d mod n

首先，在大数运算中计算 a^b 是不现实的，因此一般的想法是每两次乘法就进行一次 mod 运算，程序一共进行了$O(n)$次 mod 运算。然而，在计算 a^b 时可以只计算 $x=a^{b/2} \bmod n$，再计算 x*x mod n，利用递归，可以只进行$O(\lg n)$次 mod 运算，算法的伪码表示为：

```
QUICK-MOD(num, d, n):
if d = 0
    return 1
if d = 1
    return num mod n
half = d/2
temp = QUICK-MOD(num, half, n)
result = temp * temp mod n
if d mod 2 != 0
    result = result * num mod n
return result
```

基于这样的思想，设计如下代码：

```
public static MyInteger speedUpMod(MyInteger num, MyInteger d, MyInteger n) {

        if (d.isZero()) return MyInteger.ONE;
        if (d.compareAbs(MyInteger.ONE)==0) return num.mod(n);

        num = num.mod(n); //首先对 num 本身求 mod n
        MyInteger[] oddTest = d.divide(MyInteger.TWO); //计算 d/2
```

```
        MyInteger temp = speedUpMod(num, oddTest[0], n); //递归计算
        MyInteger result = temp.multiply(temp).mod(n);

        if (!oddTest[1].isZero()) {//如果是奇数，需要再乘一次 num
            result = result.multiply(num).mod(n);
        }
        return result;
}
```

优化 num^d mod p*q

如果已知$n = p \times q$，则可以利用剩余系和中国剩余定理对算法进行优化。将 num^d mod p*q 转化为$(\langle\langle num\rangle_p{}^d\rangle_p, \langle\langle num\rangle_q{}^d\rangle_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$，利用欧拉函数的性质，可以转化为$(\langle\langle num\rangle_p{}^{\langle d\rangle_{\phi(p)}}\rangle_p, \langle\langle num\rangle_q{}^{\langle d\rangle_{\phi(q)}}\rangle_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$，分别计算$\langle num\rangle_p$、$\langle num\rangle_q$、$\langle d\rangle_{\phi(p)}$、$\langle d\rangle_{\phi(q)}$，可以加速这个求 mod 的过程。

算法的详细代码如下：

```
public static MyInteger speedUpMod(MyInteger num, MyInteger d, MyInteger p, MyInteger q)
{
        //计算⟨num⟩_p、⟨num⟩_q
        MyInteger numP = num.mod(p);
        MyInteger numQ = num.mod(q);

        //计算⟨d⟩_φ(p)、⟨d⟩_φ(q)
        MyInteger dP = d.mod(p.subtract(MyInteger.ONE));
        MyInteger dQ = d.mod(q.subtract(MyInteger.ONE));

        //使用第一步的分治法求指数 mod n
        numP = speedUpMod(numP, dP, p);
        numQ = speedUpMod(numQ, dQ, q);

        MyInteger[] x = {numP, numQ};
        MyInteger[] m = {p,q};

        //使用中国剩余定理求解方程组
        return CrtResult(m, x);
}
```

4. 加解密字符串

将字符串理解为 ASCII 码的集合，每个字符可以用 8 位 ASCII 码来表示，如果对每一个字符进行加密，那么每个字符都会生成一个 n（n 为 RSA 算法位数）位的大数，这个 n 通常会很大，例如 1024、2048，如果采用这种方式，加密后的字符串长度会是原来字符串的上百倍，这是不能接受的。

因此，可以将多个字符的二进制 ASCII 码拼接在一起，理解为一个整数，利用该整数进行加密，理论上可以维持加密后的长度不变。为了分组方便，这里采取 bit/2 长度的 ASCII 字符串为一组，在集齐一组字符串拼接为一个二进制大数，对这个大数进行加密。

解密时选取切割同样分组的字符串，恢复为大数，解密后形成一个二进制字符串，按每 8 位进行切割，形成一个 8 位 ASCII 码，仍然能还原为最开始的字符。

解密的部分代码如下：

```
for (int i = 0; i <= message.length() / charNum; i++) {

    //计算下一次字符串截取的位置，每次递增一个分组长度
    //不能超过字符串的长度
    int maxLen = Math.min(i * charNum + charNum, message.length());
    if (i * charNum < message.length()) {
        //按该长度截取密文的一段，这一段就是加密时的一个分组
        String hexStr = message.substring(i * charNum, maxLen);
        //恢复为大数进行 mod 运算
        MyInteger cInt = new MyInteger(StringConvert.convert(hexStr,16,10));
        MyInteger resultInt = SpeedUp.speedUpMod(cInt, D, P, Q);
        //运算后的结果恢复为二进制码
        StringBuilder resultStr =
                        new StringBuilder(StringConvert.convert(resultInt.toString(),10,2));
        while (resultStr.length() % ASCII_BIT != 0)
            resultStr.insert(0, "0");
        String currentStr = resultStr.toString();
        //将二进制码每 8 位进行分割，恢复成字符
        for (int j = 0; j < currentStr.length(); j+=ASCII_BIT) {
            int c = Integer.parseInt(currentStr.substring(j, j+ASCII_BIT), 2);
            char cRes = (char) c;
            result.append(cRes);
        }
    } else break;
}
```

四、实验总结

在这次实验中，为了自己实现一个高效率的大数，我阅读了部分 Java BigInteger 的源码，并查阅了相关资料。Java 的 BigInteger 类有 5000 多行，也只是实现了大数运算的基本功能。一个在平时貌似并不难写的类如果要追求效率到极致，则有很多可以改进的方面。在自己写大数的过程中由于更换了已有的大数表达方式，思维上很多适用于 10 进制的想法不再正确，在简单的加减乘除上不断遇到新的问题。由于大数运算是 RSA 中最基础的部分，所以一个高效、易用的大数类是算法最重要的基础。

在随机生成素数这一方面，我也遇到了不少挑战。在所有的算法描述中，都指明要随机生成指定位数的奇数，判断是否为素数，如果不是，则重复这一过程。然而，随机生成指定位数的奇数本身也是一件低效率的事情，由于 Java 没有有效的随机方法，我采用的方式是随机生成多个 32 位的 int 类型整数，拼接到一起产生一个指定位数的大整数，再将最后一位修改为 1 形成一个奇数。这样生成素数的时间非常缓慢。然而，如果将算法修改为随机生成一个奇数，如果不能满足素数判定条件则在这个奇数的基础上加 2，那么算法的效率会提高很多。

在随机生成一个指定范围的大数这一方面，我也经历不少挑战。自己写的大数没有基础的带范围的随机生成算法，所以一开始采用的方案是在限定位数之后随机生成大数，如果在范围之内则返回，不在就继续随机生成。这种算法很多情况下几乎不能返回有效的值，特别是在指定范围很小的情况下，相当于进入了死循环。改进的方案是随机生成一个指定位数的大数，与上届进行求 mod 运算，再与下届比较和微调。这样只需一次随机生成就可以生成一个指定范围的大数。

Java 本身的运行速度远远慢于 c++等语言，为了满足 1s 内生成密钥对的要求，需要在很多细节上进行优化。因此，这次实验除了让我更了解 RSA 的实现细节之外，更让我开始在写每一个算法时认证考虑算法的效率，在每一个细节考虑算法的性能，这才是这次实验我最大的收获。