云南大学数学与与统计学院
上机实践报告

课程名称：近代密码学实验	**年级：**2013	**上机实践成绩：**
指导教师：陆正福	**姓名：**金洋	
上机实践名称：因子分解问题实验	**学号：**20131910023	**上机实践日期：**9.27
上机实践编号：No. 04	**组号：**	**上机实践时间：** **17:45**

一、实验目的

熟悉整数因子分解问题（IFP）及其有关的密码体制

二、实验内容

1. 编程实现整数因子分解问题(IFP)有关的算法

2. 编程实现 RSA 体制

三、实验环境

个人计算机，Java 8 平台
对于非信息与计算科学专业的学生，可以选择任意编程平台

四、实验记录与实验结果分析

（注意记录实验中遇到的问题。实验报告的评分依据之一是实验记录的细致程度、实验过程的真实性、实验结果的解释和分析。**如果涉及实验结果截屏，应选择白底黑字。**）

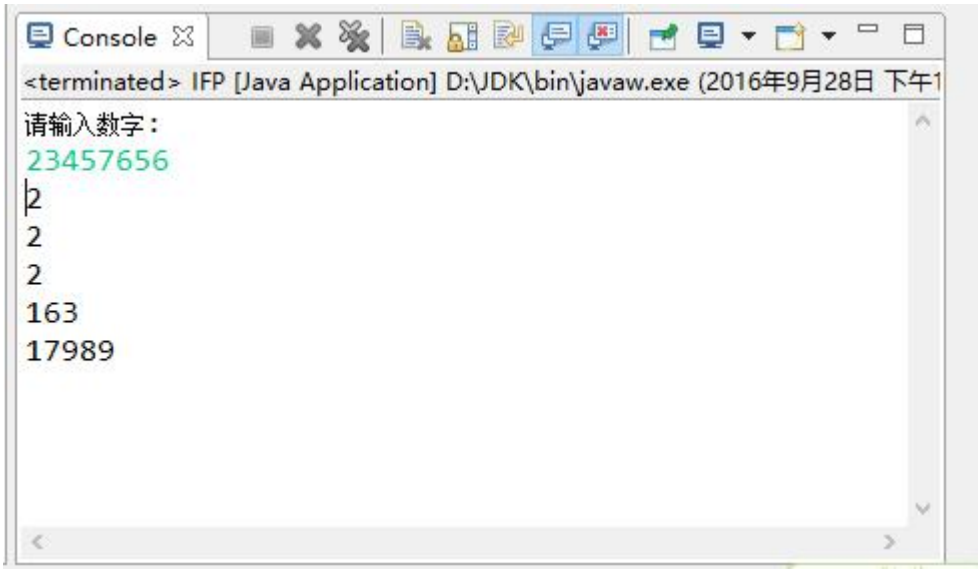
1. 编程实现整数因子分解问题(IFP)有关的算法

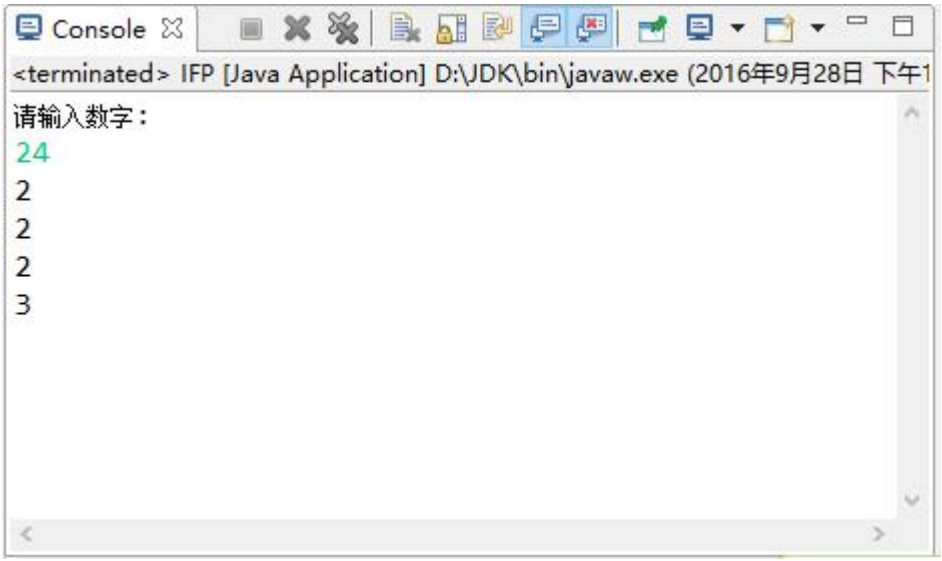
IFP.java

```
package MC04;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Scanner;


class IFP {
	private final static BigInteger ZERO = new BigInteger("0");
	private final static BigInteger ONE = new BigInteger("1");
	private final static BigInteger TWO = new BigInteger("2");
	private final static SecureRandom random = new SecureRandom();

	public static BigInteger rho(BigInteger N) {
```

```
            BigInteger divisor;
            BigInteger c = new BigInteger(N.bitLength(), random);
            BigInteger x = new BigInteger(N.bitLength(), random);
            BigInteger xx = x;
            if (N.mod(TWO).compareTo(ZERO) == 0)
                  return TWO;

            do {
                  x = x.multiply(x).mod(N).add(c).mod(N);
                  xx = xx.multiply(xx).mod(N).add(c).mod(N);
                  xx = xx.multiply(xx).mod(N).add(c).mod(N);
                  divisor = x.subtract(xx).gcd(N);
            }while((divisor.compareTo(ONE)) == 0);
            return divisor;
      }

      public static void factor(BigInteger N) {
            if (N.compareTo(ONE) == 0)
                  return;
            if (N.isProbablePrime(20)) {
                  System.out.println(N);
                  return;
            }
            BigInteger divisor = rho(N);
            factor(divisor);
            factor(N.divide(divisor));
      }

      @SuppressWarnings("resource")
      public static void main(String[] args) {
            Scanner scanner = null;
            scanner = new Scanner((System.in));
            BigInteger str = null;
            System.out.println("请输入数字:");
            str = scanner.nextBigInteger();
            factor(str);
      }

}
```

运行结果：





2. 编程实现 RSA 体制

RSA 体制的过程如下：

Bob	Alice
Key Creation	
Choose secret primes p and q. Choose encryption exponent e with $\gcd(e,(p-1)(q-1))=1$. Publish $N=pq$ and e.	
Encryption	
	Choose plaintext m. Use Bob's public key (N,e) to compute $c \equiv m^e \pmod{N}$. Send ciphertext c to Bob.
Decryption	
Compute d satisfying $ed \equiv 1 \pmod{(p-1)(q-1)}$. Compute $m' \equiv c^d \pmod{N}$. Then m' equals the plaintext m.	

程序代码：

FundAl.java

```
package MC04;
import java.math.BigInteger;

public class FundAl {
	protected BigInteger u;
	protected BigInteger v;
	protected BigInteger zero;


	public FundAl() {
		zero=new BigInteger("0");
	}

	public BigInteger getU() {
		return u;
	}
	public BigInteger getV() {
		return v;
	}

	public BigInteger euclidean(BigInteger a,BigInteger b) {
```

```
            if (b.compareTo(zero)==0) return a;
            return euclidean(b,a.mod(b));

      }

      public BigInteger extendedEuclidean(BigInteger a,BigInteger b) {
            if (b.compareTo(zero)==0) {
                  u=new BigInteger("1");
                  v=new BigInteger("0");
                  return a;
            }
            BigInteger r= extendedEuclidean(b,a.mod(b));
            BigInteger t=u;
            u=v;
            v=t.subtract(a.divide(b).multiply(v));
            return r;

      }


      public BigInteger fastPowering(BigInteger g,BigInteger A,BigInteger
N) {
            BigInteger a=g;
            BigInteger b=new BigInteger("1");
            BigInteger one=new BigInteger("1");
            BigInteger two=new BigInteger("2");
            while (A.compareTo(zero)!=0) {
                  if (A.mod(two).compareTo(one)==0) b=b.multiply(a).mod(N);
                  a=a.multiply(a).mod(N);
                  A=A.divide(two);
            }
            return b;
      }
}
```

RSA.java

```
package MC04;

import java.math.BigInteger;
import java.util.Random;

public class RSA {
      private final static BigInteger ZERO = new BigInteger("0");
      private final static BigInteger ONE = new BigInteger("1");
      private final static BigInteger TWO = new BigInteger("2");
```

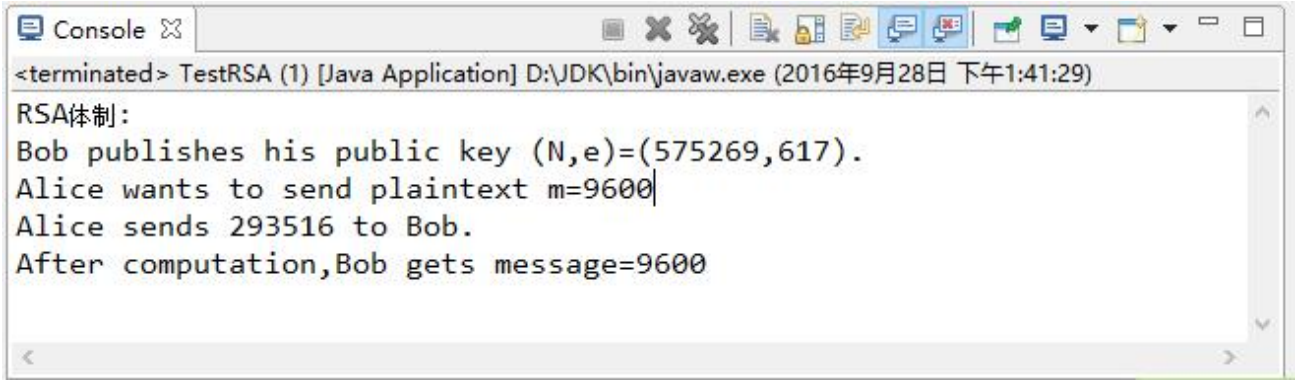
```
	public RSA() {

	}

	/*产生一个素数，是素数的概率超过 1-2^(-10)*/
	public BigInteger createBigPrime(int len) {
		BigInteger p;
		do {
			p=new BigInteger(len, 10, new Random());//此构造函数用于构造一个随机生成正 BigInteger 的可能是以指定的 len 的素数。可能性超过 1-2^(-10)
		} while (!p.isProbablePrime(10));//是素数则跳出构造
		return p;

	}


	public BigInteger createRandomInt() {
		Random rand = new Random();
		return(new BigInteger(rand.nextInt(8999)+1000+""));//产生一个四位整数

	}

	public void RSA_C_E_D() {
		BigInteger p,q,pq_1,e,N,m,c,d,t;

		/*Bob*/
		FundAl FA=new FundAl();
		p=createBigPrime(10);
		q=createBigPrime(10);
		pq_1=p.subtract(ONE).multiply(q.subtract(ONE));
		e=createBigPrime(10);
		N=p.multiply(q);
		System.out.println("Bob publishes his public key (N,e)=("+N+","+e+").");

		/*Alice*/
		m=createRandomInt();
		System.out.println("Alice wants to send plaintext m="+m);
		c=FA.fastPowering(m, e, N);
		System.out.println("Alice sends "+c+" to Bob.");

		/*Bob*/
		//d=FA.fastPowering(e, pq_1.subtract(TWO), pq_1);
		FA.extendedEuclidean(e, pq_1);
		d=FA.getU().mod(pq_1).abs();
		t=FA.fastPowering(c, d, N);
		System.out.println("After computation,Bob gets message="+t);
	}
```

```
}
```

TestRSA.java

```
package MC04;


public class TestRSA {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		RSA rsa=new RSA();
		System.out.println("RSA 体制:");
		rsa.RSA_C_E_D();
	}

}
```

实验截图：



五、实验体会

（请认真填写自己的真实体会）

1.实验中发现在计算模逆运算时，使用实验二中的算法（拓展欧几里得算法或模幂+费马小定理），花费了很多时间。当 p，q 都在 11 位以上时，计算时间超过 20s。需要考虑更有效率的算法。

因为“BigInteger(int bitLength, int certainty, Random rnd)：此构造函数用于构造一个随机生成正 BigInteger 的可能是以指定的 bitLength 的素数”，b 的规模是 b 用二进制表示时的长度，所以当 p，q 都在 11 位以上时，运用拓展欧几里得算法，问题规模在 22 位。

之后通过实验发现问题在于快速模幂算法中：

```
public BigInteger fastPowering(BigInteger g,BigInteger A,BigInteger N) {
    BigInteger a=g;
    BigInteger b=new BigInteger("1");
    BigInteger one=new BigInteger("1");
    BigInteger two=new BigInteger("2");
    while (A.compareTo(zero)!=0) {
        if (A.mod(two).compareTo(one)==0) b=b.multiply(a).mod(N);
        a=a.multiply(a).mod(N);
        A=A.divide(two);
    }
    return b;
}
```

先前的实验中都未加入标示的“mod(N)”，这导致底数不断变大，加大了乘 a 的复杂度。改正之后解密时间明显下降。

2.目前网络上大多数的 RSA 实现均为一般的整数，这里就涉及到一个大素数版本的时候，已经公钥如何求私钥的过程（这里指的是密钥产生期间）。普通的试探法无法满足需要，需要采用“扩展的欧几里德算法”才可以得到。

3.对极大整数做因数分解的难度决定了 RSA 算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA 算法愈可靠。假如有人找到一种快速因数分解的算法，那么 RSA 的可靠性就会极度下降。但找到这样的算法的可能性是非常小的。今天只有短的 RSA 密钥才可能被暴力破解。

六、参考文献

1. 主讲课教材（数学密码学导论）第三章

2. 算法导论