Generating strong prime numbers for RSA using probabilistic Rabin-Miller algorithm

An integer $p > 1$ is called a prime number in case there is no divisor $d$ of $p$ satisfying $1 < d < p$. If an integer $a > 1$ is not a prime, it is called a composite number.

Characteristics of primes

- Every integer $n$ greater than 1 can be expressed as a product of primes (with perhaps only one factor).
- There are arbitrarily large gaps in the series of primes.
- The number of primes is infinite. That is, there is no end to the sequence of primes.
- The factoring of any integer $n > 1$ into primes is unique apart from the order of the prime factors.

The RSA cryptographic algorithm uses a lot of prime numbers in generating its keys. These keys in turn will hold the strength of the encryption. In this paper I will discuss how a large number of large primes can be generated.

Public-key algorithms need prime numbers. Any reasonably sized network needs lots of them. Before discussing the mathematics of prime number generation, some of common questions are answered.

1. If everyone needs a different prime number, won’t we run out? No. In fact, there are approximately $10^{151}$ primes 512-bits in length or less. For numbers near $n$, the probability that a random number is prime is approximately one in $\ln n$. So the total number of primes less that $n$ is $n/(\ln n)$.

2. What if two people accidentally pick up the same prime number? It won’t happen. With over $10^{151}$ prime numbers to choose from, the odds of that happening are significantly less than the odds of your computer spontaneously combusting at the exact moment you win the lottery.

3. If someone creates a database of all primes, won’t he be able to use that database to break public-key algorithms? Yes, but he can’t do it. If you could store one gigabyte of information on a drive weighing one gram, then a list of just the 512-bit primes would weight so much that it would exceed the Chandrasekhar limit and collapse into a black hole &ldots; so you couldn’t retrieve the data anyway.

There are several probabilistic primality tests; tests that determine whether a number is prime with a given degree of confidence. Assuming this "degree of confidence" is large enough, these sorts of tests are good enough.

Various probabilistic primality-testing algorithms include:

- Rabin-Miller Test
- Lucas-Lehmer Algorithm
- Fermat’s Test
• Frobenius-Grantham

The algorithm everyone uses was developed by Michael Rabin, based in part on Gary Miller’s ideas. Actually, this is a simplified version of the algorithm recommended in the DSS proposal. Therefore, we start out with Rabin-Miller algorithm.
Rabin Miller Algorithm

Rabin-Miller algorithm has following characteristics

- Uses exponentiations mod p to declare that p is ‘probably’ prime
- Each iteration has worst case failure rate of \( (\frac{1}{4})^c \)
- For randomly selected large numbers this failure rate is much lower
- One can construct numbers which achieve worst case

Algorithm

Input : an odd integer \( n \geq 3 \) and number of iterations \( t \geq 1 \)
Output : “Prime” or “Composite”

1. Write \( n - 1 = 2^s r \) such that \( r \) is odd
2. For \( i \) from 1 to \( t \) do the following
   2.1 Choose a random integer \( a, 2 \leq a \leq n-2 \)
   2.2 Compute \( y = a^r \mod n \)
   2.3 If \( y \neq 1 \) and \( y \neq n - 1 \)
      \( j = 1 \)
      While \( j \leq s-1 \) and \( y \neq n - 1 \)
      Compute \( y = y^2 \mod n \)
      If \( y = 1 \) then return (“Composite”)
      \( j = j + 1 \)
      if \( y \neq n - 1 \) then return (“Composite”)
3. Return Prime

Class named RabinMiller with constructor of form MillerRabin(BigInteger \( n \), int \( t \)) has been used for experiment. This class expects arbitrary long precision number and number of iterations \( t \). We then use method isPrime() to see if this number is a probable prime.

The Rabin-Miller Probabilistic Primality Test Proof

Fermat’s Little Theorem. Let \( \rho \) be prime and suppose \( \rho \nmid a \) (\( \rho \) does not divide \( a \)) Then
\[ a^{\rho-1} \equiv 1 \pmod{\rho} \]
This is a generalization of the Chinese Hypothesis and a special case of Euler’s Theorem

Euclid’s Lemma. Let \( \rho \) be prime and suppose \( \rho \mid ab \). Then either \( \rho \mid a \) or \( \rho \mid b \).

Now we can outline a development which leads to Miller’s Test for compositeness.

By Fermat’s Little Theorem, if \( \rho \) is an odd prime and \( \rho \nmid b \), then
\[ b^{\rho-1} \equiv 1 \pmod{\rho} \]
Since \( \rho-1 \) is even,
\[ \rho \mid (b^{\frac{\rho-1}{2}} - 1) (b^{\frac{\rho-1}{2}} + 1) \]
By Euclid’s Lemma,
\[ b^{\frac{\rho-1}{2}} \equiv \pm 1 \pmod{\rho} \]
If \((\rho-1)/2\) is even and \( b^{(\rho-1)/2} \equiv 1 \pmod{\rho} \), then we know that
\[ b^{(\rho-1)/4} \equiv \pm 1 \pmod{\rho} \]
We can repeat this until \((\rho-1)/2^k\) is odd or \( b^{(\rho-1)/2^k} \equiv -1 \pmod{\rho} \) for some \( k \).
Miller’s Test. Let \( n - 1 = 2^r m \) where \( r \geq 1 \) and \( m \) is odd. Let \( \gcd(b, n) = 1 \).

If either \( b^m \equiv \pm 1 \pmod{n} \) or \( b^{2^k} \equiv -1 \pmod{n} \) for some \( k \leq r \), then, we say that \( n \) passes Miller’s Test to base \( b \).

All primes \( p \) pass Miller’s Test for \( 1 < b < p \), and if \( p \) fails Miller’s test in this interval, then \( n \) has to be composite. However, composites can pass Miller’s Test for some values of \( b \). It can also be shown that if \( n \) is an odd composite, then \( n \) passes Miller's Test for at most \( (n-1)/4 \) bases \( b \) with \( 1 < b < n-1 \). This therefore could be turned into a deterministic test if we are willing to do \( (n-1)/4 + 1 \) iterations.

**Experiment**

1. We first find probable primes that pass 20 iteration of Rabin-Miller test. We use them as test vectors to measure time it takes

2. Perform Miller Rabin test on these test vectors using 10 and 20 iterations

<table>
<thead>
<tr>
<th>Size of Number (in bits)</th>
<th>10 Iterations</th>
<th>20 Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>0.05 sec</td>
<td>0.083 sec</td>
</tr>
<tr>
<td>512</td>
<td>0.9 sec</td>
<td>1.7 sec</td>
</tr>
<tr>
<td>1024</td>
<td>8.8 sec</td>
<td>17 sec</td>
</tr>
<tr>
<td>2048</td>
<td>45 sec</td>
<td>90 sec</td>
</tr>
</tbody>
</table>

As we can see, time it takes to test for primality is proportional to the number of iterations. In addition, as we increase the number of bits, the time it took for the algorithm increased substantially.

Originally, random number was generated and time was measured to compare the time it took for different number of iterations. However, test result indicated that the time it took was very close. As the result was examined, it was noted that most numbers that was generated at random was a “Composite” number; therefore, number of iterations didn’t matter much(‘cause it always got reject in first few iteration). Thus, in this case, we used numbers that we knew were “Probable Prime” numbers to compare the time.
Lucas Test (Lucas Pseudoprimes)

We state that a number passes Lucas Test with parameter (P,Q) if it passes the algorithm illustrated below with p and q satisfies D = p^2 – 4q. Here we used Lucas Test with parameter Lucas(1, Q). D is determined by finding a number that satisfies (D/n) = -1 in {5, -7, 9, -11, 13, -15, 17, …} Although theoretically, we can spend infinite time looking for D that satisfies this (as with n = 9), based on the experiment, we found out that D is mostly single or two decimal digits (most of time, D is in {5, -7, 9, -11, -13, 15, -17, 19} Therefore, time spent looking for D was not significant.

Input:
n – large odd number tested for primality,

Output: 
composite or probably prime

Operations:
1. Find first d in the sequence {5, -7, 9, -11, 13, -15, 17, …} such that the Jacobi symbol (d/n) = -1.
2. Set \( p := 1 \), \( q := (1-d)/4 \).

If \( u_{n+1} = 0 \) mod \( n \), then \( n \) is probably prime. To compute \( u_k \), where \( k=n+1 \) you need to perform the following operations.

3. Set \( d = (p^2 - 4q) = 1 - 4q \)
4. Let \( k_r, k_{r-1}, \ldots, k_0 \) denote the binary representation of \( k \)
5. Set \( u=1 \) and \( v:= p \)
6. For \( i := r-1 \) to 0 do
   6.1 Set \( (u, v) = (uv \mod n, (v^2+du^2)*2^{-1} \mod n) \)
   6.2 If \( k_i = 1 \) do
      \( (u, v) = ((pu+v)*2^{-1} \mod n, (pv+du)*2^{-1} \mod n) \)
7. \( u_k = u, v_k = v \)
8. If \( u_k = 0 \) mod \( n \) return probably prime,
     otherwise return composite.

Experiment

<table>
<thead>
<tr>
<th>Size of Number (in bits)</th>
<th>Lucas Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>0.125 sec</td>
</tr>
<tr>
<td>512</td>
<td>0.6 sec</td>
</tr>
<tr>
<td>1024</td>
<td>5 sec</td>
</tr>
<tr>
<td>2048</td>
<td>35 sec</td>
</tr>
</tbody>
</table>

When we applied Lucas Test to our test vectors that passed 20 iterations of Miller-Rabin, some of them were determined to be Composites. Here we show some list of 1024 bit numbers that passed Miller-Rabin (20 iterations) but didn’t pass Lucas Test.

943663967303341733831073530494149595215288153105481870301659362295789602095234
218089124597953290352035102845761871600763866437004412165477329142505789342618
It is known that Lucas Test is independent of Miller-Rabin test. Therefore by combining these two algorithms, we present a powerful algorithm in the next section that utilizes one iteration of Miller-Rabin with Lucas Test.(we also noticed that if the number passed a single iterations of Miller-Rabin, it’s most likely to pass more iterations of Miller-Rabin -> Increasing Miller-Rabin increases worst case probability of error; however, in practice, Miller-Rabin yields substantially better probability of error(than worst case) and each subsequent iteration improves the algorithm moderately)
Lucas-Lehmer Test

Pomerance, Selfridge and Wagstaff have proposed a test, based on a combination of the Strong Probable Prime Test and the Lucas Probable Test, that seems very powerful. Indeed, nobody yet claimed the $620 that they offer for a composite that passes it, even though they have relaxed the conditions of the offer so that the Probable Prime Test no longer need be “Strong”. In this section, we present a test that combines Probable Prime Test(Miller Rabin test) and the Lucas Probable Test.

```java
MillerRabin test = new MillerRabin(a, 1);
if (test.isPrime() == true) {
    lt = new LucasTest(a);
    if (lt.isPrime()) System.out.println("prime1");
}
```

We measure a time taken by applying a single iteration of Miller-Rabin followed by Lucas Test.

<table>
<thead>
<tr>
<th>Size of Number (in bits)</th>
<th>Lucas-Lehmer Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>0.125 sec</td>
</tr>
<tr>
<td>512</td>
<td>0.72 sec</td>
</tr>
<tr>
<td>1024</td>
<td>6.12 sec</td>
</tr>
<tr>
<td>2048</td>
<td>40 sec</td>
</tr>
</tbody>
</table>
Grantham-Frobenius

This is a test based on Frobenius pseudoprimes developed by Grantham which is passed by Composite Numbers with probability at most 1/7710. Algorithm is presented below.

1. Choose pairs \( (b,c) \) at random with \( 1 \leq b, c < n \) until one is found with \( \left(\frac{b^2+4c}{n}\right) = -1 \) and \( \left(\frac{-c}{n}\right) = 1 \), or with \( \gcd(b^2+4c, n) \), \( \gcd(b, n) \) or \( \gcd(c, n) \) a nontrivial divisor of \( n \). If latter case occurs before the former, declare \( n \) to be composite and stop. If after \( B \) pairs are tested, none is found satisfying the above conditions, declare \( n \) to be a probable prime and stop (we need to do this since we can’t declare prime as composite at any case).

2. Perform Quadratic Frobenius Test
   1) Test \( n \) for divisibility by primes less than or equal to \( \min \{B, n^{0.5}\} \). If it is divisible by one of these primes, declare \( n \) to be composite and stop.
   2) Test whether \( n^{0.5} \in \mathbb{Z} \). If it is declare \( n \) to be composite and stop.
   3) Compute \( x^{(n+1)/2} \mod (n, x^2 - bx - c) \). If \( x^{(n+1)/2} \not\in \mathbb{Z}/n\mathbb{Z} \), declare \( n \) to be composite and stop.
   4) Compute \( x^{n+1} \mod (n, x^2 - bx - c) \). If \( x^{n+1} \neq -c \), declare \( n \) to be composite and stop.
   5) Let \( n^2 - 1 = 2^s s \), where \( s \) is odd. If \( x \equiv 1 \mod(n, x^2 - bx - c) \), and \( x^{2^{j}s} \not\equiv -1 \mod (n, x^2 - bx - c) \) for all \( 0 \leq j \leq r-2 \), declare \( n \) to be composite and stop.

3. If \( n \) is not declared composite in Steps 1-5, declare \( n \) to be a probable prime.
Generating Prime Numbers

To generate n-bit prime, we first generate a random number of n-bit \((2^n+1 < x < 2^n)\). Starting from that point, we implement either a sequential or random search. Here we present a sequential approach.

Algorithm consists of following

1. Generate an odd random number of n-bit
   \[ x = \text{one.shiftLeft}( n ) \quad \text{//} \quad 2^n \]

   //generate random number uniformly distributed on \([0, 2**\text{numBits} - 1]\)
   \[ \text{rand\_num} = \text{new BigInteger}( n, \text{rndSrc}) \]
   \[ x = x.\text{add}(\text{rand\_num}) \]

   //make sure it’s an odd number
   if (x.testBit(0))
     //ok
   else
     x = x.\text{add}( \text{one} )

2. Test for Primality

3. If not prime, increment by two(or by random even number) and test for primality again

For primes of bit length n, mean gap distance is just \(\ln(2^n) = n \ln(2)\)
Therefore on the average, for exhaustive search of odd numbers between two primes \(n* \ln(2)\).
But we note that since our starting point is random, we assume that on the average, we land in the middle of two primes. Therefore, we need to check \(n*\ln(2)/2\) numbers on the average.

Here we do an experiment for 130, 256, 512, and 1024 bits

Number of Tests required(Theoretical)

<table>
<thead>
<tr>
<th>bit(# of primes)</th>
<th>256 bit(20)</th>
<th>512 bit(2)</th>
<th>1024 bit(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td># Tested</td>
<td>Time</td>
<td># Tested</td>
</tr>
<tr>
<td>MillerRabin(20)</td>
<td>35</td>
<td>1869</td>
<td>73</td>
</tr>
<tr>
<td>Lucas-Lehmer</td>
<td>43</td>
<td>2488</td>
<td>70</td>
</tr>
</tbody>
</table>

Normalizing for single prime,

<table>
<thead>
<tr>
<th>bit(# of primes)</th>
<th>256 bit</th>
<th>512 bit</th>
<th>1024 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td># Tested</td>
<td>Time</td>
<td># Tested</td>
</tr>
<tr>
<td>MillerRabin(20)</td>
<td>1.75</td>
<td>94</td>
<td>36.5</td>
</tr>
<tr>
<td>Lucas-Lehmer</td>
<td>2.15</td>
<td>125</td>
<td>35</td>
</tr>
</tbody>
</table>

* were we very unlucky?(theoretical states that we need to check substantially less numbers)
In the code, we have a fixed seed for Random number generator
```
Random rndSrc = new Random(5);
```
Therefore, each time we run the experiment, it generates same random number.
So, why there’s a discrepancy in number of Numbers tested to find primes? For 256 and 1024 bit cases, we see that Lucas-Lehmer requires more numbers. Since we are generating same random number for each test(MillerRabin and Lucas-Lehmer), we conclude that some numbers that was declared as “probable prime” by Miller-Rabin(20) was found to be Composite by Lucas-Lehmer, thus yielding to more # Tested. Therefore, we can see that Lucas-Lehmer was more accurate than Miller-Rabin(20)

Now, let’s compare the speed of these algorithms. We know that Lucas-Lehmer is faster to test a Prime number(that is if the number is a prime number); however, if a number was not declared a Composite on the first iteration of Miller-Rabin, we perform Lucas Test for Lucas-Lehmer.
Lucas Test typically takes approximately time it takes for 6 iterations of Miller-Rabin. Therefore, Time spent on Lucas-Lehmer test is equivalent to 7 iterations of Miller-Rabin.

However, as shown on the table, Lucas-Lehmer test was not necessarily faster. This is because if a number is found to be composite less than two iterations of Miller-Rabin, we stop there. Nevertheless, with Lucas-Lehmer test, if a number passes first iteration of Miller-Rabin, we continue with Lucas Test, which takes about same time as 6 iterations of Miller-Rabin.
Therefore, actual speed depends on following factors.

1. Number of Composites that fail one iteration of Miller-Rabin
   - This definitely will speed up both algorithms; however, it does not affect our comparison of Lucas-Lehmer and Rabin-Miller.

2. Number of Composites that takes 2-5 iterations of Miller-Rabin before being declared Composite
   - if we have more numbers that falls in this category, clearly Miller-Rabin algorithm is better.

3. Number of Composites that takes more than 6 iterations of Miller-Rabin before being declared Composite.
   - in this case, Lucas-Lehmer is a better choice
class PROJECT590 {

final static int MillerRabin = 0;
final static int LucasLemer = 1;

public static void main( String argv[] ) {
    int bitLength = 256;
    int typeOfMethod = LucasLemer;
    int mrlIteration = 20;
    int count = 0, i = 0;
    int numPrimes = 20;

    // make this random deterministic so that repeated experiment
    // yields same result
    Random rndSrc = new Random(5);
    BigInteger one = new BigInteger("1"), two = new BigInteger("2");
    BigInteger x, rand_num, start;

    MillerRabin MR;
    LucasTest LT;

    start = one.shiftLeft( bitLength );  //start = 2^n

    while (count <= numPrimes ) {
        rand_num = new BigInteger( bitLength, rndSrc);
        x = start.add(rand_num);

        if (!x.testBit(0))  x = x.add( one );

        // now look for primes
        while (true) {
            if (typeOfMethod == MillerRabin) {
                i++;
                MR = new MillerRabin( x, mrlIteration );
                if (MR.isPrime() ) {
                    //System.out.println(x.toString() + " is a prime");
                    break;
                }
            } else {
                i++;
                MR = new MillerRabin( x, 1 );
                if (MR.isPrime() ) {
                    LT = new LucasTest( x );
                    if (LT.isPrime() ) {
                        //System.out.println(x.toString() + " is a prime");
                        break;
                    }
                }
            }
        }
    }
}
x = x.add( two );
count ++;
System.out.println( i + " number tested to find " + numPrimes + " primes ");
import java.math.*;
import java.util.*;

class MillerRabin {
    final int COMPOSITE = 0, PRIME = 1;
    final BigInteger _one = new BigInteger("1");

    BigInteger testVector;
    BigInteger _r;
    int _s;

    private int _isPrime;

    /**
     * constructs a class MillerRabin with a number to be tested with iteration
     */
    public MillerRabin(BigInteger n, int t) {
        int result;
        testVector = n;
        result = start(n, t);
        _isPrime = result;
    }

    /**
     * getNumber() returns number that is tested using MillerRabin
     */
    public BigInteger getNumber() {
        return testVector;
    }

    protected int start(BigInteger n, int t) {
        BigInteger y, n_minus_one, a, r;
        int i, j;
        int s;

        findS_R( n );
        // n - 1 = 2^r x s

        s = _s;
        r = _r;

        y = new BigInteger("0");
        n_minus_one = n.subtract(_one);

        for (i=1;i<=t;i++) {
            //pick a;
        }
    }

    //private methods will go here
}
// creates odd big integer {2,...,n-1}
a = createOddBigInteger(n);

y = a.modPow(r, n);
if ( !(y.intValue() == 1) && !(y.equals(n_minus_one)) ) {
    j = 1;
    while ( (j <= s-1) && !(y.equals(n_minus_one))) {
        y = y.multiply(y);
        y = y.mod(n);
        if (y.intValue() == 1) return COMPOSITE;
        j++;
    }
    if ( !(y.equals(n_minus_one)) ) return COMPOSITE;
}
}
return PRIME;

/**
 * isPrime() returns true if probable prime
 * else it returns false
 */
public boolean isPrime() {
    if (_isPrime == PRIME)
        return true;
    else
        return false;
}

public BigInteger createOddBigInteger( BigInteger n) {
    BigInteger result = new BigInteger("0");
    int numBits;
    boolean isOdd = false;
    Random rndSrc = new Random();

    numBits = n.bitLength();

    while (!isOdd) {
        result = new BigInteger(numBits, rndSrc);
        // Make sure it's an odd number
        if (result.getLowestSetBit() == 0)
            // make sure it's greater than 2 and less than n
            if ( (result.longValue() > 2) && (result.compareTo(n) == -1) )
                isOdd = true;
    }

    return result;
}
protected void findS_R(BigInteger n) {
    // temp = n-1
    BigInteger temp = n.subtract(_one);
    int length, i;

    length = temp.bitLength();

    // ex. 1011100 i = 2
    i = temp.getLowestSetBit();

    // should be true all the time as long as it's even number
    if (i < 1) {
        System.out.println("huh: " + temp.toString());
        System.out.println("Critical Error, \n is not an even number");
    }

    _s = i;

    // ex. i = 2, r = 10111
    _r = temp.shiftRight(i);
}
}
Lucas Test

import java.math.*;

class LucasTest {
    final int COMPOSITE = 0, PRIME = 1;
    final BigInteger _zero = new BigInteger("0");
    final BigInteger _one = new BigInteger("1");
    final BigInteger _two = new BigInteger("2");
    final BigInteger _three = new BigInteger("3");
    final BigInteger _four = new BigInteger("4");
    final BigInteger _five = new BigInteger("5");
    final BigInteger _seven = new BigInteger("7");
    final BigInteger _eight = new BigInteger("8");
    final BigInteger _negativeOne = new BigInteger("-1");

    BigInteger _u, _v, testVector;
    boolean _isPrime;

    public LucasTest() {
    }

    /**
     * returns number being tested
     */
    public BigInteger getNumber() {
        return testVector;
    }

    public LucasTest(BigInteger n) {
        BigInteger d, p, q, one, four, temp;
        int result;

        testVector = n;
        d = findValueOfD( n );
        System.out.println("d = " + d.toString());

        q = new BigInteger("0");
        p = new BigInteger("1");

        //q = 1-d/4
        q = q.subtract(d);
        q = q.divide(_four);

        d = p.multiply(p);
        temp = q.multiply(_four);
    }
}
d = d.subtract(temp);
result = findUk(p,q,d,n);
if (result == COMPOSITE )
   _isPrime = false;
else
   _isPrime = true;
}
/**
* isPrime() returns true if it passes Lucas test
* otherwise, returns false
*/
public boolean isPrime() {
   return _isPrime;
}
protected BigInteger findValueOfD(BigInteger n) {
   int i = 1;
   Integer candidate = new Integer(5);
   BigInteger d;
   BigInteger temp;
   //generate candidate d
   d = new BigInteger( candidate.toString() );
   while ( true ) {
      temp = findJacobi3(d, n);
      if (temp.intValue() == -1) {
         return d;
      }
      i = i*(-1);
      candidate = new Integer( (candidate.intValue()*i*(-1) + 2) * i );
      d = new BigInteger( candidate.toString() );
   }
}
protected int findUk(BigInteger p, BigInteger q, BigInteger d, BigInteger n) {
   BigInteger u, v, temp, k;
   int r, i;
   k = n.add(_one);
   r = k.bitLength();
   // u =1, v = p
   u = new BigInteger("1");
   v = new BigInteger("0");
   v = v.add(p);
   //System.out.println("u =" + u.toString());
   //System.out.println("v =" + v.toString());
_u = u;
_v = v;

for(i=r-2; i >= 0; i--) {
    setUV( d, p, n);
    if (k.testBit(i)) {
        setUV2(d, p, n);
    }
}

u = _u;
v = _v;

temp = u.mod(n);
if (temp.intValue() == 0)
    return PRIME;
else
    return COMPOSITE;
}

protected void setUV( BigInteger d, BigInteger p, BigInteger n ) {
    BigInteger u, v, u_o, v_o, temp, invTwo;

    invTwo = _two.modInverse( n );

    u = _u;
v = _v;
    // Save original values
    u_o = new BigInteger("0");
v_o = new BigInteger("0");
    u_o = u_o.add(u);
v_o = v_o.add(v);
    u = u.multiply(v);
    u = u.mod(n);
    v = v.multiply(v);
    temp = d.multiply(u_o);
    temp = temp.multiply(u_o);
    temp = temp.add(v);
    temp = temp.multiply( invTwo );
    temp = temp.mod(n);
    v = temp.mod(n);

    _u = u;
    _v = v;
}
protected void setUV2( BigInteger d, BigInteger p, BigInteger n ) {
  BigInteger u, v, u_o, v_o, temp, invTwo;

  invTwo = _two.modInverse( n );
  u = _u;
  v = _v;
  // Save original values
  u_o = new BigInteger("0");
  v_o = new BigInteger("0");
  u_o = u_o.add(u);
  v_o = v_o.add(v);

  u = u.multiply(p);
  u = u.add(v_o);
  u = u.multiply( invTwo );
  u = u.mod(n);

  v = v.multiply(p);
  temp = d.multiply(u_o);
  temp = temp.add(v);
  temp = temp.multiply( invTwo );
  v = temp.mod(n);

  _u = u;
  _v = v;
}

public BigInteger findJacobi3(BigInteger a, BigInteger n) {
  BigInteger j;
  BigInteger x, y, temp;

  temp = a.gcd(n);

  if (a.gcd(n).intValue() > 1) {
    return _zero;
  }

  j = _one;

  x = a;
  y = n;

  while (true) {
    temp = y.abs();
    x = x.mod(temp);
    temp = y.divide(_two);

    if ( x.compareTo(temp) == 1) {
x = y.subtract(x);
if (y.mod(_four).intValue() == 3) {
    j = j.negate();
}
}
while (true) {
    if (x.remainder(_four).intValue() == 0) {
        x = x.divide(_four);
    } else
        break;
}
if (x.testBit(0) == false) {
    x = x.shiftRight(1);
    if ( (y.mod(_eight).intValue() == 3) || (y.mod(_eight).intValue() == 5) ) {
        j = j.negate();
    }
}
if (x.intValue() == 1)
    return j;
if ( (x.mod(_four).intValue() == 3) && (y.mod(_four).intValue() == 3) ) {
    j = j.negate();
}

public BigInteger findJacobi2(BigInteger a, BigInteger n) {
    BigInteger j;
    BigInteger temp, temp1, temp2;
    j = _one;
    while (a.intValue() != 0) {
        temp = a.mod(_two);
        while (temp.intValue()==0) {
            a = a.divide(_two);
            temp1 = n.mod(_eight);
            temp = a.mod(_two);
        }
if ( (temp1.compareTo(_three) == 0) || (temp1.compareTo(_five) == 0) )
    j = j.negate();
}
temp = a;
a = n;
n = temp;

temp1 = a.mod(_four);
temp2 = n.mod(_four);

if ( (temp1.intValue() == 3) && (temp2.intValue() == 3) )
    j = j.negate();
a = a.mod(n.abs());
}

if (n.intValue() == 1)
    return (j);
else
    return(_zero);
}
Test Vectors for 130 bit

1200161307614228132587124344911319888847
1217061394045440770272863028410128226539
921108606988933429616620315810400467517
8533175303701040124095985275222730700651
680852455832286412234959266339294171989
7558747216136808683503755580762472058163
708845233379365237138514321248709595271
970579120976007047606697988455111051307
1078975318520442677247975877914475852561
132066756566554595872386562961365612402689
718325227243164098105241005534129831503
1346998127308036333796416179976314235187
121072844350135619623535316310907871307
1284329995011366362345904345674796615737
1131332641316196878738236781313416166959
68329680599084781280212233425096120853
1092286847468566817841840658247701405267
Test Vectors for 512 bit

1340601848582718226105106124561910107966927382381693505223898685801514051846334874516
1181977871257356339625265114480545265852708725241237818621057413123
1111043076569827739723228137506514700911488119243320945040666652913879569977
99786764934782509208259688325961015654276900869681598598040697716077
11806612758985307653174860307043541727168872169912880085110125181788315008169041939158
9904619536696132548287020645696062398554829340010797727855517679577
7636810453037092894669673697932391829579255234764917573412969410347060681575840312688
26454733373678255348947228170666923745117283266045509961151462316057
821710873554763324892845383072732659728100168728824109830952415403711946066296016313
22285074660331028910462762507189124859650982129406755093565601585947
12266764785165030162729387081296720993472143161693170466820779003275526416314273360003
1365622188540606714218777940555755353782878699241474842871472870249733
11728828176947869125905969923079452166640964198014787150461606701830705115895087461910
819710471416920813507088644745902523556045575123057112934782264425687
7724237022373706895281247525279827689805491420383621463993732717430696633093986804006
279609765689689551488531832343322768141670920172192422542577977
6760636969162029518661036975538549719080504571945441058014561650112007764003636622421
242418467842021753596876236000861199801529844122830030450903077015403
11773872679029718988610133166785963818345265677642697567491661823113308097132227502935
2787554713531245578516528117406660700198190473198996500712126545367
8855476145908697845903733451136077125319384965963019746363826630694031941361246737261
8547647765939597731592418724167151856937283706143788223282514554017
792405211236290298963167181481746795436779314844977164250748164227512994730667900560
6061741559693008004004586971230569245975919083221020277131493149584291
9229175486247271498082069808442811393404730445338998404950033929249251889341228335305
8592748365446693889649566082660748379550793255865527242121281
750159256418765532119136847067336832480885853133994678375243316471290054066846813882206
008969137047587600414525877269085583130556114395631972053194867150397
TestVector for 1024 bit(5 vectors)

94366396730334173383107353049414959521528815310548187030165936229578960209523421808912
4597953290352035028457618716007638664370044121654773291425057893426189151082714026704
35920072251607983489136394725647150554452015124613593594887954278755302310012985524522
30535485049737222714000227878890892901228389026881
14104232639093549409335498262110844478751699451363766872997477735559175907393036116352898
20809386404512851073393944150388901008444463883117113952707223795554404489449244451218
796471568843847699082688100302117700762900374621813907773379090919031429196100235579610
612355809815098951461960436805335178146468594497053
11360610111179306388223948010135385010447915110037709330181026893453973249232168998103
2567226622623957153523225328631881081241616212270943207600497544632660747936595524811
48466692479681327569567487876744209771481314457618208860179484415611981619594638107773
332491673281796678206353605533616952366511645578797
1082263414638548112856407409747600383328757987915507351632541215199296812089211097475
72617088142464679350893510811656701414129900899834374838063259388920803833943031173169
3079285323717366805638854697019375558599072778395212385117402656093740233106410211387
525683987973785065561609104502259722551424163822821
1319875765832390743517803154228613910058637331985021843908406178692896645380363964883
6683448848489044397822815653937429267029614303661153543788078182474076893164007732138
61885428867838814872547009946101026907624284479881920021341634731228098247065866684397
494312971820820784312389551449748365108896731559199
**TestVector for 2048 bit (5 vectors)**

```
22193271413026737908225156813220050145533382453698983118475999190643145336265614540013
36985846720293048796874695840111749678729693926675089998163784991236421460825398914
760145255110732541507660029294649905810063441539256875828520115880877962857723409378
4731887124693001826357104449598665241105572379090532405030183276218339444330312933640
78923011068687774053161875187390687335103640385968781285091513028511224579425656992568
7493430047570618389936192163128920273294467674744028218644466050793120248014204949931
2669635527947982053663808414577501315887149939111376419253561680572656612491920302176
0596842540906949
```

... (remaining lines of the text continue in a similar manner)