Homework 3

Implementation of Montgomery Multiplication

Montgomery multiplier calculates C = A . B mod N as follows:

1. Given integers x, y and $0 \le x,y \le m$
2. Montgomery Domain:
   a. $\tilde{x} = xR \bmod m$, $\tilde{y} = yR \bmod m$
3. Montgomery Multiplication:
   a. $\tilde{z} = \tilde{x}\tilde{y}R^{-1} \bmod m = xR(yR)R^{-1} \bmod m = xyR \bmod m$
      i. Montgomery Reduction of product $\tilde{x} * \tilde{y}$
      ii. Defined as $MP(\tilde{x},\tilde{y})$
4. Result: $z = \tilde{z}R^{-1} \bmod m = (zR)R^{-1} \bmod m$
5. Compute Montgomery Domain of x using $MP(x, R^2 \bmod m)$
6. Compute Result z using MP(z, 1)

For example:

1. m = 119 (odd value), x = 63, y = 57
   a. 63(57) mod 119 = 21
2. Montgomery Multiplication:
   a. R > m, choose $2^7 = 128$
   b. $R^2 \bmod m = 81$, $R^{-1} \bmod m = 53$
   c. Montgomery Domain:
      $\tilde{x} = xR \bmod m = MP(x, R^2 \bmod m)$
      $= 63(81)\ (53) \bmod 119 = 91$
      $\tilde{y} = yR \bmod m = MP(y, R^2 \bmod m)$
      $= 57(81)\ (53) \bmod 119 = 37$
   d. Montgomery Multiplication:
      $\tilde{z} = MP(91, 37) = 91(37)\ (53) \bmod 119 = 70$
   e. Result: $z = MP(\tilde{z}, 1) = 70(1)\ (53) \bmod 119 = 21$

Following code can coompute montgomery multiplication of 128 bit numbers:

Results are:

```
A: 270902565107982796159539501779943026603 -> 128 Bit Number
B: 251052638676463176992136739297869888188 -> 128 Bit Number
N: 272320286609021830958597652262355401393 -> Modulus
R: 179795676119011098898159522850739282410 -> Result of A x B mod N
```

```
import java.math.BigInteger;

public class Program
{
	private static final BigInteger ONE = new BigInteger("1");

	public static void main(String[] args)
	{
	  String hexStrA = "CBCDEFABCDEFABCDEFABCDEFABCDEFAB";//32 x 4 bits = 128
bits
	  BigInteger A = new BigInteger(hexStrA, 16);
	  System.out.println("A: " + A);

	  String hexStrB = "BCDEFABCDEFABCDEFABCDEFABCDEFABC";//32 x 4 bits = 128
bits
	  BigInteger B = new BigInteger(hexStrB, 16);
	  System.out.println("B: " + B);

	  String hexStrN = "CCDEFABCDEFABCDEFABCDEFABCDEFAB1";//32 x 4 bits = 128
bits
	  BigInteger N = new BigInteger(hexStrN, 16);
	  System.out.println("N: " + N);

		BigInteger Result = montgomeryMult(A, B, N);
		System.out.println("R: " + Result);
	}

	private static BigInteger montgomeryMult(BigInteger A, BigInteger B,
BigInteger N)
	{
		BigInteger RSquareModM = ChooseR(N);
		BigInteger R_I = (RSquareModM.multiply(RSquareModM)).modInverse(N);
		BigInteger montX = MontgomeryMultiplier(A,RSquareModM,R_I,N);
		BigInteger montY = MontgomeryMultiplier(B,RSquareModM,R_I,N);
		BigInteger montZ = MontgomeryMultiplier(montX,montY,R_I,N);
		BigInteger Z = MontgomeryMultiplier(montZ,ONE,R_I,N);
		return Z;
	}

	private static BigInteger MontgomeryMultiplier(BigInteger a, BigInteger
rSquareModM, BigInteger r_I,BigInteger m)
	{
		return ((a.multiply(rSquareModM)).multiply(r_I)).mod(m);
	}

	private static BigInteger ChooseR(BigInteger m)
	{
		BigInteger R = ONE;

		BigInteger Result = ONE;
		for(int i=0;i<m.bitLength();i++)
		{
			R = R.shiftLeft(1);
		}
		R = (R.multiply(R)).mod(m);
		return Result;
	}

}
```

Implementation of RSA for 128 Bit Modulus, N and Choosing Two 64-Bit Prime Number "p" and "q" and a Public Key, "e" and Computing the Corresponding Private Key, "d"

Firstly, p and q prime numbers are chosen and calculated modulus "n" by using them:

$$n = p\ x\ \text{q and}\ \ \varphi(n) = (p-1)\ x\ (\text{q}-1)$$

An "e" prime number is chosen by using (n) . "e" is the public key.

"d" which is the private key is calculated by using formula:

$$d = e^{-1} mod \varphi(n)$$

Messages are encrypted by using:

$$c = m^e mod\ n$$

Ciphered messages are decrypted by using:

$$m = c^d mod\ n$$

Following code is the implementation of the RSA:

Results are:

```
p:                  14545081761063098339
q:                  12070297673976155981
modulus:            175563466548352926442597621509006015559
publicKey:          65537
privateKey:         144904085211918557390379654824984513313
message:            sended by RSA
message:            914263013107949659769258130 3105
encryptBirol:       731892722298932608956033481257147993 05
encryptedMessage:   7_½¸³)gÁ_C«4ë<ÚÉ
decryptBirol:       914263013107949659769258130 3105
decryptedMessage:   sended by RSA
```

```
package RSA;

import java.math.BigInteger;
import java.security.SecureRandom;

public class Main
{
	private final static BigInteger ONE   = new BigInteger("1");

	static SecureRandom random = new SecureRandom();

	static BigInteger privateKey;
	static BigInteger publicKey;
```

```
	static BigInteger modulus;

	public static void main(String[] args)
	{
		BigInteger message, encryptedMessage,decryptedMessage;

		String testMessage = "sended by RSA";
		byte[] testMessageBytes = testMessage.getBytes();
		BigInteger testMessageInteger = new BigInteger(testMessageBytes);

		GenerateKeys(128);
		System.out.println("message:	" + testMessage);
		System.out.println("message:	" + testMessageInteger);
		encryptedMessage = encryptBirol(testMessageInteger);
		System.out.println("encryptedMessage: " +new
String(encryptedMessage.toByteArray()));
		decryptedMessage = decryptBirol(encryptedMessage);
		System.out.println("decryptedMessage: " + new
String(decryptedMessage.toByteArray()));	}


	/**
	 * @param message
	 * @param exponent.
	 * @param n modulo.
	 * @return message^exponent mod modulus.
	 **/
	public static BigInteger moduloPower(BigInteger message, BigInteger
exponent, BigInteger modulus)
	{
		BigInteger r = ONE;

		for(int i=exponent.bitLength()-1; i>=0; i--)
		{
			r = montgomeryMult(r,r,modulus);
			if (exponent.testBit(i))
				r = montgomeryMult(r, message, modulus);
		}

		return r;
	}

	/**
	 * @param N integer
	 * @creates key
	 **/
	public static void GenerateKeys(int N)
	{
		// choose two primes - p and q
		BigInteger p = new BigInteger("14545081761063098339");
		BigInteger q = new BigInteger("12070297673976155981");
		// calculate phi
		BigInteger phi = (p.subtract(ONE)).multiply(q.subtract(ONE));
		//calculate modulus
		modulus = p.multiply(q);
		//choose a public key e w.r.t phi
		publicKey = new BigInteger("65537"); // 2^16 +1;
		privateKey = publicKey.modInverse(phi);
```

```
            System.out.println( "p:       " + p );
            System.out.println( "q:       " + q );
            System.out.println( "modulus:       " + modulus );
            System.out.println( "publicKey:     " + publicKey );
            System.out.println( "privateKey:    "+ privateKey );
      }

      /**
       * @param message
       * @return encrypted message
       **/
      public static BigInteger encryptBirol(BigInteger message)
      {
            BigInteger result = moduloPower(message,publicKey,modulus);
            System.out.println("encryptBirol: " + result);
            return result;
      }

      /**
       * @param encrypted message
       * @return decrypted message
       **/
      public static BigInteger decryptBirol(BigInteger encryptedMessage)
      {
            BigInteger result = moduloPower(encryptedMessage, privateKey,
modulus);
            System.out.println("decryptBirol: " + result);
            return result;
      }

      /**
       * @param A
       * @param B.
       * @param N.
       * @return C = A x B mod N
       **/
      private static BigInteger montgomeryMult(BigInteger A, BigInteger B,
BigInteger N)
      {
            BigInteger RSquareModM = ChooseR(N);
            BigInteger R_I = (RSquareModM.multiply(RSquareModM)).modInverse(N);
            BigInteger montX = montgomeryDomain(A,RSquareModM,R_I,N);
            BigInteger montY = montgomeryDomain(B,RSquareModM,R_I,N);
            BigInteger montZ = montgomeryDomain(montX,montY,R_I,N);
            BigInteger Z = montgomeryDomain(montZ,ONE,R_I,N);
            return Z;
      }

      /**
       * @param a
       * @param R^2 mod M
       * @param R'
       * @param m
       * @return Number that in Montgomery domain
       **/
      private static BigInteger montgomeryDomain(BigInteger a, BigInteger
rSquareModM, BigInteger r_I,BigInteger m)
      {
```

```
            return ((a.multiply(rSquareModM)).multiply(r_I)).mod(m);//A R^2 mod M
x R' mod M
      }

      /**
       * @param m
       * @return R
       **/
      private static BigInteger ChooseR(BigInteger m)
      {
            BigInteger R = ONE;

            BigInteger Result = ONE;
            for(int i=0;i<m.bitLength();i++)
            {
                  R = R.shiftLeft(1);
            }
            R = (R.multiply(R)).mod(m);
            return Result;
      }
}
```

Encrypt and Decrypt Some Messages to Use Them in Common Modulus

To avoid generating a different modulus N = pq for each user one may wish to fix N once for all. The same N is used by all users. A trusted central authority could provide user “i” with a unique pair “ei,di” from which user “i” forms a public key(N, ei) and a secret key(N,di).

A ciphertext C = m^ea mod N indented for Alice cannot decrypted by Bob since Bob does not possess da. However, this is incorrect and resulting the system is insecure. Bob can use his own exponents eb and db to factor the modulus N. Once N is factored Bob can recover Alice’s private key da from her public key ea. This observation, due to Simmons, shows that an RSA modulus should never be used by more than one entity.

Let’s server send Alice that ciphered message:

$$y1 = m^b mod\ n$$

Let’s server send Bob that ciphered message:

$$y2 = m^c mod\ n$$

Then it can be sait that:

$$d = b^{-1} mod\ c$$

$$e = (db - 1)/mod\ c$$

Then calculate:

$$y1^d (y2^e)^{-1} mod\ n = \left(x^{bd}\right)(x^{ce})^{-1} mod\ n = x^{bd-ce} mod\ n = x\ mod\ n$$

Therefore, obtaining the message without factoring the modulus is possible.

Following Code simulates that Server send same message with same MODULUS, and Oscar listens the encrypted messages of Alice and Bob. Althoug he only knows public keys of the Alice and Bob, he can listen whatever Server tells to Alice and Bob:

Results are shown first, Server Says : `Hello Alice and Bob How Are You Today I have a Secret Message Do not Tell To Oscar`

```
message:      Hello
message:      310939249775
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 98209189237204485507999883982020794962
decryptBirol: 310939249775
Alice's decrypted Message: Hello
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 135785735749386095924839275936215343789
decryptBirol: 310939249775
Bob's DecryptedMessage: Hello
-----------------------
Oscar Listens: 310939249775
Oscar Listens: Hello
```

```
message:      Alice and Bob
message:      518338235107868297006378978083 4
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 164376045300816567038492641461653725618
decryptBirol: 518338235107868297006378978083 4
Alice's decrypted Message: Alice and Bob
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 438344133951725753681595693428823392 08
decryptBirol: 518338235107868297006378978083 4
Bob's DecryptedMessage: Alice and Bob
-----------------------
Oscar Listens: 518338235107868297006378978083 4
Oscar Listens: Alice and Bob
```

```
message:      How Are You
message:      875690391786612639940893333
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 126807372247174814540926816854539372824
decryptBirol: 875690391786612639940893333
Alice's decrypted Message: How Are You
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   405146461265429830190728250939923295l7
-----------------------
encryptBirol: 806589514798206658954954714561835237l7
decryptBirol: 875690391786612639940893333
Bob's DecryptedMessage: How Are You
-----------------------
Oscar Listens: 875690391786612639940893333
Oscar Listens: How Are You


message:      Today
message:      362646102393
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 340237627036511170420924993074101000l8
decryptBirol: 362646102393
Alice's decrypted Message: Today
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   405146461265429830190728250939923295l7
-----------------------
encryptBirol: 300182388105299198886471334496339868888
decryptBirol: 362646102393
Bob's DecryptedMessage: Today
-----------------------
Oscar Listens: 362646102393
Oscar Listens: Today


message:      I have a
message:      5269326331830935649
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 106094465012213434654852287538551019309
decryptBirol: 5269326331830935649
Alice's decrypted Message: I have a
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
```

```
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 131163760962509158965120466736828435387
decryptBirol: 5269326331830935649
Bob's DecryptedMessage: I have a
-----------------------
Oscar Listens: 5269326331830935649
Oscar Listens: I have a

message:      Secret Message
message:      169147281882964625722461289194480 5
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 145088680795590543245158393151662942323
decryptBirol: 169147281882964625722461289194480 5
Alice's decrypted Message: Secret Message
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 371097042943089607467626045336977532 60
decryptBirol: 169147281882964625722461289194480 5
Bob's DecryptedMessage: Secret Message
-----------------------
Oscar Listens: 169147281882964625722461289194480 5
Oscar Listens: Secret Message

message:      Do not Tell
message:      827317366669104219950562 36
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 291466599829527000589901140662105276 94
decryptBirol: 827317366669104219950562 36
Alice's decrypted Message: Do not Tell
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 151156468889118027219168735445819145313
decryptBirol: 827317366669104219950562 36
Bob's DecryptedMessage: Do not Tell
-----------------------
Oscar Listens: 827317366669104219950562 36
Oscar Listens: Do not Tell


message:      To Oscar
message:      6084117147211227506
-----------------------
Alice's modulus:      175563466548352926442597621509006015559
Alice's publicKey:    11
```

```
Alice's privateKey:   127682521126074855575259812417430371811
-----------------------
encryptBirol: 147804044953158052305695490182095670746
decryptBirol: 6084117147211227506
Alice's decrypted Message: To Oscar
-----------------------
Bob's modulus:      175563466548352926442597621509006015559
Bob's publicKey:    13
Bob's privateKey:   40514646126542983019072825093992329517
-----------------------
encryptBirol: 172949227208917769625034939992740530048
decryptBirol: 6084117147211227506
Bob's DecryptedMessage: To Oscar
-----------------------
Oscar Listens: 6084117147211227506
Oscar Listens: To Oscar
```

```
package RSA;

import java.math.BigInteger;
import java.security.SecureRandom;

public class Main
{
	private final static BigInteger ONE   = new BigInteger("1");

	private static final int Alice = 1;
	private static final int Bob = 2;

	static SecureRandom random = new SecureRandom();

	static BigInteger privateKey;
	static BigInteger publicKey;
	static BigInteger modulus;

	public static void main(String[] args)
	{
		BigInteger message,
encryptedMessageAlice,decryptedMessageAlice,encryptedMessageBob,decryptedMessageBo
b;

		//message = new BigInteger(128,random);

		String testMessage = "How";
		byte[] testMessageBytes = testMessage.getBytes();
		BigInteger testMessageInteger = new BigInteger(testMessageBytes);
	       //// create message by converting string to integer
	       // String s = "test";
	       // byte[] bytes = s.getBytes();
	       // BigInteger message = new BigInteger(s);

		System.out.println("message:      " + testMessage);
		System.out.println("message:      " + testMessageInteger);

		BigInteger AlicesKey = GenerateKeys(128,Alice);
		encryptedMessageAlice = encryptBirol(testMessageInteger);
		System.out.println("Alice's Encrypted Message: " +new
String(encryptedMessageAlice.toByteArray()));
```

```
            decryptedMessageAlice = decryptBirol(encryptedMessageAlice);
            System.out.println("Alice's decrypted Message: " + new
String(decryptedMessageAlice.toByteArray()));

            BigInteger BobsKey = GenerateKeys(128,Bob);
            encryptedMessageBob = encryptBirol(testMessageInteger);
            System.out.println("Bob's EncryptedMessage: " +new
String(encryptedMessageBob.toByteArray()));
            decryptedMessageBob = decryptBirol(encryptedMessageBob);
            System.out.println("Bob's DecryptedMessage: " + new
String(decryptedMessageBob.toByteArray()));
            System.out.println("-----------------------");
            BigInteger d = AlicesKey.modInverse(BobsKey);
            BigInteger e =
(d.multiply(AlicesKey).subtract(ONE)).divide(BobsKey);

            BigInteger Y1 =
encryptedMessageAlice.pow(d.intValue()).mod(modulus);
            BigInteger Y2 =
encryptedMessageBob.pow(e.intValue()).modInverse(modulus);
            BigInteger OscarListens = Y1.multiply(Y2).mod(modulus);

            System.out.println("Oscar Listens: " + OscarListens);
            System.out.println("Oscar Listens: " + new
String(OscarListens.toByteArray()));
      }



      /**
       * @param i
       * @param alice2
       */
      private static BigInteger GenerateKeys(int N, int index)
      {
            if(index == Alice)
            {
                  // choose two primes - p and q
                  BigInteger p = new
BigInteger("14545081761063098339");//BigInteger.probablePrime(N/2, random);
                  BigInteger q = new
BigInteger("12070297673976155981");//BigInteger.probablePrime(N/2,random);
                  // calculate phi
                  BigInteger phi = (p.subtract(ONE)).multiply(q.subtract(ONE));
                  //calculate modulus
                  modulus = p.multiply(q);
                  //choose a public key e w.r.t phi
                  publicKey = new BigInteger("11"); // 2^16 +1;
                  privateKey = publicKey.modInverse(phi);

                  System.out.println( "-----------------------");
                  //System.out.println( "Alice's p:      " + p );
                  //System.out.println( "Alice's q:      " + q );
                  System.out.println( "Alice's modulus:      " + modulus );
                  System.out.println( "Alice's publicKey:    " + publicKey );
                  System.out.println( "Alice's privateKey:   "+ privateKey );
                  System.out.println( "-----------------------");
                  return publicKey;
```

```
		}
		else if(index == Bob)
		{
			// choose two primes - p and q
			BigInteger p = new
BigInteger("14545081761063098339");//BigInteger.probablePrime(N/2, random);
			BigInteger q = new
BigInteger("12070297673976155981");//BigInteger.probablePrime(N/2,random);
			// calculate phi
			BigInteger phi = (p.subtract(ONE)).multiply(q.subtract(ONE));
			//calculate modulus
			modulus = p.multiply(q);
			//choose a public key e w.r.t phi
			publicKey = new BigInteger("13"); // 2^16 +1;
			privateKey = publicKey.modInverse(phi);

			System.out.println( "-----------------------");
			//System.out.println( "Bob's p:      " + p );
			//System.out.println( "Bob's q:      " + q );
			System.out.println( "Bob's modulus:      " + modulus );
			System.out.println( "Bob's publicKey:    " + publicKey );
			System.out.println( "Bob's privateKey:   "+ privateKey );
			System.out.println( "-----------------------");
			return publicKey;
		}
		else
		{
			throw new IllegalArgumentException();
		}

	}




	/**
	 * @param message
	 * @param exponent.
	 * @param n modulo.
	 * @return message^exponent mod modulus.
	 **/
	public static BigInteger moduloPower(BigInteger message, BigInteger
exponent, BigInteger modulus)
	{
		BigInteger r = ONE;

		/*for(int i=exponent.bitLength()-1; i>=0; i--)
		{
			r = (r.multiply(r)).mod(modulus);
			if (exponent.testBit(i))
				r = (r.multiply(message)).mod(modulus);
		}*/

		for(int i=exponent.bitLength()-1; i>=0; i--)
		{
			r = montgomeryMult(r,r,modulus);
			if (exponent.testBit(i))
```

```
                        r = montgomeryMult(r, message,
modulus);//(r.multiply(message)).mod(modulus);
                }

                return r;
        }

        /**
         * @param N integer
         * @creates key
         **/
        public static void GenerateKeys(int N)
        {
                // choose two primes - p and q
                BigInteger p = new
BigInteger("14545081761063098339");//BigInteger.probablePrime(N/2, random);
                BigInteger q = new
BigInteger("12070297673976155981");//BigInteger.probablePrime(N/2,random);
                // calculate phi
                BigInteger phi = (p.subtract(ONE)).multiply(q.subtract(ONE));
                //calculate modulus
                modulus = p.multiply(q);
                //choose a public key e w.r.t phi
                publicKey = new BigInteger("65537"); // 2^16 +1;
                privateKey = publicKey.modInverse(phi);

                System.out.println( "p:       " + p );
                System.out.println( "q:       " + q );
                System.out.println( "modulus:       " + modulus );
                System.out.println( "publicKey:     " + publicKey );
                System.out.println( "privateKey:    "+ privateKey );
        }

        /**
         * @param message
         * @return encrypted message
         **/
        public static BigInteger encryptBirol(BigInteger message)
        {
                BigInteger result = moduloPower(message,publicKey,modulus);
                System.out.println("encryptBirol: " + result);
                return result;
        }

        /**
         * @param encrypted message
         * @return decrypted message
         **/
        public static BigInteger decryptBirol(BigInteger encryptedMessage)
        {
                BigInteger result = moduloPower(encryptedMessage, privateKey,
modulus);
                System.out.println("decryptBirol: " + result);
                return result;
        }

        /**
         * @param A
         * @param B.
```

```
         * @param N.
         * @return C = A x B mod N
         **/
        private static BigInteger montgomeryMult(BigInteger A, BigInteger B,
BigInteger N)
        {
                BigInteger RSquareModM = ChooseR(N);
                BigInteger R_I = (RSquareModM.multiply(RSquareModM)).modInverse(N);
                BigInteger montX = montgomeryDomain(A,RSquareModM,R_I,N);
                BigInteger montY = montgomeryDomain(B,RSquareModM,R_I,N);
                BigInteger montZ = montgomeryDomain(montX,montY,R_I,N);
                BigInteger Z = montgomeryDomain(montZ,ONE,R_I,N);
                return Z;
        }

        /**
         * @param a
         * @param R^2 mod M
         * @param R'
         * @param m
         * @return Number that in Montgomery domain
         **/
        private static BigInteger montgomeryDomain(BigInteger a, BigInteger
rSquareModM, BigInteger r_I,BigInteger m)
        {
                return ((a.multiply(rSquareModM)).multiply(r_I)).mod(m);//A R^2 mod M
x R' mod M
        }

        /**
         * @param m
         * @return R
         **/
        private static BigInteger ChooseR(BigInteger m)
        {
                BigInteger R = ONE;

                BigInteger Result = ONE;
                for(int i=0;i<m.bitLength();i++)
                {
                        R = R.shiftLeft(1);
                }
                R = (R.multiply(R)).mod(m);
                return Result;
        }
}
```