RSA Algorithm

[Computer Network Security – Project 2]

Andrea Sancho

PROJECT DESCRIPTION

In this project, you will first learn how to generate large prime numbers followed by implementing the RSA algorithm. Since you will have to compute large exponents as part of the RSA algorithm, the standard integer types (int - 16 or 32 bits, long int - 32 bits and even long long int – 64 bits) of most computer languages will not suffice.

- I choose to work with Java, so I will be using the built-in BigInt that behaves exactly like an int, except that it is not bounded by a finite value.

PART 1: Prime Numbers

RSA requires finding large prime numbers very quickly. You will need to research and implement a method for primality testing of large numbers. There are a number of such methods such as Fermat's, Miller-Rabin, AKS, etc. Your task is to develop two programs (make sure you use your two programs to check each other).

The first program is called primecheck and will take a single argument, an arbitrarily long positive integer and return either True or False depending on whether the number provided as an argument is a prime number or not. You may not use the library functions that come with the language (such as in Java or Ruby) or provided by 3rd party libraries.

To do this I first created a Prime class. I added methods to this class on the way, as in the moment I needed them. All the programs will be using objects of this class (Prime numbers).

The documentation provided along with the codes contains the explanation of how it works, and what each method does.

Prime.java	**Andreas-MacBook-Air:RSA andrea$ javac Prime.java**

```
import java.math.*;
import java.util.Random;

/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.2
 * @since 04/01/2014
 */

public class Prime {

	public static final BigInteger TWO = BigInteger.ONE.add(BigInteger.ONE);
	private BigInteger n;
	private int b;

	/**
	 * Constructor used for testing primes
	 * @param BigInteger
	 */
	public Prime(BigInteger number) {
	  setN(number);
	}
	public BigInteger getN() {
	  return n;
	}
	public void setN(BigInteger number) {
	  this.n = number;
	}
	public Prime(int nbits) {
	  setB(nbits);
	}
	public int getB() {
	  return b;
```

```java
	}
	public void setB(int bits) {
	  this.b = bits;
	}


	/**
	 * Computes [base^(exponent)] (modulus n) using the binary exponent notation as shortcut.
	 * If n is a prime number, and base=n-1, then the result of this method will be 1.
	 *
	 * @param  BigInteger base,exp,mod
	 * @return BigInteger  [a^(n-1)] (mod n)
	 */
	public BigInteger expModn(BigInteger base, BigInteger exp, BigInteger mod) {
	  BigInteger x = BigInteger.ONE;
	  while (exp.compareTo(BigInteger.ZERO) > 0) {
		  BigInteger exp2 = exp.remainder(TWO);
		  if ((exp2.compareTo(BigInteger.ONE) == 0)) {
			  x = x.multiply(base);
			  x = x.remainder(mod);
		  }
		  base = base.multiply(base);
		  base = base.remainder(mod);
		  exp = exp.divide(TWO);
	  }
	  return x;
	}

	/**
	 * Picks any integer a between 2 and n-1 (inclusive). The precise integer
	 * picked isn't important. Also, since the parameters for [base] are inclusive, 2
	 * and n-1 themselves are both valid choices.
	 *
	 * @param Random
	 * @return BigInteger
	 */
	private BigInteger generateBase(Random r){
	  int low = 2;
	  BigInteger high = BigInteger.valueOf((int) Math.pow(2, 30) - low);
	  // We need to take into account the fact that Integer.MAX_VALUE
	  // may be less than the actual (n-1) value, which is a BigInteger.
	  // So we set the upper bound as the minimum of these two:
	  BigInteger minimum = high.min(getN().subtract(BigInteger.ONE));
	  int h = minimum.intValue();
	  int q = r.nextInt(h - low) + low;
	  return BigInteger.valueOf(q);
	}

	/**
	 * Check whether a^n-1 (mod n) = 1 If not, n is composite. If true, n is
	 * likely, (but not certainly) prime. See http://primes.utm.edu/index.html
	 *
	 * Repeating the test (trials times) with
	 * different values for a increase the probability in the outcome, though
	 * there are rare composite numbers that satisfy the Fermat condition for
	 * all values of a: Carmichael numbers.
	 *
	 * @param int number of trials
	 * @return true if the candidate set in n passes all tests
	 */
	public boolean testPrime(int trials) {
	  BigInteger mod = getN();
	  BigInteger exp = mod.subtract(BigInteger.ONE);
	  BigInteger base;
	  Random r = new Random();
	  for (int i = 0; i < trials; i++) {
		  base = generateBase(r);
		  if (expModn(base, exp, mod).compareTo(BigInteger.ONE) != 0) return false;
	  }
	  return testCarmichael();
	}

	/**
	 * Performs the Miller-Rabin test on a probable prime number
	 *
	 * This test is to detect false probable primes, called
	 * Carmichael numbers, which will pass the Fermat test.
	 * If x and n are positive integers such that x^2 = 1 (mod n)
	 * but x != _1 (mod n), then n is composite.
```

```
     *
     * @return true if the candidate is passes the test, so it is not
     * a Carmichael number
     */
    private boolean testCarmichael(){
      Random r = new Random();
      BigInteger mod = getN();
      BigInteger exp = TWO;
      BigInteger base = generateBase(r);
      if(expModn(base, exp, mod).compareTo(BigInteger.ONE) == 0){
              if(expModn(base, BigInteger.ONE, mod).compareTo(BigInteger.ONE) != 0) return false;
      }
      return true;
    }

    /**
     * Generates an initial BigInteger as k = (n-5)/6,
     * which is in the range fixed by the specified number of bits.
     *
     * @return BigInteger
     */
    public BigInteger initializeK(){
      int bits = getB();
      if (bits > 3) {
              BigInteger k = BigInteger.ONE.shiftLeft(bits-1).subtract(BigInteger.valueOf(5));
              k = k.divide(BigInteger.valueOf(6));
              return k.add(BigInteger.ONE);
      }else {
              return BigInteger.ZERO;
      }
    }
    /**
     * Checks if the candidate is in the ranged imposed by the number of
     * bits specified by the user.
     *
     * @param candidate BigInteger (candidate for prime number to evaualte)
     * @return true (when the number is in the range 2^[nbits-1] < candidate < 2^[nbits]
     */
    public boolean checkRange() {
      int nbits = getB();
      BigInteger max = BigInteger.ONE.shiftLeft(nbits);
      BigInteger min = BigInteger.ONE.shiftLeft(nbits - 1);
      if ((getN().compareTo(max) < 0) && (getN().compareTo(min) >= 0))
          return true;
      return false;
    }

    /**
     * Generates a number as n = 6k + 5,
     * this will be a prime candidate.
     *
     * The algorithm is conflictive when generating
     * prime numbers of less than 3 bits (because k = 0)
   * So I treated individually these three cases (bits = 1,2,3)
     *
     * @param BigInteger
     */
    public void generateCandidate(BigInteger k) {
      int bits = getB();
      // First check the conflictive cases
      if (bits<=3){
          // With 1 bit, the only prime is 1
              if(bits == 1) setN(BigInteger.ONE);
              // With 2 bits, the only primes are 2 and 3
          else if(bits == 2) {
              if(k.compareTo(BigInteger.ZERO) == 0) setN(TWO);
              else {
                  setN(TWO.add(BigInteger.ONE));
              }
          }
          // With 3 bits, the only primes are 5 and 7
              else if(bits == 3)
              if(k.compareTo(BigInteger.ZERO) == 0) setN(BigInteger.valueOf(5));
              else {
                  setN(BigInteger.valueOf(7));
              }
      // For the rest of numbers we use the general formula n=6k+5
      } else {
              BigInteger n = k.multiply(BigInteger.valueOf(6));
```

```
                n = n.add(BigInteger.valueOf(5));
                setN(n);
        }
      }
}
```

Then we are ready to use the PrimeCheck program:

PrimeCheck.java	**Andreas-MacBook-Air:RSA andrea$ javac PrimeCheck.java** **Andreas-MacBook-Air:RSA andrea$ java PrimeCheck 958918590485910** **FALSE** **Andreas-MacBook-Air:RSA andrea$ java PrimeCheck 709** **TRUE**

```
import java.math.BigInteger;

/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.1
 * @since 03/26/2014
 */

public class PrimeCheck {
      // Look for prime numbers to check at http://www.bigprimes.net/archive/prime/
      public static void main(String args[]){
        if (args.length !=1) {
                System.out.println("Please, insert an integer number as argument");
                return;
        }
        BigInteger number = new BigInteger(args[0]);

        // All numbers equal or less than 3 (less tan 4) are primes!
        if(number.compareTo(BigInteger.valueOf(4)) == -1) {
                System.out.println("TRUE\n");
                return;
        }
        Prime p = new Prime(number);
        // The argument for testPrime is the number of test to perform on the prime candidate
        if (p.testPrime(10)) System.out.println("TRUE\n");
        else System.out.println("FALSE\n");
      }
}
```

Your second program will be named primegen and will take a single argument, a positive integer which represents the number of bits, and produces a prime number of that number of bits (bits not digits). You may NOT use the library functions that come with the language (such as in Java or Ruby) or provided by 3rd party libraries.

PrimeGen.java	**Andreas-MacBook-Air:RSA andrea$ javac PrimeGen.java** **Andreas-MacBook-Air:RSA andrea$ java PrimeGen 10** **521** **Andreas-MacBook-Air:RSA andrea$ java PrimeGen 1024** **898846567431157953864652595394512366808988489471153286367150405788663** **379027504815663542386612037680105600569399356966788293948844072083112** **464237153197370621888839467124327426381511098006230470597265414760425** **028844190753411712314407369565552704136185816752553422931491199736229** **69239858152417678164812112069763**

```
import java.math.BigInteger;

/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.1
 * @since 03/26/2014
 */

public class PrimeGen {

	public static void main(String args[]){
	  if (args.length !=1){
		System.out.println("Please, insert an integer number as argument");
		return;
	  }
	  int nbits = Integer.parseInt(args[0]);
	  if (nbits <=0){
		System.out.println("Please, insert a positive number as argument");
		return;
	  }
	  Prime p = new Prime(nbits);
	  BigInteger k = p.initializeK();
	  p.generateCandidate(k);
	  if(nbits > 3) {
		while(!(p.testPrime(10)&&p.checkRange())){
			k = k.add(BigInteger.ONE);
			p.generateCandidate(k);
		}
	  }
	  System.out.println(p.getN());
	}
}
```

We are able now to check the two programs with each other:

Andreas-MacBook-Air:RSA andrea$ javac Prime.java
Andreas-MacBook-Air:RSA andrea$ javac PrimeCheck.java
Andreas-MacBook-Air:RSA andrea$ javac PrimeGen.java
Andreas-MacBook-Air:RSA andrea$ java PrimeGen 10
521
Andreas-MacBook-Air:RSA andrea$ java PrimeCheck 521
TRUE
Andreas-MacBook-Air:RSA andrea$ java PrimeGen 1024
89884656743115795386465259539451236680898848947115328636715040578866337902750481563542386612037680105600569399356966788293948844072083112464237153197370621888839467124327426381511098006230470597265414760425028844190753411712314407369565552704136185816752553422931491199736229692398581524176781648121120697 63
Andreas-MacBook-Air:RSA andrea$ java PrimeCheck 89884656743115795386465259539451236680898848947115328636715040578866337902750481563542386612037680105600569399356966788293948844072083112464237153197370621888839467124327426381511098006230470597265414760425028844190753411712314407369565552704136185816752553422931491199736229692398581524176781648121120697 63
TRUE

PART 2: RSA Implementation

You will implement three programs: keygen, encrypt and decrypt.

Key generation

Keygen will generate a (public, private) key pair given two prime numbers. Example (the $ sign is the command line prompt):
$ keygen 127 131
$ Public Key: (16637,11)
$ Private Key: (16637,14891)

In the table below you'll find some test cases for testing your keygen function. For those cases where you have to select your own numbers, make sure you select numbers that are at least 10 digits long.

KeyGen.java	**Andreas-MacBook-Air:RSA andrea$ javac KeyGen.java** **Andreas-MacBook-Air:RSA andrea$ java KeyGen 127 131** **Public key: (16637,11)** **Private key: (16637,14891)**

```
import java.math.BigInteger;

/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.0
 * @since 03/31/2014
 */
public class KeyGen {

      /**
       * It will generate a (public, private) key pair given two prime numbers
       */

      public static void main(String args[]) {
        int l = args.length;
        if(l !=2) {
                System.out.println("Please type as input two prime integer numbers ");
                return;
        }
        BigInteger p = new BigInteger(args[0]);
        BigInteger q = new BigInteger(args[1]);
        // Check if inputs are prime numbers
        Prime pp = new Prime(p);
        Prime pq = new Prime(q);
        if (!(pp.testPrime(5)) || !(pq.testPrime(5))) {
                System.out.println("The numbers typed are not prime. \n " +
                              "Use 'PrimeGen.java' to generate prime numbers as wished.");
                return;
        }
        // n = pq,
        BigInteger n = p.multiply(q);
        // Totient(n) = Totient(p)Totient(q) = (p - 1)(q - 1)
        BigInteger totient = ((p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE)));
        BigInteger e = generateE(totient);
        System.out.println("Public key: (" + n + "," + e + ")");
       // de = 1 mod(totient(n))
       BigInteger d = generateD(e,totient);
       System.out.println("Private key: (" + n + "," + d + ")");
      }

    /**
       * Generates a prime number e that does not divide totient(n).
     * This is equivalent as saying that gcd(e,totient(n)) = 1.
     * The number must be in the range: 1 < e < totient(n)
       *
       * @param  BigInteger totient(n)
```

```
        * @return BigInteger  e
        */
       private static BigInteger generateE(BigInteger totient){
         int nbits = 1;
         int aux = 1;
         Prime p = new Prime(nbits);
         BigInteger k = p.initializeK();
         p.generateCandidate(k);
         while(!(  ((totient.remainder(p.getN())).compareTo(BigInteger.ZERO) != 0)
                          &&(p.getN().compareTo(totient) < 0)
                          &&(p.testPrime(5))))){
             // To make sure we test the lowest prime candidates
             // we need to increment nbits and k slowly and not
             // at the same time:
             if (aux%2 == 0){
                 nbits = nbits + 1;
                 p.setB(nbits);
             } else if(aux%3 == 0){
                 k = k.subtract(BigInteger.ONE);
             } else{
                 k = k.add(BigInteger.ONE);
             }
             aux ++;
             p.generateCandidate(k);
         }
         return p.getN();
       }

     /**
        * Using the Euclidean Algorithm, this function finds the multiplicative
      * inverse of e mod (totient(n)). [ d*e = 1 mod(totient(n)) ]
      * To simplify things we could have used the built-in function of
      * BigIntegers: BigIntefer.modInverse(BigInteger val)
        *
        * @param  BigInteger e, totient(n)
        * @return BigInteger  d
        */
     private static BigInteger generateD(BigInteger e, BigInteger totient){
         BigInteger x2 = BigInteger.ONE;
         BigInteger x1 = BigInteger.ZERO;
         BigInteger y2 = BigInteger.ZERO;
         BigInteger y1 = BigInteger.ONE;
         BigInteger t = totient;
         BigInteger x, y, q, r;
         while (totient.compareTo(BigInteger.ZERO) == 1) {
             q = e.divide(totient);
             r = e.subtract(q.multiply(totient));
             x = x2.subtract(q.multiply(x1));
             y = y2.subtract(q.multiply(y1));
             e = totient;
             totient = r;
             x2 = x1;
             x1 = x;
             y2 = y1;
             y1 = y;
         }
         return x2.add(t);
     }
}
```

First Number	Second Number	n	e	d
1019	1021	1040399	7	890023
1093	1097	1199021	5	1675565
433	499	216067	5	172109
1061	1063	1127843	7	964903
1217	1201	1461617	7	1250743
313	337	105481	5	146765
419	463	193997	5	154493
3457	3461	11964677	7	15374263
137	139	19043	5	11261
811	853	691783	7	492943
257	271	69647	7	98743
7853	7873	61826669	5	74173133

Encryption

Encrypt will take a public key (n,e) and a single character c to encode, and returns the cyphertext corresponding to the plaintext, m.

Example:
$ encrypt 16637 11 20
$ 12046

In the table below you'll find some test cases for testing your encrypt function. For those cases where you have to select your own numbers, make sure you select n such that it is at least 20 digits long and that e is two digits long. The character c is a positive integer smaller than 256.

Encrypt.java

Andreas-MacBook-Air:RSA andrea$ javac Encrypt.java
Andreas-MacBook-Air:RSA andrea$ java Encrypt 16637 11 20
12046

```
import java.math.BigInteger;
/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.0
 * @since 04/02/2014
 */
public class Encrypt {
    /**
       * It will take a public key (n,e) and a single character c to encode, and
     * returns the cyphertext corresponding to the plaintext, m
       */

      public static void main(String args[]) {
        int l = args.length;
        if(l !=3) {
                System.out.println("Please type as input the key(n,e) and the character (int value)
\nCorrect use: java Encrypt n e c");
                return;
        }
        BigInteger n = new BigInteger(args[0]);
        BigInteger e = new BigInteger(args[1]);
        BigInteger c = new BigInteger(args[2]);

        /* As we need to compute the encryption:
         * ciphertext = (c)^e mod(n)
         * I already implementd this function inside Prime.class,
         * so I am going to create Prime objects to compute the result.
         */
        Prime aux = new Prime(1);
        BigInteger result = aux.expModn(c, e, n);
        System.out.println(result);
    }

}
```

n	e	c	m
1040399	7	99	579196
1199021	5	70	871579
216067	5	89	23901
1127843	7	98	871444
1461617	7	113	1411436
105481	5	105	36549
193997	5	85	147738
11964677	7	31	5821688
19043	5	42	18166
691783	7	200	232680
69647	7	184	51752

Decryption

Decrypt will take a private key (n, d) and the encrypted character and return the corresponding plaintext.

Example:
$ decrypt 16637 14891 12046
$ 20

In the table below you'll find some test cases for testing your decrypt function.

Decrypt.java

```
Andreas-MacBook-Air:RSA andrea$ javac Decrypt.java
Andreas-MacBook-Air:RSA andrea$ java Decrypt 16637 14891 12046
20
```

```
import java.math.BigInteger;
/**
 * @author Andrea Sancho Silgado [A20315328] asanchos@hawk.iit.edu
 * @version 1.0
 * @since 04/02/2014
 */
public class Decrypt {
    /**
       * It will take a public key (n,d) and a single character c to encrypted, and
     * returns the plaintext corresponding to that ciphertext
       */

      public static void main(String args[]) {
         int l = args.length;
         if(l !=3) {
                 System.out.println("Please type as input the key(n,d) and the encrypted character (int
value) \nCorrect use: java Encrypt n d c");
                 return;
         }
        BigInteger n = new BigInteger(args[0]);
        BigInteger d = new BigInteger(args[1]);
        BigInteger c = new BigInteger(args[2]);

        /* As we need to compute the decryption:
         * plaintext = (c)^d mod(n)
         * I already implementd this function inside Prime.class,
         * so I am going to create a Prime object to compute the result.
         */
        Prime aux = new Prime(1);
        BigInteger result = aux.expModn(c, d, n);
        System.out.println(result);
    }

}
```

n	d	m	c
1040399	890023	579196	99
1199021	1675565	871579	70
216067	172109	23901	89
1127843	964903	871444	98
1461617	1250743	1411436	113
105481	146765	36549	105
193997	154493	147738	85
11964677	15374263	5821688	31
19043	11261	18166	42
691783	492943	232680	200
69647	98743	51752	184