ECE 543 Project 2b:
RSA Algorithm Implementation

Linghan Zhao

A20313766

INTRODUCTION

In this project, I will analyze and implement RSA algorithm which is the classic public and private keys algorithm. According to the requirement, I will implement prime check, prime generate, key generate, plaintext encryption and cipher text decryption in JAVA environment. And codes will be attached in appendix part.

DESIGN AND IMPLEMENTATION

I chose JAVA to design and implement this project, as JAVA has a Java has a built-in *BigInteger* class that behaves exactly like an *int*, especially that it is not bounded by a finite value. So it is very suitable to use JAVA environment to achieve project goals.

Part 1: Prime Number

In this part, I decide to use Miller-Rabin method to check and find a big prime which will be required for RSA algorithm. During research, I find that Miller-Rabin is not absolutely effective to check if a number is a prime or not. So I decide to iterate this algorithm 30 times to increase accuracy when judging prime number.

primeCheck

This could be the most important part of this whole project. I use Miller-Rabin, whose flow diagram is showed in Figure 1, to check a big number **rnd**. First, In order to reduce time to find a big prime (e.g. 1024 bits), I search the prime numbers less than 1000 and use trial division to exclude some big number quickly if they be divided by these prime numbers. Then, I need to define a parameter **k** that determines the accuracy of prime test times, **s** and **d** which means the **rnd-1=2^{s}d**. And a random number **a** is in the range from 2 to **rnd**-2. If this function return true, it means this number is prime.

```
public boolean primeCheck(BigInteger rnd)
```

primeGen

This program will use the result of **primeCheck** method. When generating a prime number, this program will produce a random number with the specific number bits (a parameter of this program). Use **primeCheck** to check if it is a prime. This part will generate a prime if the random number passes **primeCheck**.

```
public void primeGen(int numBits);
```

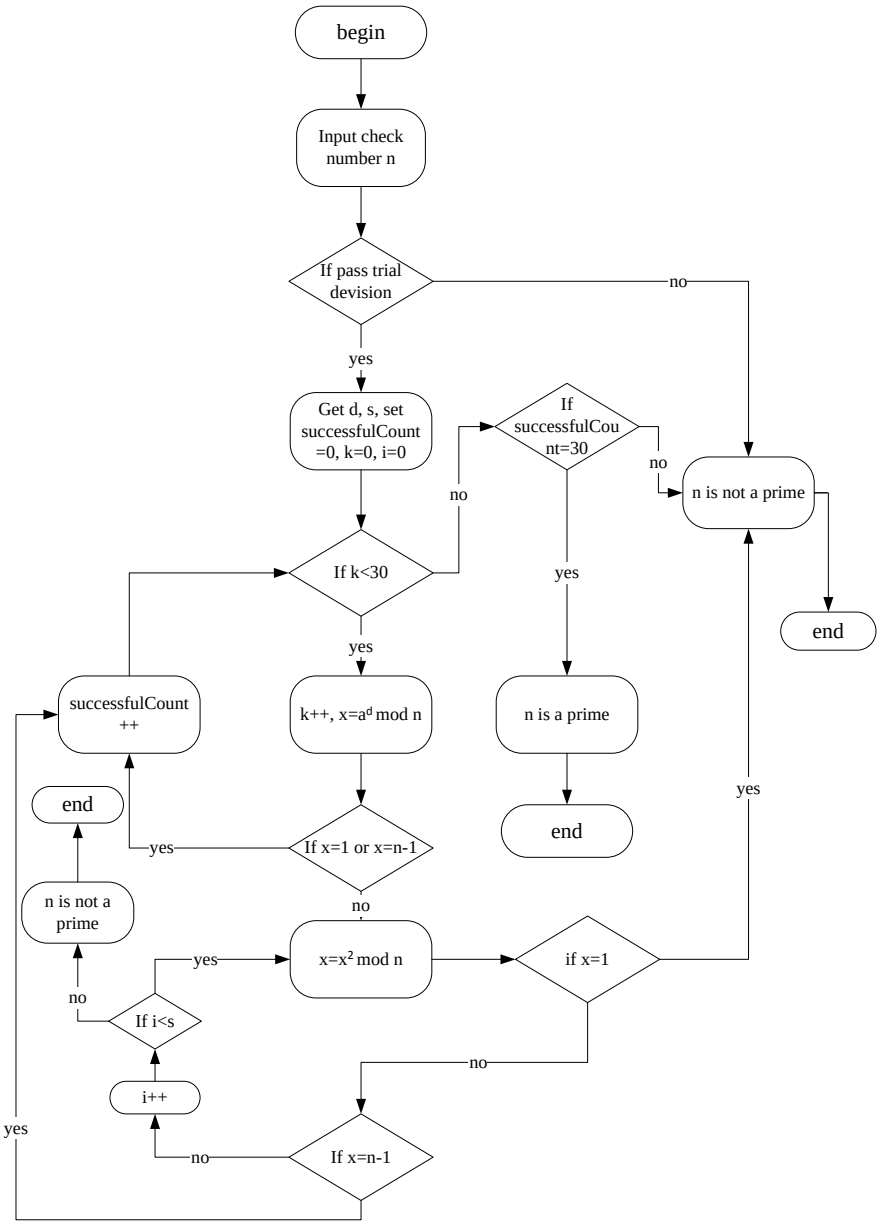

Figure 1: Miller-Rabin prime check flow chat

The Miller-Rabin algorithm pseudo code that is from wiki is displayed in Figure 2.

```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n − 1 as 2^s · d with d odd by factoring powers of 2 from n − 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n − 2]
   x ← a^d mod n
   if x = 1 or x = n − 1 then do next WitnessLoop
   repeat s − 1 times:
      x ← x^2 mod n
      if x = 1 then return composite
      if x = n − 1 then do next WitnessLoop
   return composite
return probably prime
```

Figure 2: Pseudo code of Miller-Rabin

The flow diagram of primeGen() is showed blow in Figure 3.

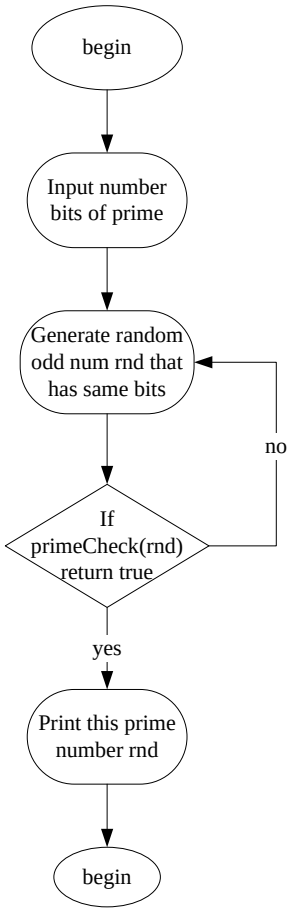

Figure 3: primeGen () flow diagram to generate a prime

Part 2: RSA Implementation

In part 2, I implement three programs including keygen, encrypt and decrypt. keygen will use two prime to generate public and private key pair, encrypt will use public key (n, e) and a plaintext m to implement encrypt, decrypt will use private key (n, d) and a cipher text c to implement decrypt.

keygen

According to textbook (Network Security: Private Communication in a Public World), RSA algorithm use formats and functions blow to produce public key **e** and private key **d**:

(p and q are prime)

n=p q;

ϕ (n) = (p-1) (q-1);

e is relative prime to ϕ (n);

de = 1 mod ϕ (n) or d= e^{-1} mod ϕ (n).

```
public void keygen(BigInteger p, BigInteger q)
```

encrypt

This function uses public key (n, e) to encrypt plaintext m:
$c=m^e \bmod n$

```
public BigInteger encrypt(BigInteger n, BigInteger e, BigInteger m)
```

decrypt

This function uses private key (d, e) to decrypt cipher text c
$m=c^d \bmod n$

```
public BigInteger decrypt(BigInteger n, BigInteger d, BigInteger c)
```

TESTS AND RESULTS

primeCheck

Try 32401 and 3244568 jsut like description and check the result.
Test results in Figure 4

```
========================prime check==========================
please input the number you want to check if it is a prime:
32401
this number is prime

========================prime check==========================
please input the number you want to check if it is a prime:
3244568
this number is not prime
```

Figure 4: Test results of primeCheck

primeGen

Generate prime number of any bits. We try 4 and 1024 bits, and use primeCheck to check 1024 bits number.
Test results in Figure 5. The 1024 bits prime number displays partially. Please wait patiently when generating 1024 bits number because JAVA is relatively slow even though I have used trial division to reduce time.

```
========================prime generate==========================
please input number bits of prime number you want to generate:
4
the prime number is: 11

========================prime generate==========================
please input number bits of prime number you want to generate:
1024
the prime number is: 17721321503644878159162057327321568471384245275098482054882932165286848481931787198559759363520747472237628574168765605

========================prime check==========================
please input the number you want to check if it is a prime:
17721321503644878159162057327321568471384245275098482054882932165286848481931787198559759363520747472237628574168765605255751952541165516131
this number is prime
```

Figure 5: Test results of primeGen

keygen

public and private pair key generate, I try the example in project description to test.
Test results show below in Figure 6.

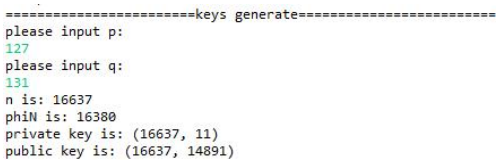

Figure 6: Test result of keys generate

encrypt

Use public key (n, e) to encrypt plaintext m and return cipher text c. I try to use example in description to test.

Test result shows in Figure 7.

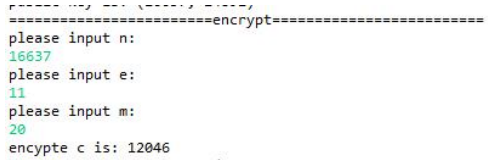

Figure 7: Test result of encrypt

decrypt

Use private key (n, d) to decrypt cipher text c and return plaintext m. I try to use example in description to test.

Test result shows in Figure 8.

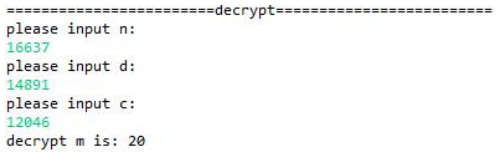

Figure 8: Test result of decrypt

Additional tests required in project description

In this part, I will finish the whole test table in the project description and select required number to test.

1. **keygen**

   My own selected number p and q are both at least 10 digits long. I use primeGen to generate the prime number for this table.

First Number	Second Number	n	e	d
1019	1021	1040399	7	890023
1093	1097	1199021	5	478733
433	499	216067	5	172109
1061	1063	1127843	7	967903
1217	1201	1461617	7	1250743
313	337	105481	5	41933
419	463	193997	5	154493
2932415249	3808178233	11167159921359075017	55	3857746152322748167
3700608113	3728195533	13796590636270159229	101	6010395917515045997
6080047423	5961217883	36244487427475665509	115	8194405850446038307

2. **encrypt**

My selected n is at least 20 digits long and m is less than 256. We use 20 digits number listed above keygen's n number as encrypt n. Then, we can check the keygen, encrypt and decrypt function together.

n	e	m	c
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
11167159921359075017	55	70	7474706001004226396
13796590636270159229	101	44	5459714453617879594
36244487427475665509	115	177	1347473421000108332

3. **decrypt**

I plan to use the same c produced from encrypt, and the related d from keygen. The result shows the implementation is correct.

n	d	c	m
1040399	890023	16560	104
1199021	478733	901767	71
216067	172109	169487	101
1127843	964903	539710	119
1461617	1250743	93069	83
105481	41933	78579	76
193997	154493	1583	122
11167159921359075017	3857746152322748167	7474706001004226396	70
13796590636270159229	6010395917515045997	5459714453617879594	44
36244487427475665509	8194405850446038307	1347473421000108332	177

SUMMERY

Through long time research and coding, I finish this project finally. All functions and parts required in project description are implemented, the tests and results are all correct. RSA algorithm and Miller-Rabin method is the essential part of this project. I learned from this project that every steps and efforts construct to the classic algorithm RSA work efficiently.

APPENDIX: CODES

Package RSA: primeNuber.java

```
package RSA;
import java.math.BigInteger;
import java.util.Random;



public class primeNumber {



    public static final int NUMBERBITS=1024;
    static BigInteger consTwo= new BigInteger("2");
    public primeNumber(){

    }



    public void primeGen(int numBits){

        primeNumber s= new primeNumber();
        Random rnd= new Random();
        BigInteger num= new BigInteger(numBits,rnd);

        while(true){
            num=new BigInteger(numBits,rnd);
            int n=numBits-num.bitLength();
            //shift left to combine a big number of 1024bits and add one to produce
odd number
            if(n!=0 || num.remainder(consTwo).equals(BigInteger.ZERO)){
                num=num.shiftLeft(n);
                num=num.add(BigInteger.ONE);
            }

            if(s.primeCheck(num)){
                break;
```

```
            }

        }
        System.out.println("the prime number is: "+num);
    }


    //check if this number is prime
    public boolean primeCheck(BigInteger rnd){

        BigInteger [] h = {new BigInteger("3"), new BigInteger("5"), new
BigInteger("7"), new BigInteger("11"),
                new BigInteger("13"), new BigInteger("17"), new
BigInteger("19"), new BigInteger("23"),
                new BigInteger("29"), new BigInteger("31"), new
BigInteger("37"), new BigInteger("41"),
                new BigInteger("43"), new BigInteger("47"), new
BigInteger("53"), new BigInteger("59"),
                new BigInteger("61"), new BigInteger("67"), new
BigInteger("71"), new BigInteger("73"),
                new BigInteger("79"), new BigInteger("83"), new
BigInteger("89"), new BigInteger("97"),
                new BigInteger("101"), new BigInteger("103"), new
BigInteger("107"), new BigInteger("109"),
                new BigInteger("113"), new BigInteger("127"), new
BigInteger("131"), new BigInteger("137"),
                new BigInteger("139"), new BigInteger("149"), new
BigInteger("151"), new BigInteger("157"),
                new BigInteger("163"), new BigInteger("167"), new
BigInteger("173"), new BigInteger("179"),
                new BigInteger("181"), new BigInteger("191"), new
BigInteger("193"), new BigInteger("197"),
                new BigInteger("199"), new BigInteger("211"), new
BigInteger("223"), new BigInteger("227"),
                new BigInteger("229"), new BigInteger("233"), new
BigInteger("239"), new BigInteger("241"),
                new BigInteger("251"), new BigInteger("257"), new
BigInteger("263"), new BigInteger("269"),
                new BigInteger("271"), new BigInteger("277"), new
BigInteger("281"), new BigInteger("283"),
                new BigInteger("293"), new BigInteger("307"), new
BigInteger("311"), new BigInteger("313"),
                new BigInteger("317"), new BigInteger("331"), new
BigInteger("337"), new BigInteger("347"),
```

```
                new BigInteger("349"), new BigInteger("353"), new
BigInteger("359"), new BigInteger("367"),
                new BigInteger("373"), new BigInteger("379"), new
BigInteger("383"), new BigInteger("389"),
                new BigInteger("397"), new BigInteger("401"), new
BigInteger("409"), new BigInteger("419"),
                new BigInteger("421"), new BigInteger("431"), new
BigInteger("433"), new BigInteger("439"),
                new BigInteger("443"), new BigInteger("449"), new
BigInteger("457"), new BigInteger("461"),
                new BigInteger("463"), new BigInteger("467"), new
BigInteger("479"), new BigInteger("487"),
                new BigInteger("491"), new BigInteger("499"), new
BigInteger("503"), new BigInteger("509"),
                new BigInteger("521"), new BigInteger("523"), new
BigInteger("541"), new BigInteger("547"),
                new BigInteger("557"), new BigInteger("563"), new
BigInteger("569"), new BigInteger("571"),
                new BigInteger("577"), new BigInteger("587"), new
BigInteger("593"), new BigInteger("599"),
                new BigInteger("601"), new BigInteger("607"), new
BigInteger("613"), new BigInteger("617"),
                new BigInteger("619"), new BigInteger("631"), new
BigInteger("641"), new BigInteger("643"),
                new BigInteger("647"), new BigInteger("653"), new
BigInteger("659"), new BigInteger("661"),
                new BigInteger("673"), new BigInteger("677"), new
BigInteger("683"), new BigInteger("691"),
                new BigInteger("701"), new BigInteger("709"), new
BigInteger("719"), new BigInteger("727"),
                new BigInteger("733"), new BigInteger("739"), new
BigInteger("743"), new BigInteger("751"),
                new BigInteger("757"), new BigInteger("761"), new
BigInteger("769"), new BigInteger("773"),
                new BigInteger("787"), new BigInteger("797"), new
BigInteger("809"), new BigInteger("811"),
                new BigInteger("821"), new BigInteger("823"), new
BigInteger("827"), new BigInteger("829"),
                new BigInteger("839"), new BigInteger("853"), new
BigInteger("857"), new BigInteger("859"),
                new BigInteger("863"), new BigInteger("877"), new
BigInteger("881"), new BigInteger("883"),
                new BigInteger("887"), new BigInteger("907"), new
BigInteger("911"), new BigInteger("919"),
```

```
                new BigInteger("929"), new BigInteger("937"), new
BigInteger("941"), new BigInteger("947"),
                new BigInteger("953"), new BigInteger("967"), new
BigInteger("971"), new BigInteger("977"),
                new BigInteger("983"), new BigInteger("991"), new
BigInteger("997"), };

        //try devide fisrt, devide small prime, if finding factor, return false;
        BigInteger smallNum= new BigInteger("1000");
        if(rnd.max(smallNum).equals(rnd)){
            for (int i=0; i<=h.length-1; i++){
                if(rnd.remainder(h[i]).equals(BigInteger.ZERO)){
                    return false;
                }
            }
        }



        int numBits=rnd.bitLength();
        BigInteger rndMinOne=rnd.subtract(BigInteger.ONE);
        int s=0;//s
        BigInteger d;//d
        int k;
        int successfulCount=0;


        BigInteger remainder=rndMinOne.remainder(consTwo);

        //computer s and d
        while(remainder.equals(BigInteger.ZERO)){
            rndMinOne=rndMinOne.divide(consTwo);
            remainder=rndMinOne.remainder(consTwo);
            s++;

        }
        d=rndMinOne;


        //witness loop Miller-Rabin
        for(k=0;k<30;k++){

            Random random=new Random();
            //pick a random base number a range is [2,rnd-2]
```

```
            BigInteger a= new BigInteger(numBits, random);
            while (a.equals(rnd) ||a.equals(rnd.subtract(BigInteger.ONE)) ||
a.equals(BigInteger.ONE) || a.equals(BigInteger.ZERO) ||
a.max(rnd).equals(a)){

                a=new BigInteger(numBits,random);
            }

            BigInteger x= a.modPow(d, rnd); //to check if x equal to rnd-1 or
1

            //if x==1 or rnd-1, return successful count
            if (x.equals(BigInteger.ONE)){
                successfulCount++;
                continue;
            }
            if (x.equals(rnd.subtract(BigInteger.ONE))){
                successfulCount++;
                continue;
            }

            for(int sCount=0; sCount<=s-1;sCount++){
                x=x.modPow(consTwo, rnd);
                if(x.equals(BigInteger.ONE))
                    return false;
                if(x.equals(rnd.subtract(BigInteger.ONE))){
                    successfulCount++;
                    break;
                }
                //return false;
            }

        }
        //every witness loop is right, successful count is equal to k times check,
return true
        if(successfulCount==30)
            return true;
        else
            return false;

    }

}
```

Package RSA: rsaimplement.java

```
package RSA;
import java.math.BigInteger;
import java.util.Random;

public class rsaImplement {

    public void keygen(BigInteger p, BigInteger q){

        Random rnd= new Random();
        BigInteger n= p.multiply(q);
        System.out.println("n is: "+n);
        BigInteger phiN=
p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        System.out.println("phiN is: "+phiN);
        BigInteger e = new BigInteger("55");
        BigInteger d;

        while(!phiN.gcd(e).equals(BigInteger.ONE) ||
e.equals(BigInteger.ONE)){
            e = new BigInteger(7, rnd);
        }
        d= e.modInverse(phiN);
        System.out.println("private key is: ("+n+", "+e+")");
        System.out.println("public key is: ("+n+", "+d+")");

    }

    //RSA encrypt, use e to encrypt plaintext m
    public BigInteger encrypt(BigInteger n, BigInteger e, BigInteger m){

        BigInteger c;
        c= m.modPow(e, n);
        return c;
    }

    //RSA decrypt, use d to decrypt cyphertext c
    public BigInteger decrypt(BigInteger n, BigInteger d, BigInteger c){

        BigInteger m;
        m= c.modPow(d, n);
        return m;
    }
}
```

Package RSA: rsa.java

```
package RSA;
import java.math.BigInteger;
import java.util.*;

public class rsa {

    private static Scanner s;

    public static void main(String[] args) {

        // TODO Auto-generated method stub
        primeNumber prn = new primeNumber();
        rsaImplement rsi = new rsaImplement();

        /*prime check, input candidate number to check if it is prime*/
        System.out.println("======================prime
check=========================");
        System.out.println("please input the number you want to check if it is
a prime:");
        s = new Scanner(System.in);
        BigInteger num= s.nextBigInteger();
        if(prn.primeCheck(num))
            System.out.println("this number is prime");
        else
            System.out.println("this number is not prime");


        /*prime generate, input number bits*/
        System.out.println("======================prime
generate=========================");
        System.out.println("please input number bits of prime number you want
to generate:");
        int numBit;
        numBit = s.nextInt();
        prn.primeGen(numBit);


        /*private key and public key generate, input p and q*/
        System.out.println("======================keys
generate=========================");
        System.out.println("please input p: ");
        BigInteger p=s.nextBigInteger();
        System.out.println("please input q: ");
```

```
        BigInteger q=s.nextBigInteger();
        rsi.keygen(p, q);


        /*encrypt, input n, e, m*/

    System.out.println("========================encrypt====================
=====");
        System.out.println("please input n: ");
        BigInteger n=s.nextBigInteger();
        System.out.println("please input e: ");
        BigInteger e=s.nextBigInteger();
        System.out.println("please input m: ");
        BigInteger m=s.nextBigInteger();
        BigInteger c;
        c = rsi.encrypt(n, e, m);
        System.out.println("encypte c is: "+c);


        /*decrypt, input n, d, c*/

    System.out.println("========================decrypt====================
=====");
        System.out.println("please input n: ");
        BigInteger n1=s.nextBigInteger();
        System.out.println("please input d: ");
        BigInteger d=s.nextBigInteger();
        System.out.println("please input c: ");
        BigInteger c1=s.nextBigInteger();
        BigInteger m1;
        m1 = rsi.decrypt(n1, d, c1);
        System.out.println("decrypt m is: "+m1);
    }



}
```