

# CS201 - RSA encryption algorithm and pseudo-random number generator

---

11811721 庄湛

## CS201 - RSA encryption algorithm and pseudo-random number generator

### Abstract

1. The principle of RSA
  - 1.1 Asymmetric encryption
  - 1.2 One-way function
2. Large prime number generator
  - 2.1 Primality testing
    - 2.1.1 Trying division method
    - 2.1.2 Fermat testing
    - 2.1.3 Miller-Rabin testing
    - 2.1.4 Comparison of three testing method
  - 2.2 Pseudo-random number generator
    - 2.2.1 Middle-square method
    - 2.2.2 Linear congruential generator
3. Program Implementation
  - 3.1 Code
  - 3.2 Demo
4. Realistic demand
  - 4.1 RSA key length problem
  - 4.2 Authentication

### Reference

## Abstract

---

In this project, I reviewed the principles of RSA and learned the difference between asymmetric encryption and symmetric encryption. Then I used a program to implement a more complete RSA algorithm. For the generation of large prime numbers, I first implement and compare of three different prime number testing methods and then analyzed the two pseudo-random number generation methods of Middle-square method and LCG. I chose Miller-Rabin and LCG for the implementation of RSA algorithm. Finally, two actual requirements of RSA encryption are analyzed.

## 1. The principle of RSA

---

### 1.1 Asymmetric encryption

**Public-key cryptography**, or **asymmetric cryptography**, is a cryptographic system that uses pairs of keys: public keys, which may be disseminated widely, and private keys, which are known only to the owner(It's from Wikipedia). In contrast to symmetric cryptography, which uses the same key for plaintext encryption and ciphertext decryption, this cryptography exists in an open public key that anyone who owns the public key can encrypt the information and send it to the sender. But the key used for decryption only exists with the recipient, so it is less likely to be

stolen by third parties, thus ensuring the security of the encryption. But in general, asymmetric encryption algorithms are slower.

The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. In such a system, any person can encrypt a message using the sender's public key, but that encrypted message can only be decrypted with the sender's private key.

## 1.2 One-way function

One-way functions are whose function values are easy to calculate for each input (polynomial time), but given a function value, it is difficult to calculate the original input.

A **trapdoor function** is a function that is easy to compute in one direction, yet difficult to compute in the opposite direction without special information, called the "trapdoor". Trapdoor functions are widely used in cryptography. (It's from Wikipedia)

If we can find such a trap door one-way function, we can design a password based on it. For example:

1. Using coloring matter, it is easy to mix two colours with paint mixture, but it is extremely difficult to tell the colours apart.

But if we know the inverse of one color, the two colors will cancel each other out, then we can separate the two colors from the mixture.

2. Using factorization, it is easy to multiply two large prime Numbers, but it is extremely difficult to factor the product.

We can use this one-way function and Euler function to construct an algorithm:

1. find two big primes  $p$  and  $q$
2. calculate their production  $n$ , which means  $n = pq$
3. calculate the  $\phi(n) = (p-1)(q-1)$
4. using the Euler function, there are  $m^{k\phi(n)} = m \mod n$
5. we can use a small prime  $e$ , and calculate  $d$ , which satisfies  $ed = k\phi(n) + 1$
6. finally, we can get  $m$  easily if  $c = m^e \mod n$  because  $c^d \mod n = m^{ed} \mod n = m \mod n$

This is the main idea of the RSA algorithm.

## 2. Large prime number generator

But the primes in RSA must not be fixed, or there would be a lack of security. So a prime number generator is required to complete the RSA algorithm. In order to improve the speed of generating prime Numbers, we usually generate a pseudo-random number and then test the prime number.

### 2.1 Primality testing

There are many ways to test prime Numbers, such as the most common trial division method, but for larger primes, the time complexity of this method does not meet our requirements.

#### 2.1.1 Trying division method

Trial division is the easiest to understand and easiest way to test prime Numbers. The principle of this method is to check whether every number less than  $\sqrt{n}$  is a factor of  $n$ .

Here is the program referenced from Wikipedia:

```
// Trying division method
private static boolean isPrime1(long n) {
    if (n <= 3) return n > 1;
    else if (n % 2 == 0 || n % 3 == 0) return false;
    else {
        double sqrtN = Math.floor(Math.sqrt(n));
        for (int i = 5; i <= sqrtN; i += 6) {
            if (n % i == 0 || n % (i + 2) == 0) return false;
        }
        return true;
    }
}
```

Its time complexity is:  $O(\sqrt{n})$

### 2.1.2 Fermat testing

**Fermat's little theorem** states that if  $p$  is a prime number, then for any integer  $a$ , the number  $a^p - a$  is an integer multiple of  $p$ . In the notation of modular arithmetic, this is expressed as  $a^p \equiv a \pmod{p}$

The converse of Fermat's theorem is not true, so this conclusion is a necessary condition for prime Numbers. But we can pick multiple  $A$  values to see **if a number is not prime**.

Here is the program I wrote using Fermat testing:

```
// Fermat testing
private static boolean isPrime2(long n, int testTime) {
    BigInteger N = new BigInteger(Long.toString(n));
    BigInteger A;
    if (BigInteger.TWO.modPow(N.subtract(BigInteger.ONE),
N).compareTo(BigInteger.ONE) != 0)
        return false;
    else {
        for (int i = 0; i < testTime - 1; i++) {
            A = new BigInteger(Long.toString((long) (Math.random() * (n - 2) +
2)));
            if (A.modPow(N.subtract(BigInteger.ONE), N).compareTo(BigInteger.ONE)
!= 0)
                return false;
        }
    }
    return true;
}
```

Using fast algorithms for modular exponentiation and multiprecision multiplication, the running time of this algorithm is  $O(k \log^2 n \log \log n)$ , where  $k$  is the number of times we test a random  $a$ , and  $n$  is the value we want to test for primality (It's from Wikipedia).

### 2.1.3 Miller-Rabin testing

According to Fermat's theorem and Euclid's lemma, the following conclusions can be drawn. Suppose  $n$  is a prime number and  $n > 2$ , so  $n - 1$  is even and can be expressed as  $2^s \cdot d$ , both  $s$  and  $d$  are odd. For some range of  $a$  and  $0 \leq r \leq s - 1$ , it must satisfy one of the following two forms:

$$\begin{aligned} a^d &\equiv 1 \pmod{n} \\ a^{2^r d} &\equiv -1 \pmod{n} \end{aligned}$$

So this is also necessary for a prime number, just like Fermat testing, we can also use it to see **if a number is not prime**. In particular, according to Wikipedia, for cases less than  $2^{64}$ , selecting seven credentials(2, 7, 325, 9375, 28178, 450775, 9780504, 1795265022) effectively excludes all composite Numbers.

Here is the program I wrote using Miller-Rabin testing:

```
// Miller-Rabin testing
private static boolean isPrime3(long n) {
    long[] witness = new long[]{ 2, 7, 325, 9375, 28178, 450775, 9780504,
1795265022L };
    // long[] witness = new long[]{ 2, 7, 61};
    if (n == 1) return false;
    int s = Long.numberOfTrailingZeros(n - 1);
    long d = (n - 1) >> s;
    BigInteger bigD = new BigInteger(Long.toString(d));
    BigInteger bigN = new BigInteger(Long.toString(n));
    for (long l : witness) {
        if (n <= l) break;
        else if (MillerRabinTest(n, l, bigD, bigN, s)) return false;
    }
    if (n > witness[witness.length - 1]) {
        for (int i = 0; i < 3; i++) {
            if (MillerRabinTest(n, (long) (Math.random() * (n - 1) + 1), bigD,
bigN, s)) {
                return false;
            }
        }
    }
    return true;
}

private static boolean MillerRabinTest(long n, long a, BigInteger bigD,
BigInteger bigN, int s) {
    BigInteger bigA = new BigInteger(Long.toString(a));
    BigInteger result = bigA.modPow(bigD, bigN);
    if (result.equals(BigInteger.ONE)) return false;
    for (int j = 0; j < s; j++) {
        if (result.longValue() == n - 1) return false;
        result = result.multiply(result).mod(bigN);
    }
    return true;
}
```

Using repeated squaring, the running time of this algorithm is  $O(k \log^3 n)$ , where  $n$  is the number tested for primality, and  $k$  is the number of rounds performed (It's from Wikipedia).

## 2.1.4 Comparison of three testing method

We can write a program to compare these prime number tests:

```

public static void main(String[] args) {
    long min = 1000000000000000000L;
    long max = 1000000000001000000L;
    int testTime = 3;
    long cnt1 = 0, cnt2 = 0, cnt3 = 0;
    long mis1 = 0, mis2 = 0, mis3 = 0;
    boolean test1, test2, test3;
    long t1, t2;
    for (long i = min; i < max; i++) {
        t1 = System.currentTimeMillis();
        test1 = isPrime1(i);
        t2 = System.currentTimeMillis();
        cnt1 += (t2 - t1);
        t1 = System.currentTimeMillis();
        test2 = isPrime2(i, testTime);
        t2 = System.currentTimeMillis();
        cnt2 += (t2 - t1);
        t1 = System.currentTimeMillis();
        test3 = isPrime3(i);
        t2 = System.currentTimeMillis();
        cnt3 += (t2 - t1);
        if (test1 != test2) mis2++;
        if (test1 != test3) mis3++;
    }
    System.out.println("Trial division: time: " + cnt1 + ", mistake: " +
mis1);
    System.out.println("Fermat testing: time: " + cnt2 + ", mistake: " +
mis2);
    System.out.println("Miller-Rabin : time: " + cnt3 + ", mistake: " +
mis3);
}

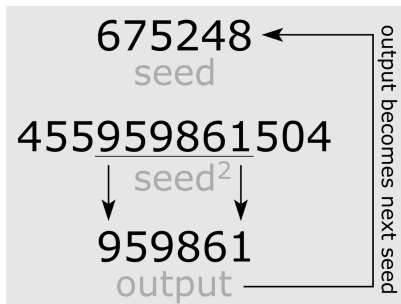
```

Testing range ---Time(ms)/mistake	Trial division	Fermat (3 rands)	Miller-Rabin (given)
1 to 10000000	6291/0	27691/114	32262/0
1000000000000000 to 1000000000100000	113691/0	577/0	618/0
1000000000000000000 to 1000000000000000100	7267/0	10/0	11/0
10000000000000000000 to 10000000000000100000 (19bits)	>> 1 hour	999/0	1177/0
90000000000000000000 to 90000000000001000000 (19bits)	>> 1 hour	6295/0	7025/0

We can see that when the number tested is small, the time of trial division is shorter and the result is accurate. But when the number is large, the time of trial division increases faster. Fermat test and Miller-Rabin prime number test have better results. And the Miller-Rabin Prime Number test was more accurate than the Fermat test, and we could have selected some given credentials to improve the accuracy, and we can also generate multiple groups of credentials with random numbers outside the given range.

## 2.2 Pseudo-random number generator

## 2.2.1 Middle-square method



The idea of the Middle-square method is to square a number that has  $2s$  bits, to leave out the head part and rear part in the number and just keep the  $2s$  bits in the middle as the next randomly generated number. And if there aren't enough bits, we need fill in them with zero.

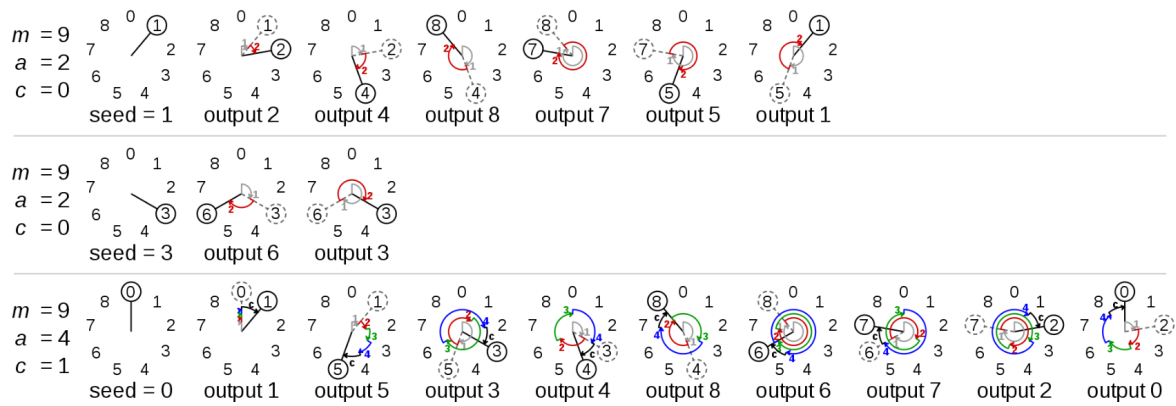
Here is my implementation:

```
private int length; private long max; private long seed;
Generator(long seed) {
    this.length = String.valueOf(seed).length();
    this.seed = seed;
    this.max = Long.parseLong(StringUtils.repeat("9", length));
}
private long nextRand() {
    long squared = seed * seed;
    String s = String.format("%0" + length * 2 + "d", squared);
    seed = Long.parseLong(s.substring(length / 2, length / 2 + length));
    return seed;
}
private double nextDouble() {
    nextRand();
    return this.seed * 1.0 / this.max;
}
```

I tried out several different sets of seedings and counted the distribution of the data and found that after multiple iterations, the results always reached 0. This illustrates the disadvantages of this method, such as easy to cycle, short cycle time and easy to produce zero value, which are rarely used nowadays.

distribution\seed	198997968	142389814	1115463111
00%-10%	983672	995216	958960
10%-20%	1795	530	4597
20%-30%	1840	494	4605
30%-40%	1708	511	4646
40%-50%	1838	521	4563
50%-60%	1836	531	4519
60%-70%	1794	527	4429
70%-80%	1857	532	4618
80%-90%	1819	565	4557
90%-100%	1841	573	4506

## 2.2.2 Linear congruential generator



The generator is defined by a recurrence relation:  $X_{n+1} = (aX_n + c) \bmod m$ , where  $X$  is the sequence of pseudorandom values.

This method can obtain random Numbers of different periods according to the selection of four parameters. Moreover, the selection of  $M$  is particularly critical. If  $M$  is prime and  $C$  is 0, a random sequence with a period of  $M-1$  will be generated. If  $m$  is a power of 2, usually  $m = 2^{32}$  or  $m = 2^{64}$ , a special LCG will be generated because this allows for modular computation by simply truncating the binary representation. In particular, when the Soldery-Dobell Theorem is satisfied, the period of a random sequence is  $M$ .

**Hull-Dobell Theorem:** The LCG used above has full period if and only if the following three conditions hold:

- (a)  $c$  is relatively prime to  $m$
- (b) If  $q$  is a prime number that divides  $m$ , then  $q$  divides  $a-1$
- (c) If 4 divides  $m$ , then 4 divides  $a-1$

Many Random Numbers in computer languages make use of this method. For example, "java.util.random" selects a combination of  $m = 2^{48}$ ,  $a = 25214903917$ ,  $c = 11$ .

Here is my implementation:

```
private BigInteger A;
private BigInteger C;
private BigInteger Seed;
private BigInteger Mod;
private int nextInt() {
    Seed = A.multiply(Seed).add(C).mod(Mod);
    StringBuilder newInt = new StringBuilder(Seed.toString());
    while (newInt.length() < 10) newInt.insert(0, "0");
    return Integer.parseInt(newInt.toString().substring(1, 9));
}
public static void main(String[] args) {
    myLCG lcg = new myLCG();
    lcg.A = new BigInteger(Long.toString(25214903917L));
    lcg.C = new BigInteger(Integer.toString(37));
    lcg.Seed = new BigInteger(Long.toString(12312)); // can use the SystemTime
    lcg.Mod = BigInteger.TWO.pow(48);
    for (int i = 1; i < 101; i++) {
        if (i % 10 == 0) System.out.println(lcg.nextInt());
        else System.out.printf("%-16s", lcg.nextInt());
    }
}
```



We can see the results of the main function as follows:

```

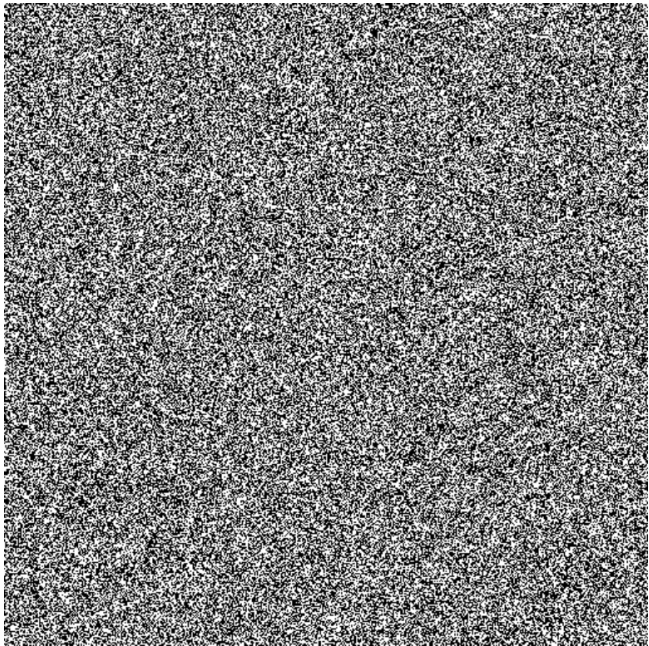
89709203      91023555      74096633      96890338      78350153      63188693      96132678      24105782      87777197      84902870
20991657      39798989      97957722      20309427      75963112      78407314      10698343      27078058      96713150      36576215
32595659      89156376      73625893      58190577      23045030      13737866      83627082      29308850      10959208      37490762
23494906      2411252      1323532      80052985      36917116      2217070      62811068      21678634      86905359      78969891
68775023      85813090      17470413      3346936      8622741      14755136      25648809      20521934      52842472      39083726
82743657      84377120      1127133      54556195      28605648      61418036      6624588      21623608      73663787      55000677
21850300      83440424      46283631      7196474      67846193      69068865      61163285      58982440      4730556      21916924
43640175      91841337      22398702      32569573      4511625      43971591      25618668      19657115      41194308      62567547
67439538      59524100      66078922      13561050      61039623      31202738      59821931      92422321      6241955      7445423
27515988      54668578      50020729      34517170      527431      82649664      49168628      29113101      16514464      75797893

```

Similarly, we can get its distribution, which is distributed more evenly:

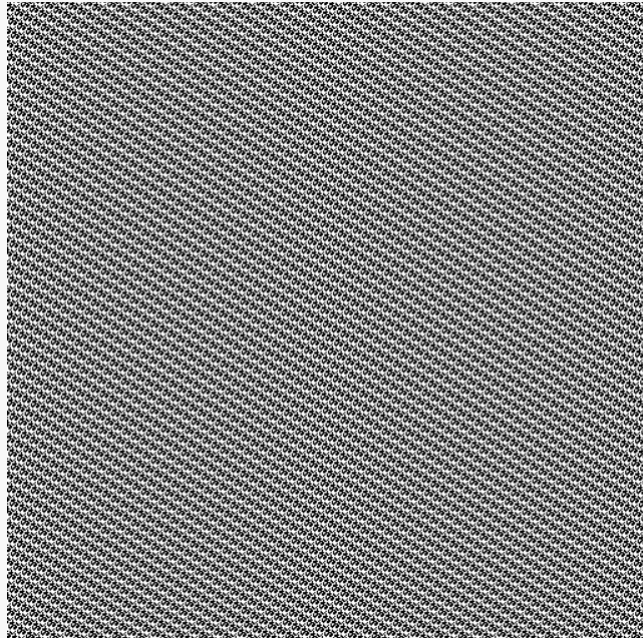
distribution\seed	12312	1
00%-10%	106849	106552
10%-20%	106420	106735
20%-30%	106846	106511
30%-40%	106410	106263
40%-50%	106997	106390
50%-60%	106138	106631
60%-70%	106743	106313
70%-80%	106645	106821
80%-90%	76677	76460
90%-100%	70275	71324

Using Python's drawing tool `Image`, we can see that the results of these values are quite random(first graph). And when taken different m values, we will get very different results:

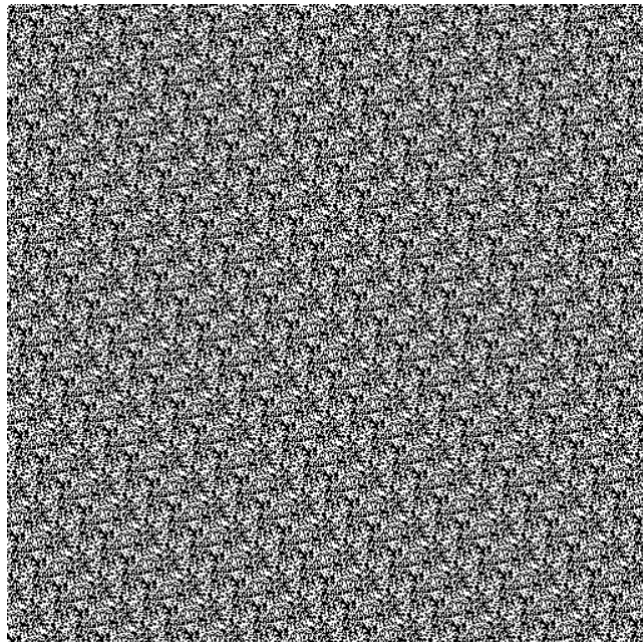
selected m	pixels[i, j] = 0 if rand > 0.5 else 1
<b>m = 281474976710656</b>	



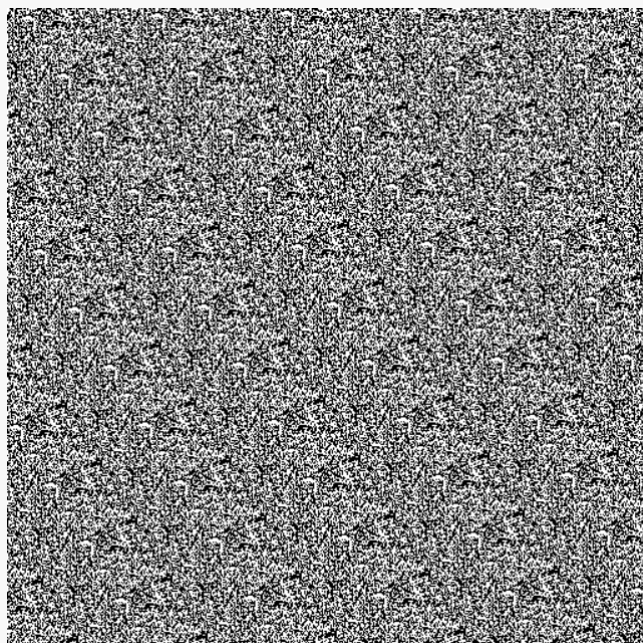
**m = 973 (not prime)**



**m = 2909**



**m = 2181271**



However, Marsaglia's research in 1968 found that its disadvantage was that the spatial distribution of generated random Numbers had a **grid structure**, that is, the points fell on an equidistant parallel hyperplane with a certain number of points. That violates randomness.

In addition, if several generated random Numbers are obtained, the sequence corresponding to the **selected seed value can be reversely solved** in polynomial time, which is also unsafe for the encryption method.

## 3. Program Implementation

### 3.1 Code

```
// I firstly have overloaded all of the above methods with BigInteger parameter
public static void main(String[] args) {
    String plaintext = "CS201";
    Generator lcg = new Generator(new BigInteger( // a c seed mod
        Long.toString(25214903917L)),
        new BigInteger(Integer.toString(37)),
        new BigInteger(Long.toString(System.currentTimeMillis())), // a
random number
        BigInteger.TWO.pow(48));
    long p, q; // p, q is generated by LCG methods & Miller-Rabin testing
    do p = Long.parseLong(lcg.nextInt().toString()); while (!isPrime(p));
    do q = Long.parseLong(lcg.nextInt().toString()); while (!isPrime(q));
    BigInteger Bp = BigInteger.valueOf(p);
    BigInteger Bq = BigInteger.valueOf(q);
    BigInteger Bn = Bp.multiply(Bq);
    BigInteger Bphi =
Bp.subtract(BigInteger.ONE).multiply(Bq.subtract(BigInteger.ONE));
    BigInteger Be = BigInteger.TWO.add(BigInteger.ONE); // 3
    BigInteger Bk = BigInteger.ONE;
    BigInteger Bd;
    while (Bphi.mod(Be).equals(BigInteger.ZERO)) Be = Be.add(BigInteger.ONE);
    while (true) { // find k to satisfy  $ed \equiv 1 \pmod{\phi(n)}$ 
        if
((Bk.multiply(Bphi).add(BigInteger.ONE)).mod(Be).equals(BigInteger.ZERO)) {
            Bd = (Bk.multiply(Bphi).add(BigInteger.ONE)).divide(Be);
            break;
        } else Bk = Bk.add(BigInteger.ONE); }
    Encrypt encrypt = new Encrypt(plaintext, Bn, Be); // message n e
    BigInteger encryptedtext = encrypt.encrypt(); //  $c = m^e \pmod n$ 
    Decrypt decrypt = new Decrypt(Bn, Bd, encryptedtext); // n d c
    String decryptedtext = decrypt.decrypt(); //  $res = c^d \pmod n$ 
    System.out.println("plaintext = " + plaintext);
    System.out.println("p = " + p);
    System.out.println("q = " + q);
    System.out.println("n = " + Bn);
    System.out.println("phi = " + Bphi);
    System.out.println("e = " + Be);
    System.out.println("k = " + Bk);
    System.out.println("d = " + Bd);
    System.out.println("encryptedtext = " + encryptedtext);
    System.out.println("decryptedtext = " + getMessage(decryptedtext));
}
```



## 3.2 Demo

```
plaintext = CS201
p = 174519364459367
q = 101075923484093
n = 17639705928587518219969349131
phi = 17639705928587242624681405672
e = 3
k = 2
d = 11759803952391495083120937115
encryptedtext = 8555277390273280859926444194
decryptedtext = CS201
```

```
plaintext = Love!
p = 186193904038099
q = 183995602405513
n = 34258859537724305048799639787
phi = 34258859537723934859293196176
e = 5
k = 4
d = 27407087630179147887434556941
encryptedtext = 17461603806926689470723907924
decryptedtext = Love!
```

## 4. Realistic demand

---

### 4.1 RSA key length problem

Obviously, the length of the text encrypted using RSA cannot exceed the product of two primes.

One solution is to send data in packets, turning long text into multiple short texts for encryption.

In another solution, in actual use, we can use RSA to encrypt the key of asymmetric encryption method in combination with symmetric encryption method (such as DES, AES). When the receiver wants to send ciphertext, he can encrypt a new private key through the public key of the sender, encrypt the information with the new key, and then send the encrypted private key to the sender together with the encrypted information with the private key. This method not only has the characteristics of fast asymmetric encryption speed, but also can have higher security with the combination of asymmetric encryption and symmetric encryption.

### 4.2 Authentication

In practice, we also need to pay attention to whether a private key is sent by the sender to prevent hackers, which requires the process of adding a digital signature during RSA encryption.

A digital signature signs the original data by signing the sender's private key, which can only be verified by signing with the sender's public key. This method can effectively prevent a third party from forging the sender, and also prevent the information from being modified in the process of transmission. The common signature algorithm is "MD5withRSA", "SHA256withRSA" and so on.

## Reference

---

[https://en.wikipedia.org/wiki/Public-key\\_cryptography](https://en.wikipedia.org/wiki/Public-key_cryptography)

[https://en.wikipedia.org/wiki/One-way\\_function](https://en.wikipedia.org/wiki/One-way_function)

[https://en.wikipedia.org/wiki/Symmetric-key\\_algorithm](https://en.wikipedia.org/wiki/Symmetric-key_algorithm)

[https://en.wikipedia.org/wiki/List\\_of\\_random\\_number\\_generators](https://en.wikipedia.org/wiki/List_of_random_number_generators)

[https://en.wikipedia.org/wiki/Linear\\_congruential\\_generator](https://en.wikipedia.org/wiki/Linear_congruential_generator)

[https://en.wikipedia.org/wiki/Fermat\\_primality\\_test](https://en.wikipedia.org/wiki/Fermat_primality_test)

[https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin\\_primality\\_test](https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test)