ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

ĐẠI HỌC BÁCH KHOA

BÀI TẬP LỚN

MÔN MẬT MÃ & AN NINH MẠNG

ĐỀ TÀI:

HIỆN THỰC HỆ MÃ RSA TRÊN JAVA

LỚP L01 --- NHÓM 20 --- HK 231

Giảng viên hướng dẫn: ThS. Nguyễn Cao Đạt

Sinh viên thực hiện	Mã số sinh viên	Điểm số
Bùi Quang Bằng	2012681	

Thành phố Hồ Chí Minh – 2023

Mục lục

1. Giới thiệu

1.1 Đề tài

Hiện thực hệ mã RSA trên Java/C/C++/Python:

- Java có lớp BigInteger được xây dựng sẵn.
- C++ có thư viện NTL(Library for doing Number Theory) or GMP (the GNU Multiple Precision Arithmetic Library).
- Các bạn có thể dùng các hiện thực big-integer để quản lý dữ liệu của bạn và thực hiện phép toán mod nhưng không được dùng các phương thức đã hiện thực (gcd, power, tìm số nguyên tố, ..). Như vậy các bạn phải tự hiện thực các phương thức này.
- Các bạn có thể dùng hàm an toàn đang tồn tại để tạo ra các số ngẫu nhiên lớn. Ví dụ Java cung cấp các công cụ để tạo số ngẫu nhiên trong java.util.random hay java.security.SecureRandom. Tương tự C++ có rand() và srand() để tạo số ngẫu nhiên.

Trong hiện thực RSA của các bạn, giả sử các số nguyên tố lớn ít nhất phải 500 bits(nhưng có thể lớn hơn) và các bạn phải viết các hàm sau:

- Tìm số nguyên tố lớn khi cho số lượng bit của số nguyên tố lớn cần tìm.
- Tính ước số lớn nhất khi cho hai số nguyên lớn.
- Tính toán khóa giải mã d khi cho khoá mã hóa e và hai số nguyên tố lớn.
- Tạo bộ khóa ngẫu nhiên khi cho 2 số nguyên tố lớn.

- Mã hóa khi cho thông điệp và khóa mã hóa e và n.
- Giải mã khi cho thông điệp mã hóa và khóa giải mã d và n.

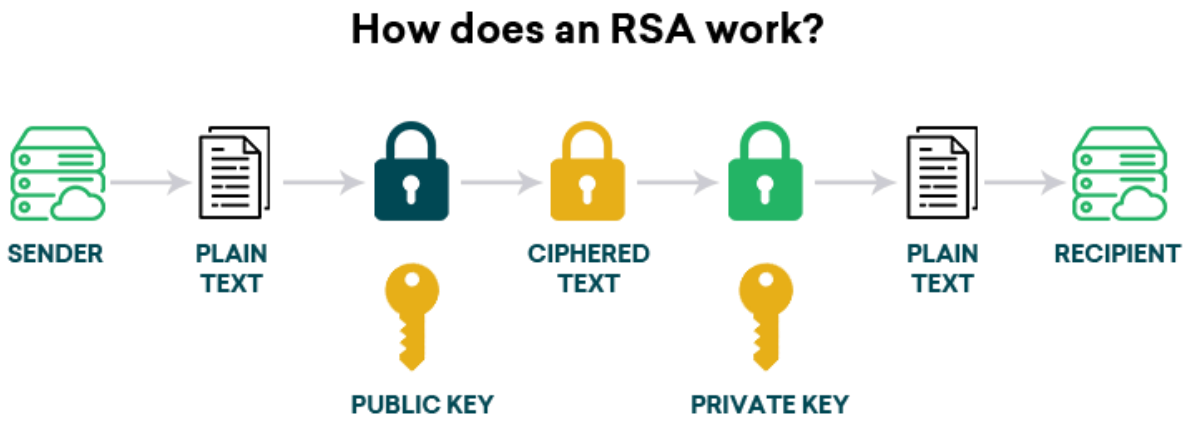
1.2 Mục tiêu cần thực hiện

- Tìm hiểu về hệ mã RSA
- Tìm hiểu và cách dụng thư viện java.math.BigInteger và java.util.Random của ngôn ngữ Java
- Cách thức hiện các phương thức (GCD, Check PrimeNumber, MillerRabin,…)
- Nghiên cứu về cách tạo bộ khóa, mã hóa và giải mã thông điệp cụ thể

2.Phân tích công nghệ được sử dụng

2.1 Cơ sở lý thuyết

RSA là một thuật toán mật mã hóa khóa công khai. Đây là thuật toán đầu tiên phù hợp với việc tạo ra chữ ký điện tử đồng thời với việc mã hóa. Nó đánh dấu một sự tiến bộ vượt bậc của lĩnh vực mật mã học trong việc sử dụng khóa công cộng. RSA đang được sử dụng phổ biến trong thương mại điện tử và được cho là đảm bảo an toàn với điều kiện độ dài khóa đủ lớn.



Một giải pháp để ngăn chặn kẻ nghe trộm và truy cập nội dung tin nhắn là mã hóa nó. Về cơ bản, điều này có nghĩa là thêm một mã vào tin nhắn, nó sẽ làm cho tin nhắn thành một mớ hỗn độn. Nếu mã của bạn đủ phức tạp, thì những người duy nhất có thể truy cập vào thư gốc là những người có quyền truy cập vào mã.

Theo mã hóa RSA, các tin nhắn được mã hóa bằng một mã gọi là public key, mã này có thể được chia sẻ công khai. Do một số đặc thù tính toán khoa học khác biệt của thuật toán RSA, một khi một thông điệp đã được mã hóa bằng public key, nó chỉ có thể được giải mã bằng một key được gọi là private key. Mỗi người dùng RSA có cặp key bao gồm public key và private key của riêng họ. Private key cần được giữ bí mật.

2.2 Mã hóa RSA hoạt động

Chúng bao gồm hàm Trapdoor, tạo số nguyên tố, hàm phi Carmichael. Và các quy trình riêng biệt liên quan đến việc tính toán các public key và private key được sử dụng trong quá trình mã hóa và giải mã.

a) Hàm Trapdoor

Hàm Secure Trapdoor là một hàm Trapdoor trong đó F là hàm một chiều, tức là có thể dễ dàng tính toán $y = F(pk, x) \forall x \in X$ nhưng không thể tính được *x* từ *y* mà không có khóa bí mật sk hay xác suất tính được *x* từ *y* mà không có khóa bí mật sk là không đáng kể.

b) Tạo số nguyên tố

Các chức năng của hàm Trapdoor đã được đề cập ở trên tạo cơ sở cho cách thức hoạt động của các lược đồ mã hóa public key và private key. Các thuộc tính của chúng cho phép chia sẻ public key mà không gây nguy hiểm cho tin nhắn hoặc tiết lộ private key. Chúng cũng cho phép dữ liệu được mã hóa bằng một key theo cách mà chỉ có thể giải mã bằng key khác.

Các số nguyên tố trong RSA cần phải rất lớn và cũng tương đối xa nhau. Các số nhỏ hoặc gần nhau sẽ dễ bị bẻ khóa hơn nhiều. Mặc dù vậy, ví dụ của chúng ta sẽ sử dụng các số nhỏ hơn để làm cho mọi thứ dễ theo dõi và tính toán hơn.

Giả sử kiểm tra tính nguyên tố cho các số nguyên tố mà chúng ta đã sử dụng ở trên, 757 và 1013. Bước tiếp theo là khám phá module *(n)*, sử dụng công thức sau:

$$n = p * q$$

Với $p = 757$ và $q = 1013$

Suy ra

$n = 757 * 1013$

$n = 766841$

c) Tạo khóa

Giả sử An và Bình cần trao đổi thông tin bí mật thông qua một kênh không an toàn (ví dụ như Internet).Với thuật toán RSA, An đầu tiên cần tạo ra cho mình cặp khóa gồm khóa công khai và khóa bí mật theo các bước sau:

- Chọn 2 số nguyên tố lớn *p* và *q* với $p \neq q$, lựa chọn ngẫu nhiên và độc lập.
- Tính: $n = pq$
- Tính: giá trị hàm số Euler $\phi(n) = (p - 1)(q - 1)$
- Chọn một số tự nhiên *e* sao cho $1 < e < \phi(n)$ và là số nguyên tố cùng nhau với $\phi(n)$.
- Tính: *d* sao cho $de \equiv 1 \ (mod\ \phi(n))$.

Một số lưu ý:

- Các số nguyên tố thường được chọn bằng phương pháp xác suất
- Các bước 4 và 5 có thể được thực hiện bằng giải thuật Euclid mở rộng

- Bước 5 có thể viết cách khác:
  Tìm số tự nhiên x, sao cho $d = \frac{x(p-1)(q-1)+1}{e}$ cũng là số tự nhiên
  Khi đó sử dụng giá trị $d\ mod\ (p-1)(q-1)$

Khóa công khai bao gồm:

- *n*: môđun.
- *e*: số mũ công khai(cũng gọi là số mũ mã hóa).

Khóa bí mật bao gồm:

- *n*: môđun, xuất hiện cả trong khóa công khai và khóa bí mật.
- *d*: số mũ bí mật (cũng được gọi là số mũ giải mã hóa).

Một dạng khác của khóa bí mật bao gồm:

- *p* và *q*: hai số nguyên tố chọn ban đầu
- *d mod (p – 1)* và *d mod (q – 1)*: thường được gọi là dmp_1 và dmq_1
- *(1/q) mod p*: thường được gọi là *iqmp*

d) Mã hóa

Giả sử Bình muốn gửi đoạn thông tin M cho An. Đầu tiên Bình chuyển M thành một số m < n theo một hàm có thể đảo ngược (từ m có thể xác định lại M) được thỏa thuận trước.

Lúc này Bình có m và biết n cũng như e do An gửi. Bình sẽ tính c là bản mã hóa của m theo công thức:

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

e) Giải mã hóa

An nhận *c* từ Bình và biết khóa bí mật *d*. An có thể tìm được d từ c theo công thức sau:

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

Biết *m*, An tìm lại *M* theo phương pháp đã thỏa thuận trước. Quá trình giải mã hoạt động vì ta có:

$$c^d \equiv (m^e)^d \equiv m^{ed} \ (\mathrm{mod}\ n)$$

Do *ed = 1 (mod p - 1)* và *ed = 1 (mod q - 1)*, theo Định lý Fermat nhỏ nên:

$$m^{ed} \equiv m \ (mod \ p)$$

$$m^{ed} \equiv m \ (mod \ q)$$

Do *p* và *q* là hai số nguyên tố cùng nhau, áp dụng định lý số dư Trung Quốc, ta có:

$$m^{ed} \equiv m \ (mod \ pq)$$

$$c^d \equiv m \ (mod \ n)$$

2.3 Kiểm tra Miller – Rabin

Kiểm tra Miller - Rabin là một thuật toán xác suất để kiểm tra tính nguyên tố cũng như các thuật toán kiểm tra tính nguyên tố: Kiểm tra Fermat và Kiểm tra Solovay - Strassen.

Khi sử dụng kiểm tra Miller - Rabin chúng ta căn cứ vào một mệnh đề *Q(p, a)* đúng với các số nguyên tố p và mọi số tự nhiên $a \in A \subset N$ và kiểm tra xem chúng có đúng với số n muốn kiểm tra và một số $a \in A$ được chọn ngẫu nhiên hay không. Nếu mệnh đề *Q(p, a)* không đúng, tất yếu n không phải là số nguyên tố, còn nếu *Q(p, a)* đúng, số n có thể là số nguyên tố với một xác suất nào đó. Khi tăng số lần thử, xác suất để n là số nguyên tố tăng lên.

a) Tiêu chuẩn kiểm tra Q(n, a)

Trước hết là một bổ đề về căn bậc hai của đơn vị trong trường hữu hạn Zp, trong đó *p* là số nguyên tố. Chắc chắn rằng 1 và -1 luôn là các căn bậc hai của 1 theo module *p*. Chúng là hai căn bậc hai duy nhất của 1. Thật vậy, giả sử rằng *x* là một căn bậc hai của 1 theo module *p*.

Khi đó:

$$x^2 \equiv 1 \ (mod\ p)$$

$$x^2 - 1 \equiv 0 \ (mod\ p)$$

$$(x - 1)(x + 1) \equiv 0 \ (mod\ p)$$

Từ đó, *x - 1* hoặc *x + 1* là chia hết cho *p*.

Bây giờ giả sử p là một số nguyên tố lẻ, khi đó p - 1 là một số chẵn và ta có thể viết p - 1 dưới dạng $2^s.m$, trong đó *s* là một số tự nhiên ≤ 1 và *m* là số lẻ. Điều này nghĩa là ta rút hết các thừa số 2 khỏi *p - 1*. Lấy số a bất kỳ trong tập {1, 2, ..., p - 1}.

Xét dãy số $x_k = a^{2\wedge k}.m$ với k = 0, 1, 2,..., s. Khi đó $x_k = (x_k - 1)^2$, với k = 1, 2, ..., s và $x_s = a^{p-1}$

Từ định lý Fermat nhỏ:

$$a^{p-1} \equiv 1 \ (mod\ p) \text{ hay}$$

$$x_s \equiv 1 \ (mod\ p) \text{ hay}$$

$$x_{s-1}^2 \equiv 1 \ (mod\ p)$$

Nếu $x_{s-1} \equiv -1 \ (mod\ p)$ thì ta dừng lại.

Nếu $x_{s-1} \equiv 1 \ (mod\ p)$ thì ta tiếp tục với x_{s-2}, sau một số hữu hạn bước.

- hoặc ta có một chỉ số *k, 0 ≤ k ≤ s - 1* sao cho $x_k \equiv -1 \ (mod\ p)$,

- hoặc tới *k = 0* ta vẫn có $x_k \equiv 1 \ (mod\ p)$.

Ta có mệnh đề *Q(p, a)* như sau:

Nếu p là số nguyên tố lẻ và $p - 1 = 2^s.m$ thì $\forall a : 0 < a < p - 1$:

- hoặc $x_k = a^{2\wedge k.m} \equiv 1 \ (mod\ p)$, ∀ k = 0, 1, 2,..., s

- hoặc tồn tại k: $0 \le k \le s$ sao cho $x_k = a^{2\wedge k.m} \equiv -1 \ (mod\ p)$.

b) Số giả nguyển tố

- Theo định lý Fermat nhỏ, với số nguyên tố p ta *có* $\forall a \in \{1, 2, ..., p - 1\} : a^{p-1} \equiv 1 \pmod p$
- Định nghĩa: Hợp số n thoả mãn $a^{n-1} \equiv 1 \pmod n$ với a nào đó được gọi là số giả nguyên tố Fermat
- Số Carmichael: Hợp số n là số giả nguyên tố Fermat với $\forall a \in \{1, 2, \ldots, n\} : GCD(a, n) = 1$ được gọi là số Carmichael
- Định nghĩa: Hợp số n được gọi là số giả nguyên tố mạnh Fermat cơ sở a nếu nó thoả mãn mệnh đề *Q(n, a)*.

c) Giải thuật kiểm tra Miller - Rabin

Định lý: Nếu n là hợp số dương lẻ thì trong các số $a \in \{2, ..., n - 1\}$ tồn tại không quá $\frac{n-1}{4}$ cơ sở a để n là số giả nguyên tố mạnh Fermat.

Gọi A là biến cố "Số n là hợp số", B là biến cố "Kiểm tra Miller - Rabin trả lời n là số nguyên tố".Khi đó xác suất sai của kiểm tra này là xác suất để số n là hợp số trong khi thuật toán cho câu trả lời TRUE, nghĩa là xác suất điều kiện *P(A|B)*.

Theo định lý trên nếu n là hợp số thì khả năng kiểm tra này trả lời TRUE xảy ra với xác suất không vượt quá $\frac{1}{4}$, nghĩa là $\leq \frac{1}{4}$. Tuy nhiên để tính xác suất sai của kiểm tra Miller - Rabin cần tính xác suất diều kiện *P(A|B)*. Dựa trên định lý về ước lượng số các số nguyên tố ta đưa ra ước lượng:

$$P(A) \approx 1 - \frac{2}{lnn} \approx \frac{lnn - 2}{lnn}$$

Theo định lý Bayes trong lý thuyết xác suất ta có công thức để tính xác suất sai của kiểm tra Miller - Rabin là:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$
$$= \frac{P(B|A) * P(A)}{P(B|A) * P(A) + P(B|\neg A) * P(\neg A)}$$

P(A) đã biết ở trên, $P(B|A) \leq \frac{1}{4}$, còn $P(B|\neg A) = 1$ vì khi *n* là số nguyên tố thì chắc chắn mệnh đề *Q(n, a)* là đúng và $P(\neg A) = 1 - P(A) = \frac{2}{lnn}$

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B|A) * P(A) + P(\neg A)} \approx \frac{P(B|A) * (lnn - 2)}{P(B|A) * (lnn - 2) + 2}$$

d) Kiểm tra Miller – Rabin lặp

Theo công thức tính xác suất sai trên đây, với n lớn (cỡ 130 chữ số thập phân), nếu thực hiện phép thử Miller - Rabin chỉ một lần, xác suất sai là khá lớn, tới trên 90%. Để giảm xác suất sai, ta lặp lại phép thử k lần với k số ngẫu nhiên a khác nhau, nếu n vượt qua 50 lần thử *th*ì $P(B|A) \leq \frac{1}{4^k}$ khi thay vào công thức với 50 lần thử nếu cả 50 lần, phép thử đều "dương tính" thì xác suất sai giảm xuống chỉ còn là một số rất nhỏ không vượt quá 9.10^{-29}

3. Hiện thực và đánh giá hệ thống

3.1 Hiện thực

3.1.1 Các hàm thực hiện

Hàm **static BigInteger calculateGCD(BigInteger a, BigInteger b)**: dung để tìm ước chung lớn nhất. Ước chung lớn nhất (GCD, viết tắt của từ Greatest Common Divisor) của hai hay nhiều số là số nguyên dương lớn nhất mà là ước chung (Common Divisor) của tất cả các số đó.

Hàm được hiện thực bằng giải thuật Euclid, lấy hai số chia lấy dư của nhau cho đến khi tìm được số

```
// Function to calculate the Greatest Common Divisor (GCD) of two BigIntegers.
static BigInteger calculateGCD(BigInteger a, BigInteger b) {
    while (true) {
        if (a.compareTo(BigInteger.ZERO) == 0) return b;
        if (b.compareTo(BigInteger.ZERO) == 0) return a;
        if (a.compareTo(b) == 1) {
            a = a.mod(b);
        } else {
            b = b.mod(a);
        }
    }
}
```

Hàm **static BigInteger[] calculateModularInverse(BigInteger a, BigInteger b)**: dung để trả về giá trị x, y sao cho $a * x - b * y = 1$ với $x \leq b$ và $y \leq a$ và *x, y* là nguyển tố cùng nhau

Việc tìm *x, y* được thực hiện bằng giải thuật Euclid mở rộng. Cho *a, b*:

- Nếu $b = 1$: trả về cặp $\{x, y\}$ là {1, a - 1}
- Nếu $a = 0$: trả về cặp $\{x, y\}$ là {1, 0}
- Nếu $a > b$: Dùng ý tưởng đệ quy dựa vào tính chất:

  Nếu tìm được cặp x_1, y_1 thỏa $b.x_1 + (a \bmod b).y_1 = g$ (g bất kì)

thì x, y sẽ có giá trị là

$$x = y_1$$

$$y = x_1 - y_1.[a/b]$$

```java
// Function to find the modular multiplicative inverse of a modulo b.
static BigInteger[] calculateModularInverse(BigInteger a, BigInteger b) {
    if (b.compareTo(BigInteger.ONE) == 0) {
        return new BigInteger[]{
                BigInteger.ONE,
                a.subtract(BigInteger.ONE)
        };
    }
    if (a.compareTo(BigInteger.ONE) == 0) {
        return new BigInteger[]{
                BigInteger.ONE,
                BigInteger.ZERO
        };
    }
    if (a.compareTo(b) != -1) {
        BigInteger[] vals = calculateModularInverse(a.mod(b), b);
        return new BigInteger[]{
                vals[0],
                a.divide(b).multiply(vals[0]).add(vals[1])
        };
    }
    BigInteger[] vals = calculateModularInverse(a, b.mod(a));
    return new BigInteger[]{
            b.divide(a).multiply(vals[1]).add(vals[0]),
            vals[1]
    };
}
```

Hàm **static BigInteger modularExponentiation(BigInteger a, BigInteger b, BigInteger n)**: dung để tìm a^b mod n, đối với 2 số *a*, *b* thuộc tập $[1, 2^{512}]$ thì việc tính trước a^b rồi mới lấy dư n là việc không thể do chi phí tính toán là rất lớn và tốn rất nhiều tài nguyên bộ nhớ

Do đó hàm được hiện thực theo các bước:

- Nhận giá trị cơ số *a*, mũ *b*, số chia lấy dư *n*
- Đặt *res* = 1
- Mỗi vòng lặp (lặp cho đến khi *b* = 1)
    - Nếu b chẵn:
        * $b = b/2$
        * $a = a^2 \, mod \, n$
    - Nếu b lẻ
        * $b = (b - 1)/2$
        * $a = a^2 \, mod \, n$
        * $res = res \times a \, mod \, n$
- Trả về *res* là kết quả của $a^b \, mod \, n$

```java
// Function to calculate a^b mod n using recursive exponentiation.
static BigInteger modularExponentiation(BigInteger a, BigInteger b, BigInteger n) {
    BigInteger two = new BigInteger(val:"2");
    if (b.compareTo(BigInteger.ZERO) == 0) {
        return BigInteger.ONE;
    }
    BigInteger res = modularExponentiation(a, b.divide(two), n);
    if (b.mod(two).compareTo(BigInteger.ZERO) == 0) {
        return res.multiply(res).mod(n);
    }
    return res.multiply(res).mod(n).multiply(a).mod(n);
}
```

Hàm **static BigInteger calculatePhiInverse(BigInteger e, BigInteger p, BigInteger q)**: Tìm nghịch đảo modulo $\phi(n)$ của e gọi là d với $n = pq$, và giá trị $\phi(n) = (p - 1) * (q - 1)$.

Trong hiện thực mã hóa RSA thì

- p, q là hai giá trị nguyên tố lớn được chọn ngẫu nhiên (512, 1024 hoặc 2048 bit).
- $\phi(n)$ là số nguyên tố cùng nhau với n nằm trong đoạn [1, n – 1].

- e là một giá trị được chọn sao cho thỏa tính chất nguyên tố cùng nhau với $\phi(n)$.
- d là nghịch đảo modulo $\phi(n)$ của e.

- Khóa công khai là (e, n) khóa bí mật là (d, n).

```java
// Function to calculate the modular multiplicative inverse of e modulo (p-1)*(q-1).
static BigInteger calculatePhiInverse(BigInteger e, BigInteger p, BigInteger q) {
    BigInteger pMinus1 = p.subtract(BigInteger.ONE);
    BigInteger qMinus1 = q.subtract(BigInteger.ONE);
    BigInteger phi = pMinus1.multiply(qMinus1);
    BigInteger[] arr = calculateModularInverse(e, phi);
    BigInteger d = arr[0];
    return d;
}
```

Hàm **static BigInteger generateRandomBigInteger(BigInteger m, BigInteger n)**

Tìm một số nguyên ngẫu nhiên nhỏ hơn *n* cho trước.

```
// Function to generate a random BigInteger in the range [m, n).
static BigInteger generateRandomBigInteger(BigInteger m, BigInteger n) {
    Random randNum = new Random();
    int len = n.bitLength();
    BigInteger res = new BigInteger(len, randNum);
    while ((res.compareTo(n) != -1) || (res.compareTo(m) == -1)) {
        randNum = new Random();
        res = new BigInteger(len, randNum);
    }
    return res;
}
```

Hàm **static boolean millerRabinTest(BigInteger n, BigInteger a)**: Hàm kiểm tra Miller – Rabin nhận vào *n, a*. Trong đó n là giá trị cần kiểm là số nguyên hay không, *a* là một giá trị được chọn để kiểm thử tính chất (để kiểm tra n có phải nguyên tố sẽ cần có nhiều giá trị *a*).

Hàm được hiện thực theo các bước:

- Nhận input là *n* với *a*.
- Tính *S*, *d* thỏa $2^S.d = n - 1$ và *d* lẻ
- Gán $x = n - 1 / 2^S$
- Lặp với i từ 0 đến n – 1
    - $y = x^2 \, mod \, n$

- Kiểm tra nếu $y = 1 \, mod \, n$ mà $x \neq 1 \, mod \, n$ (tức $x = -1 \, mod \, n$) và $x \neq n - 1$ thì hàm Miller – Rabin trả kết quả sai, n chắc chắn không phải là số nguyên tố
- Không thì gán $x = y$ và tiếp tục vòng lặp

- Kết thúc vòng lặp kiểm tra $y \neq -1$ thì trả về false còn nếu $y = 1$ thì túc là n có 25% khả năng là số nguyên tố *a* là căn bậc 2^S của *1 modulo n* và *a* là một giả nguyên tố mạnh Ferma.

```java
// Function to perform the Miller-Rabin primality test.
static boolean millerRabinTest(BigInteger n, BigInteger a) {
    BigInteger nMinus1 = n.subtract(BigInteger.ONE);
    BigInteger two = new BigInteger(val:"2");
    BigInteger copy = n.subtract(BigInteger.ONE);
    int s = 0;
    while (copy.mod(two).compareTo(BigInteger.ZERO) == 0) {
        s++;
        copy = copy.divide(two);
    }
    BigInteger y = BigInteger.ONE;
    BigInteger d = nMinus1.divide(two.pow(s));
    BigInteger x = modularExponentiation(a, d, n);
    for (int i = 0; i < s; i++) {
        y = modularExponentiation(x, two, n);
        if ((y.compareTo(BigInteger.ONE) == 0) && (x.compareTo(BigInteger.ONE) != 0) && (x.compareTo(nMinus1) != 0)) {
            return false;
        }
        x = y;
    }
    if (y.compareTo(BigInteger.ONE) != 0) {
        return false;
    }
    return true;
}
```

Hàm **static boolean isProbablePrime(BigInteger n)**

Hàm kiểm tra số nguyên tố sử dụng giải thuật Miller – Rabin. Hàm hiện thức gồm các bước:

- Nhận n là số nguyên tố cần kiểm tra
- Lặp 15 lần

- Chọn một số a bất kì nhỏ hơn n
- Kiểm tra nếu *Q(n, a)* trả về false hoặc *a* và *n* không nguyên tố cùng thì trả về kết quả *n* là hợp số.

Nếu cả 15 chọn *a* mà Miller – Rabin đều trả về true thì xác suất *n* là số nguyên tố rất cao. Tuy nhiên, để chắc chắn hơn thì ta cũng sẽ kiểm tra thử xem *n* có chia hết cho các số nguyên tố nhỏ hơn 200 hay không.

```java
// Function to check if a number is prime using the Miller-Rabin test and additional checks.
static boolean isProbablePrime(BigInteger n) {
    boolean bool = true;
    int t = 15;
    BigInteger a = new BigInteger(val:"1");
    BigInteger one = a;

    for (int i = 0; i < t; i++) {
        a = generateRandomBigInteger(one, n);
        if (calculateGCD(a, n).compareTo(BigInteger.ONE) != 0) {
            return false;
        }
        bool = millerRabinTest(n, a);
        if (!bool) {
            return bool;
        }
    }

    String[] primes = {"2", "3", "5", "7", "11", "13", "17", "19", "23", "29", "31", "37", "41", "43", "47", "53",
            "59", "61", "67", "71", "73", "79", "83", "91", "101", "103", "107", "109", "113", "127", "131", "137", "139", "149", "151", "157", "163"};
    for (int i = 0; i < 37; i++) {
        a = new BigInteger(primes[i]);
        if (n.mod(a).compareTo(BigInteger.ZERO) == 0) {
            return false;
        }
    }
    return bool;
}
```

Hàm **static BigInteger generateRandomPrime(int bit)**: có tác dụng tạo một số nguyên tố ngẫu nhiên với độ dài bit nhất định

- Chọn một số random gán với *res*
- Lặp cho đến khi *res* chứa số nguyên tố
    - Nếu *res* chia 6 dư 1 thì gán $res = res + 4$
    - Nếu không thì gán $res = res + 2$

```java
// Function to generate a random prime number with a given number of bits.
static BigInteger generateRandomPrime(int bit) {
    Random randNum = new Random();
    BigInteger res = new BigInteger(bit, randNum);
    BigInteger two = new BigInteger(val:"2");
    BigInteger four = new BigInteger(val:"4");
    BigInteger six = new BigInteger(val:"6");

    boolean isPrime = isProbablePrime(res);
    if (res.mod(two).compareTo(BigInteger.ZERO) == 0) {
        res = res.add(BigInteger.ONE);
    }
    while (!isPrime) {
        if (res.mod(six).compareTo(BigInteger.ONE) == 0) {
            res = res.add(four);
        } else {
            res = res.add(two);
        }
        isPrime = isProbablePrime(res);
    }
    return res;
}
```

Hàm **static BigInteger[] generateStrongPrime(int bit, BigInteger e)**: dung để tạo 1 số nguyên tố mạnh ngẫu nhiên và thỏa tính chất nguyên tố cùng nhau với khóa private *e* (để hiện thực giải thuật RSA).

Định nghĩa số nguyên tố mạnh p nhóm lựa chọn để hiện thực, thỏa các tính chất:

- $p = 1 \ mod \ p_0$ với p_0 là một số nguyên tố lớn
- $p = -1 \ mod \ p_1$ với p_1 là một số nguyên tố lớn

p_0 và p_1 trong hiện thực là những số nguyên tố ngẫu nhiên 128 bit. Các bước thực hiện gồm

- Sinh ra một số nguyên tố lớn 1024 bits gán với *res*.
- Sinh 2 giá trị p_0 và p_1 là những số nguyên tố 128 bit sao cho p_0 khác p_1

- Tính inv_1, inv_2 lần lượt là nghịch đảo modulo của $p_1 \% p_0$, $p_0 \% p_1$
- Gán $crt_1 = inv_1 * p_1$ vậy thỏa:
  - $crt_1 = 1 \, mod \, p_0$
  - $crt_1 = 0 \, mod \, p_1$
- Gán $crt_2 = (p1 - inv_2) * p_0$ vậy crt_2 thỏa:
  - $crt_2 = -1 \, \% \, p_1$
  - $crt_2 = 0 \, \% \, p_0$
- Biến đổi *res*
  - $increment = 2 * p_0 * p_1$
  - $crt = crt_1 + crt_2 + increment$
  - $res = res + (crt - res \, \% \, increment) \, \% \, increment$
- Lúc này res là một tổng có chứa crt, chỉ mơi làm một số có khả năng thỏa tính chất $crt = 1 \, mod \, p_0$, $crt = -1 \, mod \, p_1$
- Chạy vòng lặp, trong mỗi lần lặp kiểm tra nếu res chưa thỏa tính chất một số nguyên tố mạnh và nguyên tố cùng nhau với e thì ra gán $res = res + 4 * p_0 * p_1$. Sau nhiều lần lặp thì ta tìm được một giá trị nguyên tố và số nguyên tố đó cũng sẽ thỏa tính chất của một số nguyên tố mạnh

```java
static BigInteger[] generateStrongPrime(int bit, BigInteger e) {
    BigInteger two = new BigInteger(val:"2");

    BigInteger ten = new BigInteger(val:"10");
    BigInteger lowerBound = ten.pow(exponent:154);
    BigInteger upperBound = two.pow(exponent:512).subtract(BigInteger.ONE);
    BigInteger res = generateRandomBigInteger(lowerBound, upperBound);
    boolean result = false;
    BigInteger p0 = generateRandomPrime(bit:128);
    while (e.mod(p0).compareTo(BigInteger.ZERO) == 0) {
        p0 = generateRandomPrime(bit:128);
    }
    BigInteger p1 = generateRandomPrime(bit:128);
    while ((p1.compareTo(p0) == 0) || (e.mod(p1).compareTo(BigInteger.ZERO) == 0)) {
        p1 = generateRandomPrime(bit:128);
    }
    p1 = p1.multiply(two);

    BigInteger increment = p0.multiply(p1);

    BigInteger[] inv1 = calculateModularInverse(p1, p0);
    BigInteger crt1 = inv1[0];
    crt1 = crt1.multiply(p1);

    BigInteger[] inv2 = calculateModularInverse(p0, p1);
    BigInteger crt2 = p1.subtract(inv2[0]);
    crt2 = crt2.multiply(p0);

    BigInteger crt = crt1.add(crt2).add(increment);
    BigInteger resmod = res.mod(increment);
    res = res.add(crt.subtract(resmod).mod(increment));
    increment = increment.multiply(two);

    while (true) {
        boolean possiblePrime = true;
        if (possiblePrime) {
            if (calculateGCD(e, res.subtract(BigInteger.ONE)).compareTo(BigInteger.ONE) != 0) {
                possiblePrime = false;
            }
        }
        if (possiblePrime) {
            result = isProbablePrime(res);
            if (result) {
                break;
            }
        }
        res = res.add(increment);
    }
    return new BigInteger[]{res, p0, p1.divide(two)};
}
```

Hàm mã hóa và giải mã hóa

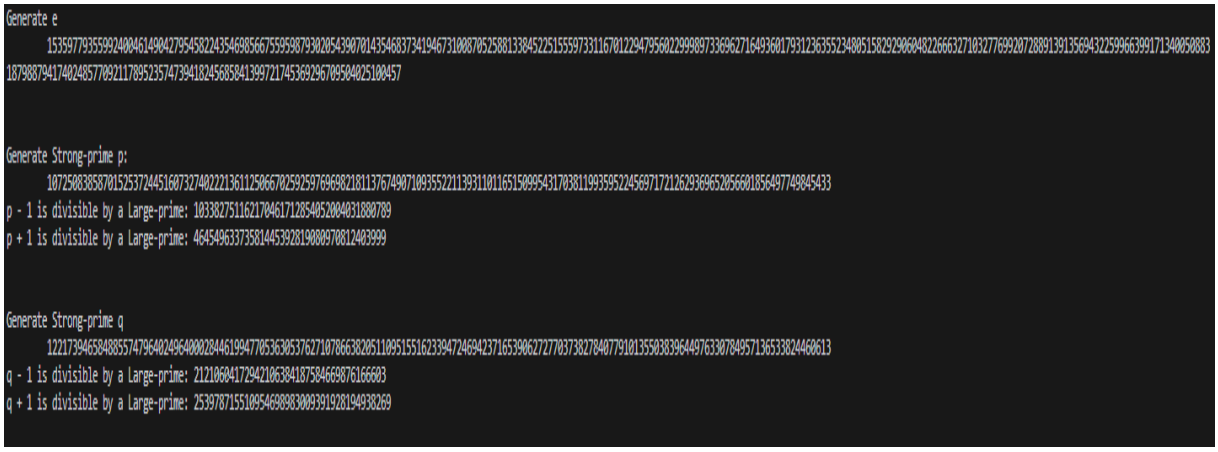
```java
// Function to encrypt a message m using public key (e, p, q).
static BigInteger performEncryption(BigInteger m, BigInteger e, BigInteger p, BigInteger q) {
    BigInteger N = p.multiply(q);
    return modularExponentiation(m, e, N);
}

// Function to decrypt a ciphertext c using private key (d, p, q).
static BigInteger performDecryption(BigInteger c, BigInteger d, BigInteger p, BigInteger q) {
    BigInteger N = p.multiply(q);
    return modularExponentiation(c, d, N);
}
```

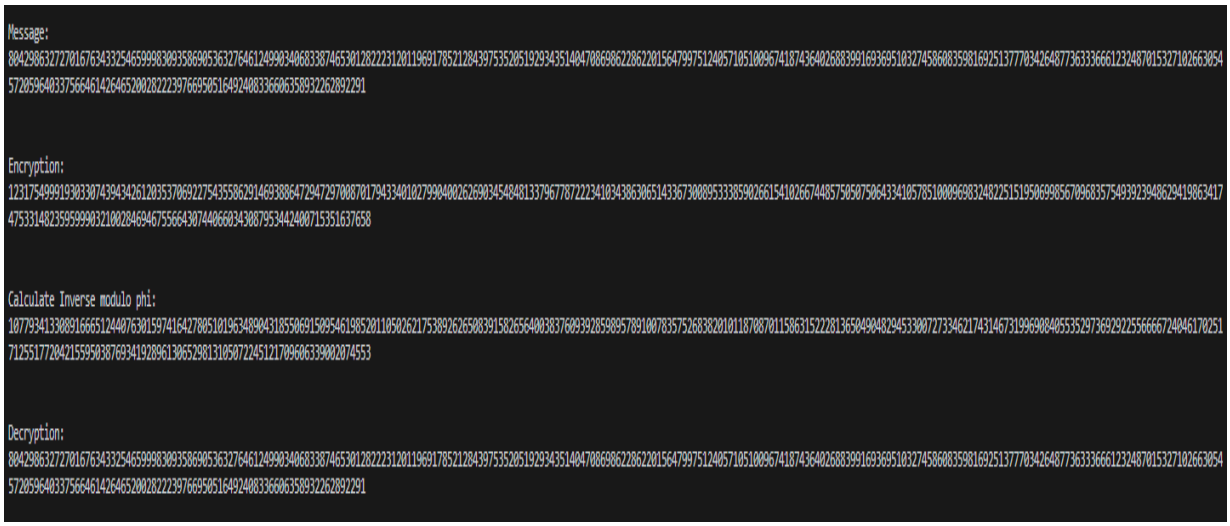
3.1.2 Demo

Toàn bộ source code của project được upload tại:

https://github.com/QuangBang681/RSA



Để minh họa cho thuật toán, ta tiến hành tạo các số ngẫu nhiên có giá trị lớn (512 bit), với p, q là số nguyên tố, thông điệp (message) là một số ngẫu nhiên. Sử dụng thuật toán mã hóa của RSA ta tiến hành mã hóa, sau đó lại giải mã thông điệp đó ra. Toàn bộ thông tin được in lên màn hình.



3.2 Đánh giá

Hệ mã hóa RSA là một thuật toán mật mã bất đối xứng phổ biến, sử dụng cặp khóa công khai và riêng tư. Thuật toán này đảm bảo tính an toàn trong truyền thông và lưu trữ dữ liệu bằng cách mã hóa thông tin với khóa công khai và giải mã bằng khóa riêng tư. RSA thường được áp dụng trong việc tạo chữ ký điện tử và xác thực thông tin, đóng vai trò quan trọng trong bảo mật thông tin và giao tiếp trực tuyến

4. Kết luận

4.1 Chương trình đã làm được

- Tìm số nguyên tố lớn khi cho số lượng bit của số nguyên tố lớn cần tìm.
- Tính ước số lớn nhất khi cho hai số nguyên lớn.
- Tính toán khóa giải mã d khi cho khoá mã hóa e và hai số nguyên tố
- Tạo bộ khóa ngẫu nhiên khi cho 2 số nguyên tố lớn.
- Mã hóa khi cho thông điệp và khóa mã hóa e và n.
- Giải mã khi cho thông điệp mã hóa và khóa giải mã d và n.

4.2 Chương trình chưa làm được

Nhận vào thông điệp và khóa từ bên ngoài.

4.3 Ưu điểm

- Số dòng code ít (khoảng 300 dòng)
- Sử dụng rất ít các thư viện cho sẵn của Java.
- Tốc độ thực thi tương đối nhanh.

4.4 Nhược điểm

- Chưa cung cấp cho người dùng khả năng tương tác từ các thiết bị ngoại vi.
- Chưa thể sử dụng như một module tích hợp vào chương trình khác.
- Độ phức tạp giải thuật còn khá cao (O(n2) ở hàm kiểm tra số nguyên tố khi sử dụng hàm Miller - Rabin có chứa vòng lặp, bên trong 1 vòng lặp khác).

5. Tài liệu tham khảo

References

[1] Wikipedia - RSA. (cryptosystem)

https://en.wikipedia.org/wiki/RS

[2] Hệ mã RSA và chữ ký số.

https://viblo.asia/p/he-ma-hoa-rsa-va-chu-ky-so-6J3ZgkgMZmB

[3] RSA là gì? Cơ chế hoạt động và ứng dụng của thuật toán RSA

https://bkhost.vn/blog/rsa/