

Algoritmo RSA

Autor

- Thiago Senhorinha Rose (12100774)

Implementação

Será apresentado apenas os pontos mais importantes. O código completo está no final do relatório.

Setup

Durante a construção da classe RSA é chamado o método `setup()` que tem o objetivo de preparar todas as variáveis necessárias para criptografar e descriptografar utilizando o algoritmo RSA

1. São gerados dois primos de 2048 bits **p** e **q**.
2. Calculamos o número **n** que será utilizado como modulus tanto para computação da chave pública como chave privada.
3. Calculamos $\phi(n)$
4. Sorteamos um número **e** no intervalo $(1, \phi(n))$ que seja coprimo a $\phi(n)$. O **e** representa o expoente da chave pública.
5. Calculamos **d**, expoente da chave privada, que é o inverso multiplicativo de **e** mod $\phi(n)$.

Primeiramente. Após isso, calculamos o número **n** que será utilizado como modulus tanto para chave pública como chave privada.

```
private void setup() {  
    p = generatePrimeWithNumberOfBits(NUMBER_OF_BITS);  
    q = generatePrimeWithNumberOfBits(NUMBER_OF_BITS);  
    n = computeN();  
    eulerPhiFunctionOfN = computeEulerPhiFunctionOfN();  
    e = generateE();  
    d = computeMultiplicativeInverse(e, eulerPhiFunctionOfN);  
}
```

1. BigInteger generatePrimeWithNumberOfBits(int numberOfBits)

Retorna um provável primo aleatório com o número de bits especificado no parâmetro **numberOfBits**.

Utilizado para criar os dois primos bases para geração das chaves.

```
private BigInteger generatePrimeWithNumberOfBits(int numberOfBits) {  
    return BigInteger.probablePrime(numberOfBits, random);  
}
```

2. BigInteger computeN()

Realiza a multiplicação entre os primos sorteados anteriormente.

O número resultado da multiplicação é o modulus utilizado nas chaves pública e privada.

```
private BigInteger computeN() {  
    return p.multiply(q);  
}
```

3. BigInteger computeEulerPhiFunctionOfN()

Retorna a função euler do **n**, ou seja, $\phi(n)$. Como **n** = **p****x****q** podemos escrever $\phi(n) = \phi(p)\phi(q)$. Sendo **p** e **q** primos a fórmula se resume a $(p-1)(q-1)$.

A função calculada é utilizada para sortear o expoente da chave pública.

```
private BigInteger computeEulerPhiFunctionOfN() {  
    BigInteger eulerPhiFunctionOfP = p.subtract(ONE);  
    BigInteger eulerPhiFunctionOfQ = q.subtract(ONE);  
    return eulerPhiFunctionOfP.multiply(eulerPhiFunctionOfQ);  
}
```

4. BigInteger generateE()

Sorteamos um número **e** no intervalo $(1, \phi(n))$ que seja coprimo a $\phi(n)$. O número sorteado representa o expoente da chave pública.

```
private BigInteger generateE() {  
    BigInteger propablyE = null;  
    boolean eNotFound = true;  
    while (eNotFound) {  
        propablyE = BigIntegerRandomGenerator.generate(TWO, eulerPhiFunctionOfN.subtract(ONE));  
        if (propablyE.gcd(eulerPhiFunctionOfN).compareTo(ONE) == 0) {  
            eNotFound = false;  
        }  
    }  
    return propablyE;  
}
```

5. computeMultiplicativeInverse(BigInteger e, BigInteger m)

Faz o cálculo do **d**, expoente da chave privada, que é o inverso multiplicativo de **e** mod $\phi(n)$.

```
private BigInteger computeMultiplicativeInverse(BigInteger e, BigInteger m) {  
    return e.modInverse(m);  
}
```

Formação das chaves

- Pública: **(n, e)**
- Privada: **d**

Cifrando

A cifra é resultado da fórmula: **$m^e \bmod n$**

```
public BigInteger encrypt(char c) {  
    BigInteger m = new BigInteger(String.valueOf((int) c));  
    m = m.modPow(e, n);  
    return m;  
}
```

Decifrando

O caractere original é obtido através da formula **$c^d \bmod n$**

```
public char decrypt(BigInteger c) {  
    return (char) c.modPow(d, n).intValue();  
}
```

Extra. Cifrando e Decifrando String

Para cifrar uma String **c** esta é separada em caracteres e cifrada caractere por caractere utilizando o encrypt(char c).

```
public List<BigInteger> encrypt(String c) {  
    List<BigInteger> result = new ArrayList<>();  
    for (char f : c.toCharArray()) {  
        result.add(encrypt(f));  
    }  
    return result;  
}
```

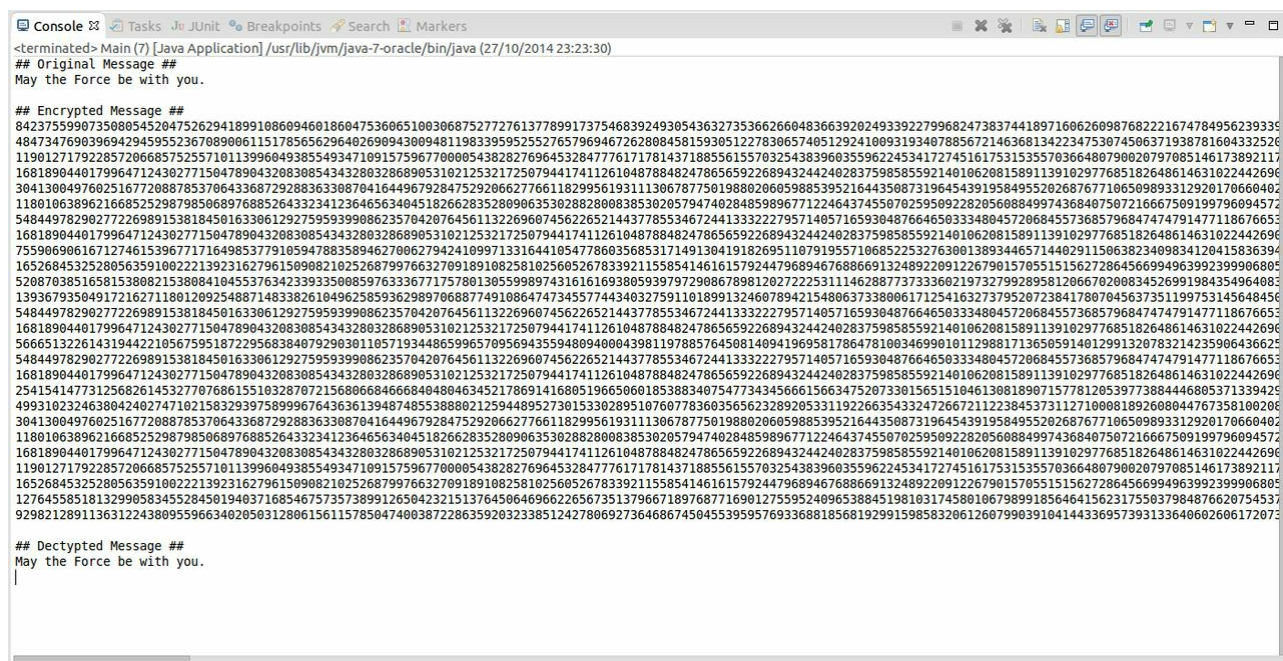
No deciframento é passado por parâmetro uma lista de cifras que são individualmente decifradas utilizando o `decrypt(BigInteger c)` e concatenadas em uma `String`.

```
public String decrypt(List<BigInteger> list) {
    String result = "";
    for (BigInteger c : list) {
        result = result.concat(String.valueOf(decrypt(c)));
    }
    return result;
}
```

Execução

Código main

```
public static void main(String[] args) {
    RSA rsa = new RSA();
    String c = "May the Force be with you.";
    System.out.println("## Original Message ##");
    System.out.println(c);
    System.out.println("");
    List<BigInteger> encrypt = rsa.encrypt(c);
    System.out.println("## Encrypted Message ##");
    for (BigInteger character : encrypt) {
        System.out.println(character);
    }
    System.out.println("");
    String decrypt = rsa.decrypt(encrypt);
    System.out.println("## Decrypted Message ##");
    System.out.println(decrypt);
}
```



```
<terminated> Main (7) [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (27/10/2014 23:23:30)
## Original Message ##
May the Force be with you.

## Encrypted Message ##
8423755990735080545204752629418991086094601860475360651003068752772613778991737546839249305436327353662660483663920249339227996824738374418971606260987682221674784956239335
48473476903969429459552367089006115178565629640269094300948119833959525527657969467262808458159305122783065740512924100931934078856721463681342234753074506371938781604332526
11901271792285720668572557101139960493855493471091575967700005438282769645328477617181437188556155703254383960355962245341727451617531535570366480790820797085146173892117
16818904401799647124302771504789043208308543432803286890531021253217250794417411261048788482478656592268943244240283759858559214010620815891139102977685182648614631022442696
30413004976025167720887853706433687292883633087041644967928475292066277661182995619311130678775019880206059885395216443508731964543919584955202687677106509893312920170660402
11801063896216685252987985068976885264332341236465634045182662835280906353028828080385302057947402848598967712246437455070259509228205608849974368407507216667509199796094572
548449782902772269891538184501633061292759593990862357042076456113226960745622652144377855346724413332279571405716593048766465033480457206845573685796847474791477118676655
16818904401799647124302771504789043208308543432803286890531021253217250794417411261048788482478656592268943244240283759858559214010620815891139102977685182648614631022442696
7595069061671274615396771164985377910594788358946270062794241099713316441054778603568531714913041918269511079195571068522532763081309344657144029115063823409834120415836394
165268453252805635910022213923162796150908210252687997663270918910825810256052678339211558541461615792447968946768869132489220912267901570551515627286456699496399239906805
52087038516581538082153608410455376342339350808597633667175780130559897431616938059397929086789812027222531114628877373336021973279289581206670208834526991984354964083
13936793504917216271180128925488714833826104962585936298970688774910864747345577443403275911018991324607894215406637338006171254163273795207238417807045637351199753145648456
548449782902772269891538184501633061292759593990862357042076456113226960745622652144377855346724413332279571405716593048766465033480457206845573685796847474791477118676655
16818904401799647124302771504789043208308543432803286890531021253217250794417411261048788482478656592268943244240283759858559214010620815891139102977685182648614631022442696
566651322614319442105675951872295683840792903811057193448659965709569435948094000439811978857645081409419695817864781003469901011298817136505914012991320783214235906436625
548449782902772269891538184501633061292759593990862357042076456113226960745622652144377855346724413332279571405716593048766465033480457206845573685796847474791477118676655
16818904401799647124302771504789043208308543432803286890531021253217250794417411261048788482478656592268943244240283759858559214010620815891139102977685182648614631022442696
25415414773125682614532770768615510328707215608068466684048046345217869141680519665060185388340754773434566615663475207330156515184613081890715778120539773884446805371339425
49931023246380424027471021583293975899967643636139487485538880212594489527301533028951076077836035656232892053311922663543324726672112238453731127100081892608044767358100208
30413004976025167720887853706433687292883633087041644967928475292066277661182995619311130678775019880206059885395216443508731964543919584955202687677106509893312920170660402
11801063896216685252987985068976885264332341236465634045182662835280906353028828080385302057947402848598967712246437455070259509228205608849974368407507216667509199796094572
16818904401799647124302771504789043208308543432803286890531021253217250794417411261048788482478656592268943244240283759858559214010620815891139102977685182648614631022442696
11901271792285720668572557101139960493855493471091575967700005438282769645328477617181437188556155703254383960355962245341727451617531535570366480790820797085146173892117
165268453252805635910022213923162796150908210252687997663270918910825810256052678339211558541461615792447968946768869132489220912267901570551515627286456699496399239906805
12764585181329905834552845019403716854675735738991265042321513764506496622656735137966718976877169012759524096538845198103174580106798991856464156231755037984876620754537
92982128911363122438095596634020503128061561157850474003872286359203233851242780692736468674504553959576933688185681929915985832061260799039104144336957393133640602606172073
```

```
## Decrypted Message ##
May the Force be with you.
```

Código completo

Classe responsável pelo ciframento e deciframento.

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RSA {

    private final int NUMBER_OF_BITS = 2048;
    private final BigInteger ONE = BigInteger.ONE;
    private final BigInteger TWO = BigInteger.valueOf(2);

    private Random random;

    private BigInteger p;
    private BigInteger q;
    private BigInteger n;
    private BigInteger eulerPhiFunctionOfN;
    private BigInteger e;
    private BigInteger d;

    public RSA() {
        random = new Random();
        setup();
    }

    public void changeKeys() {
        setup();
    }

    public BigInteger encrypt(char c) {
        BigInteger m = new BigInteger(String.valueOf((int) c));
        m = m.modPow(e, n);
        return m;
    }

    public char decrypt(BigInteger c) {
        return (char) c.modPow(d, n).intValue();
    }

    public List<BigInteger> encrypt(String c) {
        List<BigInteger> result = new ArrayList<>();
        for (char f : c.toCharArray()) {
            result.add(encrypt(f));
        }
        return result;
    }
}
```

```

public String decrypt(List<BigInteger> list) {
    String result = "";
    for (BigInteger c : list) {
        result = result.concat(String.valueOf(decrypt(c)));
    }
    return result;
}

private void setup() {
    p = generatePrimeWithNumberOfBits(NUMBER_OF_BITS);
    q = generatePrimeWithNumberOfBits(NUMBER_OF_BITS);
    n = computeN();
    eulerPhiFunctionOfN = computeEulerPhiFunctionOfN();
    e = generateE();
    d = computeMultiplicativeInverse(e, eulerPhiFunctionOfN);
}

private BigInteger generatePrimeWithNumberOfBits(int numberOfBits) {
    return BigInteger.probablePrime(numberOfBits, random);
}

private BigInteger computeN() {
    return p.multiply(q);
}

private BigInteger computeEulerPhiFunctionOfN() {
    BigInteger eulerPhiFunctionOfP = p.subtract(ONE);
    BigInteger eulerPhiFunctionOfQ = q.subtract(ONE);
    return eulerPhiFunctionOfP.multiply(eulerPhiFunctionOfQ);
}

private BigInteger generateE() {
    BigInteger propablyE = null;
    boolean eNotFound = true;
    while (eNotFound) {
        propablyE = BigIntegerRandomGenerator.generate(TWO, eulerPhiFunctionOf
N. subtract(ONE));
        if (propablyE.gcd(eulerPhiFunctionOfN).compareTo(ONE) == 0) {
            eNotFound = false;
        }
    }
    return propablyE;
}

private BigInteger computeMultiplicativeInverse(BigInteger e, BigInteger
m) {
    return e.modInverse(m);
}
}

```