E-fólio A | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Segurança em Redes e Computadores

CÓDIGO: 21181

DOCENTE: Henrique S. Mamede

A preencher pelo estudante

NOME: Ricardo Ferreira da Conceição Dias Marques

N.º DE ESTUDANTE: 1100281

CURSO: Licenciatura em Engenharia Informática

DATA DE ENTREGA: 21 de novembro de 2018

TRABALHO / RESOLUÇÃO:

Para a implementação de ambas as Cifras (Simétrica e Assimétrica), usei a linguagem de programação **Java**, num ambiente **Linux**. Especificamente, usei o **OpenJDK versão 1.8.0_181** num computador com **Ubuntu MATE 16.04.5 LTS**. Já este Relatório foi elaborado usando o processador de texto **LibreOffice Writer 5.1.6.2** com a extensão **TexMaths** (extensão usada para introduzir expressões escritas em LaTeX).

1 - Cifra Simétrica (*Cifra de Vigenère - Vigenère cipher)*

Para a Cifra Simétrica, escolhi a **Cifra de *Vigenère***. Conforme é descrito em Stallings (2007, pág. 102), a Cifra de Vigenère é uma cifra polialfabética de substituição (***polyalphabetic substitution cipher***), na qual o conjunto de regras de substituição monoalfabéticas consiste em 26 Cifras de César com deslocações (*shifts) de* 0 a 25.

Na implementação que desenvolvi, por simplificação, o texto a cifrar (*plaintext*) e a chave (key*)* são ambos convertidos em **maiúsculas**. Outra simplificação que assumi foi que o *plaintext* introduzido **não** contém algarismos, carateres acentuados, espaços ou outros carateres especiais.

A *source* Java está no ficheiro "**Vigenere.java"**. Para o programa correspondente ser executado, deve ser executado da seguinte forma (o símbolo "$" é apenas a *prompt* da *shell*):

```
$ java -classpath . Vigenere
```

O programa pede ao utilizador se quer cifrar ("**C"**) ou decifrar ("**D**") um texto.

Se o utilizador quiser **cifrar** um texto, então deverá começar por introduzir a opção "**C".** O programa pedirá então ao utilizador qual o texto a cifrar (*plaintext*) e qual a chave respetiva (*key).* O programa irá então o algoritmo da Cifra de Vigenère e irá mostrar ao utilizador o texto cifrado correspondente (*ciphertext*). Segue abaixo um exemplo de execução para o *plaintext "***criptografia"** e para a *key* "**chave**"**:**

```
$ java -classpath . Vigenere
CIFRA DE VIGENERE

Digite a opção pretendida seguida de ENTER:
C = Cifrar
D = Decifrar
C
Qual e' o plaintext a cifrar?
criptografia
Qual e' a Key?
chave
plaintext = CRIPTOGRAFIA
key = CHAVE
ciphertext = EYIKXQNRVJKH
```

Se o utilizador quiser **decifrar** um texto, então o processo será semelhante mas inverso. Ou seja: o utilizador deverá começar por introduzir a opção "**D".** O programa pedirá então ao utilizador qual

o texto cifrado (*ciphertext*) e qual a chave respetiva (*key).* O programa irá então aplicar o algoritmo da Cifra de Vigenère e irá mostrar ao utilizador o texto correspondente (*plaintext*). Segue abaixo um exemplo de execução para o ciphertext ***"EYIKXQNRVJKH"*** (igual ao do exemplo acima) e para a *key* "**chave**", para demonstrar que obtemos o *plaintext* original ("**CRIPTOGRAFIA**")**:**

```
$ java -classpath . Vigenere
CIFRA DE VIGENERE

Digite a opção pretendida seguida de ENTER:
C = Cifrar
D = Decifrar
D
Qual e' o ciphertext a decifrar?
EYIKXQNRVJKH
Qual e' a Key?
chave
ciphertext = EYIKXQNRVJKH
key = CHAVE
plaintext Decifrado = CRIPTOGRAFIA
```

O método "**cifrar**" implementado realiza os seguintes passos:

1. – Converte a capitalização das letras do *plaintext* e da *key* para ficarem as letras todas em maiúsculas, guardando as *strings* capitalizadas nas variáveis do tipo String "*plaintextMaiusculas*" e "*keyMaiusculas*" respetivamente

2. – Cria *arrays* de carateres para cada uma das *strings* capitalizadas*,* guardando os *arrays* nas variáveis "*plaintextMaiusculasCharArray*" e "*keyMaiusculasCharArray*" respetivamente.

3. – Percorre, num ciclo "for", os carateres do *array* de carateres do *plaintext*. Cada iteração do ciclo trata um carater do "*plaintextMaiusculasCharArray*", em que:

3.1. – É obtido o número (código numérico) da tabela ASCII do carater. Por exemplo, o código numérico de **A** ("letra a maiúscula") é **65**

3.2. – É obtido o número (código numérico) da tabela ASCII do carater cifrado resultante da aplicação do Algoritmo da cifra de Vigenère. Em Stallings (2007, pág. 103) é explicitado, na equação (3.3), que o processo de cifra é o seguinte:

$$\mathrm{C}_i = (p_i + k_{i \ mod \ m}) \ mod \ 26$$

em que C_i é o carater cifrado para uma posição "i" ; "p_i" é o carater em *plaintext* para a mesma posição ; $\mathrm{k}_{i \ mod \ m}$ representa o carater da chave que vai "rodando" (m é o comprimento em carateres da chave ; mod é o operador de "resto" da divisão inteira) e, no final, é aplicado o resto da divisão por 26, dado que 26 é o número de carateres do alfabeto. A linha de código correspondente é a seguinte:

```
int novoCodigoNumerico = (codigoNumerico + keyMaiusculasCharArray[indiceKey]) %
NUMERO_CARATERES_ALFABETO;
```

Para "acertar" o código numérico com o código correspondente da tabela ASCII, torna-se necessário somar o código ASCII da primeira letra do alfabeto (correspondente) à letra **A**:

```
int novoCodigoNumericoAlfabetico = novoCodigoNumerico + CODIGO_ASCII_LETRA_A_MAIUSCULA;
```

É obtido o carater correspondente cifrado e guardado na variável "charCifrado" e o mesmo é concatenado à variável *string* que tem o texto cifrado (*ciphertext*).

Nesse ponto, avançamos para o carater seguinte da chave. Caso já tenhamos chegado ao último carater da chave, regressamos ao carater inicial.

O método "**decifrar**" implementado é semelhante ao cifrar, embora com um mecanismo "inverso":

1. – Guarda as *strings* capitalizadas do *ciphertext* e a *key* nas variáveis do tipo String "*ciphertextMaiusculas*" e "*keyMaiusculas*" respetivamente.

2. – Cria arrays de carateres para cada uma das strings capitalizadas, guardando os arrays nas variáveis "*ciphertextMaiusculasCharArray*" e "*keyMaiusculasCharArray*" respetivamente.

3. – Percorre, num ciclo "for", os carateres do *array* de carateres do *ciphertext*. Cada iteração do ciclo trata um carater do "*ciphertextMaiusculasCharArray*", em que:

3.1. – É obtido o número (código numérico) da tabela ASCII do carater.

3.2. – É obtido o número (código numérico) da tabela ASCII do carater decifrado resultante da aplicação do Algoritmo da cifra de Vigenère. Em Stallings (2007, pág. 103) é explicitado, na equação (3.4), que o processo de decifra é o seguinte:

$$\mathrm{p}_i = (C_i - k_{i \bmod m}) \bmod 26$$

… em que os vários termos (p_i, C_i ...) têm o mesmo sentido que na equação (3.3) que foi descrita anteriormente. A linha de código correspondente é a seguinte:

```
int novoCodigoNumerico = (codigoNumerico - keyMaiusculasCharArray[indiceKey] +
NUMERO_CARATERES_ALFABETO) % NUMERO_CARATERES_ALFABETO;
```

Para "acertar" o código numérico com o código correspondnete da tabela ASCII, torna-se necessário somar o código ASCII da primeira letra do alfabeto (correspondente) à letra **A**:

```
int novoCodigoNumericoAlfabetico = novoCodigoNumerico + CODIGO_ASCII_LETRA_A_MAIUSCULA;
```

É obtido o carater correspondente cifrado (variável "*charCifrado*") e o mesmo é concatenado à variável *string* que tem o texto cifrado (variável "*ciphertext*").

Nesse ponto, avançamos para o carater seguinte da chave. Caso já tenhamos chegado ao último carater da chave, regressamos ao carater inicial.

2 - Cifra Assimétrica (Algoritmo RSA – *RSA Algorithm)*

Para a Cifra Assimétrica, escolhi o **Algoritmo RSA** criado por **Ron Rivest, Adi Shamir** e **Len Adleman**, no MIT, em 1977. O Algoritmo é descrito, por exemplo, em Stallings (2007, págs. 294-302). Trata-se de um algoritmo em que é gerado um par de chaves (**chave pública** e **chave privada**). Na pág. 297 do Livro, é indicado o processo de geração de chave ("***Key generation by***

Alice"), que foi seguido de perto no código que implementei. O código que implementei usa bastante a classe **BigInteger** para várias operações necessárias.

1º – São gerados 2 números primos inteiros grandes ("**p**" e "**q**"). Foi definida uma constante NUMERO_BITS à qual foi atribuído o valor 512 (numa implementação real, para maior segurança seria usado um tamanho de chave superior; pelo menos, 2048 bits). Cada um dos números primos inteiros terá o tamanho de metade dessa chave, para que o produto de ambos tenha o tamanho de NUMERO_BITS. Para a geração dos números primos inteiros é usado o método **ProbablePrime** da classe **BigInteger** que é um método probabilístico (não determinístico) de geração de números primos e que recebe 2 parâmetros (o número de bits e um número aleatório):

```
BigInteger p = BigInteger.probablePrime(NUMERO_BITS / 2, new Random());
BigInteger q = BigInteger.probablePrime(NUMERO_BITS / 2, new Random());
```

2º – É calculado o produto de "p" e "q" e atribuído ao número compósito "**n**" :

```
   BigInteger n = p.multiply(q);
```

3º – É calculado $\phi(n) = (p-1)(q-1)$:

```
BigInteger p_menos_1 = p.subtract(BigInteger.ONE);
BigInteger q_menos_1 = q.subtract(BigInteger.ONE);
BigInteger phi_de_n = p_menos_1.multiply(q_menos_1);
```

4º – É selecionado um número inteiro "**e**" tal que $\gcd(\phi(n), e) = 1; 1 < e < phi(n)$em que "gcd" = *greatest common divisor* (máximo divisor comum). Para acelerar as operações, foi usada a sugestão apresentada em Stallings (2007, pág. 299) de usar o valor de 65537 (2^{16} + 1) para "e". Esse valor foi guardado numa constante (EXPOENTE_PUBLICO):

```
static final BigInteger EXPOENTE_PUBLICO = BigInteger.valueOf(65537);
(…)
BigInteger e = EXPOENTE_PUBLICO;
```

5º – É calculado um expoente privado "**d"** tal que $\mathrm{d} \equiv e^{-1}\ mod\ \phi(n)$ Para tal, recorreu-se ao método **modInverse** também da classe BigInteger:

```
BigInteger d = e.modInverse(phi_de_n);
```

A Chave Privada é o par {d, n} e a Chave Pública é o par {e, n}.

O programa solicita ao utilizador o *plaintext* a cifrar. Para efeitos de demonstração, o programa gera o par de chaves, cifra o texto com a Chave Pública e depois volta a decifrá-lo com a Chave Privada, obtendo o *plaintext* original (capitalizado) para mostrar que o algoritmo funciona.

Para o processo de **cifra**, o programa começa por converter a *string* num *array de bytes e depois converte* esse *array de bytes num* **BigInteger** (na função "stringToBigInt" no ficheiro "RSA.java"):

```
     byte[] arrBytes = plaintext.getBytes();
     BigInteger bi = new BigInteger(arrBytes);
```

A seguir é usado o processo de geração do Ciphertext, a partir da Chave Pública, que em Stallings (2007, pág. 297) é formalizado como $\mathrm{C} = \mathrm{M}^e\ mod\ n$ No código, foi usado o método **modPow** da classe **BigInteger** para esse efeito:

```
BigInteger biCiphertext = biPlaintext.modPow(e, n);
```

Para ilustrar o processo de **decifra** (obtenção do *Plaintext* correspondente) é aplicado a seguir o processo de obtenção do *Plaintext*, a partir da Chave Privada, que em Stallings (2007, pág. 297) é formalizado como $\mathrm{M} = \mathrm{C}^d \ mod \ n$ Voltei a usar o método **modPow** da classe **BigInteger**:

```
BigInteger biPlaintextDecifrado = biCiphertext.modPow(d, n);
```

Finalmente, o número obtido é convertido de volta num *array de bytes* e, a partir daí, é obtida a *string correspondente*:

```
byte[] arrBytesDecifrado = biPlaintextDecifrado.toByteArray();
String strPlaintextDecifrado = new String(arrBytesDecifrado);
```

Uma implementação real, para maior segurança, teria algumas outras caraterísticas, como seja a utlização de *padding* - ver, por exemplo, a referência a ***optimal asymmetric encryption padding (OAEP)*** *em* Stallings (2007, pág. 307).

A *source* Java está no ficheiro "**RSA.java"**. Para o programa correspondente ser executado, deve ser executado da seguinte forma (o símbolo "$" é apenas a *prompt* da *shell*):

```
$ java -classpath . RSA
```

Ilustra-se aqui a execução do programa:

```
$ java -cp . RSA
RSA
Qual e' o plaintext a cifrar?
criptografia
Texto convertido para Maiusculas = 'CRIPTOGRAFIA'
p = 908332448369402957702139743814212188973114673079315551200054721765342274965l7
q = 108604351148151522782979193186274201164416662426406486972015809992757924070901
n =
986488561819708518978402121723105999876438858580596968319921858116310087571892289656498552519536691104119853686416423249715778165840562442738741443855181 7
p - 1 = 908332448369402957702139743814212188973114673079315551200054721765342274965l6
q - 1 = 108604351148151522782979193186274201164416662426406486972015809992757924070900
phi de n =
986488561819708518978402121723105999876438858580596968319921858116310087571872345896900043337681371787363084144410250436742344361631360314521812228698440 0
e = 65537
d =
438145002936479783487546408274961768839028064994797084911672573447816562080077822365487685757255128705716841681424135522112641098591110914980864402596307 3
Plaintext em formato de BigInteger = 208349737898064474187050704O1
Ciphertext em formato de BigInteger =
208855481463499522266762626570228698078481041593012742789951209356692284647779869307188099057926467182617193024782057827594794430469181212530110805294084 8
Ciphertext decifrado em formato de String = CRIPTOGRAFIA
```

Referências Bibliográficas

Stallings, William (2007) *Cryptography and Network Security: Principles and Practice.* 7th Edition (Global Edition), Prentice Hall.

Java™ Platform, Standard Edition 8 API Specification – Class BigInteger. Disponível em https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html