
Criptografia ingénua

	1	2	3	4	5	6	7	8	9
1	R	E	U	N	I	O	N	T	O
2	M	O	R	R	O	W	A	T	T
3	H	R	E	E	P	M	B	R	I
4	N	G	A	R	M	S	A	S	W
5	E	S	H	A	L	L	A	T	T
6	E	M	P	T	T	O	B	O	M
7	B	T	H	E	R	A	I	L	W
8	A	Y	S	T	A	T	I	O	N
9	C	H	I	E	F	A	B	C	D

A **cifra** obtém-se juntando as letras *por colunas*.

Código da tabela

Uma forma ingénua de escrever mensagens “secretas” é o chamado **código da tabela**.

Primeiro removem-se os espaços do texto original (**texto**) e escrevem-se as letras numa tabela.

Por exemplo, a mensagem “Uma forma ingénua de escrever mensagens secretas é o chamado código quadrado”, depois de *normalizado* é o **texto**

umaformaingenuadeescrevermensagenssecretaseochamadocodigoquadrado

que tem $65 = 5 \times 13$ letras e pode ser escrito, por exemplo, numa tabela com 5 colunas:

umafo
rmain
genua
deesc
rever
mensa
genss
ecret
aseoc
hamad
ocodi
goqua
drado

Depois, a mensagem codificada (**cifra**) obtém-se desta tabela escrevendo as letras ao longo das colunas, de cima para baixo, percorrendo as colunas da esquerda para a direita. A primeira coluna é a sequência de 13 letras urgdrmgeahogd, a segunda mmeeeeecsacor, etc. Juntando estas palavras, obtém-se a seguinte **cifra**:

urgdrmgeahogdmeeeeecsacoraanevnremoqafiusesseoadudonacrastcdiao

De seguida a cifra pode ser **enviada por um canal público** (por exemplo, colocando um *post* numa rede social). Em termos sérios **este esquema não tem qualquer segurança criptográfica** (como vai ser mostrado a seguir) mas é suficientemente difícil para efeitos lúdicos.

Finalmente, para **decifrar** a mensagem é preciso saber o *complemento* do número das colunas (isto é, o número das linhas da mensagem) e aplica-se à **cifra** o mesmo processo que a gerou, mas dividindo no número de linhas em vez de no número de colunas. Neste exemplo, como o **texto** tem $65 = 5 \times 13$ letras e a **cifra** foi obtida com 5 colunas, para decifrar escreve-se a **cifra** em 13 colunas:

urgdrmgeahogd
mmeeeeecsacor
aanevnremoqa
fiusesseoadud
onacrastcdiao

Agora, lendo esta tabela por colunas recupera-se o **texto** original:

uma forma ingenua de escrever mensagens secretas chamado código quadrado

Exercício

Implemente uma biblioteca para processar mensagens cifradas segundo o “Código da Tabela”. A biblioteca deve incluir as classes e os métodos descritos nas alíneas seguintes.

A biblioteca fica separada em várias classes:

- A classe **Cipher**, que define **métodos estáticos** para as operações básicas: **normalizar** e **cifrar** textos, encontrar os **divisores** de números inteiros e **quebrar** cifras.
- A classe *abstracta* **AbstractProvider** é a raiz de uma **hierarquia de classes** que proporcionam as **listas de palavras** usadas para quebrar as mensagens cifradas. São pedidos três subclasses concretas:
 - As instâncias de **MemoryProvider** definem listas de palavras acrescentando uma palavra de cada vez.
 - As instâncias de **SimpleFileProvider** lêm a lista de palavras de ficheiros especialmente formatados, apenas com uma palavra por linha.
 - As instâncias de **TextFileProvider** lêm ficheiros de texto normais, com várias palavras por linha.

As cotações das alíneas para a **nota total deste trabalho** são:

Alínea	1	2	3	4	5a	5b	5c	total
Cotação (%)	06%	12%	18%	24%	10%	14%	16%	100%

Alínea 1 | Normalizar o texto | 06%

```
public static String Cipher.normalize(String naturalText);
```

- O **parâmetro** naturalText é um texto em linguagem natural.
- O **resultado** é o *texto normalizado* que resulta de naturalText.

Texto normalizado: os espaços, pontuação, acentos e cedilhas são retirados e todas as letras convertidas para a forma minúscula. **Isto é**, o texto normalizado é formado *apenas* por letras minúsculas, de a a z, sem acentos ou cedilhas, e por algarismos, de 0 a 9. **Por exemplo**, depois de normalizado "É o 1º troço, João!" obtém-se "eo1trocojoao". Note que É, ã e ç perderam os acentos e a cedilha. Também o espaço, a vírgula, o ponto de exclamação e º foram eliminados e que as maiúsculas passaram a minúsculas. No limite, um texto normalizado tem, pelo menos, um carácter. Isto é, #@!** não pode ser normalizado.

Sugestão. Consulte a documentação de `java.lang.Character` para saber sobre *classes de carateres* e `java.text.Normalizer` para lidar com “adornos” de letras, como acentos e cedilhas. Também é recomendado que use a classe `java.lang.StringBuilder` ou `java.lang.StringBuffer`.

Alínea 2 | Cifrar o texto | 12%

```
public static String Cipher.encode(String plainText, int cols);
```

- O **parâmetro** plainText é uma *String normalizada* de um texto em linguagem natural.
- O **parâmetro** cols define o número de colunas que vai ser usado para **cifrar** este texto.
- O **resultado** é a **cifra** (uma `String`) que resulta de aplicar o método acima ao **texto** plainText usando uma tabela com cols colunas.

Atenção. Nem sempre o comprimento do texto é múltiplo do número de colunas. Nesse caso, aumente o texto com *letras escolhidas ao acaso no próprio texto* até obter uma `String` com comprimento múltiplo de cols.

Por exemplo, para cifrar "Bom dia, Alegria!" com 4 colunas, depois de normalizar, é preciso acrescentar três letras, escolhidas ao acaso entre as que estão presentes no texto: "abdegilmor". Um resultado possível seria "bomdiaalegriaarm", com as últimas três letras, arm, escolhidas ao acaso de bomdiaalegria.

Alínea 3 | Encontrar divisores | 18%

```
public static List<Integer> Cipher.findDividers(int x);
```

-
- O **parâmetro** x é um número inteiro positivo.
 - O **resultado** é a lista dos *divisores positivos* de x . Nesta lista consta 1, x e, se existirem, outros divisores de x .

Para descodificar uma **cifra** é necessário saber o número de linhas na tabela de codificação. Se esse valor não for dado, ainda assim é possível explorar alguns casos.

O comprimento da cifra é o número de colunas multiplicado pelo número de linhas: `cipher.length == cols * rows`. Portanto, **o número de linhas usadas para fazer a cifra é um divisor inteiro do comprimento dessa cifra**.

Alínea 4 | Quebrar uma cifra | 24%

O “Código da Tabela” não é criptograficamente seguro porque é (muito) fácil descobrir a mensagem secreta. Assim, um espião não teria dificuldade em ler mensagens supostamente secretas. Uma forma de *quebrar uma cifra* obtida pelo “Código da Tabela” assenta no processo ilustrado a seguir.

Quebrar uma cifra | Exemplo

Suponha que um **Espião** quer decifrar as mensagens que a **Alice** envia ao **Bruno** no *caralivro*. Previamente a **Alice** e o **Bruno** encontram-se e combinam que a Alice vai fazer cifras com 11 colunas. Se a Alice tiver escrito no *caralivro* a **cifra**

hcsmoieojnaseerevmnoaaomojououanspraaadaosia

1. Esta cifra tem 44 letras.
2. O João sabe que foi cifrada com 11 colunas e portanto vai decifrar com $44 / 11$ colunas.
3. O espião não sabe sobre as 11 colunas. Mas, como a mensagem tem 44 letras, o número de colunas para a gerar é um divisor de 44: 1, 2, 4, 11, 22 ou 44 colunas (os divisores de 44).
4. O espião vai tentar desencriptar (apenas!) para esses números de colunas e obtém as seguintes possibilidades:

```
rows: 01, text: hcsmoieajnaeeercvmnaaaoomojeouaasprvaaduosim  
rows: 02, text: hsoejaervnaomjoasradoicmianeecmaaooeuapvausm  
rows: 04, text: hojevamosaocinemaoupassearnojardimaecaoeavum  
rows: 11, text: heopceorsemvroaocjaivedemouanuojaasnnaiaasm  
rows: 22, text: hocosmmoojieeoaujanaaseperevracavdmunoasaiam  
rows: 44, text: hcsmoieajnaeeercvmnaaaoomojeouaasprvaaduosim
```

Agora o segredo foi quebrado: destas alternativas, apenas *hoje vamos a cinema ou... faz “sentido”*. O *texto original* mais provável será “*Hoje vamos ao cinema ou passear no jardim*”, com as últimas letras, *ae- caoeavum* acrescentadas ao acaso para a mensagem ficar com um comprimento adequado (múltiplo do número de colunas).

Sendo assim o espião fica também a saber que a Alice usou 11 colunas para produzir a **cifra** intersetada (porque a decifrou com 4 colunas e $44 = 11 \times 4$).

Este método para **quebrar** a cifra usa o **reconhecimento humano** de palavras para escolher os *candidatos* que fazem “sentido”. Para fazer uma seleção **automática** pode-se usar um **dicionário**, isto é uma lista com “todas” as palavras, para validar as possibilidades obtidas. Por exemplo, em português:

```
hcsmoieajnaeeeercvmna... // nenhuma palavra começa por "hc"  
hsoejaervnaomjoasrad... // nenhuma palavra começa por "hs"  
hojevamosaocinemaoup... // Pode ser separado em "hoje vamos ao cinema ou p..."  
heopceorsemvroaocja... // nenhuma palavra começa por "heo"  
hocosmmoojieeoaujana... // nenhuma palavra começa por "hoc"  
hcsmoieajnaeeeercvmna... // nenhuma palavra começa por "hc"
```

Assim, supondo que `words` é a lista das palavras portuguesas normalizadas, apenas "hojevamosaocinemaoupassearnojardimaecaoeavum" tem (vários) **textos possíveis**:

- "hoje vamos ao cinema ou passear no jardim a e cao e a vum"
- "hoje vamos ao cinema ou pas se ar no jardim aecaoeavum"
- "hoje vamos ao cinema o upas se ar no jardim a e cao e a vum"
- ...

Quebrar uma cifra | Método `breakCipher`

O processo descrito acima deve ser implementado no seguinte método:

```
public static List<String> Cipher.breakCipher(String cipherText,  
                                              List<String> words);
```

- O **parâmetro** `cipherText` é uma **cifra**.
- O **parâmetro** `words` é uma lista de **palavras normalizadas e válidas**, que faz de dicionário.
- O **resultado** é a lista de **todos os textos possíveis** para a **cifra** `cipherText`, usando palavras de `words`.

Um **texto possível usando palavras de words**:

- Resulta da concatenação de palavras da lista `words`, separadas por espaços. Por exemplo, se `words` é a lista {"um", "uma", "dia", "noite", "flor"}, então "um dia" é um texto possível, assim como "uma flor", "um um", "noite uma dia", etc.
- **Apenas a última palavra** de um texto possível pode não constar na lista de palavras dada. Por exemplo, "um dia idmiua".

Alínea 5 | Fornecedores de palavras | 40% (total)

A lista de palavras usada na alínea anterior pode ter várias proveniências: Um ficheiro de texto, um documento html, uma base de dados, uma *stream*, etc. Para abranger todas as possíveis proveniências usa-se a classe *abstrata* `AbstractProvider`, que especifica um único método:

```
import java.util.List;
abstract class AbstractProvider {
    abstract List<String> getWords();
}
```

- **Não pode alterar de qualquer forma o ficheiro `AbstractProvider.java`.**
 - As implementações concretas do método `getWords()` devolvem uma lista de **palavras normalizadas, sem duplicados e ordenadas alfabeticamente**.
-

Implemente os descendentes concretos de `AbstractProvider` indicados a seguir.

Alínea 5a | `MemoryProvider` | 10%

```
public class MemoryProvider extends AbstractProvider {
    public List<String> getWords();
    public void addWord(String word);
}
```

- A classe pública `MemoryProvider` estende `AbstractProvider` e proporciona uma implementação concreta do método `getWords()`.
- No método `getWords()`:
 - O **resultado** é formado pelas palavras adicionadas através do método `addWord` (depois de *normalizadas, ordenadas e removidos os duplicados*). **Note bem** que só pode adicionar uma palavra normalizada se esta tiver comprimento maior que zero.
- No método `void addWord(String word)`:
 - O **parâmetro** `word` poderá ter de ser normalizado.

Por exemplo

```
AbstractProvider memprovider = new MemoryProvider();
memprovider.addWord("É");
memprovider.addWord("o");
memprovider.addWord("1º");
memprovider.addWord("troço,");
memprovider.addWord("João!");
List<String> words = memprovider.getWords();
// words == ["1", "e", "joao", "o", "troco"]
```

Alínea 5b | SimpleFileProvider | 14%

- A classe pública SimpleFileProvider estende AbstractProvider e proporciona uma implementação concreta do método getWords().
- As palavras obtidas por SimpleFileProvider.getWords() resultam dum ficheiro de texto formatado de forma que **em que cada linha há uma única palavra, não necessariamente normalizada.**
- Esta classe tem o construtor

```
SimpleFileProvider(String fileName) throws java.io.IOException
```

- O **argumento** fileName identifica um ficheiro no sistema de ficheiros.
- Se não existir o ficheiro indicado, o construtor deve levantar uma exceção do tipo indicado.
- Caso contrário, as palavras desse ficheiro (uma por linha) definem a lista que vai ser devolvida por getWords().

- Não é necessário confirmar que o conteúdo do ficheiro é conforme esperado (*i.e.* exatamente uma palavra por linha).

Sugestão: A extensão de AbstractProvider pode ser feita via MemoryProvider, o que **evita duplicações de código** e simplifica a resolução desta alínea.

Alínea 5c | TextFileProvider | 16%

- A classe pública TextFileProvider estende AbstractProvider e proporciona uma implementação concreta do método getWords().
- As palavras obtidas por TextFileProvider.getWords() resultam de ler um ficheiro de texto formatado de forma que **podem haver várias palavras não normalizadas em cada linha. Numa linha as palavras são separadas por espaços.**
- Esta classe tem o construtor

```
TextFileProvider(String fileName) throws java.io.IOException
```

- O **parâmetro** fileName identifica um ficheiro no sistema de ficheiros.
- Se não existir o ficheiro indicado, o construtor deve levantar uma exceção do tipo indicado.
- Caso contrário, as palavras do ficheiro (talvez várias por linha) definem a lista que vai ser devolvida por getWords().

- Não é necessário confirmar que o conteúdo do ficheiro é conforme esperado (*i.e.* um ficheiro de texto).

Ficha técnica

Curso Engenharia Informática

Disciplina Programação II

Ano letivo 2021-2022

Autores fc@uevora.pt e vbn@uevora.pt
