

Criptografia ingénu

	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 juntado as letras *por colunas*.

Uma forma ingénu 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 ingénu 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 **urgdrmg**eahogd, a segunda **mm**eeee**csacor**, etc. Juntando estas palavras, obtém-se a seguinte **cifra**:

```
urgdrmgeahogdmmeeeecsacoraanevnnremoqafiusseseoadudonacrastcdiao
```

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
aanevnnremoqa
fiusesseoadud
onacrastcdiao
```

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

```
umaformaingenuadeescrevermensagenssecretaseochamadocodigoquadrado
```

Exercício

Implemente uma biblioteca para processar mensagens cifradas segundo o "Código das Tabelas". A biblioteca deve incluir as classes e métodos descritos nas alíneas seguintes.

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 "eoltrocojoao". Note que "É", "ã" e "ç"

perderam os acentos e a cedilha, que " ", "°", ", " e "!" foram eliminados e que as maiúsculas passaram a minúsculas.

Sugestão. Consulte a documentação de `java.lang.Character` para detetar *classes de caracteres* e `java.text.Normalizer` para lidar com "adornos" de letras, como acentos e cedilhas. Também é recomendado considerar o uso de `java.lang.StringBuilder`.

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 calcular a **cifra** deste 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.findDivisors(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. Um espião não teria dificuldade em ler mensagens supostamente secretas. A forma de o fazer assenta no método seguinte.

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

- O **parâmetro** `cipherText` é (supostamente) uma **cifra**.
- O **parâmetro** `words` é uma lista de *palavras normalizadas e válidas*, possivelmente obtidas de um dicionário.
- O **resultado** é a lista de **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.
 - Exceccionalmente, a **última palavra** de um texto possível pode não constar na lista de palavras dada. Por exemplo, "um dia idmua".

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**

```
hcsmoieojnaseerevmnoaaomojoouanspraaadaosia
```

1. Esta cifra tem 44 letras.
2. O João sabe que foi obtida com 11 colunas e portanto vai descriptar a cifra usando $4 = 44 / 11$ 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 pode tentar decodificar para esses números de colunas e obtém as seguintes possibilidades:

```
rows: 01, text: hcsmoieajnaeeercvmnaaaomojeouaasprvaaduosim
rows: 02, text: hsoejaervnaomjoasradoicmianeecmaaooeuapvausm
rows: 04, text: hojevamosaocinemaoupassearnojardimaecaoeavum
rows: 11, text: heopceorsevmvroacjaivedemouanuojaasnaaiaasm
rows: 22, text: hocosmmoojieeoaujanaaseperevracavdmunoasaiam
rows: 44, text: hcsmoieajnaeeercvmnaaaomojeouaasprvaaduosim
```

Agora o segredo foi quebrado: destas alternativas, apenas

`hojevamosaocinemaoupassearnojardimaecaoeavum` faz "sentido". O **texto original** mais provável será "*Hoje vamos ao cinema ou passar no jardim*", com as últimas letras, `aecaoeavum` 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:

```
hcsmoieajnaeeercvmna... // nenhuma palavra começa por "hc"
hsoejaervnaomjoasrad... // nenhuma palavra começa por "hs"
hojevamosaocinemaoup... // Pode ser separado em "hoje vamos ao cinema ou
p..."
heopceorsevmroaocja... // nenhuma palavra começa por "heo"
hocosmmoojieoaujana... // nenhuma palavra começa por "hoc"
hcsmoieajnaeeercvmna... // 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"`
- ...

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 abarcar todas as possíveis proveniências usa-se a classe *abstrata* `AbstractProvider`, que especifica um único método:

```
abstract class AbstractProvider {
    abstract List<String> getWords();
}
```

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()`.
- As instâncias de `MemoryProvider` têm uma **lista interna (`protected`) de palavras normalizadas, sem duplicados e ordenadas alfabeticamente**.
- O construtor `MemoryProvider()` inicializa vazia a lista interna de palavras.
- No método `getWords()`:

- O **resultado** é a lista interna de todas as palavras adicionadas através de `addWord`.
- No método `void addWord(String word)`:
 - O **parâmetro** `word`, depois de normalizado, é adicionado à lista interna de palavras.

Alínea 5b | `SimpleFileProvider` | 14%

- A classe pública `SimpleFileProvider` estende `AbstractProvider` e proporciona uma implementação concreta do método `getWords()`.
- Uma instância de `SimpleFileProvider` tem uma lista interna de palavras, tal como em `MemoryProvider`, normalizadas, sem duplicados e ordenada alfabeticamente.
- Essa lista resulta de ler um 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.nio.file.NoSuchFileException
```

- Neste construtor:
 - 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 do ficheiro (uma por linha) devem ser lidas para a lista interna de palavras.
- O método `getWords()` devolve todas as palavras da lista interna.

Sugestão: A extensão de `AbstractProvider` também pode ser feita estendendo a classe `MemoryProvider`, o que vai **evitar duplicações de código** e simplificar a resolução do exercício.

Alínea 5c | `TextFileProvider` | 16%

- A classe pública `TextFileProvider` estende `SimpleFileProvider`.
- Uma instância de `TextFileProvider` tem uma lista interna de palavras normalizadas, sem duplicados e ordenada alfabeticamente.
- Essa lista resulta de ler um ficheiro de texto formatado de forma que **podem haver várias palavras não normalizadas em cada linha. Na mesma linha as palavras são separadas por espaços**.
- Esta classe tem o construtor

```
TextFileProvider(String fileName) throws java.nio.file.NoSuchFileException
```

- Neste argumento:
 - 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.
- Se existir o ficheiro indicado no argumento `fileName`, as palavras do ficheiro devem ser lidas para a lista interna de palavras, de acordo com o seguinte pré-processamento:
 1. Substituir todos os espaços por `'\n'`, de forma a obter-se uma palavra por linha.
 2. Proceder como na classe `SimpleFileProvider`.

- O método `getWords()` devolve todas as palavras da lista interna.

Ficha técnica

Curso	Engenharia Informática
Disciplina	Programação II
Ano letivo	2021-2022
Autores	<code>fc@uevora.pt</code> e <code>vbn@uevora.pt</code>