Guião de Trabalho Autónomo

Exercício E1

[1] Na área da construção de edifícios de habitação, os vários profissionais precisam de gerir a informação sobre cada unidade habitacional (casa ou apartamento), bem como extrair diversas propriedades. São fornecidas em anexo classes para representar pontos no espaço cartesiano (`Point`), habitações (`House`), e divisões de uma habitação (`Room`) bem como uma classe para testes (`TestHouse`). Assume-se que cada divisão terá uma forma rectangular, estando alinhada com os eixos de um determinado sistema de coordenadas. A unidade usada no sistema de coordenadas e nas distâncias é o metro.

Pretende-se que integre novas funcionalidades neste programa:

a. Como pode constatar através dos métodos da classe `House`, as divisões de uma habitação são guardadas num vector (array). Em particular, o método `addRoom(room)` na classe `House` adiciona uma divisão ao vector. Modifique esse método de forma a que retorne o índice da posição do vector em que a divisão foi armazenada. Esse índice funcionará como identificador da divisão.

b. Com vista ao registo das portas entre divisões de uma habitação, complete a definição da classe `Door`, com pelo menos os seguintes métodos:

   - `Door(RoomId1,RoomId2,Width,Height)` - construtor que recebe como argumentos os identificadores das duas divisões ligadas pela porta bem como a largura e altura da porta;
   - `area()` - um método que calcula a área da porta.

c. O construtor de `House` cria um vector para armazenar a informação das portas. A capacidade desse vector é igual à capacidade inicial do vector das divisões. O método `addDoor(Door)` na classe `House` adiciona uma nova porta, mas não prevê a possibilidade de o vector encher. Altere este método de forma a que, caso o vector que armazena a informação das portas esteja cheio, a sua capacidade seja extendida com mais `extensionSize` posições.

d. Crie um método `roomClosestToRoomType(roomType)` na classe `House` que, dado um tipo de divisão, retorna o identificador da divisão mais próxima de uma qualquer divisão do tipo dado. Considere distâncias em linha recta entre os centros geométricos das divisões.

e. Crie um método `maxDoorsInAnyRoom()` na classe `House` que devolve o máximo número de portas numa qualquer divisão da habitação.

[1] Exercícios retirados da API nº1.

Exercício E2

[2] Os ficheiros `JogaJogoDoGalo.java` e `jogos/JogoDoGalo.java` definem um programa e um módulo para implementar um "jogo-do-galo" mas onde, propositadamente, foram inseridos vários erros.

a. Corrija o módulo `JogoDoGalo.java` de forma a eliminar os erros sintácticos (de compilação) do programa.

   - Pode compilar com o comando: `javac JogaJogoDoGalo.java`

b. O programa principal `JogaJogoDoGalo.java` contém também um erro semântico.

   Detecte-o e corrija-o.

   - Pode executar o seu programa com: `java -ea JogaJogoDoGalo`
   - Pode executar uma versão correcta com: `java -ea -jar JogaJogoDoGalo.jar`

```
   1   2   3
1  X |   | O
  ---+---+---
2    | X | O
  ---+---+---
3    |   | X


Jogador X ganhou!
```

c. Torne o programa principal robusto na utilização do módulo (não é necessário usar Excepções).

d. Altere o programa `JogaJogoDoGalo.java` de forma a realizar campeonatos de até 10 jogos, terminando quando um dos jogadores atinja 3 vitórias. No fim de cada jogo deve indicar a pontuação de cada jogador.

Exercício E3

Implemente uma função recursiva – `invertDigits` – que recebendo (pelo menos[3]) um `String` como argumento, devolve um novo `String` em que as sequências de dígitos lá contidas são invertidas mantendo a ordem dos restantes caracteres.

Por exemplo, a invocação do programa aplicando a função a cada um dos seus argumentos:

```
java -ea P2 1234 abc9876cba a123 312asd a12b34c56d
```

deve ter como resultado:

[2] Problema da AIP de 2009-2010.

[3] Pode acrescentar mais argumentos se considerar conveniente.

```
1234 -> 4321
abc9876cba -> abc6789cba
a123 -> a321
312asd -> 213asd
a12b34c56d -> a21b43c65d
```

Exercício E4

Implemente uma função recursiva `factors` que recebendo um número inteiro como argumento devolve uma `String` com o produto dos seus factores.

Por exemplo, a invocação do programa:

```
java -ea Factors 0 1 10 4 10002
```

deve ter como resultado:

```
0 = 0
1 = 1
10 = 2 * 5
4 = 2 * 2
10002 = 2 * 3 * 1667
```

Exercício E5

[4] Construa uma função recursiva – `countPairs` – que recebendo (pelo menos[5]) um `String` como argumento, devolve o número de vezes que dois caracteres iguais estão em posições consecutivas nesse texto.

Para testar a função crie um programa – `P2.java` – que aplique a função a todos os seus argumentos.

Seguem alguns exemplos da execução pretendida do programa:

`java -ea P2 112233`	`"112233" contains 3 pairs of consecutive equal characters`
`java -ea P2 aaaa`	`"aaaa" contains 3 pairs of consecutive equal characters`

`java -ea P2 a abba sfffsff`	`"a" contains 0 pairs of consecutive equal characters`
	`"abba" contains 1 pairs of consecutive equal characters`
	`"sfffsff" contains 3 pairs of consecutive equal characters`

Pode executar uma versão correcta do programa com o comando:

```
java -ea -jar P2.jar <arg> ...
```

Exercício E6

Crie um programa que, dado um número inteiro positivo como argumento, escreva todos os divisores do número que não o próprio nem o número 1, e, recursivamente, faça o mesmo para todos esses divisores. Seguem alguns exemplos da execução pretendida do programa:

[4] Problema da AIP de 2012-2013, 1º semestre.

[5] Pode acrescentar mais argumentos se considerar conveniente.

`java -ea AllDivisors 12`	`java -ea AllDivisors 23`	`java -ea AllDivisors 81`	`java -ea AllDivisors 32`

12
  6
  3
  2
  4
  2
  3
  2
	23	81
		  27
		  9
		  3
		  3
		  9
		  3
		  3
			32
			  16
			  8
			  4
			  2
			  2
			  4
			  2
			  2
			  8
			  4
			  2
			  2
			  4
			  2
			  2

Exercício E7

Crie um programa que dado um número racional pertencente ao intervalo aberto entre zero e um, e expresso como uma fracção (n/d), escreva essa fracção como sendo uma soma de fracções unitárias com denominadores diferentes[6]. Uma fracção unitária é uma fracção em que o numerador é igual a um. O programa a desenvolver deve fazer uso de um algoritmo recursivo.

Seguem alguns exemplos da execução pretendida do programa:

`java -ea UnitaryFractionSum 3 4`	`3/4 = 1/2 + 1/4`
`java -ea UnitaryFractionSum 3 7`	`3/7 = 1/3 + 1/11 + 1/231`
`java -ea UnitaryFractionSum 1 8`	`1/8 = 1/8`
`java -ea UnitaryFractionSum 2 20`	`2/20 = 1/10`

Para resolver o problema considere a seguinte estratégia (designada por "gananciosa" e proposta no Séc. XIII por Fibonacci):

a. Tentar subtrair da fracção a maior fracção unitária possível. Para descobrir essa fracção unitária $(1/d)$ considere a seguinte fórmula:

$$\frac{\text{num}}{\text{den}} - \frac{1}{d} \geq 0 \quad \Leftrightarrow \quad d \geq \frac{\text{den}}{\text{num}} \quad \Rightarrow \quad d = \left\lceil \frac{\text{den}}{\text{num}} \right\rceil$$

b. A fracção será a soma da fracção unitária $1/d$ somada à fracção unitária obtida da fracção resultante da diferença (para a qual se deverá aplicar o mesmo algoritmo);

c. O procedimento termina quando o numerador for divisor do denominador (indicando que já é uma fracção unitária).

Exercício E8

Construa uma função recursiva – `isPrefix` – que recebendo (pelo menos[7]) dois `Strings` como argumentos, indica se a segunda *string* é um prefixo da primeira.

[6] Fibonacci demonstrou que qualquer número racional pode ser expresso por uma soma finita de fracções unitárias com denominadores diferentes.

[7] Pode acrescentar mais argumentos se considerar conveniente.

Para testar a função crie um programa – `P3.java` – que aplique a função a todos os seus argumentos (dois a dois, pelo que o número de argumentos deve ser par). Seguem alguns exemplos da execução pretendida do programa:

`java -ea P3 dedo de`	`"dedo" is prefixed by "de" -> true`
`java -ea P3 assim sim`	`"assim" is prefixed by "sim" -> false`
`java -ea P3 s sd ff f`	`"s" is prefixed by "sd" -> false` `"ff" is prefixed by "f" -> true`
`java -ea P3 tudo "" "" nada`	`"tudo" is prefixed by "" -> true` `"" is prefixed by "nada" -> false`

Pode executar uma versão correcta do programa com o comando:

```
java -ea -jar P3.jar <arg> ...
```

Exercício E9

O programa `ArraySorting.java` contém a implementação de vários algoritmos de ordenação de vectores bem como uma função `main()` que aplica esses algoritmos a vectores com conteúdos gerados aleatóriamente (pela função `randomArray()`). Os algoritmos de ordenação têm já algumas asserções que permitem verificar pré- e pós-condições. Se executar o programa com verificação de asserções (opção `-ea`), a conclusão normal da execução indicará que os algoritmos conseguiram ordenar correctamente os vectores gerados para o efeito. Reveja a implementação dos algoritmos e corrija quaisquer erros que encontre.

Exercício E10

Crie um módulo `LeakyQueue`, baseado na estrutura de dados fila, de forma a que o programa `ProgX` funcione devidamente[8].

Uma fila "rota" (*leaky queue*) é uma estrutura de dados baseada numa fila, mas em que só ficam armazenados, no máximo, os últimos N números inseridos. Quando a fila está preenchida (N elementos) a inserção de um novo número implica a saída do primeiro (que deixa de existir).

Exemplos de utilização ($N = 3$) e resultados esperados:

```
java -ea ProgX 1 2 3 4 5 6
i = 0    1.0              (Min = 1.0)
i = 1    1.0  2.0         (Min = 1.0)
i = 2    1.0  2.0  3.0    (Min = 1.0)
i = 3    2.0  3.0  4.0    (Min = 2.0)
i = 4    3.0  4.0  5.0    (Min = 3.0)
i = 5    4.0  5.0  6.0    (Min = 4.0)
```

```
java -ea ProgX 9 8 7 6 5 4 3 2 1
i = 0    9.0              (Min = 9.0)
i = 1    9.0  8.0         (Min = 8.0)
i = 2    9.0  8.0  7.0    (Min = 7.0)
i = 3    8.0  7.0  6.0    (Min = 6.0)
i = 4    7.0  6.0  5.0    (Min = 5.0)
i = 5    6.0  5.0  4.0    (Min = 4.0)
i = 6    5.0  4.0  3.0    (Min = 3.0)
i = 7    4.0  3.0  2.0    (Min = 2.0)
i = 8    3.0  2.0  1.0    (Min = 1.0)
```

[8] Não pode usar os módulos do pacote `exameP2` neste problema.

```
java -ea ProgX 1 3 - 5 7 - 9 11 -
i = 0    1.0              (Min = 1.0)
i = 1    1.0  3.0         (Min = 1.0)
i = 2    3.0              (Min = 3.0)
i = 3    3.0  5.0         (Min = 3.0)
i = 4    3.0  5.0  7.0    (Min = 3.0)
i = 5    5.0  7.0         (Min = 5.0)
i = 6    5.0  7.0  9.0    (Min = 5.0)
i = 7    7.0  9.0 11.0    (Min = 7.0)
i = 8    9.0 11.0         (Min = 9.0)
```

```
java -ea ProgX 2 - - 4 - 6 8
i = 0    2.0              (Min = 2.0)
i = 1
i = 2
i = 3    4.0              (Min = 4.0)
i = 4
i = 5    6.0              (Min = 6.0)
i = 6    6.0  8.0         (Min = 6.0)
```

Exercício E11

O programa `ProgX` serve para verificar se uma expressão aritmética (formada por dígitos, operações elementares e parêntesis) é sintacticamente válida. Construa o módulo `PilhaX`, baseado na estrutura de dados pilha, de forma a que este programa funcione devidamente[9].

Exemplos de utilização e resultados esperados:

```
java -ea ProgX "2+2"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+D
REDUCE: e+e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "2+(2-3)"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+(
  PUSH: e+(D
REDUCE: e+(e
  PUSH: e+(e-
  PUSH: e+(e-D
REDUCE: e+(e-e
REDUCE: e+(e
  PUSH: e+(e)
REDUCE: e+e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "3*(4/(3))"
  PUSH: D
REDUCE: e
  PUSH: e*
  PUSH: e*(
  PUSH: e*(D
REDUCE: e*(e
  PUSH: e*(e/
  PUSH: e*(e/(
  PUSH: e*(e/(D
REDUCE: e*(e/(e
  PUSH: e*(e/(e)
REDUCE: e*(e/e
REDUCE: e*(e
  PUSH: e*(e)
REDUCE: e*e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "2+"
  PUSH: D
REDUCE: e
  PUSH: e+
Bad expression!
```

```
java -ea ProgX "(3*(2+4)+5))"
  PUSH: (
  PUSH: (D
REDUCE: (e
  PUSH: (e*
  PUSH: (e*(
  PUSH: (e*(D
REDUCE: (e*(e
  PUSH: (e*(e+
  PUSH: (e*(e+D
REDUCE: (e*(e+e
REDUCE: (e*(e
  PUSH: (e*(e)
REDUCE: (e*e
REDUCE: (e
  PUSH: (e+
  PUSH: (e+D
REDUCE: (e+e
REDUCE: (e
  PUSH: (e)
REDUCE: e
  PUSH: e)
Bad expression!
```

```
java -ea ProgX "2+4*(4++5)"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+D
REDUCE: e+e
REDUCE: e
  PUSH: e*
  PUSH: e*(
  PUSH: e*(D
REDUCE: e*(e
  PUSH: e*(e+
  PUSH: e*(e++
  PUSH: e*(e++D
REDUCE: e*(e++e
  PUSH: e*(e++e)
Bad expression!
```

[9] Não pode usar os módulos do pacote `exameP2` neste problema.

Exercício E12

Construa um programa (`JustifiedText.java`) que permita alinhar um texto simultaneamente às margens esquerda e direita ("justificar" o texto). O programa recebe como parâmetros o comprimento de cada linha e o nome de um ficheiro de entrada, e deve escrever o texto justificado na consola.

Para resolver este problema tem de utilizar pelo menos uma estrutura adequada do pacote `exameP2.jar`.

Por exemplo, dado o seguinte ficheiro:

```
If one cannot enjoy reading a book over and over again, there is no use
in reading it at all.
Perfect day for scrubbing the floor and other exciting things.

You
are      standing   on my
toes.  You have taken yourself too seriously.
```

Dois exemplos de utilização do programa serão:

`java -ea JustifiedText 40 texto.txt`	`java -ea JustifiedText 30 texto.txt`

`If one cannot enjoy reading a book over and over again, there is no use in reading it at all. Perfect day for scrubbing the floor and other exciting things.`
 	
`You are standing on my toes. You have taken yourself too seriously.`
	`If one cannot enjoy reading a book over and over again, there is no use in reading it at all. Perfect day for scrubbing the floor and other exciting things.`
	
	`You are standing on my toes. You have taken yourself too seriously.`

Detalhes a ter em consideração:

- Cada linha escrita deve conter o maior número possível de palavras sem ultrapassar o comprimento definido. Considera-se "palavra" qualquer sequência de caracteres delimitada por espaços em branco.
- Não se podem juntar palavras (espaçamento nulo) nem dividir nenhuma palavra entre linhas.
- Os espaçamentos entre palavras de uma linha devem ter comprimentos iguais ou diferir no máximo de um espaço.
- A última linha de cada parágrafo deve ficar alinhada à esquerda (com um único espaço entre palavras). Considere que um parágrafo termina com uma linha vazia ou com o fim do ficheiro.

Exercício E13

O programa `P1` gere uma fila de espera que comprime elementos consecutivos repetidos. Crie o módulo `CompressedQueue` de forma que este programa compile e funcione devidamente. Consulte os comentários em ambos os ficheiros e os exemplos abaixo.

IMPORTANTE: Não pode usar os módulos do pacote `exameP2` neste problema.

```
java -ea P1 1 1 1 2 2 3 4 4
```

```
IN 1
IN 1
IN 1
IN 2
IN 2
IN 3
IN 4
IN 4
QUEUE: {[1:3],[2:2],[3:1],[4:2]}
```

```
java -ea P1 2 5 5 max 5 5 4 max min
         show clear 1
```

```
IN 2
IN 5
IN 5
MAX: 3
IN 5
IN 5
IN 4
MAX: 3
MIN: 1
QUEUE: {[2:1],[5:4],[4:1]}
CLEAR
IN 1
QUEUE: {[1:1]}
```

```
java -ea P1 2 3 3 3 out 3 3
```

```
IN 2
IN 3
IN 3
IN 3
OUT: [2:1]
IN 3
IN 3
QUEUE: {[3:5]}
```

```
java -ea P1 1 2 out out out
```

```
IN 1
IN 2
OUT: [1:1]
OUT: [2:1]
ERROR: a non-empty queue is required!
QUEUE: {}
```

```
java -ea P1 1 max min a
```

```
IN 1
ERROR: a queue with a least two elements is required!
ERROR: a queue with a least two elements is required!
ERROR: invalid argument!
QUEUE: {[1:1]}
```