Guião de Trabalho Autónomo

Este guião contém uma coleção de exercícios complementares aos fornecidos nos guiões das aulas práticas de Programação II. Muitos destes exercícios são adaptações de problemas propostos em provas de avaliação prática em anos passados. Para cada problema é fornecido um diretório com o material necessário: ficheiros de código, ficheiros de dados, documentação, etc.

Para resolver alguns dos problemas deverá usar estruturas de dados definidas nas classes do pacote `pt.ua.p2utils`, que é fornecido na forma de um arquivo pré-compilado `p2utils.jar`. Descarregue-o do página da disciplina e siga as instruções indicadas. Outros problemas, pelo contrário, proibem o uso dessas classes. Deverá usar as classes fornecidas no diretório (ou subdiretórios) do problema.

Exercício E1. (Problema da AIP de 2009-2010.)
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 E2. (Exercícios retirados da API nº1 de 2014-2015.)
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 devolva 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, devolve 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.

Exercício E3. (Baseado na API nº1 de 2015-2016.)
Embora já existam diversas redes sociais de âmbito mundial, existe um novo nicho de mercado que procura plataformas de interação social em âmbito mais restrito, por exemplo numa empresa ou organização. Neste trabalho, pede-se que acrescente novas funcionalidades a uma aplicação deste tipo.

A implementação parcial já disponibilizada em anexo inclui as classes `Person` (pessoas que podem aderir a uma rede social), `FriendshipRequest` (um pedido de amizade no âmbito de uma rede social) e `SocialNetwork` (uma rede social caracterizada pelo conjunto dos seus membros e pelo conjunto de pedidos de amizade já registados). Cada pedido de amizade pode estar num de quatro estados: `"pending"`, `"accepted"`, `"rejected"` e `"cancelled"`. Disponibiliza-se ainda uma classe `TestSocialNetwork` para teste das funcionalidades da aplicação.

Pretende-se que integre novas funcionalidades nesta aplicação.[1]

a. Inclua no método `setStatus()` da classe `FriendshipRequest` as asserções necessárias para garantir a sua correcção.

b. Relativamente ao armazenamento de pedidos de amizade, desenvolva na classe `SocialNetwork` os seguintes métodos:

   - `addFriendshipRequest(m1, m2)` que, dados os nomes de dois membros da rede, regista um pedido de amizade que o primeiro envia ao segundo. Pressupõe-se que os dois membros existem e que não existe ainda nenhum pedido entre eles. Para este efeito, inclua em local apropriado a criação do vector `requests`, que vai conter todos os pedidos de amizade, e tenha em conta que esse vector pode encher. A dimensão inicial do vector deverá ser `maxSize`. Sempre que o vector encher, deve ser expandido com mais `sizeExpansion` posições.
   - Inclua no método anterior as asserções necessárias para garantir a sua correcção.
   - `numRequests()`, que devolve o número de pedidos de amizade existentes.
   - `maxNumRequests()`, que devolve o número máximo de pedidos de amizade que é possível armazenar com a capacidade actual.

c. Na classe `SocialNetwork`, desenvolva ainda os seguintes métodos:

   - `numPendingRequests(memberName)` que, dado o nome de um membro da rede, devolve o número de pedidos enviados ainda pendentes de resposta.
   - `oldestFriend(memberName)` que, dado o nome de um membro da rede, devolve o nome do seu amigo mais velho (e não o que é amigo há mais tempo). Para este efeito considera-se pedidos enviados e recebidos que estejam em estado `"accepted"`. Caso o membro não tenha nenhum amigo aceite, a função deve devolver `null`.

Exercício E4.

Implemente uma função recursiva – `invertDigits` – que recebendo (pelo menos[2]) um

[1] Pode observar o comportamento desejado da aplicação com o comando `java -jar TestSocialNetwork.jar`.

[2] Pode acrescentar mais argumentos se considerar conveniente.

`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:

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

Exercício E5.

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 E6. (Problema da AIP de 2012-2013, 1º semestre.)

Construa uma função recursiva – `countPairs` – que recebendo (pelo menos[3]) 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> ...
```

[3]Pode acrescentar mais argumentos se considerar conveniente.

Exercício E7.

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:

```
java -ea AllDivisors 12
12
   6
      3
      2
   4
      2
   3
   2
```

```
java -ea AllDivisors 23
23
```

```
java -ea AllDivisors 81
81
   27
      9
         3
      3
   9
      3
   3
```

```
java -ea AllDivisors 32
32
   16
      8
         4
            2
         2
      4
         2
      2
   8
      4
         2
      2
   4
      2
   2
```

Exercício E8.

Crie um programa que dado um número racional pertencente ao intervalo aberto entre zero e um, e expresso como uma fração (N/D), escreva essa fração como uma soma de fracções unitárias com denominadores distintos.[4] Uma fração unitária é uma fração 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. Encontrar a maior fração unitária que não supera a fração dada. Isto equivale a encontrar o menor inteiro d tal que $\frac{1}{d} \leq \frac{N}{D}$ ou equivalentemente, o inteiro d que satisfaz $d \geq \frac{D}{N} > d - 1$. Note que a divisão de inteiros positivos é arredondada para baixo, em Java.

b. O resultado será expresso por $\frac{N}{D} = \frac{1}{d} + E$, onde E é a expansão em frações unitárias da diferença $\frac{N}{D} - \frac{1}{d} = \frac{dN-D}{dD}$, obtida com o mesmo algoritmo.

c. O procedimento termina quando o D é múltiplo de N e portanto fica simplesmente $\frac{N}{D} = \frac{1}{d}$, com $d = D/N$.

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

Exercício E9.

Construa uma função recursiva – `isPrefix` – que recebendo (pelo menos[5]) dois `Strings` como argumentos, indica se a segunda *string* é um prefixo da primeira.

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 E10.

Desenvolva na classe `LinkedList` (dentro do subdiretório `p2utils`) implementações iterativas e recursivas para os seguintes métodos:

- `count(e)` - devolve o número de ocorrências de um dado elemento na lista.
- `indexOf(e)` - devolve a posição da primeira ocorrência de um dado elemento na lista, ou `-1` caso a lista não contenha o elemento.
- `cloneReplace(x, y)` - devolve uma cópia da lista em que todas as ocorrências de `x` estão substituídas por `y`.
- `cloneSublist(start, end)` - devolve uma nova lista com os elementos entre as posições `start` (inclusive) e `end` (exclusive).
- `cloneExceptSublist(start, end)` - devolve uma cópia da lista exceto dos elementos entre as posições `start` e `end` (exclusive).
- `removeSublist(start, end)` - remove da lista os elementos nas posições `start` até `end-1`.

Exercício E11.

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 aleatoriamente (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

[5]Pode acrescentar mais argumentos se considerar conveniente.

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 E12.

Crie uma classe `LeakyQueue`, baseada na estrutura de dados fila, de forma a que o programa `ProgX` funcione devidamente.

Atenção: Não pode usar as classes do pacote `pt.ua.p2utils` neste problema.

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)
```

```
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 E13.

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

Atenção: Não pode usar as classes do pacote `pt.ua.p2utils` neste problema.

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!
```

Exercício E14.

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 `pt.ua.p2utils`.

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 E15.

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

Atenção: Não pode usar as classes do pacote `pt.ua.p2utils` 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 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 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 1 2 out out out
```

```
IN 1
IN 2
OUT: [1:1]
OUT: [2:1]
ERROR: 1 or more elements are required!
QUEUE: {}
```

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

```
IN 1
ERROR: 2 or more elements are required!
ERROR: 2 or more elements are required!
ERROR: invalid argument!
QUEUE: {[1:1]}
```

Exercício E16.

O programa `MainTrain` demonstra a utilização de uma estrutura de dados para gerir a carga e descarga de vagões num comboio de mercadorias. Crie a classe `Train` de forma que este programa compile e funcione devidamente.

Atenção: Não pode usar as classes do pacote `pt.ua.p2utils` neste problema.

Um objecto da classe Train representa um comboio composto de vários vagões de mercadorias a granel. Quando se cria um comboio, é necessário especificar a capacidade de cada vagão e a capacidade total que o comboio suporta, ambas em toneladas. Pode acrescentar-se um vagão com certa carga à cauda de um comboio (addWagon) ou pode retirar-se um vagão da cauda (removeWagon), segundo uma política LIFO (último a entrar é o primeiro a sair). Naturalmente, a carga de um vagão não pode superar a sua capacidade e só se pode acrescentar um vagão que não faça ultrapassar a carga total máxima do comboio. Também é possível pedir para descarregar (unload) uma dada quantidade, o que será feito pela descarga completa e retirada de zero ou mais vagões da cauda e pela descarga parcial de outro vagão para completar a quantidade pedida. Em qualquer altura é possível obter uma lista da carga nos vagões do comboio (list); saber o número de vagões (size) ou a carga total transportada (totalCargo).

Exemplos de utilização e resultados esperados:

```
java -ea MainTrain 10 100 1 2 3 R R 4.5 0.1
```

```
(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 100.0 ton.)
args[2]="1": Junta vagão com 1.0 ton
  (1 vagões, 1.0 ton): Loc0_[1.0]
args[3]="2": Junta vagão com 2.0 ton
  (2 vagões, 3.0 ton): Loc0_[1.0]_[2.0]
args[4]="3": Junta vagão com 3.0 ton
  (3 vagões, 6.0 ton): Loc0_[1.0]_[2.0]_[3.0]
args[5]="R": Retira vagão com 3.0 ton
  (2 vagões, 3.0 ton): Loc0_[1.0]_[2.0]
args[6]="R": Retira vagão com 2.0 ton
  (1 vagões, 1.0 ton): Loc0_[1.0]
args[7]="4.5": Junta vagão com 4.5 ton
  (2 vagões, 5.5 ton): Loc0_[1.0]_[4.5]
args[8]="0.1": Junta vagão com 0.1 ton
  (3 vagões, 5.6 ton): Loc0_[1.0]_[4.5]_[0.1]
```

```
java -ea MainTrain 10 100 4 2 5 7 -2 -11 -1
```

```
(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 100.0 ton.)
args[2]="4": Junta vagão com 4.0 ton
  (1 vagões, 4.0 ton): Loc0_[4.0]
args[3]="2": Junta vagão com 2.0 ton
  (2 vagões, 6.0 ton): Loc0_[4.0]_[2.0]
args[4]="5": Junta vagão com 5.0 ton
  (3 vagões, 11.0 ton): Loc0_[4.0]_[2.0]_[5.0]
args[5]="7": Junta vagão com 7.0 ton
  (4 vagões, 18.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[7.0]
args[6]="-2": Descarrega 2.0 ton e retira 0 vagões vazios.
  (4 vagões, 16.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[5.0]
args[7]="-11": Descarrega 11.0 ton e retira 2 vagões vazios.
  (2 vagões, 5.0 ton): Loc0_[4.0]_[1.0]
args[8]="-1": Descarrega 1.0 ton e retira 1 vagões vazios.
  (1 vagões, 4.0 ton): Loc0_[4.0]
```

```
java -ea MainTrain 10 20 2 10 11

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 20.0 ton.)
args[2]="2": Junta vagão com 2.0 ton
  (1 vagões, 2.0 ton): Loc0_[2.0]
args[3]="10": Junta vagão com 10.0 ton
  (2 vagões, 12.0 ton): Loc0_[2.0]_[10.0]
args[4]="11": ERRO: Sobrecarga de vagão!
```

```
java -ea MainTrain 10 20 5 7 9

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 20.0 ton.)
args[2]="5": Junta vagão com 5.0 ton
  (1 vagões, 5.0 ton): Loc0_[5.0]
args[3]="7": Junta vagão com 7.0 ton
  (2 vagões, 12.0 ton): Loc0_[5.0]_[7.0]
args[4]="9": ERRO: Sobrecarga do comboio!
```

Exercício E17. (Problema da AIP-2 de 2014-2015.)
Muitas empresas e instituições prestam serviços diferenciados, gerindo a prestação de cada serviço por ordem de chegada dos clientes. Neste trabalho, pretende-se que colabore no desenvolvimento de uma aplicação de gestão de serviços em que cada cliente é identificado pelo seu nome. Assim, é já fornecida uma classe que suporta a gestão de um serviço usando uma fila (classe `ServiceQueue`) que está implementada com base num vector de dimensão fixa. O atributo `beingServed` da classe `ServiceQueue` deve conter sempre o nome do cliente que está ser atendido, ou `null`, caso nenhum esteja a ser atendido. É também fornecida uma implementação bastante incompleta de uma classe para suporte da gestão de um conjunto de serviços (classe `ServiceManager`) com base num vector de instâncias de `ServiceQueue`. Finalmente, pode usar a classe `TestServices` para testar as funcionalidades pedidas.

a. Implemente um método `queueFor(clientName, serviceName)` na classe `Service Manager` que, dado o nome de um cliente e o nome de um serviço, coloca o nome do cliente na fila desse serviço. Introduza as asserções (pré-condições, pós-condições ou outras) que entenda adequadas neste método.

b. Implemente um método `serveNext(String serviceName)` na classe `ServiceManager` que actualize a estrutura de dados por forma a assinalar o início da prestação do serviço `serviceName` ao próximo cliente, retirando-o da respectiva fila. Note, entretanto, que um cliente pode estar em várias filas ao mesmo tempo, mas não pode ser atendido ao mesmo tempo em dois ou mais serviços. Assim, se no momento em que este método é chamado o primeiro cliente da fila estiver já a ser atendido noutro serviço, ele é simplesmente retirado da fila. O mesmo se aplica ao segundo, etc. O cliente a atender será o primeiro da fila que não esteja já a ser atendido em outro serviço.

c. Implemente um método `endService(serviceName, time)` na classe `ServiceManager` que, dado o nome de um serviço terminado e o respectivo tempo de atendimento, registe a conclusão do serviço. Este método deve invocar o método `logServiceData( serviceIndex, client, time)` que, dado o índice de um serviço, o nome de um cliente e o tempo de atendimento, regista estas informações numa estrutura de histórico.

d. Implemente um método `maxServiceTime()` na classe `ServiceManager` que devolva o maior tempo de atendimento registado no histórico de serviços. O histórico está implementado como uma lista ligada em que os nós são instâncias da classe `HistoryNode`,

já disponível. O método `logServiceData(...)`, já referido, acrescenta informação a esta estrutura. Para cotação total, a implementação de `maxServiceTime()` deverá ser recursiva.

e. A função `validServiceName(serviceName)`, já disponível na classe `ServiceManager`, verifica se uma dada cadeia de caracteres é um nome válido de um serviço. Implemente uma função equivalente `validServiceNameRec(serviceName)` usando um algoritmo recursivo. Se necessário, a sua solução pode incluir uma função auxiliar além da função pedida.

f. Implemente na classe `ServiceManager` um método estático `sort(a, start, end)` que, dado um vector de cadeias de caracteres `a` e os limites `start` e `end` de um subvector, tal como usado nas aulas, proceda à ordenação por fusão do vector. Para fundir subvectores ordenados, pode usar a função `mergeSubarrays(a, start, middle, end)` já disponibilizada na classe. Caso não consiga implementar a ordenação por fusão, pode obter metade da cotação implementando outro algoritmo de ordenação.

g. Implemente um método `alphabeticalClientList(serviceName)` na classe `Service Manager` que, dado o nome de um serviço, devolve um vector ordenado alfabeticamente com os clientes presentemente na fila desse serviço

Exercício E18. (Problema da AIP-2 de 2014-2015.)

Num terminal de contentores, os contentores são armazenados num conjunto de pilhas. Cada pilha suporta no máximo um certo número de contentores. Sempre que chega um novo contentor para armazenar, tem de ser empilhado numa das pilhas que não esteja cheia. Para expedir um certo contentor, é necessário encontrá-lo numa das pilhas, retirar os contentores que estejam por cima e rearmazená-los noutras pilhas. Neste trabalho pretende-se desenvolver uma aplicação de gestão de um terminal de contentores. Para isso, são fornecidas as seguintes classes:

`Container` que representa um contentor, caraterizado pelo tipo de carga que contém ("arroz", "bananas", etc) e por um identificador único que lhe é atribuído automaticamente na criação. O contentor também inclui um contador do número de operações (movimentações) a que foi sujeito.

`ContainerStack` que implementa uma pilha de contentores usando um vector (array) de dimensão fixa.

`ContainerTerminal` que trata da gestão de um array de pilhas que formam o terminal. Nesta classe já encontra um construtor bem como algumas funções auxiliares que lhe poderão ser úteis.

`TestContainers` que é tem um programa que faz diversas operações e testes a estas classes. Use-o como indicador da funcionalidade esperada das classes, e como forma de as testar. Execute-o sempre com verificação das asserções: `java -ea TestContainers [...]`.

Integre as seguintes funcionalidades:

a. Crie uma função `toString()` na classe `ContainerStack` que devolva uma string representando os contentores da pilha, do mais antigo (bottom) ao mais recente (top). Repare que a classe `Container` já tem um método análogo para representar cada contentor.

b. Crie uma função `store(container)` na classe `ContainerTerminal` que permita armazenar um novo contentor. Para isso, deverá encontrar a primeira pilha que não esteja cheia e empilhar aí. Naturalmente, só pode funcionar se o terminal não estiver lotado. Introduza as asserções (pré-condições, pós-condições ou outras) que entenda adequadas neste método.

c. Crie uma função `retrieve(type)` na classe `ContainerTerminal` que procura um contentor com carga de um certo tipo, e se encontrar, retira-o e devolve-o. Para retirar o contentor desejado, pode ser necessário retirar os contentores que estejam por cima e rearmazená-los noutras pilhas, um de cada vez. Se não houver nenhum contentor do tipo certo no terminal, deve devolver a referência `null`. Faça uso das funções de pesquisa já fornecidas na classe. A função `retrieve` também deve invocar `logContainerInfo`, passando o contentor que retirou, de forma a atualizar um registo histórico dos contentores retirados.

d. Acresente um método `averageOpsPerContainer()` que percorra o registo histórico e devolva o número médio de operações de empilhamento por contentor. Este registo está implementado como uma lista ligada em que os nós são instâncias da classe `HistoryNode`, já fornecida. O método `logContainerInfo(...)`, já referido, acrescenta informação a esta estrutura. Para cotação total, a implementação de `averageOpsPerContainer()` deverá ser recursiva.

e. A classe `ContainerStack` já inclui uma função `search(type)` que procura um contentor de certo tipo e devolve um inteiro que indica a sua posição relativa na pilha, a contar do topo, ou -1 se não encontrar. Faça uma função equivalente `searchRec(type)` que use um algoritmo recursivo. Pode ser necessário criar uma função auxiliar para resolver o problema.

f. Implemente na classe `ContainerStack` um método estático `sort(a, start, end)` que, dado um vector `a` e os limites `start` e `end` de um subvector, tal como usado nas aulas, proceda à ordenação por fusão do vector. Para fundir subvectores ordenados, pode usar a função `mergeSubarrays(a, start, middle, end)` já disponibilizada na classe. Caso não consiga implementar a ordenação por fusão, pode obter metade da cotação implementando outro algoritmo de ordenação.

g. Implemente um método `containersInStack()` na classe `ContainerStack` que devolve um vector com todos os contentores da pilha, ordenados por ordem crescente dos respetivos identificadores.

Exercício E19.

Pretende-se construir um programa (`PhoneCalls.java`) que processe uma lista de chamadas telefónicas que estão descritas em ficheiros do tipo `*.cls` (por exemplo: `calls.cls`, com a organização seguinte: *número de origem*, *número de destino* e *duração* em segundos). Tem ainda disponível ficheiros do tipo `*.nms` (por exemplo: `names.nms`) com a informação seguinte: *número* e *nome*.[6]

calls.cls

```
009047362 269633507 287
269633507 545065453 723
269633507 021693118 680
513512774 269633507 265
564359070 564359070 751
503512774 396659735 475
071356756 181964754 719
```

names.nms

```
396659735 Sergio Tavares
269633507 Paula Nunes
208974207 Mario Nunes
462589991 Maria Nunes
564359070 Joao Nunes
181964754 Ana Nunes
503512774 Paula Melo
009047362 Miguel Silva
482318937 Pedro Oliveira
071356756 Tomas Alberto
```

a. Utilizando da melhor forma possível o pacote `pt.ua.p2utils`, faça um programa que leia a informação dos ficheiros do tipo `*.nms` passados como argumento para uma (ou mais) estrutura de dados apropriada. Por outro lado, para os argumentos terminados em `.cls`, o programa deve listar o seu conteúdo, mas trocando o número de telemóvel pelo nome do dono, se já for conhecido. Os ficheiros devem ser processados pela ordem dos argumentos. Por exemplo:

java -ea PhoneCalls names.nms calls.cls

```
Miguel Silva to Paula Nunes (287 seconds)
Paula Nunes to 545065453 (723 seconds)
Paula Nunes to 021693118 (680 seconds)
513512774 to Paula Nunes (265 seconds)
Joao Nunes to Joao Nunes (751 seconds)
Paula Melo to Sergio Tavares (475 seconds)
Tomas Alberto to Ana Nunes (719 seconds)
```

b. Altere o programa de forma que qualquer argumento que não termine com as extensões definidas (`.nms` ou `.cls`) seja considerado um número de telemóvel. Para cada um desses argumentos, o programa deve escrever imediatamente a lista de chamadas feitas por esse telemóvel, e a lista de chamadas recebidas. Tal como na alínea anterior trocando o número de telemóvel pelo nome do dono, sempre que possível.

java -ea PhoneCalls names.nms calls.cls 269633507

```
...
Calls made by Paula Nunes:
  - to phone 021693118 (680 seconds)
  - to phone 545065453 (723 seconds)
Calls received by Paula Nunes:
  - from phone 513512774 (265 seconds)
  - from Miguel Silva (287 seconds)
```

[6] O número é a primeira palavra da linha, sendo as restantes palavras consideradas como sendo o nome.

Exercício E20.

Pretende-se construir um programa (`CityTraveler.java`) que apresente as cidades visitadas por cada um dos funcionários de uma empresa. Para cada cidade existe um ficheiro com a lista dos funcionários que a visitaram.

Por exemplo, dados os seguintes ficheiros:

Aveiro
Maria
Marisa
Miguel
António
Luis
José

Porto
Luis
Miguel
António
Rui
Pedro
Francisco

Lisboa
Manuel
Miguel
Maria

Se executar:

```
java -ea CityTraveler Aveiro Porto Lisboa
```

a saída do programa seria (eventualmente com os funcionários numa outra ordem):

Luis	:	Aveiro Porto
José	:	Aveiro
Rui	:	Porto
Maria	:	Aveiro Lisboa
Miguel	:	Aveiro Porto Lisboa
Francisco	:	Porto
Pedro	:	Porto
António	:	Aveiro Porto
Marisa	:	Aveiro
Manuel	:	Lisboa

a. Utilizando o pacote `pt.ua.p2utils`, comece por escolher uma estrutura de dados adequada para resolver este problema e crie uma função que preencha essa estrutura com a informação retirada de um único ficheiro. Detalhes:

   - Cada linha (não vazia) do ficheiro corresponde ao nome de um funcionário.
   - O nome do ficheiro é o nome da cidade.

b. Complete o programa para fazer o pretendido, tendo em consideração que:

   - O programa recebe como argumentos os ficheiros com as listas de funcionários.
   - A lista de todos os funcionários deve ser escrita no dispositivo de saída.

Exercício E21.

Pretende-se construir um programa (`Restaurante.java`) que faz a gestão da entrada de alimentos e saída de refeições de um restaurante. A saída de uma refeição só pode ocorrer

assim que o restaurante tiver as quantidades de ingredientes para ela requeridos e depois de todos os anteriores pedidos de refeições terem sido servidos. O programa recebe os alimentos e os pedidos de refeições através de um ou mais ficheiros de entrada (passados como argumentos do programa). Estes ficheiros têm o seguinte formato: Uma entrada de um ingrediente é uma linha com o prefixo "`entrada: `" seguido de uma palavra com o nome do ingrediente. Os pedidos para refeições são linhas com o prefixo "`saida: `", seguido de uma lista de palavras compostas pelo nome do ingrediente e a quantidade requerida (separadas pelo símbolo `:`). Considere que nestes pedidos não pode haver a repetição do mesmo ingrediente. Um desses ficheiros é exemplificado em baixo.

O programa a implementar deve processar todos os seus argumentos interpretando-os como indicado em cada alínea (na dúvida, verifique o comportamento do ficheiro `jar`).

a) Utilizando da melhor forma possível o pacote `pt.ua.p2utils`, faça um programa que leia e registe a informação das entradas de ingredientes dos ficheiros de entrada e a escreva com o formato exemplificado à frente (nesta alínea, pode ignorar completamente as saídas). Deve ser criada uma função para a leitura da informação de cada ficheiro. A seguir exemplifica-se o comportamento que o programa deve ter.

food-data01.txt

```
entrada:  cerveja
entrada:  sopa
entrada:  carne
entrada:  carne
entrada:  feijao
saida:  sopa:1 carne:1 cerveja:1
saida:  sopa:1 peixe:1 agua:1 torta:1
entrada:  sumo
entrada:  sopa
saida:  carne:1 sumo:1
entrada:  torta
saida:  carne:1 feijao:1
entrada:  peixe
entrada:  agua
```

java -ea Restaurante food-data01.txt

```
Comida em stock:
  feijao: 1
  torta: 1
  cerveja: 1
  sumo: 1
  peixe: 1
  carne: 2
  sopa: 2
  agua: 1
```

b) Altere o programa por forma a servir também as refeições. As refeições têm de ser servidas exactamente pela ordem com que aparecem nos ficheiros de entrada e logo que possível (portanto terá de modificar a função anterior). No exemplo dado, a primeira refeição pode de imediato ser servida, já que existem em stock todos os ingredientes, mas o mesmo já não acontece com a refeição seguinte. Por outro lado, a terceira refeição fica "suspensa" pelo facto da segunda ainda não ter sido servida. No final, o programa deve indicar, para além da comida em stock (alínea a), as refeições que ficaram pendentes ou por falta de ingredientes ou por estarem atrás de uma refeição pendente.[7]

java -ea Restaurante food-data02.txt

```
Refeicao servida:  sopa:1 carne:1 cerveja:1
Refeicao servida:  sopa:1 peixe:1 agua:1 torta:1
Refeicao servida:  carne:1 sumo:1
Comida em stock:
  feijao: 1
Refeicao pendente:  carne:1 feijao:1
```

Exercício E22. (Problema da APF de 2015-2016.)

A exportação de produtos baseia-se no transporte de produtos em contentores por via

[7] Para facilitar a depuração do programa existem vários comandos de teste (`test*.sh`) que pode utilizar (quer para o seu programa quer para o programa `jar` fornecido).

terrestre e marítima. Os portos são o centro desta actividade. Neste exercício, um porto (classe `Harbour`) é composto por um conjunto de cais ou docas (classe `Dock`). Em cada doca confluem contentores (transportados normalmente em camiões) através de um ramal de acesso (*access extension*). Os contentores (classe `Container`) são temporiamente armazenados numa pilha, e são finalmente enviados para o seu destino por via marítima. Para o controlo das actividades de cada doca, utiliza-se: uma fila (classe `Queue` do pacote `pt.ua.p2utils`), para os contentores que aguardam no ramal de acesso; e uma pilha (classe `Stack` do mesmo pacote), para os contentores já empilhados na doca. O programa `TestHarbour` permite testar as funcionalidades pedidas em seguida.

Implemente em primeiro lugar as seguintes funcionalidades na classe `Dock`:

a) `enterContainer(c)` - Regista a entrada de um contentor na fila do ramal de acesso.

b) `shipContainer()` - Regista a saída de um contentor. O contentor que sai é o que está no topo da pilha. A função devolve o contentor retirado.

c) `moveFromAccessToStack()` - Transfere da fila para a pilha, por ordem de chegada, todos os contentores disponíveis que couberem na pilha. A pilha não pode ter mais do que `maxStack` contentores.

d) `insertFirstInStack()` - Transfere o primeiro contentor da fila para a pilha. Neste caso, a pilha tem que estar sempre ordenada por distância ao destino (distância maior no topo), por isso poderá ser necessário remover temporiamente alguns contentores para uma pilha auxiliar.

Implemente agora na classe `Harbour` os seguintes métodos:

e) `count()` - Devolve o número total de contentores em todas as docas.

f) `countToDestination(dest)` - Dado o nome de uma cidade, devolve o número de contentores destinados a essa cidade. De preferência, deverá usar uma estrutura do pacote `pt.ua.p2utils` que facilite a obtenção dessa informação. Poderá acrescentar código em outros métodos por forma a manter a informação actualizada.

g) `currentDestinations()` - Devolve um vector com os nomes das cidades de destino de todos os contentores existentes no porto.

Exercício E23. (Problema da APF de 2015-2016.)

As palavras "isentas", "sentais" e "sinetas" são anagramas umas das outras, isto é: são formadas exatamente pelas mesmas letras, em igual número, mas por ordens diferentes. É fácil de confirmar se ordenarmos as suas letras: todas dão "aeinsst".

Pretendemos descobrir todos os grupos de anagramas de uma lista de n palavras. Complete a função `findAnagrams` do programa `AllAnagrams` para fazer isso com complexidade algorítmica $O(n)$. A função devolve uma lista de listas, representando o conjunto de grupos

de anagramas que foram identificados. Para garantir a complexidade pretendida, considere utilizar estruturas de dados adicionais.

O ficheiro `words100` tem uma lista de 100 palavras distintas e inclui vários grupos de anagramas. Teste o programa com esta lista e com outras progressivamente maiores também disponíveis em anexo.

```
$ java AllAnagrams words100
[[hooping, poohing], [dungaree, underage], [erodes, redoes],
 [enlists, listens, silents, tinsels], [pester, peters, preset],
 [equip, pique], [bin's, nib's], [countered, recounted],
 [clinkers, crinkles], [helot's, hotel's]]
Found 10 anagram groups in 0.021 seconds

$ java AllAnagrams words99171
[[sensor, snores], [tenors, tensor], [owners, worsen],
 [periled, replied], [glass, slags], [deliver, relived, reviled],
 [coronas, racoons], [dietitian, initiated], ...]
Found 4446 anagram groups in 0.234 seconds
```