Exercises for Autonomous Practice

[This document does not match the 2017 portuguese version of guiaoTA, yet!]

Exercise E1

[1] The files `JogaJogoDoGalo.java` and `jogos/JogoDoGalo.java` define a program and a module, respectively, implementing a "Tic-Tac-Toe" game. Intentionally, some errors were introduced.

a. Correct module `JogoDoGalo.java` in order to remove its syntactical (compilation) errors.

   - For compiling it use the following command: `javac JogaJogoDoGalo.java`

b. The main program `JogaJogoDoGalo.java` contains a semantic error. Find and correct it.

   - You may execute it with: `java -ea JogaJogoDoGalo`
   - You may execute a correct version with: `java -ea -jar JogaJogoDoGalo.jar`

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



Jogador X ganhou!
```

Note: The sentence in the output means "Player X won!"

c. Make the program robust regarding the module exploitation (exceptions are not needed).

d. Change the program `JogaJogoDoGalo.java` in order to deploy championships up to 10 games, finishing when one of the players scores 3 victories. At the end of each game the score of each player should be displayed.

Exercise E2

In the construction of residential buildings, the various professionals need to manage information on each housing unit (house or apartment) and extract various properties. Some classes are provided to represent points in Cartesian space (`Point`), housing units

[1] Problem from the AIP of 2009-2010.

(`House`) and rooms within a house (`Room`), as well as a class for testing (`TestHouse`). It is assumed that each room will have a rectangular shape, aligned with the axes of a given coordinate system. Coordinates and distances are specified in meters.

You should incorporate the following new features in the program:

a. As you can see through the methods of class `House`, the rooms of a house are stored in an array. In particular, the method `addRoom(room)` in the class `House` adds a room to the array. Change this method so that it returns the index of the element of the array in which the room was stored. This index will serve as an identifier of the room.

b. Regarding the addition of doors to a house, complete the definition of class `Door` with the following methods (at least):

   - `Door(RoomId1,RoomId2,Width,Height)` - constructor that receives the identifiers of the two rooms connected by this door, as well as the dimensions of the door.
   - `area()` - a method that returns the area of the door.

c. The constructor of `House` creates an array for storing the doors. The capacity of this array is equal to the initial capacity of the array of rooms. The method `addDoor(Door)` in class `House` adds a new door, but fails when the doors array is full. Change this method so that the array capacity is extended with an extra `extensionSize` elements whenever that happens.

d. Create a method `roomClosestToRoomType(roomType)` in class `House` that, given a type of room, returns the identifier of the room closest to any room of that type. Consider the distance in a straight line between the centers of the rooms.

e. Create a method `maxDoorsInAnyRoom()` in class `House` that returns the maximum number of doors in any room of the house.

Exercise E3

Available only in Portuguese.

Exercise E4

Available only in Portuguese.

Exercise E5

Implement a recursive function `factors` receiving an integer number as argument and returning a `String` with the product of its factors. For example, the following program invocation

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

should output:

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

Exercise E6

Available only in Portuguese.

Exercise E7

Write a program that receives an integer number as argument and writes all its divisors other than itself and the unity and, recursively, does the same for all those divisors.

Here are some use cases of the intended program:

`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

Exercise E8

Write a program that receives a rational number belonging to $]\,0, 1\,[$, expressed as a fraction (n/d), and write that fraction as the sum of unitary fractions (with unity as numerator) with different denominators[2]. The program to develop must use a recursive algorithm.

Here are some use cases of the program:

`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`

To solve the problem consider the following strategy (called "greedy" and proposed by Fibonacci in the 13th Century):

a. Attempt to subtract from the fraction the highest possible unitary fraction. To find out that unitary fraction $(1/d)$, with the lowest possible d, consider the following expression:

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

[2]Fibonacci has demonstrated that any rational number can be expressed by a finite sum of unitary fractions with different denominators.

b. The fraction will be the sum of the unitary fraction $1/d$ added to the unitary fraction obtained from the fraction resulting from the difference (for which you should apply the same algorithm);

c. The process ends when the numerator is divisor of the denominator (an evidence that it is already an unitary fraction).

Exercise E9

Available only in Portuguese.

Exercise E10

Develop in the `LinkedList` class iterative and recursive implementations of the following methods:

- `count(e)` - return the number of occurrences of the given element in the list.
- `indexOf(e)` - returns the index of the first occurrence of the specified element in this list, or `-1` if this list does not contain the element.
- `cloneReplace(x,y)` - returns a copy of the list with all occurrences of `x` replaced by `y`.
- `cloneSublist(start,end)` - returns a copy of the portion of this list between the `start` index, inclusive, and to the `end` index, exclusive.
- `cloneExceptSublist(start,end)` - returns a copy of the list excluding portion between the `start` index, inclusive, and to the `end` index, exclusive.
- `removeSublist(start,end)` - removes from the list the portion between the `start` index, inclusive, and to the `end` index, exclusive.

Exercise E11

Program `ArraySorting.java` contains the implementation of several sorting algorithms as well as a `main()` function that applies these algorithms to randomly generated arrays of numbers (by the function `randomArray()`. The implementations of the sorting algorithms include some assertions to check pre- and post-conditions. If the program is run with assertions enabled (option `-ea`), normal termination of the program indicates that the sorting algorithms correctly sorted the randomly generated arrays. Revise the whole implementation and correct any errors that you find.

Exercise E12

Create a `LeakyQueue` module, based on the queue data structure, in order to enable program `ProgX` to work properly[3].

[3]You cannot use the modules from package `p2utils` in this problem.

A N-long leaky queue is a queue-based data structure that keeps only the last N inserted values. When the queue is full (with N elements), inserting a new one implies dropping the first one from the queue.

Here are some use cases (with $N = 3$) and expected results:

```
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)
```

Exercise E13

The program `ProgX` verifies if an arithmetic expression (formed by algarisms, elementary operations and parenthesis) is syntactically correct. Write the module `PilhaX`, based on the stack data structure, in order to enable this program to work properly[4].

Here are some use cases and expected results:

```
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!
```

Exercise E14

Write a program (`JustifiedText.java`) for aligning a text to both margins, simultaneously (justified alignment). The program receives as parameters the line length and the name of the file with the text to align, which should be written in the standard output.

To tackle this problem you should use at least one suitable data structure from package `p2utils.jar`.

For instance, given the following text on the file `texto.txt`:

[4]You cannot use the modules from package `p2utils` in this problem.

```
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.
```

these are two use cases of the program:

`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.`

Details to take into consideration:

- Each output line must contain the maximum possible words without trespassing the maximum line length. A "word" should be considered any sequence of characters delimited by white spaces (space characters, tab characters, etc.).
- Words cannot be joined or merged (null spacing) nor split across lines.
- The lengths of spaces between words of the same line should differ at most by one character.
- The last line of each paragraph must be left aligned (with a single space between words). Consider that a paragraph terminates with an empty line or with the end of the file.

Exercise E15

Available only in Portuguese.

Exercise E16

`MainTrain` is a program that demonstrates the usage of a data structure for managing the loading and unloading of wagons on a freighter train. Create the `Train` module in a way that allows this program to compile and work properly[5].

An object belonging to the Train class represents a train composed by different freighter wagons in bulk. When a train is created, it is necessary to specify the capacity of each wagon, and the total capacity supported by the train, both in tons. You can add a wagon with a certain amount of cargo to a train (addWagon) or you can remove a wagon from its tail (removeWagon), according to a LIFO policy (the Last one In is the First one Out). Naturally, the load of a wagon cannot surpass its capacity and you can only add a wagon

[5]You cannot use the `p2utils` modules in this problem.

which doesn't overcome the maximum total cargo of the train. It is also possible to request the unload (unload) of a certain amount. This can be accomplished through completely unloading and removing zero or more wagons from the tail, as well as partially unloading another wagon to complete the requested amount. At any time it is possible to obtain a list with the cargo list of the train wagons (list); know the number of wagons (size) or the total transported cargo (totalCargo).

Usage examples and expected results:

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

(Wagons capacity: 10.0 ton.)
(Train capacity: 100.0 ton.)
args[2]="1": Joins wagon with 1.0 ton
  (1 wagons, 1.0 ton): Loc0_[1.0]
args[3]="2": Joins wagon with 2.0 ton
  (2 wagons, 3.0 ton): Loc0_[1.0]_[2.0]
args[4]="3": Joins wagon with 3.0 ton
  (3 wagons, 6.0 ton): Loc0_[1.0]_[2.0]_[3.0]
args[5]="R": Removes wagon with 3.0 ton
  (2 wagons, 3.0 ton): Loc0_[1.0]_[2.0]
args[6]="R": Removes wagon with 2.0 ton
  (1 wagons, 1.0 ton): Loc0_[1.0]
args[7]="4.5": Joins wagon with 4.5 ton
  (2 wagons, 5.5 ton): Loc0_[1.0]_[4.5]
args[8]="0.1": Joins wagon with 0.1 ton
  (3 wagons, 5.6 ton): Loc0_[1.0]_[4.5]_[0.1]
```

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

(Wagons capacity: 10.0 ton.)
(Train capacity: 100.0 ton.)
args[2]="4": Join wagon with 4.0 ton
  (1 wagons, 4.0 ton): Loc0_[4.0]
args[3]="2": Join wagon with 2.0 ton
  (2 wagons, 6.0 ton): Loc0_[4.0]_[2.0]
args[4]="5": Join wagon with 5.0 ton
  (3 wagons, 11.0 ton): Loc0_[4.0]_[2.0]_[5.0]
args[5]="7": Join wagon with 7.0 ton
  (4 wagons, 18.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[7.0]
args[6]="-2": Unloads 2.0 ton and removes 0 empty wagons.
  (4 wagons, 16.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[5.0]
args[7]="-11": Unloads 11.0 ton and removes 2 empty wagons.
  (2 wagons, 5.0 ton): Loc0_[4.0]_[1.0]
args[8]="-1": Unloads 1.0 ton and removes 1 empty wagons.
  (1 wagons, 4.0 ton): Loc0_[4.0]
```

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

(Wagons capacity: 10.0 ton.)
(Train capacity: 20.0 ton.)
args[2]="2":  Join wagon with 2.0 ton
  (1 wagons, 2.0 ton): Loc0_[2.0]
args[3]="10": Join wagon with 10.0 ton
  (2 wagons, 12.0 ton): Loc0_[2.0]_[10.0]
args[4]="11": ERROR: Wagon overload!
```

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

(Wagons capacity: 10.0 ton.)
(Train capacity: 20.0 ton.)
args[2]="5": Join wagon with 5.0 ton
  (1 wagons, 5.0 ton): Loc0_[5.0]
args[3]="7": Join wagon with 7.0 ton
  (2 wagons, 12.0 ton): Loc0_[5.0]_[7.0]
args[4]="9": ERROR: Train overload!
```

Exercise E17

Available only in Portuguese.

Exercise E18

In a container terminal, containers are stored in a set of stacks. Each stack can store container up to a certain maximum of a number. Whenever there comes a new container for storing, it must be stacked on a stack that is not full. To retrieve a certain container, you must find it in one of the cells, removing the containers which are above, and storing them again in other stacks. In this work, you will develop some functions for container terminal management application.

For this, the following classes are provided:

`Container` Characterized by containing a type ("rice", "banana", etc.) and a unique identifier, which is automatically assigned to it at creation. The container also includes a counter operations counter (movements) it has undergone.

`ContainerStack` Implements a stack of containers using a vector (array) of fixed dimension.

`ContainerTerminal` Deals with the management of an array of container stacks that make up the terminal. In this class you already fund a constructor and some auxiliary functions that may be useful.

`TestContainers` Has a program that makes various operations and tests to these classes. Use it as an indicator of the expected functionality of classes, and as a way to test them. Run it always with verification of assertions: `java -ea TestContainers [...]`.

Integrate the following new functionalities:

a. Create a function toString() in the class ContainerStack that returns a string representation of the cell container, from the oldest (bottom) to the latest (top). Note that the Container class already has an analogous method to represent each container.

b. Create a function store(container) in ContainerTerminal class which enables to store a new container. Therefore, it finds the first stack that is not full and stacks the container there. Of course, this can only work if the terminal is not crowded. Enter the assertions (preconditions, postconditions or other) deemed appropriate for this method.

c. Create function retrieve(type) in ContainerTerminal class looking for a container of a certain type, removes it and returns it. To remove the desired container, it can be necessary to remove the containers that are above them and store them in other stacks, one at a time. If the container type does no exist in the container terminal, the function return null. Make use of the search functions already provided in the class. The retrieve function should also invoke logContainerInfo, passing the container that was retrieved in order to update a historical record of retrieved containers.

d. Acresente a method averageOpsPerContainer() to scroll through the historical record and return the average number of stacking operations per container. This log is implemented as a linked list in which nodes are instances of HistoryNode, a class already provided. The logContainerInfo (...) method, cited above, adds information to this structure. For full scoring, the implementation of averageOpsPerContainer() should be recursive.

e. The ContainerStack class already includes a function search(type) for finding a certain type of container and returns an integer that indicates their relative position in the stack, from the top, or -1 if you the type is not find. Make an equivalent role searchRec(type) that uses a recursive algorithm. You may need to create a helper function to solve the problem.

f. Implement a static method sort(a,start,end) in class ContainerStack that, given an array and the start and end limits of a subarray, sorts the array using the mergeSort algorithm. To merge sorted subarrays, you can use the function mergeSubarrays(a,start,middle,end) already available in the module. If you cannot implement mergeSort, you can get half the full score by implementing another sorting algorithm.

g. Implement a method containersInStack() in class ContainerStack which returns a array of all containers sorted in ascending order of the respective identifiers.

Exercise E19

Write a program (`PhoneCalls.java`) for processing a list of phone calls described in `*.cls` files (for example: `calls.cls`, with the following per-line structure: *caller number*, *callee number* and *call duration* in seconds). You can also make use of `*.nms` files (for example: `names.nms`) with the following per-line information: *number* and *name*[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. Making the best possible use of package `p2utils`, write a program for reading the information from `*.nms` files, given as program argument, to one (or more) suitable data structures. On the other hand, for all program arguments referring `*.cls` files, the program must list their contents, replacing phone numbers by the name of their owners, if known. The files are to be processed in the order they are placed as arguments. For instance:

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. Change the program in order to allow any program argument not terminating with the previously used extensions (`.nms` or `.cls`) to be considered a phone number. For each of those arguments, the program must immediately write the list of calls it made, as well as the list of calls it received. As in the previous exercise, whenever possible replace a phone number by the name of its owner.

[6]The number is the first word in the line, while the name is formed by the remaining text until the end of the line.

```
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)
```

Exercise E20

Write a programa (`CityTraveler.java`) for presenting the cities visited by each employee of a company. For each city there is a file described the employees that visited it.

For example, given these two files:

Aveiro
Maria
Marisa
Miguel
António
Luis
José

Porto
Luis
Miguel
António
Rui
Pedro
Francisco

Lisboa
Manuel
Miguel
Maria

Executing:

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

the program output is (possibly with employees in a different order):

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. Using package `p2utils`, start by choosing the appropriate data structure for solving this problem and create a function for filling that structure with the information gathered from a single file. File details:

   - Each (non-empty) line contains the complete name of a single employee.

- The name of the file is the name of the city.

b. Complete the program to achieve its final goal, taking into consideration that:

- The program receives as arguments the (city) files with the list of (visiting) employees.
- The list of all the employees must be written in the standard output stream.

Exercise E21

The objective of the (`Restaurante.java`) program is to manage the input of ingredients and output of meals in a restaurant. The output of a meal can only occur as soon as the restaurant has the required amount of ingredients and after all previous meal requests have been fulfilled. The program receives the ingredients and meal orders through one or more input files (passed as arguments to the program). These files have the following format: An ingredient input is a line with the prefix: "`input:` " followed by a word with the name of the ingredient.

The meal orders are lines with the prefix: "`output:` ", followed by a list of words composed by the name of the ingredient and the required amount (separated by the symbol `:`). Consider that in these orders the same ingredient cannot be repeated. One of these files is exemplified below.

The program must process all its arguments interpreting them as indicated in each paragraph (in case of doubt, verify the behaviour of the `jar`) file).

a) By using in the best way possible the `p2utils` package, implement a program that reads and registers the ingredient input information from the input files, and writes it in the format explained next (in this exercise, you can completely ignore the outputs). A function for reading the information of each file should be written. Next, the program behaviour is exemplified.

food-data01.txt

```
input: beer
input: soup
input: meat
input: meat
input: beans
output: soup:1 meat:1 beer:1
output: soup:1 fish:1 water:1 pie:1
input: juice
input: soup
output: meat:1 juice:1
input: pie
output: meat:1 beans:1
input: fish
input: water
```

java -ea Restaurante food-data01.txt

```
Ingredients in stock:
  beans: 1
  pie: 1
  beer: 1
  juice: 1
  fish: 1
  meat: 2
  soup: 2
  water: 1
```

b) Change the program in a way that it also serves meals. Meals have to be served in the exact same order they appear in the input files and as soon as possible (as such, you need to modify the previous function). In the given example, the first meal can immediately be served, since all the ingredients are in stock, but the same doesn't happen with the next meal. In the other hand, the third meal stays on "hold" since the second hasn't been served yet. In the end, the program should indicate, besides the

food in stock (paragraph a), the meals that were on hold, due to lack of ingredients or for being behind a held meal[7].

java -ea Restaurante food-data02.txt

```
Served meal:  soup:1 meat:1 beer:1
Served meal:  soup:1 fish:1 water:1 pie:1
Served meal:  meat:1 juice:1
Ingredients in stock:
  beans: 1
Meals on hold:  meat:1 beans:1
```

Exercise E22

Available only in Portuguese.

Exercise E23

Available only in Portuguese.

[7]To facilitate debugging of the program there are several test commands (`test*.sh`) which you can use (whether for your program or for the provided `jar`