PG4200 Exam February 2022

The exam should **NOT** be done in group: each student has to do the exercises on his/her own. During the exam period, you are not allowed to discuss any part of this exam with any other student or person, not even the lecturer (i.e., do not ask questions or clarification on the exam). In case of ambiguities in these instructions, do your best effort to address them (and possibly explain your decisions in some code comments). Failure to comply to these rules will result in a failed exam and possible further disciplinary actions.

This exam for **PG4200** is composed of 5 exercises. You have **24** hours to complete these tasks. This exam will have a grade in the *A-F* range. Each exercise will be scored between 0 and 10 points. In order to pass, you **must** have **at least** 25 points, and you must provide an answer to all the exercises (i.e. **do not leave any exercise empty**).

Each exercise must be in its own file (e.g., *.txt* and *.java*), with name specified in the exercise text. All these files need to be in a single folder, which then needs to be zipped in a ***zip*** file with name *pg4200_<id>.zip*, where you should replace <id> with the unique id you received with these instructions. If for any reason you did not get such id, use your own student id, e.g. *pg4200_701234.zip*. No "*rar*", no "*tar.gz*", etc. If you submit a *rar* or *a tar.gz* format (or any other format different from *zip*), then an examiner will mark your exam as failed without even opening such file.

You need to submit all source code (eg., *.java*), and no compiled code (*.class*) or libraries (*.jar, .war*).

Note: there is **NO** requirement about writing test cases. However, keep in mind that, if your implementations are wrong, you will fail the exam. As such, it is recommended to write test cases for your implementations, to make sure your code works correctly. However, if you do so, do **NOT** submit them as part of your delivery (as the writing of the test cases will not be evaluated).

Exercise 1:
File: Ex01.java

a. Write a regular expression that matches an exercise definition for this exam. An exercise in this exam starts with the following 2 lines:
   - The first line contains "Exercise **<no>**", where **<no>** is the number of the exercise.
   - The second line contains the name and type of file you are expected to deliver. E.g. Ex**<no>** for name and .java, .txt as file types.

   You do **not** need to check that the numbers match between the file and the exercise, but both **<no>** instances should be numeric only.

   Examples.
   Your expression
   Should match:
   - "" Exercise 5:
     File: Ex05.txt ""
   - "" Exercise 3:
     File: Ex03.java ""
   - "" Exercise 13:
     File: Ex13.java""

Should not match:

- “”” Exercise no5:
  File: Ex05.txt “”” – the exercise number is not numeric only
- “”” Exercise 5: File: Ex05.txt“”” – they are not on separate lines
- “”” Exercise 5:
  File: Ex05.pdf “”” – not of a supported type.
- “”” Exercise 5:
  File: Ex05 “”” – no file type
- “”” Exercise 5:
  File: Ex_something05.txt “”” – file name does not match.

b. Write a regular expression that matches all questions asked by user @Bogdan, about the course. For example, this could be a filter on mattermost that allows someone to quickly see all the questions asked by a particular user.

In this case, you can assume that each sentence starts with the name of the user that has typed it followed by a colon (‘**:**’), and that there is only one sentence per line.

Examples:
Should match:
“@Bogdan: Has everyone completed all the exercises?”
“@Bogdan: Are there any additional questions?”
“@Bogdan: Where can I find the solutions for this exercise?”

Should not match:
“@Sven: Why do we need to do the exercises?” – different user asking the question
“@Bogdan: Exercises will be useful in understanding the topic better.” – not a question
“@Bogdan – advises that you do all the exercises” – different format than expected
“@Sven: Have you asked @Bogdan about this?” – different user asking the question

Exercise 2:
File Ex02.java
The task is to write a class called *InitiativeHandler*, that decides the initiative order for characters in an RPG. Your solution can be **inspired** from the code in the course, but it **must** be specialized to deal with the situation in this particular example.

Assume the Character class:

```
enum Initiative{
   FAST, MEDIUM, SLOW
}

public class Character {
   private Initiative initiativeRating;
   private int id;
   private int initiativeRoll;

   public int getInitiativeRoll() {
      return initiativeRoll;
   }
```

```
    public int rollInitiative(){
        int roll = (int) Math.floor(Math.random() * 20 + 1);
        initiativeRoll = roll * 10000 + id;
        return initiativeRoll;
    }
}
```

You need not worry about the details of how the initiative roll is computed (but you have the example above, if it helps).

The rules of initiative:

- FAST characters always go first
- MEDIUM characters go before SLOW ones
- For characters with the same initiative rating, they will be sorted based on initiativeRoll (which is computed as you can see above). This consists of a random roll. If two characters have the same roll, they will be differentiated based on id (which is unique, so no more ties should emerge). For this exercise you can assume that the initiativeRoll was computed earlier, is unique, and will not change during the execution of your code.

Implement the class *InitiativeHandler* to implement *InitiativeTemplate*.

```
public interface InitiativeTemplate {
    /***
     * This method adds the Character character to the initiative handler.
     * @param character - the character to be added.
     */
    public abstract void addCharacter(Character character);

    /***
     * This method removes the character with the given initiative (if one exists)
     * from the initiative handler.
     * If a character was removed, return true. Otherwise, return false.
     * @param characterInitiative - the character to be added.
     * @return - return true if a character was removed, false otherwise.
     */
    public abstract boolean removeCharacter(int characterInitiative);

    /***
     * This method returns an array of characters, sorted in order of initiative.
     * Initiative rules:
     * - FAST characters go first
     * - MEDIUM characters go before SLOW ones
     * - characters with similar rating go based on a die roll
     * - ties are broken by id.
     * @return - the ordered list of characters, based on the initiative rules.
     */
    public abstract ArrayList<Character> getInitiativeOrder();
}
```

You can assume the **id** is between 1 and 9000.

Initiative handler should contain 3 binary trees, one each for FAST, MEDIUM, and SLOW characters. The class should implement InitiativeTemplate and the methods described there, with the semantics described there. Use binary trees as a basis for your implementation.

NOTE: The name of the file will be Ex02.java, but it must contain the class InitiativeHandler. You can accomplish this by:

- developing the solution under InitiativeHandler, and renaming before submission (and after making sure the solution works as you expect).
- alternatively, you can have a different public class Ex02 in that file. In this case, make sure that the InitiativeHandler class fulfills the requirements stated above.

Exercise 3:
File: Ex03.txt
Compare your implementation of InitiativeHandler in Exercise 2 with a theoretical implementation based only on an ArrayList. Does your implementation perform better than the ArrayList? If yes, explain why. If not, explain in which cases your implementation could perform worse.

Details:

- How does adding a new character to the initiative handler happen in your implementation vs the array list? What is the worst case (O-) complexity? How did you calculate this?
- How does removing a character happen?

Assume a method called “getByName”, that returns a character from the initiative handler whose name matches an input.

```
public Character getByName(String name)
```

What is the worst case (O-) complexity of this method in the two implementations? Why is this the case?

Note: In this exercise you are expected to support your conclusions with analysis and arguments.

Exercise 4:
File: Ex04.java

Consider a collection of diploma projects belonging to students in the University.

```
public class DiplomaProject {
   String title;
   ArrayList<Student> authors;
}
```

Each project has a title and a list of authors.

```
public class Student {
    String name;
    HashMap<Course, Integer> courseList;
    DiplomaProject diplomaProject;
}
```

Each author is a Student, that has a name, a diploma project they participate in, and a list of courses they have taken in their studies (with an associated numerical grade, between 0 and 5, where 0 is failed).

```
public class Course {
    String courseName;
    String courseId;
}
```

The task is, starting from a collection of **diploma projects** and using streams:
Get a list (ArrayList) of diploma project titles, from those students that have taken a given course (identified by courseId, for example "pg4200"), and that have passed that course with at least a grade of 2 (see numerical grade mentioned above). Since several students may contribute to a given diploma project, make sure each title only appears once.

Exercise 5:
File: Ex05.java

Write a compression algorithm for a catalog of archival documents. The idea is that there is a large amount of documents stored in archives throughout Norway, and a central repository is needed to keep track of it.

There are 7 possible archives, for this exercise:
NationalArchive – NAT
OsloArchive – OSL
BergenArhive – BER
KristiansandArchive – KRS
TrondheimArchive – TRO
TromsoArchive – TRM
MaritimeArchive - MAR

For each document you have:
<Date> - the date at which the document was written.
The date is in the format: year – month – day. Where the month is written as a 3-letter code: (JAN, FEB, MAR, etc.).
<Archive> - which archive stores the document (Can be one of 7 archives, for this exercise, see above).
Location – where the document is stored. Consists of:
<ArchiveCode> - a 3 letter code that identifies the archive where the document is stored. (see the list of archives above).
<Building> - a 2 digit code that identifies which building stores the document. (NOTE: for this exercise, no archive has more than 14 buildings, numbered 1 to 15).
<RoomNumber> - the number of the room. We can safely assume that no building has more than 1023 rooms.

<Shelf> - the number of the shelf. We can safely assume that no room has more than 2047 shelves.

Each document is described by a separate line with the format:
<Date>; <Archive>; <ArchiveCode>; <Building>; <RoomNumber>; <Shelf>;

Example inputs:
1815-JUN-18; OsloArchive; OSL; 13; 42; 1500;
1805-DEC-02; NationalArchive; NAT; 07; 84; 1780;
1805-OCT-21; MaritimeArchive; MAR; 03; 126; 1111;
1814-MAY-17; NationalArchive; NAT; 01; 01; 0001;
1905-JUN-07; NationalArchive; NAT; 02; 42; 0042;

You **MUST** come up with a compression algorithm that is **customized** for this problem (i.e. do not use Huffman or LZW, but rather use *DnaCompressor* and the exercises in class as inspiration).

In your implementation, you can use the classes BitWriter and BitReader in package *org.pg4200.les11*;

NOTE: You must provide both a **compression** and a **decompression** function. If one applies the compression function to some data, followed by the decompression function, the results should be identical to the original data (i.e. the compression is to be lossless).

The algorithm you propose should be **efficient**. You should strive to compress the data as much as you can. At a minimum, the size of the compressed data should be less than the size of the decompressed data.