Mastery I — Data Structures (T. III/21–22)

Directions:

- This mastery examination starts at 2pm on Thursday June 2, 2022. You have until 5pm of the same day to complete the following *four* problems. Each problem has been broken down into multiple goals. Each goal will be graded on an all-or-nothing basis by a machine grader. No collaboration of any kind whatsoever is permitted during the exam.
- You are expected to use an IDE.
- **WHAT IS PERMITTED:** The exam is open- book, notes, Internet, Google, Stack Overflow, etc. The Internet can only be used in *read-only* mode.
- **WHAT IS *NOT* PERMITTED:** Communication/collaboration of any kind. Obviously, asking a question online is strictly *not* allowed.
- We're providing a starter package. The fact that you're reading this PDF means you have successfully downloaded the starter pack.
- At the top level, the starter package contains 4 folders, one for each problem. We're supplying a `build.gradle` suitable for each problem. For this reason, it is recommended that you use one IntelliJ project for each problem.
  To save you some typing, we're supplying some JUnit tests and driver code in the starter package. Passing these tests doesn't mean you will pass our further testing. Failing them, however, means you will not pass the real one.
- To submit your work, run `gradle clean` for each problem, and zip the folders for all problems as one zip file called `mastery1.zip` and upload it to Canvas.

Problem 1: Triple Vowels (10 points)

The letters `a`, `e`, `i`, `o`, and `u` are vowels in English. In this problem, you'll implement a function that detects the presence of three consecutive vowels—that is to say, three consecutive letters that are vowels.

Inside a public class `TripleVowels`, write a static method

```
public static boolean hasTripleVowels(String st)
```

that takes as input a `String` and returns a boolean value indicating whether the input string contains three *consecutive* letters that are vowels. The input string may have both upper- and lower- cased letters. For example:

```
hasTripleVowels("OoO") // => true
hasTripleVowels("baZaa") // => false
hasTripleVowels("fooA") // => true
hasTripleVowels("moraiene") // => true
```

Goals: There are 4 goals:

1. The method correctly detects the condition when the occurrence of triple vowels is neither at the start nor the end—and the input string contains only lower-cased letters.
2. The method correctly detects the condition when the occurrence of triple vowels is neither at the start nor the end—and the input string contains mixed-case letters.
3. The method correctly detects the condition when the input contains only lower-cased letters.
4. The method correctly detects the condition on all possible inputs.

Problem 2: Pan Digit Super Power (10 points)

A number $n \geq 1$ is a *super power* if the 7-th power of n contains every digit from 0 to 9 at least once—that is, the value of n^7 contains each of $0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9. The smallest super power number is 56, where 56^7 is 1727094849536, which contains all the digits from 0 to 9. If you're curious, the first couple of super power numbers are:

$$56, 118, 126, 139, 148, 171, 177, 184, 212, 216, 222, 231, 251, \ldots$$

Into a public class called `SuperPower`, you are to implement two static methods, though you can write as many helper functions as you see fit.

1. Implement a function **public static int** `numSPUpTo(`**int** `n)` that counts the number of super power numbers that are at most (≤) `n`. Note that `n` is included in the range as well.

2. Implement a function **public static int** `kthSP(`**int** `k)` that returns the `k`-th smallest super power number, where we use the convention that the first (i.e., $k = 1$) super power is 56. As another example, the 7-th super power number is 177.

(*Hints:* Raising a number to the 7-th power will quickly overflow the capacity of an **int** or even a **long**. But you already know what to do.)

Grading: There are 4 goals:

1. Correctly implement `numSPUpTo` for $n \leq 500$. The function returns within 2 seconds.
2. Correctly implement `numSPUpTo` for $n \leq 100,000$. The function returns within 2 seconds.
3. Correctly implement `kthSP` for $n \leq 500$. The function returns within 2 seconds.
4. Correctly implement `kthSP` for $n \leq 100,000$. The function returns within 2 seconds.

Problem 3: Books and Bookshelves (10 points)

You are to implement two classes: `Book` and `Bookshelf`. An instance of `Book` represents a book and an instance of `Bookshelf` represents a bookshelf. In more detail: A book (in this problem) has a title, a book code, and a page count. A bookshelf houses books. Translating this into Java, this means: the `Book` class has the following constructors/methods:

```
// constructor for Book with a title, a book code, and a page count
public Book(String title, String bookCode, int pageCount)

// returns the title
public String getTitle()

// returns the book code
public String getBookCode()

// returns the page count
public int getPageCount()
```

The instance variables for this class must *all* be private. Furthermore, the `Bookshelf` class implements the `IBookshelf` interface and has the following constructors/methods:

```
// a default constructor with no input parameters
public Bookshelf()

// adds a book to the shelf
public void add(Book book)

// returns how many books the shelf houses
public int size()

// returns the Book object of any book whose book code matches the given book code
// if the shelf has no matching books, return null.
public Book getBookByBookCode(String bookCode)

// returns the total page count of all books matching the predicate (see more below)
public int totalPageCountMatching(Predicate<Book> p)
```

Importantly, your `Bookshelf` class can contain exactly one instance variable, declared as

```
  private List<Book> shelvedBooks;
```

(Do not change this line or add any more instance variables.)

Notice that `totalPageCountMatching(Predicate<Book> p)` takes in a predicate "higher-order function" as input. We wish for this method to return the total page count of the books for which the predicate `p` returns true. See `java.util.function.Predicate`'s documentation for details.

Ground Rules: The only two files you can modify here (aside from creating/writing tests) are `Book.java` and `Bookshelf.java`.

Grading: There are 4 goals:

1. Correctly implement the `Book` class.
2. Correctly implement `add` and `size` in the `Bookshelf` class.
3. Correctly implement `getBookByBookCode` in the `Bookshelf` class.
4. Correctly implement `totalPageCountMatching` in the `Bookshelf` class.

Problem 4: Row-by-Row Iterator (10 points)

Given a two-dimensional structure (example below), a row-by-row iterator (aka. natural Pusheen walker) yields elements from the first row from left to right, then proceed to the second row, so on so forth. As an example, the 2d structure on the left results in the sequence on the right. Observe that the empty rows—which can appear anywhere, including at the start and/or the end—are skipped.

```
List<List<Integer>> twoD = List.of(
    List.of(3, 7),
    List.of(),
    List.of(1),
    List.of(5, 0, 2)
)
```

⇒ 3, 7, 1, 5, 0, 2

You will implement a public class

```
public class PusheenWalker<T> implements Iterable<T> { ... }
```

with the following specifications:

- The only constructor takes as input a `List<List<T>>` representing the said 2D structure. As an example, this lets us write `new PusheenWalker<>(twoD)` and the result has type `PusheenWalker<Integer>`.
- The iterator given by this class—as described earlier—yields elements from left to right, then onto the following row, skipping each empty row, until all the elements have been returned. There may be many consecutive empty rows.

Ground Rules: Your code cannot store a collection of any kind, aside from a reference to the input structure. This means, for example, that you *cannot* make an output list ahead of time and simply return from that list. Moreover, do *not* use Java streams.

Grading: There are 5 goals:

1. The code has some legit implementation and compiles clean.
2. The iterator works correctly on inputs involving just one non-empty row of data.
3. The iterator works correctly on inputs involving multiple rows, each with the same non-zero length.
4. The iterator works correctly on inputs involving multiple rows of varying non-empty lengths
5. The iterator works correctly on all possible inputs.