Java Interim Test

Tuesday 20th March 2018

10:00 – 13:00

THREE HOURS

(including 10 minutes planning time)

- Please make your swipe card visible on your desk throughout the test.
- After the planning time, log in using your username as **both** your username and password.

There are **TWO** sections: Section A and Section B, each worth 50 marks.

Credit will be awarded throughout for code that successfully compiles, which is clear, concise, usefully commented, and has any pre-conditions expressed with appropriate assertions.

Important note:

- In each section, the tasks are in increasing order of difficulty. Manage your time so that you attempt both sections. You may wish to solve the easier tasks in both sections before moving on to the harder tasks
- It is critical that your solution compiles for automated testing to be effective. Up to **TEN MARKS** will be deducted from solutions that do not compile (-5 marks per Section). Comment out any code that does not compile before you log out.
- You can use the terminal or an IDE like IDEA to compile and run your code. **Do not ask an invigilator for help on how to use an IDE.**
- Before you log out at the end of the test, you **must** ensure that your source code is in the correct directory otherwise your marks can suffer heavy penalties. Only code in the original directories provided will be checked.

Section A

Problem Description

Your task is to write some methods and functions that generalise multiplication of square matrices. You are familiar with matrix addition and multiplication for numeric matrices. For example, we can add and multiply a pair of 2x2 matrices as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \times e + b \times g & a \times f + b \times h \\ c \times e + d \times g & c \times f + d \times h \end{bmatrix}$$

More generally, suppose that T is some type and that we have two binary operators $\mathsf{sum} : T \to T \to T$ and $\mathsf{prod} : T \to T \to T$. If A and B are $N \times N$ matrices with entries of type T then we can "add" A and B just as we would add two numerical matrices, applying the binary operator sum wherever we would usually apply numeric addition, $+$. Similarly, we can "multiply" A and B by following the standard procedure for matrix multiplication, but applying sum and prod wherever we would apply numerical $+$ and $\times$, respectively.

Example: Let us define the sum of two strings to be their concatenation, so that $\mathsf{sum}(\texttt{cat}, \texttt{dog}) = \texttt{catdog}$, and the product of two strings to be the first string, followed by `!`, followed by the second string, so that $\mathsf{prod}(\texttt{cat}, \texttt{dog}) = \texttt{cat!dog}$. We can use these operators to add and multiply string matrices as illustrated by the following examples:

$$\begin{bmatrix} \texttt{hi} & \texttt{fi} \\ \texttt{wi} & \texttt{fi} \end{bmatrix} + \begin{bmatrix} \texttt{si} & \texttt{fi} \\ \texttt{py} & \texttt{py} \end{bmatrix} = \begin{bmatrix} \texttt{hisi} & \texttt{fifi} \\ \texttt{wipy} & \texttt{fipy} \end{bmatrix}$$

$$\begin{bmatrix} \texttt{hi} & \texttt{fi} \\ \texttt{wi} & \texttt{fi} \end{bmatrix} \times \begin{bmatrix} \texttt{si} & \texttt{fi} \\ \texttt{py} & \texttt{py} \end{bmatrix} = \begin{bmatrix} \texttt{hi!sifi!py} & \texttt{hi!fifi!py} \\ \texttt{wi!sifi!py} & \texttt{wi!fifi!py} \end{bmatrix}$$

You will write a number of classes and methods that implement generalised matrices and various operations on them.

Getting Started

The skeleton files are located in the `generalmatrices` package and its sub-packages. This is located in your Lexis home 'smmtbs' directory at:

- `~/smmtbs/SectionA/src/generalmatrices`

During the test you will need to make use of (though should **NOT** modify) the following:

- `generalmatrices/pair/Pair.java`: an implementation of an integer-integer pair
- `generalmatrices/operators/RingElement.java`: a generic interface describing abstract "sum" and "product" operators

You should not modify these files in any way: auto-testing of your solution depends on the files having exactly their original contents.

During the test, you will populate the following:

- `generalmatrices/matrix/Matrix.java`: an empty class, which you will fill to represent a generic matrix, and then equip with "sum" and "product" operators
- `generalmatrices/pair/PairWithOperators.java`: an empty class, which you will fill to represent an integer-integer pair equipped with specific "sum" and "product" operators
- `generalmatrices/examples/Example.java`: a class containing an empty static method, which you will implement

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed in the `generalmatrices` package or one of its sub-packages.

Testing

The `generalmatrices/Tests.java` class, which will not be marked, contains four JUnit tests to help you gauge your progress during Section A. The tests in this file are commented out initially. **As you progress through the exercise you should uncomment the test associated with each question in order to test your work.**

You can also add extra tests in this file as you see fit, to help you develop your solution.

What to do

1. **Populating the `Matrix` class.**

   The `Matrix` class, in `generalmatrices/matrix/Matrix.java`, is generic with respect to some type `T`. The class body is initially empty.

   Your first task is to populate this class so that it represents an $N \times N$ square matrix of elements of type T, which we say has *order* N. It is up to you how to represent the matrix internally, but you should follow good design principles and maximise encapsulation. You should also take steps to make `Matrix` *immutable*.

   `Matrix` should have a single constructor that takes a list of elements of type `T`. You may assume that the size of this list is a perfect square.

   If the list is empty, an `IllegalArgumentException` (an *unchecked* exception) should be thrown. Otherwise, the new matrix should have order $\sqrt{k}$, and the first $\sqrt{k}$ elements in the list should correspond to the first row of the matrix, the next $\sqrt{k}$ elements to the second row of the matrix, etc. You may find the `Math.sqrt` method useful in implementing this constructor.

   `Matrix` should have the following public instance methods:

   - `T get(int row, int col)`: returns the element at row `row` and column `col`, which you can assume are valid rows and columns of the matrix.

- `int getOrder()`: returns the order of the matrix.
- `String toString()`: overrides the `toString()` method from `Object`, to turn a `Matrix` into a string *exactly* as follows:
  - the matrix string should start with "`[`" and end with "`]`";
  - each row should have the form "`[`*row contents*`]`";
  - the contents of a row should consist of string representations of the row elements, in order, each separated by a space;
  - except for white-space that might appear in the string representations of matrix elements, the spaces that separate elements should be the only white-space that occurs in the string representation of the matrix.

  For example, the integer matrix written mathematically as:

$$\begin{bmatrix} 1 & 20 \\ 300 & 4000 \end{bmatrix}$$

  should have string representation `[[1 20][300 4000]]`.

Un-comment the `testQuestion1()` test in `Tests.java` and and debug your solution until this test passes.

[15 marks]

2. **Implementing a pair with operators.**

The `Pair` class in `generalmatrices/pair/Pair.java` implements a simple integer-integer pair.
The `RingElement` interface in `generalmatrices/operators/RingElement.java` is generic with respect to a type `T`, and specifies that implementing classes should provide methods `sum` and `product`, each of which takes an argument of type `T` and returns a result of type `T`. The idea is that if a class implements `RingElement<T>` then it should be possible to add and multiply members of the class with elements of type `T`.[1]

Adapt the (initially empty) `PairWithOperators` class in the `generalmatrices.pair` package so that it extends `Pair` without adding any additional fields, and also implements `RingElement` in such a way that the `sum` and `product` methods have the following signatures and behaviour:

- `PairWithOperators sum(PairWithOperators other)`:
  returns a `PairWithOperators` whose x coordinate is the sum of the x coords of the target object and `other`, and whose y coordinate is similarly computed.
- `PairWithOperators product(PairWithOperators other)`:
  returns a `PairWithOperators` whose x coordinate is the product of the x coords of the target object and `other`, and whose y coordinate is similarly computed.

Un-comment the `testQuestion2()` test in `Tests.java` and debug your solution until this test also passes.

[1]The name "ring element" comes from the fact that abstract mathematical objects called rings support addition and multiplication, generalising the integers.

[10 marks]

3. **Abstract addition and multiplication of matrices.**

   Your task is now to add `sum` and `product` methods to the `Matrix<T>` class from Question 1, so that it is possible to add and multiply generic matrices.

   The problem is that we cannot directly add or multiply two matrices of type `Matrix<T>` without knowing how to add and multiply elements of type `T`. However, as discussed in the *Problem Description* above, we can define matrix addition and multiplication if we are provided with *binary operators* that describe addition and multiplication for `T`.

   Recall that the `BinaryOperator<T>` interface, from package `java.util.function`, specifies a single (non-default) method:

   - `T apply(T first, T second);`

   The `apply` method can be invoked to apply the binary operator to two given arguments, returning the associated result.

   Use this interface to add two additional public methods to `Matrix<T>`:

   - `Matrix<T> sum(Matrix<T> other, BinaryOperator<T> elementSum)`:
     given a matrix `other`, which you can assume has the same order as the target matrix, and a binary operator `elementSum`, returns the sum of the target matrix and `other` (in that order). Matrix elements should be added using `elementSum`.
   - `Matrix<T> product(Matrix<T> other, BinaryOperator<T> elementSum, BinaryOperator<T> elementProduct)`:
     given a matrix `other`, which you can assume has the same order as the target matrix, and a binary operators `elementSum` and `elementProduct`, returns the product of the target matrix and `other` (in that order). In computing the product, matrix elements should be multiplied using `elementProduct` and added using `elementSum`.

   Un-comment `testQuestion3()` in `Tests.java` and debug your solution until this test also passes.

   [15 marks]

4. **Implementing an example method.**

   The class `Example` in `generalmatrices/examples/Example.java` provides the following un-implemented method:

   - `Matrix<PairWithOperators> multiplyPairMatrices( List<Matrix<PairWithOperators>> matrices)`:
     takes a non-empty list of matrices whose entries have type `PairWithOperators`. Returns the product of all these matrices, in left-to-right order, where in each matrix product operation the `sum` and `product` methods of `PairWithOperators` should be used to add and multiply matrix elements.

   Your task is to implement the method to provide this functionality. Your implementation should reuse code that you have written in solving Questions 1–3 as much as possible: only minimal credit will be given for a "from scratch" solution.

Consider using lambdas and/or method references to express your solutions in a concise manner.

Un-comment the `testQuestion4()` test in `Tests.java` and debug your solution until this test also passes.

[10 marks]

Total for Section A: 50 marks

Useful commands

```
cd ~/smmtbs/SectionA

mkdir out

javac -g -d out
-cp /usr/share/java/junit4-4.12.jar
-sourcepath src:test
src/generalmatrices/**/*.java test/generalmatrices/Tests.java

java
-cp /usr/share/java/junit4-4.12.jar:/usr/share/java/hamcrest-core-1.3.jar:out
org.junit.runner.JUnitCore generalmatrices.Tests
```

Section B

You are required to implement a tree-based data structure representing a set of words in a memory-efficient way, while preserving their order. Each word can have an arbitrary length, but can be composed only of the 26 lower-case English characters (a..z). Being a set, your data structure will not allow duplicates.

The tree-based structure you are required to implement is a simplified version of a prefix tree, which we will call a `CompactWordsSet`.

In a `CompactWordsSet`, every node is labeled with a char and has 26 children nodes, each labeled with a char between 'a' and 'z'. The root node is labeled by an empty char (“”), which is not part of the valid alphabet. A word is represented by the concatenation of the chars labeling the nodes along a path from the root (whose label is skipped because invalid) to one of the tree's nodes.

To distinguish between words that belong to the set and their prefixes, each node has a flag `isWord`, which is true if the node corresponds to the last char of words that has been added to the set, and false otherwise.

For example, the `CompactWordsSet` in Figure 1 contains the words “ae”, “az”, “e”, “ef”, but not “a”.

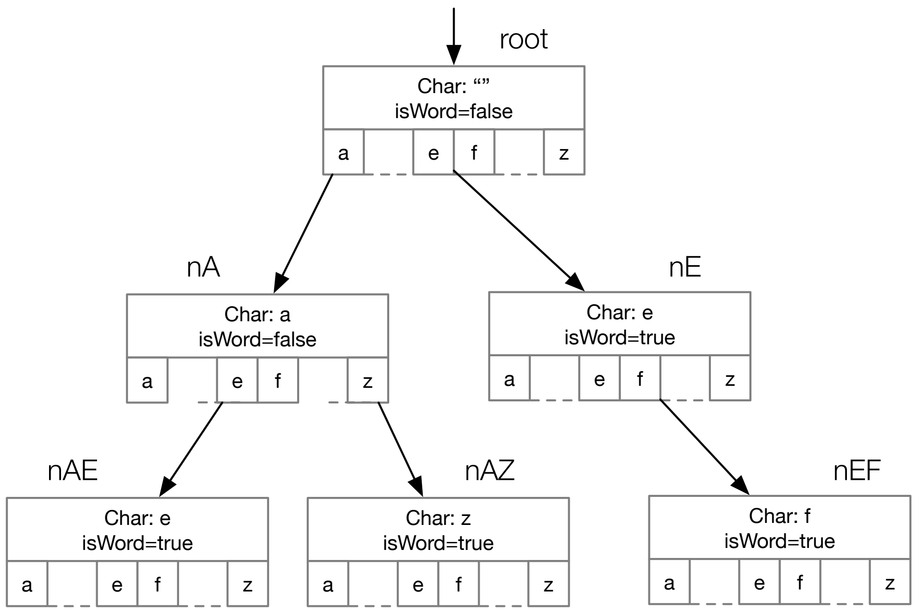

Figure 1: `CompactWordsSet` containing the words: “ae”, “az”, “e”, “ef”.

Overview of `CompactWordsSet`'s operations

Search for a word. The first char of the word indicates which child node of the root to followed next (if not null). Then, the second char of the word indicates the next node to be followed from the root's child, and so on.

For example, to find the word “ae” in Figure 1, the search begins in the root node. Because the first char is “a”, the first child of the root, nA, is visited. Similarly, from this node, the child corresponding to the char “e”, nAE, is visited. At this point, all the chars of the searched word “ae” have been matched, reaching nAE. Because in nAE the flag `isWord` is true, the word “ae” belongs to the set.

Hint: the second char of a string is the first char of its substring starting from position 1.

Add a word. Adding a word into a `CompactWordsSet` requires traversing the chain of nodes that match the chars of the new word from the root down to the node matching the last char of the word; then setting the flag `isWord` of this latter node.

For example, adding the word “aeaz” would require to traverse the tree from the root to nA and then to nAE, to create a new child of nAE matching the char “a” (say nAEA), and finally a new child of nAEA that matches the char “z” (nAEAZ) with the flag `isWord` true.

Remove a word. For this test *only logical removal is required.* This means that if the word belongs to the `CompactWordsSet` the flag `isWord` of the node matching its last char at the end of the search procedure should be set to false. No physical disconnection of the nodes is required.

Hints on chars and substrings

`char` in Java is a primitive type representing an ASCII symbol, like “a”, “b”, “?” or others not printable. Chars correspond also to numerical values. In particular, for the lower-case English alphabet char “a” has numerical value 97, char “b” has numerical value 98, and so on until “z” that has numerical value “122”.

To see the numerical value of a char, one can runs a code similar to the following:

```
int c = 'z';
System.out.println(c);
```

Chars have numerical values, positions in arrays and lists are also numerical values.

Hint: Another method that may come handy is `String.substring`.

Getting Started

The skeleton files are located in the `src` folder of the project located in your Lexis home ‘smmtbs’ directory at:

- `~/smmtbs/SectionB/`

The source files are located in `src` which contains a `collections` subfolder with the source code of the classes you are going to use or implement and a `test` subfolder with a set of JUnit test classes you can use to check your implementation and to gain additional clues about what it is expected to do.

During the test you will need to make use of (though should **NOT** modify) the following:

- `CompactWordsSet`: all the method signatures must remain unchanged.
- `InvalidWordException`: you can add constructors if needed, but cannot remove or rename the class.

- `SimpleCompactWordTree`: cannot change its name, nor the fact that it implements `CompactWordsSet`. Apart from this, you can add any method/field you see fit.

During the implementation, you will populate and/or modify `SimpleCompactWordTree`.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed (in a suitable package) in the `src` or `test` directories, depending if it is part of the implementation or of its tests, respectively.

Testing

The `test` folder, which will not be marked, contains some JUnit tests to help you check your progress during Section B.

`CompactWordsSetTest` contains a set of functional tests to help you find problems in the behavior of your `SimpleCompactWordTree` implementation.

`StressTest` contains two test cases to help you identify possible race-conditions in your implementation (`stressTestWithManyOperations`) or to get an idea about how the performance of your implementation changes when multiple threads access it (`performanceTest`).

You can also add extra tests in this file as you see fit, to help you develop your solution.

If you decide you will not implement some methods, you may want to remove the statements and the assertions where these methods are used.

Remember that tests can find bugs, not exclude their presence :)

What to do

General observations. Most of the methods you are required to implement can be implemented either iteratively or recursively. You are free to choose your preferred way, though recursively is probably more compact and easier to design in most cases. As a suggestion, prefer simplicity over time or memory efficiency for these operations. Any correct, complete, well-designed, and non-redundant implementation can achieve the maximum score. Extreme improvements of time or memory performance are not required to achieve the maximum score, so follow good design principles and common sense, and keep your code readable and easy to fix.

Important: you are required to implement the tree-based data structure `SimpleCompactWordTree` and its operations, as described before. Builtin Java collection can be used only as auxiliary data structures, for example the children of a node can be stored in an array, and ArrayList, or and AtomicReferenceArray, but cannot be used as backend implementation of `SimpleCompactWordTree`.

1. **Implement the static method `checkIfWordIsValid` of the `CompactWordsSet` interface.** This method returns void, but throws an `InvalidWordException` if:
    - the input string is null
    - the input string is empty (i.e., has length 0)

- the input string contains any char that is not in the range "a"-"z" (the methods `String.chars` or `String.charAt` may come handy)

This method should be called to check the input of `add`, `contains`, `remove` (as in the skeleton source code).

[**3 marks**]

2. **Implement the methods `add`, `contains`, `size` of the `SimpleCompactWordTree` class.** The methods should implement the behavior described previously in this document.

   As customary for sets in Java, `add` returns true if the collection has been modified (i.e., the new word was not already in the collection), and false otherwise.

   Contains returns true if the word is contained in the set, false otherwise.

   Size returns the number of words currently in the set. You are free to decide whether to add a field to the `SimpleCompactWordTree` class to keep track of the number of elements in the collection (recommended in general, but especially if you plan to make the collection thread-safe later) or if you prefer to traverse the tree every time and count the number of words.

   See also the test suite `CompactWordsSetTest` for additional examples of expected behaviors.

   [**17 marks**]

3. **Implement the method `remove`, and, if needed, update the implementation of the method `size` of the `SimpleCompactWordTree` class.** The methods should implement the behavior described previously in this document for removing a word from the collection.

   As customary for sets in Java, `remove` returns true if the collection has been modified (i.e., the word was in the collection and has been removed), and false otherwise.

   Remember that only logical removal is required for this test. You are free to implement also a physical removal (disconnecting the removed nodes), but this is not required to achieve the maximum mark.

   If needed, update your implementation of the method `size` to consistently return the number of words contained in the `SimpleCompactWordTree` collection.

   See also the test suite `CompactWordsSetTest` for additional examples of expected behaviors.

   [**6 marks**]

4. **Implement the method `uniqueWordsInAlphabeticOrder` of the `SimpleCompactWordTree` class.**

   This method should traverse the tree and collect all the words in a list of strings to be returned as result. The words in the list should be in alphabetic order, but you are not allowed to use any sorting method: the order should result from the order in which the nodes in the tree are traversed.

   An implementation that correctly returns all the words contained in the collection, but not in alphabetic order can achieve up to 6 marks out of 10.

[10 marks]

5. **Make your `SimpleCompactWordTree` thread-safe.**

   The methods `add`, `remove`, `contains`, `size` should be safe. `uniqueWordsInAlphabeticOrder` can return a possibly partially consistent snapshot of the collection. In other words, there is no need to block the entire collection while `uniqueWordsInAlphabeticOrder` collects the words from the tree: if a word is added or deleted by another thread during the time `uniqueWordsInAlphabeticOrder` executes, this word may or may not appear in the list returned by `uniqueWordsInAlphabeticOrder`.

   You can use any synchronization strategy you see fit. A correct coarse-grained synchronization is worth half of the marks. *Any* correct and appropriate finer-grained synchronization can obtain the maximum mark.

   If you are running out of time, feel free to add a comment at the beginning of `SimpleCompactWordTree.java` to succinctly explain what kind of fine-grained synchronization strategy you did not manage to complete, how you would have implemented it (as clearly and precisely as possible), and why and, if applicable, under which assumptions it should perform well. If you manage to fully implement your synchronization strategy, there is no need for additional comments.

   The test suites `stressTestWithManyOperations` and `performanceTest` may help you find possible race conditions. It is not necessary that `performanceTest` shows a significant performance improvement; it is only intended to help yourself assess the performance and correctness of your implementation.

   [14 marks]

Total for Section B: 50 marks

Useful commands

```
cd ~/smmtbs/SectionB

mkdir out

javac -g -d out
-cp /usr/share/java/junit4-4.12.jar
-sourcepath src:test src/collections/**/*.java test/*.java test/**/*.java

java
-cp /usr/share/java/junit4-4.12.jar:/usr/share/java/hamcrest-core-1.3.jar:out
org.junit.runner.JUnitCore StressTest

java
-cp /usr/share/java/junit4-4.12.jar:/usr/share/java/hamcrest-core-1.3.jar:out
org.junit.runner.JUnitCore collections.CompactWordsSetTest
```

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning