Java Interim Test

Imperial College London

Department of Computing

Java Interim Test

Friday 17 March 2023
14:00 to 16:00
TWO HOURS
(including 10 minutes planning time)

- There are **TWO** parts: Section A and Section B, each worth 50 marks.
- The maximum total is **100 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:** 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.
- 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.
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided. No additional marks are awarded for extra tests.
- You can use the terminal or an IDE like IDEA to compile and run your code. **You are advised to create a separate project for each Section**. Do not ask for help on how to use an IDE.
- The files are in your Home folder, under the "`javainterimtestparta`" and "`javainterimtestpartb`" subdirectories. **Do not move any files**, or the test engine will fail resulting in a compilation penalty.
- When you are finished, simply **save everything and log out**. **Do not shut down your machine**. We will fetch your files from the local storage.

Useful commands

```
cd ~/javainterimtestparta/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore generalmatrices.Question#Tests

java
-ea
-cp "./lib/*:out"
generalmatrices.TestSuiteRunner



cd ~/javainterimtestpartb/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/*.java test/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore
ListArrayBasedTest
PriorityQueueTest
NineTailsWeightedGraphTest

java
-ea
-cp "./lib/*:out"
TestSuiteRunner
```

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, a pair of 2x2 matrices can be added and multiplied 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 class and methods that implement generalised matrices and operations on them.

Getting Started

The skeleton files are located in the `generalmatrices` package. This is located in your Lexis home directory at:

- `~/javainterimtestparta/src/generalmatrices`

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

- `Pair.java`: an implementation of an integer-integer pair

You should not modify this file in any way: auto-testing of your solution depends on the file having exactly its original contents.

During the test, you will populate the following:

- `Matrix.java`: an empty class, which you will fill to represent a generic matrix, and then equip with "sum" and "product" operators

- `ExampleMethods.java`: a class containing empty static methods, which you will implement to provide specific matrix-related operations

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.

Testing

You are provided with a set of test classes under:

- `~/javainterimtestparta/test/generalmatrices`

There is a test class `Question`*i*`Tests.java` for each question i. These contain initially commented-out JUnit tests to help you gauge your progress during Section A. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.**

These tests are not exhaustive and are merely intended to guide you: while it is good if your solution passes these tests, your work will be assessed with respect to a larger, more thorough test suite. You should thus think carefully about whether your solution is complete, even if you pass all of the given tests.

What to do

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

   The `Matrix` class, in `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 take steps to make `Matrix` *immutable*.

   You should ensure that it is impossible to create subclasses of `Matrix`.

   `Matrix` should have a single constructor that takes a list of elements of type `T`. If the list is empty, or if the size k of the list is not a perfect square, 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 as *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]]`.

Test your solution using (at least) the tests in `Question1Tests`.

[14 marks]

2. **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`.

Test your solution using (at least) the tests in `Question2Tests`.

[20 marks]

3. **Implementing some example methods.**

Note: at this point you might want to consider getting started on Section B if you have not already, to ensure that you secure marks for the easier parts of Section B, returning to complete Section A later.

The class `ExampleMethods` in `ExampleMethods.java` provides signatures and for two methods that you should implement, to provide some specific matrix operations.

In implementing these methods you should, wherever possible, reuse the code that you have written in solving Questions 1–2. Only minimal credit will be given for "from scratch" solutions that implement these algorithms without reusing your previous solutions.

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

The methods are as follows:

- `Matrix<Matrix<Integer>> multiplyNestedMatrices(Matrix<Matrix<Integer>> first, Matrix<Matrix<Integer>> second)`:
  takes two matrices of integer matrices, which you can assume have the same order, N say, and such that every sub-matrix of each matrix has the same order, M say. Returns the product of `first` and `second`, in that order, where inner matrices are added and multiplied using standard integer matrix addition and multiplication.
- `Matrix<Pair> multiplyPairMatrices(List<Matrix<Pair>> matrices)`:
  takes a non-empty list of matrices whose entries have type `Pair`, which you can assume all have the same order. Returns the product of all these matrices, where the sum of pairs (x_1, y_1) and (x_2, y_2) is defined to be $(x_1 + x_2, y_1 + y_2)$, and the product of pairs (x_1, y_1) and (x_2, y_2) is defined to be $(x_1 \times x_2, y_1 \times y_2)$.

Test your solution using (at least) the tests in `Question3Tests`.

[10 marks]

4. **Matrix equality.**

   Override the `equals` method of `Object` in the `Matrix` class so that two matrices are deemed equal if and only if they have the same order and they are element-wise equal. Take care to ensure that you satisfy all requirements associated with overriding `equals` on a Java class.

   Test your solution using (at least) the tests in `Question4Tests`.

   [6 marks]

Total for Section A: 50 marks

Section B

Problem Description

Many problems can be solved using graphs and graph algorithms. The *Weighted Nine Tails Problem* is one example. Nine coins are placed in a $3x3$ matrix configuration with some face up (heads) and some face down (tails). The aim is to find the minimum number of flips that leads to all coins being face down (the **target configuration**). A legal move takes a face-up coin and flips it over together with the coins immediately adjacent (but not diagonal to it) regardless of whether they are face up or down. The **weight** (or cost) of a move is the number of coins that have been flipped in that move. A solution is a sequence of legal moves leading from a given configuration to the target configuration. An optimal solution is one where the total number of flips is the smallest.

For example, Figure 1 shows the optimal solution from the given configuration to the target configuration. The first move starts by flipping the coin in the bottom left which causes both the coin immediately above it and the coin immediately to its right to flip. The result is shown in *b*. Now flipping the coin in the bottom right corner of *b* causes both the coin immediately above it and the coin immediately to its left to flip. The result of this is *c*. Now flipping the top-left coin and its two adjacent coins attains the goal of all coins being tails up. The three moves each results in three coins being flipped (a weight of 3), giving a total weight of 9.

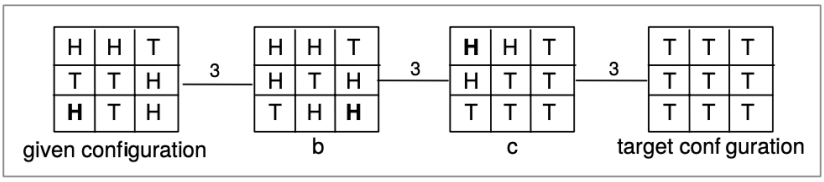

Figure 1: Configuration

The *Weighted Nine Tails Problem* is interesting because it can be reduced to solving the more general problem of finding the cheapest path (the optimal solution) between two vertices in an edge-weighted graph.

You need to complete the provided code that solves the *Weighted Nine Tails Problem* described above. The program:

1. constructs a nine tails **weighted graph** that stores all possible configurations and legal moves with their respective weights (the method `constructGraph` of the class `NineTailsWeightedGraph`);

2. generates a **minimum spanning tree** from the *target configuration* to all the configurations (vertices) in the graph. This step also computes the weight of each path, that is the number of flips needed from the vertex at the end of the path back to the target configuration (the `constructMinimumSpanningTree` method of class `NineTailsWeightedGraph`);

3. prompts the user for a starting configuration as a `String` of 9 'H' and 'T' characters;

4. finds (using the minimum spanning tree) the **cheapest path** from the user entered starting configuration to the target configuration, then prints this path together with the total number of flips needed (see the `printShortestPath` method of the class `NineTailsWeightedGraph`).

In a **nine tails weighted graph**, vertices are indexed from 1 to 512, as there are 512 (2^9) possible configurations of 9 coins. Given any configuration, it is possible to generate its index as an integer; and, given any index, it is possible to generate its configuration (methods `configurationToIndex` and `indexToConfiguration` respectively).

Edges are instances of an inner class `WeightedEdge`. An edge stores links between two configurations, the parent and child, and the weight of the move from parent to child. The parent is the configuration to which the legal move is applied, while the child is the configuration resulting from that move. For instance, from Figure 1, consider the edge between configuration c (whose index is 96) and the target configuration (whose index is 512). Since the legal move is applied to configuration c, the link between these two configurations is represented by an edge object where the parent is 96, the child is 512 and the weight is 3 (since three flips are made in this move).

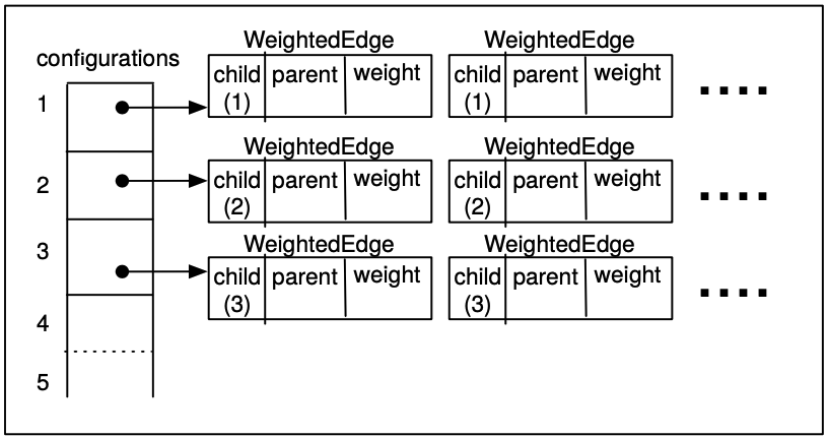

Figure 2: Visualization of the NineTailsWeightedGraph

A neat way to represent a nine tails weighted graph is as a list of coin configurations where each element in that list is a priority queue of weighted edges, prioritised by weight: the lower the weight, the higher the priority. The child of a weighted edge in the nth priority queue is always n. The size of the configurations list is the number of vertices in the graph. See Figure 2.

Minimum Spanning Tree Algorithm

The construction of the minimum spanning tree starts from the *target configuration* (the root) and computes the tree of minimum (cheapest) paths from this configuration to all the other configurations in the graph. Given the chosen implementation of the nine tails weighted graph, the minimum spanning tree can simply be stored in an array of configuration indices. The size of the array (called `nextMoves`) is equal to the number of vertices in the graph. An array index corresponds to a parent configuration index. Given an array index (parent configuration index) the value is the chosen child configuration index as the next best move.

The minimum spanning tree can be constructed using the algorithm outlined in Listing 1, where s denotes the index of the target configuration, V the set of vertices in the graph, and $w(u, v)$ denotes the weight of the edge between u and v, where u is the child configuration and v is the parent:

```
Let nextMoves be the array storing the minimum spanning tree
Let visited be the set of visited vertices in the graph
Let costs be the array of the minimum weight from a parent
    configuration to s computed so far

Initially visited contains just s
costs[s] = 0
nextMove[s] = -1 /* meaning no next move from s */

while (size of visited < size of V){
    find a triple (u,v,c) where u and v are configuration
        indices and c is a cost such that:
        1. v is a not-yet-visited parent of u
        2. c is costs[u] + w(u,v)
        3. there is no other triple (u',v',c'), with u' in the
            visited set, such that c' < c
    add v to visited
    costs[v] = c
    nextMoves[v] = u
}
```

Listing 1: constructMinimumSpanningTree pseudocode

Getting Started

The files used in Section B can be found in the SectionB directory in your Lexis home directory: `~/javainterimtestpartb/src/`

Also a UML class diagram describing the architecture of the system is provided (see Figure 3 at the end of the section).

During Section B, you will be working in the following files:

- `ListArrayBased.java`: class that implements `ListInterface`. You will need to complete a method in the class.
- `NineTailsWeightedGraph.java`: class that uses a graph for the weighted nine tails problem. This class includes inner classes. It also has auxiliary

methods for printing and for generating: legal moves, index from configuration, and configuration from index. It also includes the auxiliary method `constructGraph` for constructing a nine tail weighted graph, and `constructMinimumSpanningTree` for constructing the minimum spanning tree. You will have to implement these two methods.

- `PriorityQueue.java`: class that implements `PriorityQueueInterface`. You will need to complete two methods in this class.

You may also need to make use of (though you should **NOT** need to edit) the following:

- `WeightedNineTailsProblem.java`: this class creates a graph, reads the initial configuration of coins, generates the index of the initial configuration and prints the shortest path from this configuration to the target configuration.
- `ListIndexOutOfBoundsException.java`: this class implements the out of bounds exception for a List class.
- `PriorityQueueException.java`: this class implements the PriorityQueue exception
- `PriorityQueueInterface.java`: adds a new entry to the priority queue according to the priority value.
- `ListInterface.java`: represents a collection of abstract methods to be used by a List class.

Testing

You've been provided with a variety of tests to help you compile and test your code, located under the directory `test`. `TestSuiteRunner` currently runs a few sample tests from `ListArrayBasedTest`.

- When you finish *Task 1* you can uncomment more tests in the `ListArrayBasedTest` class.
- When you finish *Task 2* you can uncomment the tests in the `PriorityQueueTest` class. Don't forget to add `PriorityQueueTest.class` in the runClasses in the `TestSuiteRunner` class.
- When you finish *Task 3* you can uncomment the `@Before` test in the `NineTailsWeightedGraphTest` class. Don't forget to add `NineTailsWeightedGraphTest.class` in the `runClasses` in the `TestSuiteRunner` class.
- When you finish *Task 4* you can uncomment the remaining tests in the `NineTailsWeightedGraphTest` class.

Feel free to add your own tests as you work through your solution.

What to do

Your task is to complete the provided implementation. You must also carefully read the commented source files.

1. `add(int givenPosition, T newItem)` for the class `ListArrayBased<T>`. [**7 marks**]
2. `add(T newEntry)` and `priorityQueueRebuild(int root)` for the class `PriorityQueue< T extends Comparable<T> >`. [**13 marks**]
3. `constructGraph()` for the class `NineTailsWeightedGraph` that constructs a nine tails weighted graph as described above. [**12 marks**]
4. `constructMinimumSpanningTree()` for the class `NineTailsWeightedGraph` that computes the minimum spanning tree as described in the Minimum Spanning Tree algorithm (see Listing 1). [**18 marks**]

Total for Section B: 50 marks

Good Luck!

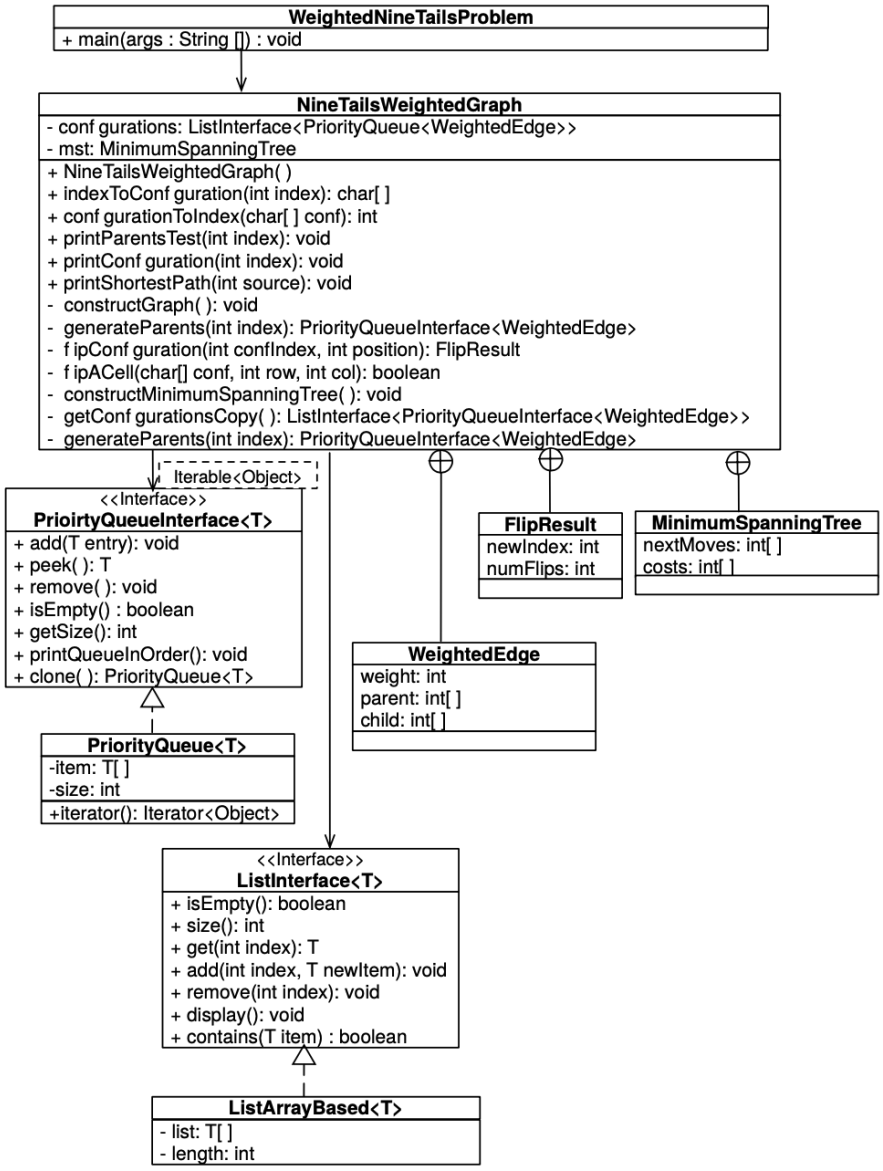

Figure 3: UML Diagram illustrating the Software Architecture