ALGORITHMICS - Bachelor of Software Engineering, 2020

LAB GUIDE. SESSION 1.1

GOALS:

- **Measure execution times**
- **Work with benchmarks**

1. Code to start working on this lab

You will now work with the `Vector1.java` class provided, together with the `MatrixOperations.java` class you developed the previous week. This way of working, **packaging classes**, has the advantage of being able to structure and use the information much better, so it will be **our way of working from now on.**

If we use the JDK with the command line, we just need to place ourselves in the folder with the class we want to compile and type:

- `javac Vector1.java`

Next, we can verify that the `Vector1.class` file appears in that same folder, doing:

- `dir`

Since the package of the class is `session1_1` we should create that folder structure and place the file in there. That is, we should create a folder called `session1_1`. Then, we can copy and paste the `Vector1.class` file there. Thus, we can type:

- `java session1_1.Vector1`
- `java session1_1.Vector1 5`
- `java session1_1.Vector1 50`
- `java session1_1.Vector1 500`
- `java session1_1.Vector1 5000`
- …

Nevertheless, for this course we will use the **Eclipse IDE**. To compile and execute the code, we use the option "**Run as ...**". To add the arguments in the execution, you must configure them in "**Run configurations ...**".

2. Measuring execution times

The idea is to perform an **empirical study** of the execution time of some programs. That way, we can determine whether they match with the theoretical behavior obtained from the **analytical study** of their time complexity.

We are going to use the Java method called `currentTimeMillis()` from the `System` class of the `java.lang` package (that is the only Java package that is preloaded, without the need for explicitly importing it using the `import` statement). The `currentTimeMillis()` method returns an integer of type `long` which is the current millisecond that the computer is living in that moment (the value of 0 has been more than 40 years ago). Put another way, it is the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. So, if we call such a method twice, at the beginning and at the end of the measurement process, and if we subtract those two values, we will get the time spent between the calls in milliseconds. Considering that, 1- how many more years can we continue using this way of counting?

If when taking times, made as explained in the previous paragraph, we get only a few milliseconds (by putting a threshold, we will consider a number less than 50 milliseconds) we will not use it for lack of reliability. The reason is that there are internal processes of the system (e.g., the so-called "garbage collector"), which are executed with a higher priority than our program (in fact, they stop it, although we may not realize that fact). The time of those system processes, in addition to what it takes to end our process, sensitively distort the time obtained in the case of low times.

CONCLUSION: We reject times below 50 milliseconds, and the longer the time the more reliable it is (3578 is more reliable than 123).

Next, we create a new class called `Vector2.java`, which is a program that should be able to measure the time of an operation (algorithm), namely the **addition of the `n` elements of a vector**, that we have implemented in the `Vector1` class. We will pass an argument in line of commands with the size of the vector as in the previous case.

2. What does it mean that the time measured is 0?
3. From what size of problem (`n`) do we start to get reliable times?

3. Grow of the problem size

It is impractical to vary the size of the problem by hand, especially if we consider that we usually want to draw a graph of the time depending on the size of the problem for an operation. So, how can we obtain the different values to draw the graph?

Create a `Vector3.java` class that will increase the size of the vector, obtaining times for each case. In this way, you will be able to follow more conveniently the evolution of the execution time.

We are going to increase the size exponentially and compare the times for different sizes of the problem.

1. What happens with time if the size of the problem is multiplied by 5?
2. Are the times obtained those that were expected from linear complexity O(n)?
3. Use a spreadsheet to draw a graph with Excel. On the X axis we can put the time and on the Y axis the size of the problem.

4. Taking small execution times (<50 ms)

When the process takes very little time, we can run the process to be measured `nTimes`, adjusting this parameter, since on the one hand, we must ensure that the total execution time exceeds the 50 milliseconds and on the other hand, the process must end in a reasonable time.

Although `nTimes` can be any value, it is advisable to test with values that are powers of 10 because the conversion of times is as easy as applying the following table:

nTimes	*Time inits:*	
1	Milliseconds	(10^{-3} sg.)
10	Tens of millisec.	(10^{-4} sg.)
100	Hundreds of millisec.	(10^{-5} sg.)
1 000	Microseconds	(10^{-6} sg.)
10 000	Tends of micros.	(10^{-7} sg.)
100 000	Hundreds of micros.	(10^{-8} sg.)
1 000 000	Nanoseconds	(10^{-9} sg.)
…….	……..	
10^9	Picoseconds	(10^{-12} sg.)

Table 1. Unit conversion table

Create the `Vector4.java` class using the previous idea.

> If for any reason (e.g., it takes too long) an execution should be aborted, press Control + C. In Eclipse, you can use the red square button above the Console panel.

5. Operations on matrices

In the `MatrixOperations.java` class created in the previous session, several operations are programmed on a square matrix of order `n`.

Create a new class `MatrixOperationsTimes.java` to measure times of operations `sumDiagonal1()` and `sumDiagonal2()`, using everything we have learned so far.

6. Benchmarking

You are provided with two versions of the same program in Java (`JavaTimes.java`) and Python 3 (`PythonTimes.py`). Check the source code of the 2 versions of the program and verify that they are equivalent implementations. Run the 2 programs and look at the results they offer in the console.

1. Why you get differences in execution time between the two programs?

2. Regardless of the specific times, is there any analogy in the behavior of the two implementations?

TO DO:

A. Work to be done

- An Eclipse `session1_1` **package** in your course project. The content of the package should be:
    - All the files that were given with the instructions for this session together with the implementation of `Vector2.java`, `Vector3.java`, `Vector4.java` and `MatrixOperationsTimes.java`.
- A **PDF document** using the course template. The activities of the document should be the following:
    - **Activity 1. Measuring execution times**
        - Provide an answer to the 3 questions marked in blue color in section 2 of this document.
    - **Activity 2. Grow of the problem size**
        - Provide an answer to the 3 questions marked in blue color in section 3 of this document.
    - **Activity 3. Taking small execution times**
        - You should use the previous concepts for the three following methods: `fillIn()`, `sum()` and `maximum()`. With the values obtained, you should complete the following table:

n	*fillIn(t)*	*sum(t)*	*maximum(t)*
10	…..	…..	…..
30	…..	…..	…..
90	…..	…..	…..
270	…..	…..	…..
810	…..	…..	…..
2430	…..	…..	…..
7290	…..	…..	…..
21870	…..	…..	…..
65610	…..	…..	…..
196830	…..	…..	…..
590490	…..	…..	…..
1771470	…..	…..	…..
…	…..	…..	…..
Until it crashes	…..	…..	…..

- What are the main components of the computer in which you did the work (process, memory)?
- Do the values obtained meet the expectations? For that, you should calculate and indicate the theoretical values (a couple of examples per column) of the time complexity. Explain briefly the results.

- **Activity 4. Operations on matrices**
  - You must measure the times of the 2 previous methods seen:

n	*sumDiagonal(t)*	*sumDiagonal(t)*
10	…..	…..
30	…..	…..
90	…..	…..
270	…..	…..
810	…..	…..
2430	…..	…..
7290	…..	…..
21870	…..	…..
65610	…..	…..
196830	…..	…..
590490	…..	…..
1771470	…..	…..
…	…..	…..
Until it crashes	…..	…..

  - What are the main components of the computer in which you did the work (process, memory)?
  - Do the values obtained meet the expectations? For that, you should calculate and indicate the theoretical values (a couple of examples per column) of the time complexity. Explain briefly the results.

- **Activity 5. Benchmarking**
  - Provide an answer to the 2 questions marked in blue color in section 6 of this document.

B. Delivery method

You should **commit and push** your project with your new `session1_1` package in your Github repository with the following content inside it:

- All the requested source files.
- The requested PDF document called `session1_1.pdf` with the corresponding activities.

Important:

- Make sure your Github course project is up to date after the delivery and that the last commit is strictly before the deadline. If not, the mark for this session will be 0 without any exception.
- Your mark will be eventually included in the `Marks.xlsx` file in your repository. Please, do not modify anything in that file to avoid conflicts between versions.

Deadlines:

- Group L.I-01 (Thursday): February 19, 2020 at 11:55pm.
- Group L.I-02 (Wednesday): February 18, 2020 at 11:55pm.
- Group L.I-03 (Tuesday): February 17, 2020 at 11:55pm.
- Group L.I-04 (Monday): February 23, 2020 at 11:55pm.