XJTLU | Computing

CPT108: Data Structures and Algorithms

Semester 2, 2023-24

Laboratory 1: Complexity analysis

We are going to use the *debugger* to trace the code at the beginning of this lab. You are **required** to get yourself familiar with the software by going through the material in the optional lab[a]: "Debugging with the IDE" first if you are not familiar with or using it before.

[a] Available at Learning Mall under the topic: "Problem analysis and procedural abstraction"

1 Purpose

This laboratory session aims to develop you insight into the performance of algorithms. We shall look at both the actual running time of algorithms (in terms of central processing unit (CPU) usage) and the time complexity of the algorithms (in terms of operations performed), and compare and contrast these.

The main emphasis is not so much on coding but is on understanding the performance of the algorithms and why they perform well and poorly on different inputs. Marks are allocated not just for running code but for your understanding and explanation of the performance of algorithms. So, be prepared to give clear explanations to the teaching assistants (TAs). To understand the performances you may need to consult textbooks in the area.

We shall deal with the process of sorting of lists of integers into ascending order and shall compare the perform of three sorting algorithms, namely: insertion sort, heap sort, and merge sort.

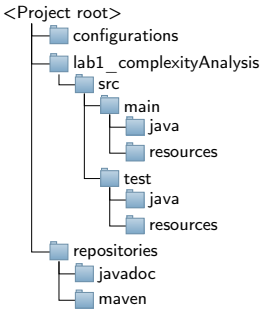
We are interested in two aspects of performance:

- How the performance varies as the size of the input increases, i.e., the rate of growth of the performance measures;
- How the algorithms perform on special kinds of inputs, e.g., those that also already in ascending or descending order, and those in which there are few items repeated many times; and
- Without changing the default Java Virtual Machine (JVM) heap size, what is the maximum length of the list the implementations can handle.

2 Tasks

Step 0: Prepare the evaluation environment

Extract the file that you have download from Learning Mall (LM). The folder structure after extracting the file is as shown in the image on the right.



Here, `configurations` stores Gradle[a]'s common configuration that is going to be used on all of our lab sessions; while `repositories` and `lab1_complexityAnalysis` store the dependencies (libraries and their Javadoc, if available) and the boilerplate code used in the lab, respectively.

[a] Gradle: `http://gradle.org`

You can now go into the project folder (`lab1_complexityAnalysis`) and type "`gradlew compileJava`" in the terminal to compile the Java code (using Gradle), and you will see a message similar to the following in the terminal.

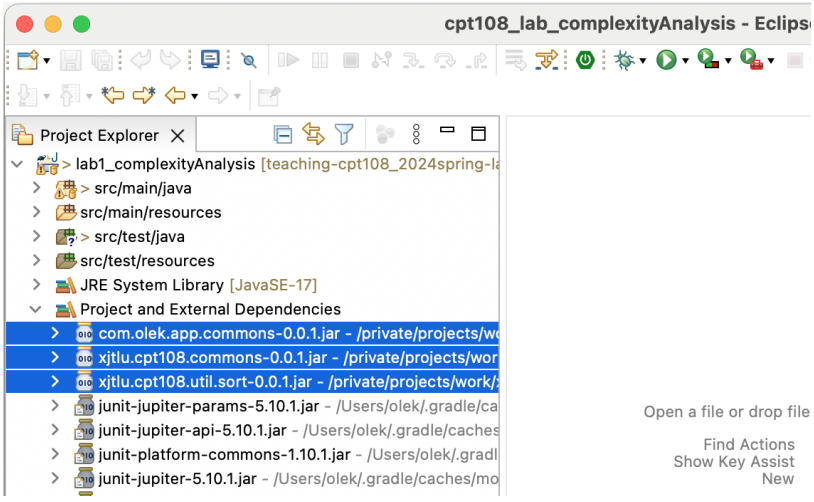
```
<YOUR_PROJECT_FOLDER>\> gradlew compileJava

> Configure project :
localMavenRepository folder exist
⋮
BUILD SUCCESSFUL in 635ms
1 actionable task: 1 executed
```

You can then run the sample Java class by executing the command "`gradlew run`" in the terminal, and a message similar to one below will be shown in the terminal.

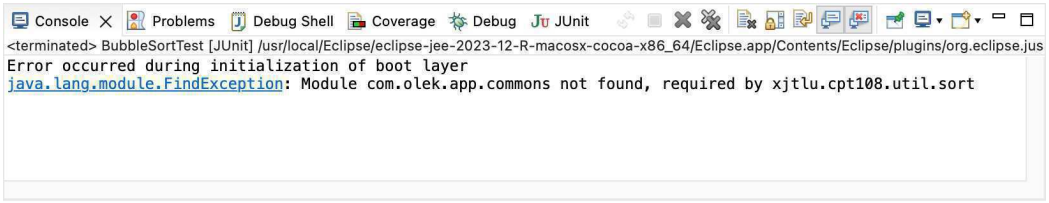
```
<YOUR_PROJECT_FOLDER>\> gradlew run

> Configure project :
localMavenRepository folder exist

> Task :run
xjtlu.cpt108.util.sort.InsertionSort [SORTING_START]: items size: 10
  inputs:  [10]: 10 9 8 7 6 5 4 3 2 1
xjtlu.cpt108.util.sort.InsertionSort [SORTING_END]:
  inputs :  [10]: 10 9 8 7 6 5 4 3 2 1
  outputs:  [10]: 1 2 3 4 5 6 7 8 9 10
==
== Test completed!
==
```

Now, import the project from the folder `lab1_complexityAnalysis` into an Integrated Development Environment (IDE). Note that information on how to import a Gradle project into an IDE is available on LM.

After importing the project, you should be able to find the folder structure and the dependency libraries from the IDE's project explorer, as illustrated in the figure below (using Eclipse).



You may encounter the following error message when running a Java class in Eclipse.



Solution: For some unknown reason, the Eclipse Java Runtime Environment (JRE) is having problem with the Java module descriptor `module-info.java`. There are some solutions available online (by updating the JRE modulepath, etc.) but most of them are not effective. Hence, the most simplest solution is to delete the file `module-info.java` *temporary* from the package.

Step 1: Test data preparation

In this exercise, we need different types of inputs to evaluate the performance of different sorting algorithms.

Task 1

In this task, you are required to create a new class and implements the methods described below.

- A method to generate *ascending* and *descending* lists of numbers;
- A method to generate lists of randomly generated numbers. To do this you may need to use the random number generator from Java (`java.util.Random`);
- A method to generate lists of numbers which consists of a few items *repeated many times* in different patterns of your choice (i.e., you can decide how the numbers in a list appear). To simplify the implementation, you can restrict the possible numbers of repeat to a small range, say from 1 to 9.

All methods above should allow users to input the *size* of the list to be generated.

To simplify the test scenarios, you can assume all numbers generated are to be *integers*.

Before start implementing your code, you should start analysing the problem by identifying:

- Input: What is given?
- Output: What is required?
- Constraints: Under what conditions?
- Abstraction: What information are essential?

ALL must be presented in some form of data that can be processed by computer.

Step 2: Running the sorting classes

The implementation of the three sorting algorithms, i.e., *insertion sort* (`InsertionSort`), *heap sort* (`HeapSort`), and *merge sort* (`MergeSort`), have been provided to you as a dependency library and has been configured to your IDE by Gradle already.

However, it is better if you can understand how they work. Details of the algorithms can be found in Sections 2.1, 2.3 and 7.1 of (Cormen et al., 2022).

A sample Java file `SortingSamples.java` has been provided to you in the source folder (i.e., `src‣main‣java`), illustrating how these Java classes can be used.

Note that all sorting classes can accept input in a form of an array, or any data types that implement the `java.util.Collection` interface, such as `java.util.ArrayList`, `java.util.LinkedList`, `java.util.Stack`, etc.

You can use the methods that you created in Step 1 above to test the implementations.

Note that the package provided also include implementations of other sorting algorithms such as *bubble sort* (`BubbleSort`), *heap sort* (`HeapSort`), *selection sort* (`SelectionSort`), and *shell sort* (`ShellSort`). You can also explore the performance of these sorting algorithms if you are interested.

Step 3: Timing analysis

Performing timing analysis on the three sorting algorithms on the various types of linear searchs above (i.e., random, ascending, descending, and few items), is the main objective of this laboratory session.

There are a few ways that we can use to capture the running time of a program. Some commonly used classes include: `java.time.Duration`, and `java.time.Instant`. Besides, the Java system functions: `System.currentTimeMillis()` and `System.nanoTime()` will return you the current time in millisecond and nanosecond, respectively, which may be useful in measuring the execution time used.

Your work now is to *design* and *implement* your own approach(es) to measure the execution time of sorting a list and applies it to the sorting algorithms. Again, you should start by first analysing the *input*, *output*, *constraints*, and *abstraction* of the problem before starting your implementation.

Task 2

You should run each sorting algorithm on the various types of lists of increasing lengths and measure their performance. The actual lengths of the lists for realistic timing in milliseconds depend on the speed of your processor. A maximum length of 10,000 may be appropriate, but on slower processors this may be too large. *"Experiment" is the only mean of knowing the answer!*

For each type of input, create about 10 lists of lengths *evenly spaced* up to the maximum (of your choice), and *plot* graphs of the timing of each algorithm against the size of the lists.

What conclusion can you make from the results of the experiment?

- You should be aware that the actual CPU time can be affected not just by the algorithm running, but by other processes which may interrupt, so try to minimize other processes running at the same time.
- Make sure the *same* set of data has been used when evaluating the performance of different sorting algorithms, including the case of randomly generated lists!

It is a bit time consuming to record the execution time of the lists one by one. Try to think of a better way of doing it, such as storing the results to some variables and save it to a file, such as a common-separated values (CSV) file, at the end.

An utility class, `com.olek.util.FileManager` has been provided to you to save the content of a file into the file system. A sample file `FileWriterTest.java` illustrating how it can be used can be found inside the `test` folder.

Task 3

In addition, try measuring the execution time of different sorting algorithms with small sizes: 5, 10, 15, . . . , 100, and plot your results on another graph.

Together with the results in the previous task, ***what conclusion can you make with respect to the size of the input***?

In the package, there is a class `StringDataGenerator` that provides you methods to generate random strings with (i) fixed length, and (ii) variable length. You can make use of this class to generate different set of string data and compare the performance of the sorting classes with the results above, if you are interested.

This will give you the experience of how the types of inputs can affect the performance of the computations.

Step 4: Memory usage

Task 4

Without changing the default heap size of the JVM, find the *maximum* size of the array that each sorting algorithms can handle on your machine before the *stack overflow* appear.

What conclusion can you make from the results of the experiment?

3 Marking Scheme

Be prepared to explain to the TAs all the graphs of timing analysis that you produce, and the conclusions that you can derive from the graphs and the result from Task 4, including the effect, if any, when the size of list is small.

#	Item	Weight
1	Code for generating different types of lists	9
2	Code for timing analysis	3
3	Explanation of timing on lists of ascending and descending orders	6
4	Explanation of timing on lists with repeated patterns	3
5	Explanation of timing on randomly generated lists	3
6	Explanation of timing when list sizes is small	3
7	Explanation on memory consumption of different sorting algorithms	3
	Total	30

You should *submit* all your code and the lab report that includes all the results and conclusions that you have derived to the Learning Mall.

References

Cormen, Thomas H. et al. (2022). *Introduction to Algorithms*. 4th. MIT Press. ISBN: 9780262046305. URL: https://mitpress.mit.edu/9780262046305.