Empirical Analysis

COMP 2210 Assignment

Problem Overview

This assignment is a departure from previous assignments insomuch as its focus is not on program construction, but is instead on experimentation, analysis, critical thinking, and applying the scientific method. The provided resources for this assignment are the following.

- `resources.jar` — A jar file containing the `TimingLab` class for which you have to experimentally determine time complexity of a method, and the `SortingLab` class for which you have to experimentally determine the sorting algorithms implemented in various methods.
- `TimingLabClient.java` — Source code of a Java class that illustrates basic use of the `TimingLab` class. *This class is for illustration only.* You can modify and use it for your own purposes or you can use it as an example to create a different but similar class of your own.
- `SortingLabClient.java` — Source code of a class that illustrates basic calls to public methods of `SortingLab`. *This class is for illustration only.* You can modify and use it for your own purposes or you can use it as an example to create a different but similar class of your own.
- `sample_report.pdf` — A sample report that you are required to use as a style, structure, and formatting guide for the deliverables that you submit for this assignment.

There are two parts to this assignment. Each has its own deliverable and is to be done independently of the other.

Part A: Experimental Discovery of Time Complexity

You must develop and perform a repeatable experimental procedure that will allow you to empirically discover the time complexity (in term of big-Oh) of the `timeTrial(int N)` method in the `TimingLab` class. The parameter N represents the *problem size.* Thus, by iteratively calling `timeTrial` with successively larger values of N, you can collect timing data that is useful for characterizing the method's time complexity.

The constructor `TimingLab(int key)` will create a `TimingLab` object whose `timeTrial` method is tailored specifically to the given key value. You will use your Banner ID number as the key required by the constructor. For example, one student might invoke the constructor `TimingLab(903111111)` and another student might invoke the constructor `TimingLab(903222222)`. This will create two distinct objects whose `timeTrial` methods would (very likely) have different time complexities. Note that you **must** use your own Banner ID since grading will be based on Banner ID rather than student name.

You are guaranteed that for any key value used, the associated time complexity will be proportional to N^k for some positive integer k. Thus, you can take advantage of the following property of polynomial time complexity functions $T(N)$.

$$T(N) \propto N^k \implies \frac{T(2N)}{T(N)} \propto \frac{(2N)^k}{N^k} = \frac{2^k N^k}{N^k} = 2^k \tag{1}$$

This property tells us that as N is successively doubled, the ratio of the method's running time on the current problem size to the method's running time on the previous problem size (i.e., $T(2N)/T(N)$) converges to a numerical constant, call it R, that is equal to 2^k, and thus $k = log_2 R$.

As an example, consider the data shown in Table 1. Each row in this table records data for a given run of some method being timed. The first column (N) records the problem size, the second column (Time) records the time taken for the method to run on this problem size, the third column (R) is the ratio discussed above (i.e., $Time_i/Time_{i-1}$), and the third column (k) is $log_2 R$. From Property 1 and Table 1 we can hypothesize that the method being timed has $O(N^4)$ time complexity.

Table 1: Running-time data and calculations

N	**Time**	**R**	**k**
8	0.04	—	—
16	0.08	2.25	1.17
32	0.84	10.37	3.37
64	7.59	9.03	3.18
128	113.56	14.97	3.91
256	1829.28	16.11	4.01
512	29689.21	16.23	4.02

You are required to use `System.nanoTime()`, as illustrated in `TimingLabClient.java`, to generate timing data like that shown in Table 1. Running time values must be expressed in seconds.

To document your work you must write a lab report that fully describes the experiment used to discover the big-Oh time complexity of the given method. The lab report must discuss the experimental procedure (what you did), data collection (all the data you gathered), data analysis (what you did with the data), and interpretation of the data (what conclusions you drew from the data). The lab report must be well written, it must exhibit a high degree of professionalism, and it must be structured like the provided sample. Your experiment must be described in enough detail that it could be reproduced by someone else.

Part B: Experimental Identification of Sorting Algorithms

You must develop and perform a repeatable experimental procedure that will allow you to empirically discover the sorting algorithms implemented by the five methods of the `SortingLab` class — `sort1`, `sort2`, `sort3`, `sort4`, `sort5`. The five sorting algorithms implemented are merge sort, randomized quicksort, non-randomized quicksort, selection sort, and insertion sort.

Insertion sort, selection sort, and merge sort are implemented exactly as described in lecture and the associated note set. Quicksort has two implementations, randomized and non-randomized. Both implementations choose the pivot in the same way: the pivot is always the left-most element (i.e., `a[left]`) of the current partition. Both implementations also use the same algorithm for the partitioning operation (although it is slightly different than the one presented in lecture). The two implementations are different, however, in the following way. The randomized quicksort implementation makes the worst case probabilistically unlikely by randomly permuting the the array elements before the quicksort algorithm begins. The non-randomized quicksort exposes the algorithm's worst case by never shuffling the array elements.

To generate timing data you are required to use `System.nanoTime()`, as illustrated in `SortingLabClient.java`. Running time values must be expressed in seconds. Although timing data will be an important part of your experimental procedure, it will (likely) not be sufficient to distinguish between merge sort and randomized quicksort. To do this you should think about *stability*.

The constructor `SortingLab(int key)` will create a SortingLab object whose sort method implementations are tailored specifically to the given key value. You will use your Banner ID number as the key required by the constructor. For example, one student might invoke the constructor `SortingLab(903111111)` and another student might invoke the constructor `SortingLab(903222222)`. This will create two distinct objects whose `timeTrial` methods would (very likely) have different time complexities. Note that you **must** use your own Banner ID since grading will be based on Banner ID rather than student name.

To document your work you must write a lab report that fully describes the experiments used to discover the the sorting algorithm associated with each of the five sort methods. The lab report must discuss the experimental procedure (what you did), data collection (all the data you gathered), data analysis (what you did with the data), and interpretation of the data (what conclusions you drew from the data). The lab report must be well written, it must exhibit a high degree of professionalism, and it must be structured like the provided sample. Your experiment must be described in enough detail that it could be reproduced by someone else.

Grading

Two rubrics (*Experimentation Rubric* and *Written Communication Rubric*, both available on Canvas) will be used to score each lab report. Marks of Little/No Ability, Basic, Intermediate, and Advanced are worth 1, 2, 3, and 4 points respectively. Note that you are required to structure and style your reports according to the sample provided. Significant deviation from this sample will negatively impact your grade.

Using the provided `jar` file

To compile and run the provided source files and any of your own that might use for this assignment, you will need to include `resources.jar` on the classpath. You can do this from the command line or from within your IDE.

From the command line. In the following example, `>` represents the command line prompt and it is assumed that the current working directory contains both the source code and the jar file. The example applies to both source code files although only one is used for illustration.

```
> javac -cp .:resources.jar TimingLabClient.java
> java -cp .:resources.jar TimingLabClient
```

From the jGRASP IDE.

1. Create a project for the assignment (suppose you call it A3Project), and include `SortingLabClient.java` and `TimingLabClient.java`.
2. Click on *Settings* → *PATH/CLASSPATH* → *Project* → *<A3Project>*.
3. Click on the *CLASSPATHS* button.
4. Click *New*.
5. In the "Path or JAR File" text box, use the *Browse* button to navigate to `resources.jar` and select it.
6. Click on *OK*.
7. Click on *Apply*.
8. Click on *OK*.

Increasing the JVM stack size

In Part B it may be necessary for you to increase the amount of memory allocated to the JVM's runtime stack. The `Xss` flag is used to request an amount of memory in bytes. For example, the following command runs the `SortingLabClient` class with four gigabytes of stack memory.

```
java -Xss4G -cp .:resources.jar SortingLabClient
```

You can set the `Xss` flag in jGRASP at *Settings* → *Compiler Settings* → *Workspace* → *Flags/Args* → *FLAGS2 or ARGS2*.

Assignment Submission

This assignment appears as two separate assignments in Canvas – A3A and A3B, for Part A and Part B respectively. You must submit a separate lab report as a PDF file to each. Submissions made within the 24 hour period after the published deadline will be assessed a late penalty of 15 points. No submissions will be accepted more than 24 hours after the published deadline.