ESOF-3251 Compiler & Algorithm Design
Project 2
Performance Analysis of Sorting Algorithms

1. Aims

The aim of this project is to design and implement classes for various sorting algorithms (Selection-Sort, Bubble-Sort, Insertion-sort, and QuickSort) with a strong emphasis on performance. Also, the sorting algorithms need to be designed with the ability to sort any data type (using generic data types).

2. Objectives

On completion of this project the student should be able to:

a) Design and implement classes for different sorting algorithms
b) Use the *Comparable* interface to build APIs that manipulate generic data types
c) Use *interfaces* for building a single driver that can test-various sorting algorithms
d) To analyze the performance (Big-O notation) of different sorting algorithms

3. Background

Sorting is a frequent operation in many applications. It is used not only to produce sorted output but also in many sort-based algorithms such as grouping with aggregation, duplicate removal, sort merge join, as well as set operations including union, intersect, etc. The main goal of this project is to analyze the performance of a set of sorting algorithms. One way of achieving this goal is to add counters to the sorting algorithms in order to empirically measure the number of comparisons performed by each algorithm. You will then write code to analyze the measured data and automatically determine the BigO function for each set of data.

Counting compares is not always a straightforward process. It is particularly “tricky” when used inside of for or while loop tests. The following example explains the problem.

```
for (int i = 0; i < items.length && items[a].compareTo(items[b])>0; i++)
```

Here, there are two parts to the for loop condition. If you count inside the loop, you could miss the case where the loop didn't run because items[a].compareTo(items[b]) ≤ 0. However, if you just add an extra double increment to your compare count after the loop, you won't be accounting for the case where i < items.length (because java boolean expression evaluation is short circuited).

Comparison-based sorting algorithms make ordering decisions using only pairwise comparisons of two items at a time. Algorithms such as insertion sort, selection sort, bubble sort, and quick sort are all comparison-based sorting algorithms. Any objects that implement the *Comparable* interface can be sorted this way. The efficiency of a sorting algorithm is in this case evaluated by counting the number of compares the algorithm uses to sort the data. A compare is counted each time two items are compared.

4. Required Tasks

Implement and Test the Sort Interface

The file Sort.java, available in the course website on Desire2Learn, defines the `Sort` interface. Classes that implement the sort interface are capable of sorting arrays of `Comparable` objects and counting the numbers of compares used to do the sorting.

The `Sort` interface defines two methods: `void`

```
     sort( Comparable [ ] a );
```

This method sorts the array `a` in ascending order, smaller items come first and larger items come later.

```
     long getCompares();
```

This method returns the number of compares used in the sort.

Write four classes that implement the `Sort` interface. Name your classes `InsertionSort`, `SelectionSort`, `BubbleSort`, and `QuickSort`. Put your classes in the `engi3255.sort` package. Your classes must use the algorithms defined in the files InsertionSortstd.java, SelectionSortstd.java, BubbleSortstd.java, and QuickSortstd.java. These are made available to you in the course website on Desire2Learn. However, you must modify the provided code to count compares and to implement the Sort interface. Don't change the workings of the algorithms as this may cause your compare counts to be wrong.

Testing Your Sort Classes

The first half of the test driver (TestDriver.java) tests the sorting classes using a number of different types and sizes of test data. The driver verifies that each sort actually sorts the data and that the numbers of compares are counted correctly. You have to test and debug your sorting classes before continuing.

Empirical BigO Analysis

Empirical BigOh analysis is the process of estimating the BigO of a program using measurements obtained from actually running the program. Suppose you want to show that the number of compares in a sort is `O(F(N)).` (The function `F(N)` could be `N`, `N log N`, or `N^2` for example.) You could run the sort for a number of different input sizes and record the number of compares (`C(N)`) for each input size. If the function `F(N)` is a good estimate of the correct BigO, the ratio `C(N)/F(N)` should converge to a positive constant (refer to the definition of Big-O notation).

Implement the Analyzer Interface

The file Analyzer.java, available in the course website on Desire2Learn, defines the `Analyzer` interface. Classes that implement the `Analyzer` interface are capable of estimating the BigOh of a program using empirical data. Write a class named `AnalyzerImpl` that implements the `Analyzer` interface. Put your class in the `engi3255.sort` package.

The `Analyzer` interface defines four methods:

```
void analyze(int[] sizes, long[] data);
```

This method is given two arrays. The first array provides the sizes of the datasets that were input to the program. The second array gives the values that were measured when running the program with each dataset. From this data analyze decides which BigO function gives ratios that converge to a constant as described above. Analyze must consider the following possible BigO functions:

```
F(N) = 1

F(N) = log N

F(N) = N

F(N) = N log N

F(N) = N^2

F(N) = N^3

F(N) = 2^N
```

Analyze tries each BigO function and computes the ratio for each size of input data. How do you decide if the ratios converge to a constant? First compute the mean of the ratios. Then compute the relative error of each ratio with respect to the mean. If the average of the relative errors is a small number, the ratios are probably close to the same constant. The relative error can be computed as follows:

$$error = |ratio - mean|/mean$$

Note that the bars around 'ratio - mean' represent absolute value.

So if you compute the average relative error in the ratios for each possible BigO function and select the BigO function that gives the smallest error, its likely that you've chosen the best BigO function.

```
double[] getRatios();
```

This method returns the ratios computed by analyze for the BigO function that has the smallest error.

```
double getError();
```

This method returns the error computed by analyze for the BigO function that has the smallest error.

```
String getBigO();
```

This method returns a string describing the BigO function selected by analyze. One of the following strings must be returned.

```
O(1)

O(log N)

O(N)

O(N log N)

O(N^2)

O(N^3)

O(2^N)
```

Testing Your Analyzer

The second half of the test driver tests the `AnalyzerImpl` class using a number of different types and sizes of test data. The driver verifies that `analyze` selects the correct BigO function for each set of test data. Note that the test driver does not test `getRatios` and `getError`. It is strongly recommended that you test and debug your analyze method before continuing.

Test Your Code using the TestDriver Program

The file TestDriver.java, available in the course website on Desire2Learn, contains a test program you can use to test your code. If the test program fails when you run it with your code, your code has a problem that needs to be fixed. Using the `TestDriver` program should help you get your code working properly.

NOTE: The `TestDriver` program uses Java `assert` statements to verify that your code works properly. To enable `assert` statements you must do the following:

(1) When you run `javac` compiler, you must use the `-source 1.4` command-line option. For example,

```
    javac -source 1.4 TestDriver.java
```

(2) You must run the java interpreter with the `-ea` command-line option. If you run the program without the `-ea` option, it will successfully complete without actually testing anything.

NOTE: You will also need to use the `-classpath` command-line option when running the virtual machine (`java`) to tell it where to find your `.class` files. For example, if your `.java` source files are stored in a directory named `\projects\engi3255\sort` the following command would be used to run the `TestDriver` program:

```
java -ea -classpath \projects engi3255.sort.TestDriver
```

The above mentioned settings can also be set in Java environment settings if you are using for instance Visual Studio or a similar environment.

Write a driver program that empirically determines the BigO for each of your sorting algorithms.

Write a class named `SortAnalyzer`. Put your class in the `engi3255.sort` package. Within `SortAnalyzer` write a `main` method and supporting methods that do the following:

1. Run each of your four sorting algorithms.

2. Run each algorithm with three types of

   data: *data that is already sorted*

   data that is sorted in reverse order

   data that is in random order

3. Run each algorithm with five sizes of each type of data. The required five sizes are 100, 200, 400, 800, and 1600 elements. This means you will execute a total of 4*3*5 = 60 runs.

4. Record the numbers of compares used in each sorting run.

5. Pass the compare data to an `Analyzer` object.

6. Print the results of each run in a format similar to the following:

```
BUBBLE sort with SORTED data

sizes: 100 200 400 800 1600

compares: 99 199 399 799 1599

ratios: 0.99 0.995 0.997 0.998 0.999 O(N) error 0.0026
```

```
BUBBLE sort with REVERSE data

sizes: 100 200 400 800 1600

compares: 4950 19900 79800 319600 1279200

ratios:  ...........................


BUBBLE sort with RANDOM data

sizes: 100 200 400 800 1600

compares: 4935 19809 79239 319275 1278900

ratios: ............................
```

Deliverables

Once you have your code working with the `TestDriver` program and you have written a `SortAnalyzer` driver, you must submit the following:

1. Java code of all implemented classes, interfaces, and test-drivers
2. Report the final output of the two test-drivers (*TestDriver* and *SortAnalyzer*)
3. Arrange for a software demonstration of your software

References and Acknowledgements

More information about *Interface Comparable* .can be found at
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html

More information about Java Interfaces can be found at
http://java.sun.com/docs/books/tutorial/java/IandI/createinterface.html

Thanks to Dr. J Crandall, the Department of Computer Science, at Brigham Young University, who developed the original version of this project. However, most sorting algorithms used in this version of the project are the ones used in this course. All students are reminded that acquiring solutions from other people is a punishable academic offence.