CSC 222: Programming Project # 2: Sorting Algorithms and Complexity

Instructor: V. Paúl Pauca

due date: Tuesday March 3

In this programing project you will implement three well-known sorting algorithms in C++ and will conduct experiments on them to understand and compare their complexities. The three algorithms to implement are: selection sort, merge sort, and quick sort.

Code provided to you. You are provided with working C++ code and a Makefile to help you run the timing experiments. Here is a summary of the code that is provided:

- Class definitions (header files) and implementations (cpp files) for two classes: `Data` and `Experiment`
- Abstract class definition (header file) for class `SortAlgorithm`
- Class definition (header file) and (empty) implementation (cpp file) for `SelectionSort`
- A driver function in file `main.cpp`
- A makefile

Here is the driver `main()` function and an explanation of what it does:

```
1  int main() {
2      // Read file web2random which contains 235886 word entries
3      Data data("web2random");
4
5      // Set the standard deviation for the Gaussian weight function to be
6      // small so that the number of trials decays quickly as problem size
7      // increases.
8      data.setStd(1.0/1024);
9
10     Experiment experiment(&data);
11
12     // Add the algorithms to the experiment
13     experiment.addAlgorithm(new SelectionSort());
14     //experiment.addAlgorithm(new MergeSort());
15     //experiment.addAlgorithm(new QuickSort());
16
17     // Run 98 timing experiments with list sizes between 2 and 100
18     // 2048 is the maximum number of trials to do for each problem size
19     experiment.run(2, 100, 98, 2048);
20     experiment.save("Results.txt");
21 }
```

An executable called `test` is created upon successful compilation with the Makefile. The executable in turn creates a text file (e.g. `Results.txt`) containing the timing results. This output file is formatted as follows:

- first column: problem size of the different lists to sort
- second column: average time (in seconds) that it took the first sort algorithm added to the experiment to sort a list of the size specified in the first column
- third column: average time for the second sort algorithm added the experiment
- etc.

What you are required to do.

1. Complete the implementation of `SelectionSort` in the cpp file. Notice that class `SelectionSort` is a subclass of `SortAlgorithm`
2. Following the class definition shown in the header file for `SelectionSort`, define and implement classes for `MergeSort` and `QuickSort`. It's very important that these are subclasses of `SortAlgorithm` and that you implement the pure virtual function `sort()` as shown in the code.
3. Verify the accuracy of your implementations by running an experiment with small list sizes, e.g.

   ```
   // 2 sort problems with lists of size 4 and 5 and 1 trial for each
   experiment.run(4, 5, 2, 1);
   ```

   and by removing the `#` in the definition of `CFLAGS` in the makefile:

   ```
   CFLAGS = -g -I/usr/local/include #-DPRINT
   ```

   Make sure you add the `#` back if you don't want to continue to see a whole bunch of output.
4. Run timing experiments

What to turn in. You must turn in a well-written paper of at least 3 pages explaining what you have done and the results you have obtained. Here are the different parts required for the paper:

1. Introduction. Brief description (1/2 page) of the sort algorithms you have implemented and their complexities
2. Method. Description of the experiments that you have run with the provided code, e.g. specify what parameters you choice and explain those decisions. (1 page)
3. Timing results and discussion. Plots showing your results (you can load file `Results.txt` into Excel (or Matlab if you are familiar with it)) and discussion regarding the following:
   - contrast the performance of the different algorithms for small list sizes (below 30 or so),
   - contrast the performance over larger list sizes (less than 10000 or so), and
   - show that in fact the timing values you get are Θ of n^2 or $n \log n$.

Grading rubric

- A range: a well-written and well-organized paper showing all the different required parts. There are nice plots and the timing results are clearly discussed showing great understanding of the experiment. In addition, the complexity of each algorithm is experimentally verified to be Θ of n^2 or $n \log n$.
- B range: a well-written and well-organized paper showing all the different required parts. There are nice plots and the timing results are clearly discussed showing good understanding of the experiment.
- C range: a well-written and well-organized paper showing reasonable understanding of the experiment.
- D range: not well-written paper. Incomplete results.
- F: fails to turn in assignment.