Programming Assignment #13: Empirical Testing of Sorting Algorithms[1]

This program focuses on learning about program efficiency through empirical testing of specific sorting algorithms. You will turn in two files: `Empirical.java` and `Sorters.java`. You can get the shell of `Sorters.java` from the website and download it to your working directory.

The assignment has two parts: a client program that uses the static sort methods in the `Sorters` class, and an `Empirical.java` program of your own that creates a test of all the sorts with varying sized and pre-sorted arrays.

Part A (`Sorters.java`, typing in the sorting algorithms):

The first part of this assignment is to download `Sorters.java` and fill in the three sorting algorithms, `InsertionSort`, `SelectionSort` and `MergeSort`. You will find the code for these algorithms in the book. These three methods will be declared as `public` methods, any methods these three methods call should be declared as `private`.

You may also choose to implement other sorts to test, such as `QuickSort`, `HeapSort`, `BozoSort`, etc, whatever you can find on the web. If you choose to implement other sorting algorithms, you must cite the source from which you found them.

- `public static void insertionSort(int[] array)`
  Sorts the array, in place, using the Insertion Sort algorithm.

- `public static void selectionSort(int[] array)`
  Sorts the array, in place, using the Selection Sort algorithm.

- `public static void mergeSort(int[] array)`
  Sorts the array, in place, using the Merge Sort algorithm.

Part B (`Empirical.java`, an empirical testing program):

The second part of this assignment asks you to implement a program, much the way we did in Computer Science I. You will need to functionally decompose the problem into parts, create methods that make sense, use the proper data structures, and match the output. This program will be in the file called `Empirical.java`.

[1] Idea from Java For AP Computer Science, page 749-750

In this program you will create integer arrays of size 1000, 5000, 10,000, 20,000, and 50,000, and you will test them at a minimum with, `InsertionSort`, `SelectionSort`, `MergeSort` and `Arrays.sort()`.

In order to determine the running time of a section of code, you can use the method `System.currentTimeMillis` which returns a `long` variable containing the number of milliseconds since Jan 1, 1970 of the current time. For example, the following main method determines how long it takes to sort 1,000 strings using the `InsertionSort` method

```
    private static int LIST_SIZE= 1000;
    public static void main(String[] args)
    {
         // create your string array list and fill
         // it up
         String[] list = {"c", "b", "a"};
         System.gc();
         long beginTime = System.currentTimeMillis();
         Sorters.insertionSort(list);
         long endTime = System.currentTimeMillis();
         System.out.println((endtime - beginTime) + "ms");
    }
```

Note the other system call here: `System.gc()`. This is called just before the critical section we want timed to make sure that the garbage collector doesn't run stall that part of the program. This will give us a more accurate view of the timing of our sorting algorithm.

Your program will compare, at a minimum, `SelectionSort`, `MergeSort`, and `InsertionSort`, and `Arrays.sort()` with the following data sets:

- Fully ordered arrays with 1000, 5000, 10,000, 20,000, and 50,000 elements
- Reverse ordered arrays with 1000, 5000, 10,000, 20,000, and 50,000 elements
- Randomly ordered arrays with 1000, 5000, 10,000, 20,000, and 50,000 elements. The random numbers can be anything, for instance from `0.. array.length`, with repeats (in other words, you can use the `nextInt()` in the Java `Random` class)
- **IMPORTANT NOTE**: In order to have a proper comparison for the randomly ordered elements, you will need to create **ONE** array that you copy for each of your tests (so your tests are even). I would recommend making one really large array, and copying parts of it using `Arrays.copy` to copy the first n elements of it to test `SelectionSort`, then copy it again to do `MergeSort`, etc. Do NOT include the time it takes to copy the array in your empirical results.

As a bonus you can find and implement other sorts such as `QuickSort` and `HeapSort`.

Your results will be printed in a nice table, as shown below. You will need to use `System.out.printf()` in order to get the table to align correctly. (In fact the first column of the table below uses `System.out.printf("%15s\t",...)`.

```
     Increasing 1000    5000    10000   20000   50000
  selectionSort 6       24      35      137     866
  insertionSort 0       0       0       1       1
      mergeSort 1       12      24      37      23
    Arrays.sort 1       5       7       1       2
     mergeSort2 1       6       1       2       3

     Decreasing 1000    5000    10000   20000   50000
  selectionSort 0       16      61      235     1470
  insertionSort 11      8       31      124     766
      mergeSort 1       0       1       3       6
    Arrays.sort 1       0       0       1       1
     mergeSort2 0       0       0       1       3

         Random 1000    5000    10000   20000   50000
  selectionSort 0       9       37      139     870
  insertionSort 0       10      10      70      362
      mergeSort 0       0       2       2       7
    Arrays.sort 0       1       1       2       5
     mergeSort2 0       1       1       2       6
```

Realize that the actual times for the sorting times will vary, but will be close to these values.

Testing Your Program:

Note that you will not get exactly the same running time as the numbers listed here, but the table should look fairly similar. You will expect to see quadratic growth for `InsertionSort`, and `SelectionSort` and log linear growth for `MergeSort` and `Arrays.sort()`.

Development Strategy and Hints:

Complete Part A before Part B.

Write a small test program to test that all of your sorts work properly before you start work on `Empirical.java`.

There is a LOT of duplicated code in part B. **Make sure that you use good functional decomposition and refactoring**. Methods that take parameters such as which sort you're doing, the size of the array your testing on, whether it's an ascending, descending or randomly ordered array, etc. Part of your grade will be how effectively you refactor your code.

Java has a concept called an "Enum" or enumerated type. This may be helpful to write your methods for refactoring your code.