CSci 3501 Lab 2
Due Friday, September 14th by midnight
30 points
Work in pairs

- All lab submissions should be done by canvas. Please be sure to include your group members.
- When working on the lab, please comment your work so that it is clear what contributions of each person are.
- At the end of the lab each group should submit the results of their in-class work. Please indicate if this is your final submission. Don't forget to answer all the questions below.
- If your submission at the end of the lab time was not final, please submit a final copy before the due time.

Overview and goals

In this lab you will study the efficiency of the standard Java algorithms for objects(`Tim Sort`) and `quicksort` (see `http://en.wikipedia.org/wiki/Quicksort`) for different types of data (such as completely random, ordered, partially ordered). The efficiency of the algorithms is measured in the number of comparisons. Your program will call both sorting methods on the same data and measure the number of comparison operations performed by each sorting algorithm. The goal is to observe and analyze practical aspects of sorting. The lab also helps you learn testing methodology for algorithms analysis.

Technical details

You will use a predefined method `sort` that implements `Tim Sort` which is specifically designed ot produce high efficiency on partially sorted arrays. Note that this is a static method of a class `Arrays` which works on arrays of objects.

You will need to write your own implementation of `quicksort` according to the algorithm in the book (p. 171). Make sure that you implement the in-place quicksort given in the book (i.e. your algorithm should not create additional arrays). Your quicksort should take an array of `TestInteger` type (see below). Use `compareTo` method to compare elements, do not use <. Try to minimize the number of comparisons.

In addition, please write a method `isSorted` that takes an array of `TestInteger` and returns `true` if it is sorted and `false` otherwise. Use this method to test your quicksort.

In order to count the number of comparisons you need to create your own class `TestInteger` that has an integer field `value` for comparison and a static `long` field counter to count the number of comparisons performed on all `TestInteger` objects.

Your class `TestInteger` should have the following components:

- Implements `Comparable<TestInteger>` interface
- Have an integer field `value` (which is used by `compareTo` for the comparison)
- A static `long` field called `counter` to counter the number of comparisons performed on all `TestInteger objects`
- The `compareTo` mehtod returns the result of comparison of `value` of "this" `TestIntetger` and the parameter `TestInteger` according to the specification in `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html` (see `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html\#compareTo-T-`)

When generating arrays to be sorted, create two arrays and insert the same elements in both in the same order. Sort one array using `Tim Sortt` and the other one using `quicksort`. Make sure to record and reset the `TestInteger` counter between the two sortings.

Tasks

- Implement `TestInteger` and `quicksort` method as described above. Test your implementation using `isSorted` method.
- Generate 10,000 random `TestIntegers` in the range from 1 to 1,000,000 and put them (in the same order) into two arrays. Use Java random number generator Random (`Math.random`). **Make sure to use only one Random object in your program** - create it in the very beginning outside of any loops. Sort one using the `Tim Sort` and the other using a `quicksort`. Run the program 5 times measuring the time to sort the two arrays (separately) in the number of comparisons. Record your results.
- Repeat the sorting, but instead of random elements use the arrays where elements are stored in increasing order. Record the results.
- Repeat the measurements for arrays that consist of 10 sorted sequences of 1,000 elements each (randomly choose the starting number in each sequence). Record the results of 5 measurements.
- Repeat the measurements for arrays that consist of 100 sorted sequences of 100 elements each (randomly choose the starting number in each sequence). Record the results of 5 measurements.

For each kind of data, which of the two algorithms performs better? Submit your program, the results of measurements, and noted on your observations.

In case you are getting a Stack Overflow exception: numbers in increasing order may generate a stack overflow because there are 10,000 calls (that's the worse case of quicksort when there is one call per element) which is more than the default stack size. You need to increase the stack size in the JVM (Java Virtual Machine) which you can do by setting a JVM flag in Eclipse: `http://stackoverflow.com/questions/15313393/how-to-increase-heap-size-in-eclipse`