Complementary Task 1:
Efficient Sorting Algorithms

1 General Instructions Applying to All Tasks

Each of the complementary tasks of the course results in a written report that will be graded. As the language of this course is English, your report must be written in English, too. Each task is divided into four individual steps that should roughly correspond to sections of your report. Each step has its own deadline at which you must hand in a draft of your report (or, in step 4, the final version). The drafts corresponding to steps 1–3 must each receive a pass grade in order to continue. The pass grade merely signifies that you have worked on the task according to its specification and have made reasonable progress. As a consequence, a fail grade cannot be rectified by handing in a new version. On each of the drafts handed in in steps 1–3 you will receive comments that must be taken into account when going ahead. The final report will be graded according to the scale fail, 3, 4, 5.

The following are the deadlines. – Make sure to keep them!

**Step 1** 2014-09-21, 23:59
**Step 2** 2014-10-05, 23:59
**Step 3** 2014-10-19, 23:59
**Step 4** 2014-11-02, 23:59

Please submit your drafts as well as the final report [here](#). If implementations are required (i.e., for tasks 1 and 2), make sure that the files are organized in a reasonable way, and that it does not require a wizard to make the implementation work outside your personal environment.

2 The Task

In this task, you will implement different sorting algorithms in Java, compare their running times, and write a report on your implementation of the algorithms and your findings.

Step 1: A Basic Environment for Testing Running Times

Your Java package will consist of

1. an abstract class `Sorter` containing a single method
   ```java
   public abstract void sort(int[] arr, int last),
   ```
2. a number of concrete subclasses of `Sorter` that implement several sorting algorithms, and
3. a class `Tester` that provides some basic functionality for testing the efficiency of the `Sorter` implementations,

4. a variant of `Tester` described later.

Naturally, an implementation of `sort(int[] arr, int last)` should sort `arr[0, . . . , last]`. By running `Tester`, it should be possible to measure and compare the running times of different sorters (i.e., classes implementing `Sorter`). For this, the sorters that shall be tested are applied to randomly populated integer arrays of increasing size, so that a curve is obtained. Passing the desired testing parameters can be done via command line arguments or by redefining variables in `Tester.java` and recompiling. (Feel free to invent fancier methods if you consider this to be too inelegant.)

The testing parameters that shall be used are:

- values `start` and `stop` that determine the minimum and maximum number of entries to be sorted,
- a value `steps` that determines into how many intervals the range from `start` to `stop` shall be divided (see Figure 1 for an example where `start = 0, stop = 2.000 .000, and steps = 20`), and
- an array of sorter instances to be tested.

In addition, there should be a constant (that can of course be altered) called `rep` here. It determines how many times the sorters are run on arrays of a each size (with different array contents as input, of course). This number should not be too small, in order to make the tests less prone to random fluctuations caused by an unlucky choice of input data.

The output of `Tester` is a matrix showing the results in a tab-separated format. There are `steps + 1` rows corresponding to the progression of array sizes, and there are as many columns as there are sorters tested. Measured time is given in microseconds. The time reported in each entry of the matrix is the average of the time the `rep` repetitions required. You may precede the table by a row showing the class names of the sorters, and also by a column showing the respective array sizes. This output format allows you easily to paste the matrix into a spreadsheet and create a diagram such as the one in Figure 1. (If you prefer to use Matlab or the like for drawing your diagrams, feel free to do it.)

The implementation and application of your `Tester` class should be done in such a way that the running times of the sorters are affected as little as possible, and are measured as accurately as possible. In particular, pay attention to the following:

- Search for good ways to measure elapsed time in Java. Select the one which is most suitable and make a few tests to ensure that it works as expected.
- Avoid unnecessary memory allocation, especially repeated allocation of large input arrays. Instead, use and re-use sufficiently large arrays.
- Try to make sure that the garbage collector is unlikely to interrupt the run of a sorter. Otherwise, the measured time of one sorter could be affected, especially if previously running sorters allocated a lot of memory.
- For the sake of fairness, let all sorters run on the same input arrays, i.e., populate an array with random integers and apply all sorters to these data before turning to the next round. (Obviously, you need two arrays for this, one holding the original data and a copy to which the sorters are applied.)
- During your test runs, try to make sure that the workload caused by other processes is as low as possible and does not change.
For each of the coming implementations of sorters, make sure to run some basic correctness tests. (It is very easy to make efficient implementations if correctness is disregarded.) In the sorter implementations, avoid wasting time by unnecessary comparisons, assignments or parameter passing. Since we want to find out how efficient the algorithms are, they should not be implemented in a suboptimal way, because otherwise comparisons are meaningless. Note that textbook versions sometimes can be tuned a bit, as they sometimes trade efficiency for conciseness of the description. If you are in doubt, it may be a good idea to make two versions of the same sorter and use your tester to see which one to keep. If the tests reveal a strange behaviour
of some or all sorters, study your implementation to find the reason and improve it.

Besides Tester, in this first step you should implement heapsort, in a class called Heapsorter that extends Sorter. Draw the diagram of the average running time of Heapsorter if applied to arrays of size up to stop elements, where stop is large enough to obtain a reasonably clear picture of the asymptotic behaviour of the algorithms. For example, in Figure 1, five different sorters (rather than Heapsorter alone) were tested with a stop value of 2,000,000.

Think about whether or not there are any details in the implementation of Heapsorter that might slow it down, e.g. because of comparisons or assignments that could be avoided. If you identify one or more, make alternative implementations and use your Tester in order to check which version is better.

To complete the first step of your complementary task, write a first section in your report that

1. explains your implementation of the Tester class and highlights any important points
2. explains your implementation(s) of the heapsort algorithm, and
3. shows and briefly discusses your results for Heapsorter.

In particular, your discussion should discuss whether the results confirm your expectations and whether they coincide with the theoretical results. If you discover discrepancies, try to find possible explanations. The latter will not explicitly be mentioned again below, but it applies to the entire Complementary Task. Always ask yourself whether your results reflect what you would expect, and discuss this. Try to find explanations for any deviations (after having checked that the implementations are correct).

**Step 2: Further Sorters**

The second step of the task is to implement four additional sorters, namely two versions each of mergesort and quicksort, as follows:

**SimpleMergesorter** This should be a straightforward implementation of mergesort in which sort calls a recursive method of the form mergesort(int[] dest, int first, int last) that sorts source[first, . . . , last] into dest by allocating to new auxiliary destination arrays, making two recursive calls, and merging the resulting arrays into dest.

**Mergesorter** Develop a more efficient variant of mergesort that avoids the recursive allocation of arrays. Instead, use a second array of the same size as the input array, and use two mutually recursive methods mergesort1(int first, int last) and mergesort2(int first, int last) that use the two arrays alternatingly as source and destination.

**Quicksorter** Unsurprisingly, this sorter should be an implementation of quicksort. Try to implement the partition method in a good way, but do not use a randomly chosen pivot element. Pick, for example, the first element of the input array as the pivot.

**Quicksorter2, . . .** Choose at least one more version of quicksort known from the literature, and implement it.

Extend your report by a section in which you explain your implementations and compare the different versions of mergesort and quicksort with each other. In particular, this should include a discussion of your implementation of Mergesorter, where you should argue why it is correct. Again, make sure to choose reasonable values for the parameters of the tester. In particular, choose a sufficiently high number of repetitions.
and a sufficiently large maximum array size. The exact value may depend on your implementation, computer, operating system, and a few further factors; on my desktop computer, a stop value between 2,000,000 and 3,000,000 turned out to be reasonable. For example, using the same sorter implementations as in Figure 1, but a smaller stop value, the diagram in Figure 2 is obtained. Comparing the two figures, we see that the second does not do sorters A and B justice (unless we know that our sorters are unlikely to be applied to really large arrays).

Feel free to add a third family of related sorters to the two above and study them in the same way if you like.

**Step 3: Sorting Presorted Arrays**

We know from the theory that the worst-case running time of quicksort is $O(n^2)$. It seems intuitively reasonable to assume that quicksort will become more efficient the less sorted the input array is. Here, you are asked to design and implement a little experiment that tries to provide evidence for the correctness of this hypothesis. For this, the idea is to create arrays that are presorted to various degrees and apply quicksort to them in order to see how the running time increases if the degree of presortedness does.

First of all, think about a reasonable way of creating presorted arrays. Try to make sure that the data is not particularly structured except for being presorted to some degree, because any kind of structure may influence the running time in some unknown way.

---

1 You may have to use the `-Xmx` option of the java virtual machine to avoid an out of heap space error.
and thus invalidate the results. The basic procedure for obtaining a presorted array should be to start from an entirely sorted array. To this array, you should apply a method that, depending on a parameter \texttt{presortedness} between 0 and 1, randomly mixes up array elements. The value \texttt{presortedness} = 0 should yield very low order (i.e., many elements get mixed up randomly) whereas \texttt{presortedness} = 1 should correspond to total order (i.e., nothing gets mixed up at all).

Think about a good way to do this and implement it in a variant of \texttt{Tester} that tests only one sorter (namely \texttt{Quicksorter}), but for different values of \texttt{presortedness}. For this, the \texttt{Tester} variant has additional constants \texttt{low}, \texttt{hi} and \texttt{num}, where $0 \leq \texttt{low} < \texttt{hi} \leq 1$ and \texttt{num} $\geq 1$. When it is executed, it runs tests on arrays of increasing size, precisely as before. However, for each array size, it measures the running time on \texttt{num} more and more presorted arrays of the given size, where the degree of presortedness is given by \texttt{num} values of \texttt{presortedness} ranging from \texttt{low} to \texttt{hi}. Thus, the diagram you obtain shows \texttt{num} graphs, each corresponding to one value of the parameter \texttt{presortedness}.

Note that it may require a bit of experimentation to find out which values of the various parameters result into meaningful data. (I do not include a diagram containing the results of my own implementation to avoid influencing you, especially because the diagrams may look different if different ways of creating presortedness are used.)

Extend your report by a section that describes what you have done, why you have done it in this particular way and, most importantly, which conclusions you can or cannot draw from your test results.

**Step 4: The Final Touch**

In addition to this, add a first and a last section to your report, containing an introduction and a conclusion. The introduction should explain what the report is about, and it should also contain a short introduction to the heapsort, mergesort, and quicksort algorithms and their (theoretical) running time. The conclusion should consist in a brief discussion of the main sections, highlighting and critically reflecting on the main findings, and perhaps mention further things that could be studied. Add a few references (like the course textbook) to your report pointing the reader to the sources that you have used.

In addition, polish and re-work the already existing parts of your report in order to support all claims with convincing arguments. Look for gaps in your argumentations or experiments and try to close them. If you, during your work regarding the previous steps, have come across any other interesting observation, hypothesis or sorter variants, feel free to design some experiments to find out more, and include a discussion of your findings in an appropriate place. A possible addition would be a discussion of measures of presortedness known from the literature (do not forget references in this case!) and how they relate to your work.

*I hope that this task has been interesting and inspiring to work with. Please do not hesitate to give me any type of feedback.*