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** 2013-09-22, 23:59
**Step 2** 2013-10-06, 23:59
**Step 3** 2013-10-20, 23:59
**Step 4** 2013-11-03, 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 abstract method `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, 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. In particular, pay attention to the following:

- 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 kick in during the run of a sorter. Otherwise, the measured time of one sorter could be affected by the memory allocation of previously running sorters.
- 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 make an implementation of 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] the value chosen was 2,000,000.

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 of the heapsort algorithm, and
3. shows and briefly discusses your test results for Heapsorter.

Step 2: Mergesort and Quicksort

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.

HybridSorter In the literature, one can find the claim that insertion sort is quick for small input arrays because of its simplicity. In theory, this may make it possible to speed up quicksort by stopping the recursion when the input size drops below a certain threshold, and calling insertion sort instead. Implement this variant of quicksort and try to determine a good threshold value. For this, add a constructor HybridSorter(int threshold) to your implementation and use the Tester to find such a value.

Extend your report by a section in which you explain your implementations and findings. In particular, this section should discuss your implementation of Mergesorter and argue why it is correct. Explain also how you determined a good threshold value for HybridSorter, and include a discussion of a test that compares the efficiency of the five sorters (Heapsorter included). 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
Figure 2: Running times of the sorters in Figure 2 applied to smaller arrays.

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).

Step 3: Sorting Almost Sorted 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, the more unsorted the input array is, the more efficient will quicksort become. Here, you will implement a little experiment that tries provide evidence for the correctness of this hypothesis. Maybe the most interesting question is how we can easily create arrays that are “more and more unsorted”. You may wish to think a bit about this question before continuing to read.

A simple solution one may came up with works as follows: As before, we use random arrays. However, before passing them to the sorter we are testing, we sort them with a certain degree of carelessness. For this, we may use the quicksort variant from the textbook, whose partition method puts an array element $a[i]$ into a new place by means of swapping if $a[i] < \text{pivot}$ (or, equivalently, if $\text{pivot} - a[i] > 0$). In our variant, we do the same, but only if $\text{pivot} - a[i] > t$, where $t$ is a threshold value. In other words,

1 You may have to use the -Xmx option of the java virtual machine to avoid an out of heap space error.
2 There will certainly be better ones, especially from the mathematical point of view, but this one has the advantage of being easy to implement.
the sorting algorithm does not bother putting $a[i]$ into the right place if it is not considerably smaller than the pivot, and the value of $t$ determines what considerably exactly means. If $t$ is zero, we obtain a completely sorted array, if it is a small positive value, there will be some errors in the “sorted” array, and if it is Integer.MAX_VALUE then no sorting at all takes place.

Your task is to implement this variant of Quicksort. The implemented class, a sorter called ThresholdQuicksorter, should have a constructor to which the threshold is passed as a parameter. Furthermore, implement a variant of Tester that tests only one sorter (namely Quicksorter), but for different values of $t$. For this, it has additional constants low, hi, and num. When it is executed, it runs tests on arrays of increasing size, precisely as before. However, for each array, it measures the running time on num lesser and lesser sorted copies of this array, where the degree of sortedness is given by num thresholds between low and hi. Thus, the diagram you obtain shows num graphs, each corresponding to one threshold.

Extend your report by a section that describes what you have done, and discuss your test results.

Step 4: The Final Touch

Re-run the tests of the two preceding steps (possibly with slightly adjusted parameters to complete the tests in reasonable time) by executing the Java virtual machine in interpreted mode (option –Xint). In a new section of your report, highlight and discuss the differences compared with the earlier results, if any.

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 this section.

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.

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