Halmstad University course DA4002
Introduction to Algorithms, Data Structures, and Problem Solving

Report Template for Project 1: Sorting Algorithm Benchmarks

Student Name One <student.email@one>
Student Name Two <student.email@two>

September 25, 2013

The report is an essential part of the project work. It counts for approximately half the points. The structure of this template should be followed to write your report. *Most aspects of the report are clearly specified in this template, others are more open-ended and you have to decide how best to structure them to illustrate what you have done and what conclusions you draw from your results.*

The report should start with a short summary of what you have done (which tasks and which algorithms you have chosen). Then follows a section on individual algorithm benchmarks, each under its own section heading. It should contain at least one plot of the runtime in function of problem size. If you did the second mandatory task, this plot should show the minimum, average, and maximum runtimes. If you did the third mandatory task, you should add a plot that illustrates how well your empirical measurements match the theoretical complexity. The plots should be followed by a short text (or bullet list) which

- *specifies references that you used;*
- *explains where to find your implementation in your code;*
- *states how to reproduce your plots;*
- *finishes with a more free-form text about other things you wish to point out, for example where to find test code and what you conclude from the plots.*

After the individual algorithms, you should write a section which compares them with each other. This should include some plots that clearly show how the algorithms in relation to each other, possibly using various ranges on the axes or even logarithmic scale if you think that is the most useful. The figures should again be followed by a short text that explains how to reproduce the plots with your program, and concludes with another open-ended free-form text. If you did the first and/or second bonus task, include them in the individual and comparison sections. If you did the third bonus task, choose your own reporting format such that it properly conveys your insights.

Do not forget to

- ***give the course name, project title, and student names;***
- ***specify references to all the sources you have used, for example links to websites were you found pseudo-code;***
- ***number all the figures and give them short descriptive captions;***
- ***generally include all the information necessary to reproduce your results with your code;***
- ***finish the report with a short overall conclusion.***

Summary

This template report illustrates how to present your project results. It relies on insertion sort and merge sort, which are included in the project assignment. It does not include a section on bonus task three. Students who do that bonus task are required to come up with their own structure to clearly demonstrate their findings.

Individual Algorithms

Insertion Sort

The implementation of insertion sort can be found in the source code `insertion-sort--benchmark.c` in the `insertion_sort()` function. It is based on code examples readily available on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Insertion_sort`.

Figure 1 shows the minimum, average, and maximum runtimes measured by running insertion sort on five different random arrays that are initialized at the beginning of the `main()` function[1]. The benchmark prints the measured times in individual columns, with the minimum, average, and maximum printed in the last three columns. The figure can be reproduced with the following Gnuplot commands, assuming you have saved the benchmark output into a file called `data`.

```
set key left
set title 'insertion sort on random data'
set ylabel 'time [ms]'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
plot 'data' u 1:7 w lp t 'min', '' u 1:8 w lp t 'avg', '' u 1:9 w lp t 'max'
```

According to Wikipedia, the theoretical complexity of insertion sort is $O(N^2)$. Figure 2 shows that, if the measured runtimes are divided by N^2, the resulting values do converge to a non-zero constant. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
O(N) = N * N
set key right
set title 'insertion sort on random data, convergence for O(N^2)'
set ylabel 'time [ms] / (N^2)'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
set yrange [0:2e-6]
plot 'data' u 1:($7/O($1)) w lp t 'min', \
         '' u 1:($8/O($1)) w lp t 'avg', \
         '' u 1:($9/O($1)) w lp t 'max'
```

Merge Sort

The implementation of merge sort can be found in the source code `merge-sort-benchmark.c` in the functions `merge()`, `msort_rec()`, and `merge_sort()`. Examples of merge sort implementations are easily found on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Merge_sort`. In our implementation, the `merge_sort()` function is

[1]Note that the code included in the project assignment does *not* do this, it is part of mandatory task 1 that you have to do. Also note that five runs may not be enough to reduce the noise in the measurements, this is for you to judge based on your measurements.

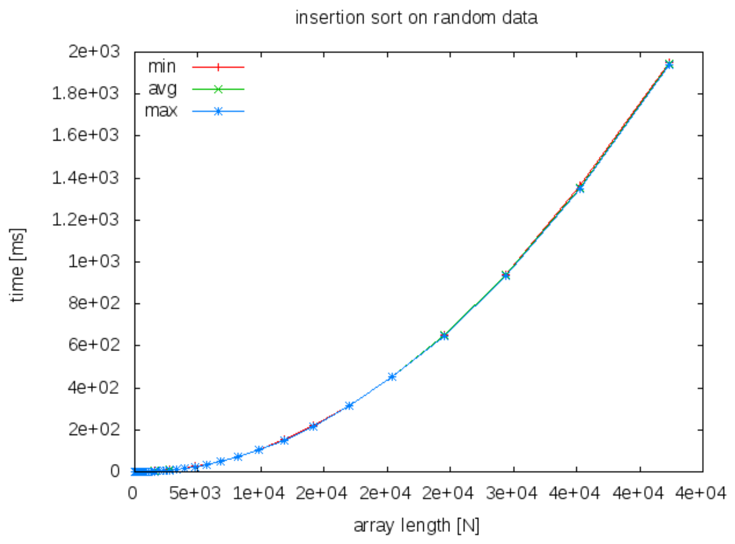

Figure 1: Runtimes of insertion sort on various arrays sizes.

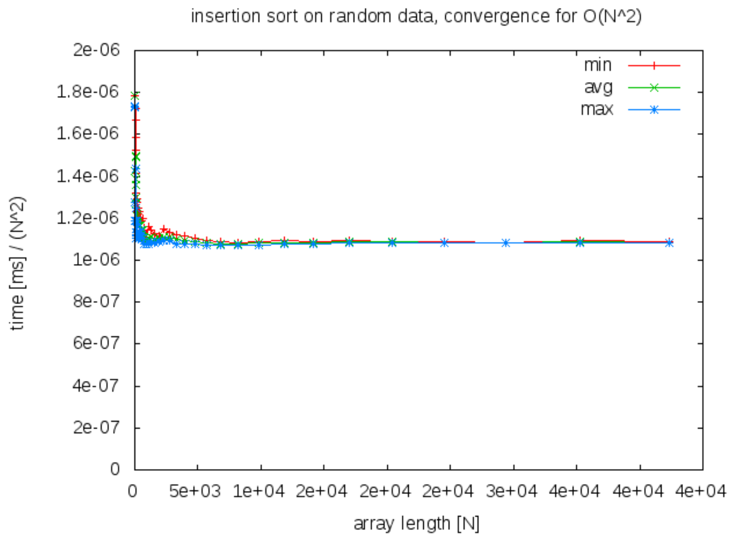

Figure 2: Dividing the measured runtimes of insertion sort by N^2 shows a clear convergence to a non-zero value.

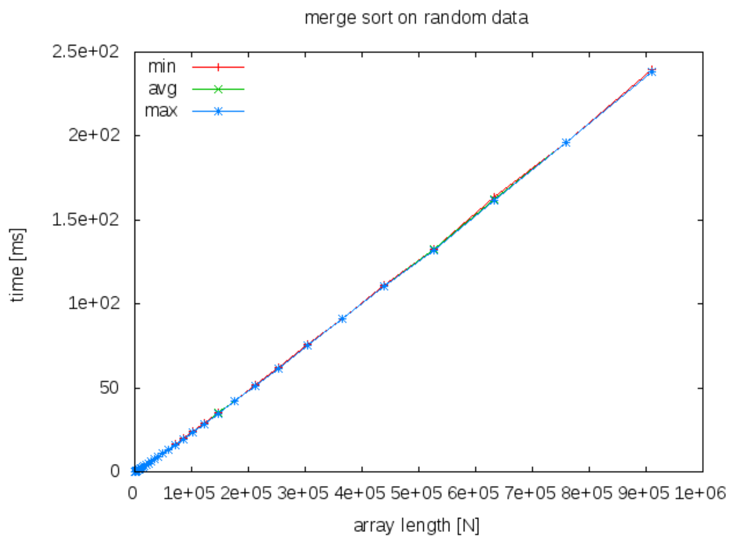

Figure 3: Runtimes of merge sort on various arrays sizes.

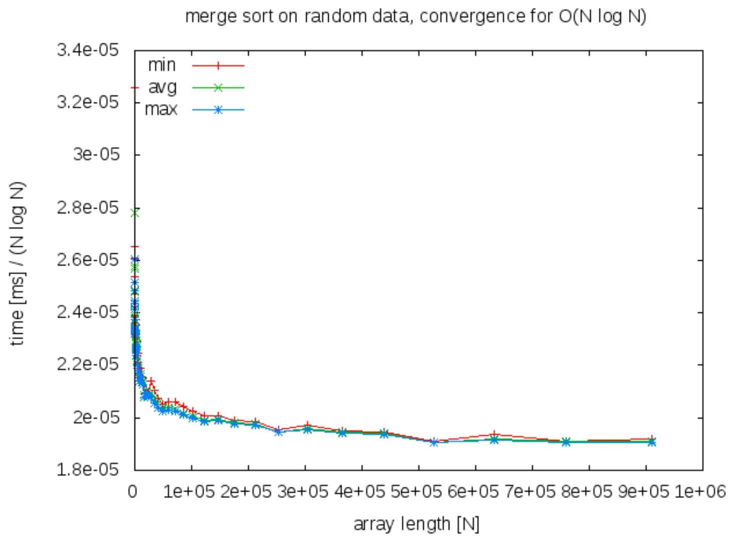

Figure 4: Dividing the measured runtimes of insertion sort by $N \log N$ shows a clear convergence to a non-zero value.

the high-level entry point. It allocates a temporary array, calls the recursive `msort_rec()` which implements the actual algorithm, and then frees the temporary array. The `merge()` function takes two arrays (stored within the same memory range but at different indices) that are already sorted, and produces a merged sorted array from them.

Figure 3 shows minimum, average, and maximum runtimes over five different runs, similarly to what was done with insertion sort (see the previous section for details). The figure can be reproduced with the following Gnuplot commands.

```
set key left
set title 'merge sort on random data'
set ylabel 'time [ms]'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
plot 'data' u 1:7 w lp t 'min', '' u 1:8 w lp t 'avg', '' u 1:9 w lp t 'max'
```

According to Wikipedia, the theoretical complexity of insertion sort is $O(N \log N)$. Figure 4 shows that, if the measured runtimes are divided by $N \log N$, the resulting values do converge to a non-zero constant. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
O(N) = N * log(N)
set key left
set title 'merge sort on random data, convergence for O(N log N)'
set ylabel 'time [ms] / (N log N)'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
plot 'data' u 1:($7/O($1)) w lp t 'min', \
         '' u 1:($8/O($1)) w lp t 'avg', \
         '' u 1:($9/O($1)) w lp t 'max'
```

Algorithm Comparison

Figure 5 shows the average runtime measurements for all implemented algorithms. It can be produced with the following Gnuplot commands, assuming that the insertion sort runtime data is stored in a file called `isort.data` and the merge sort data in `msort.data`:

```
set key right
set title 'average runtimes on random data'
set ylabel 'time [ms]'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
plot 'isort.data' u 1:8 w lp t 'insertion sort', 'msort.data' u 1:7 w lp t 'merge sort'
```

To better separate the slow-growing from the fast-growing curves, Figure 6 shows the same data on logarithmic scales. It can be produced with the following commands:

```
set key left
set title 'average runtimes on random data (logarithmic scales)'
set ylabel 'time [ms]'
set xlabel 'array length [N]'
set format x '%.1g'
set format y '%.2g'
set logscale xy
plot 'isort.data' u 1:8 w lp t 'insertion sort', 'msort.data' u 1:7 w lp t 'merge sort'
```

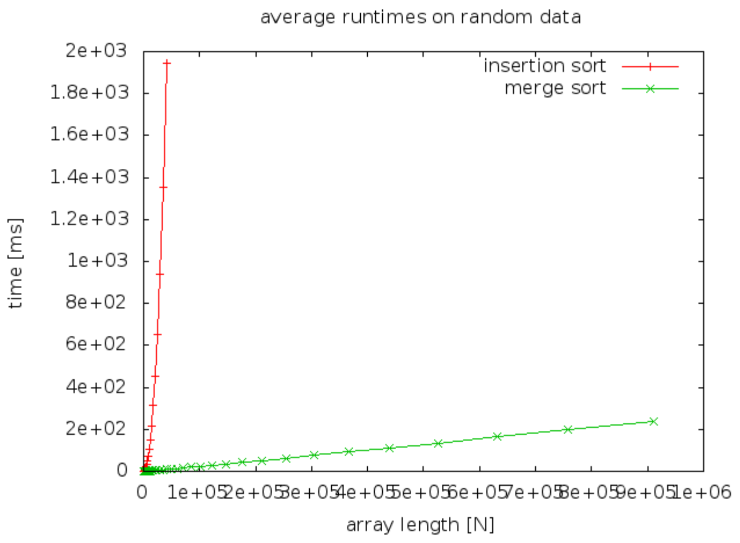

Figure 5: Average runtimes of insertion sort and merge sort on various array sizes.

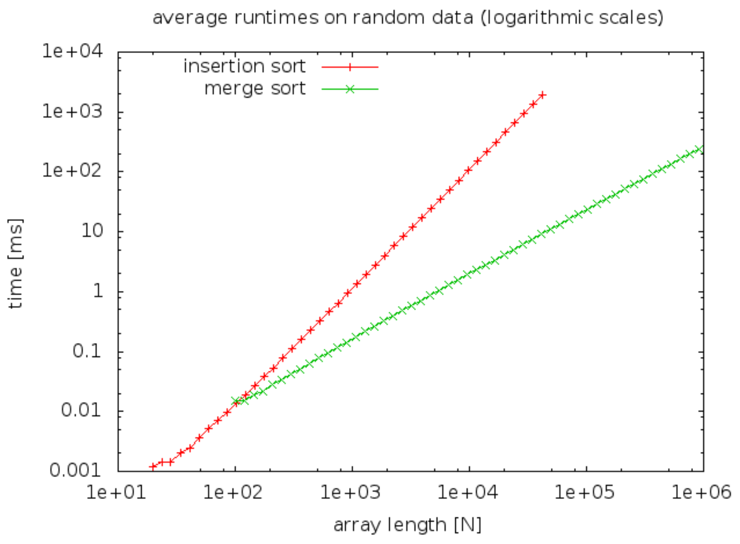

Figure 6: The same data plotted with logarithmic scales for both N and T.

These figures clearly show that merge sort is orders of magnitude faster than insertion sort on the larger problem sizes. This is as expected, given that merge sort has a lower runtime complexity of $O(N \log N)$ whereas insertion sort is $O(N^2)$ which grows much faster. On the other hand, both algorithms take approximately the same time for $N \approx 100$. It can be expected that for smaller N, insertion sort would actually run faster than merge sort.

Conclusion

The runtime measurements performed with our benchmark programs for insertion sort and merge sort shows a good match between theoretical and empirical runtime complexity on random data. In our measurements, the minimum, maximum, and average values were close together, so the average was chosen in the comparison section to represent each algorithm. If more noise is present, it can be expected that using the minimum would be the best representative.