Introduction to Algorithms, Data Structures, and Problem Solving

DA4002 (HT12) Halmstad University

Example Report with Guidelines
Project 1: Sorting Algorithms Benchmark

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

September 28, 2012

1 Introduction

A short written report is an essential part of the project. It must clearly identify the team and present the work performed by them. This example report uses the two benchmark applications given to students when they start working on project 1. It also gives some additional guidelines in paragraphstypeset in italics, like the following:

> *The introduction is the place to introduce the problem and briefly state what results have been achieved. This means stating which mandatory and bonus tasks you have performed. For example, list the names of the sorting algorithms that you have implemented, and whether you have investigated the dependence of runtime on the type of input data.*
>
> *In case you hand in a partial solution, this should already be mentioned in the introduction, but details should wait until the implementation or results section. If you need rather long explanations for some of these aspects, you can put that into a separate discussion section.*

2 Implementation

> *The implementation section should introduce any new files and functions, and explain how you have integrated your work with what was given at the beginning of the project. For example, if you implemented bubble sort [1] as part of your mandatory tasks*[1]*, you could write something like:*

The source file `main-bubble-sort.c` is an adapted copy of `main-insertion-sort.c` which contains our implementation of bubble sort (see function `bubble_sort`) and measures its runtime over 5 runs with different random data for array lengths from 20 to 42329. In order to manage the 5 different runs, we maintain 5 separate data arrays called `data1`, `data2`, `data3`, `data4`, and `data5`. Those arrays get initialized before the main loop. We based our `bubble_sort` function on the pseudo code provided on Wikipedia [1], shown in figure 1.

In order to produce plots, we wrote `bubble-sort.plot` and `bubble-sort-N2.plot` scripts based on the ones provided for insertion sort. They are given to gnuplot [4] and expect the measured runtimes of bubble sort to be stored in a file called `bubble-sort.data`. Both plot scripts, and the data file which we used to produce the figures in this report, are included in our source archive.

[1] Note that bubble sort is *not* on the list of algorithms you can choose for the mandatory tasks, because it was part of an earlier exercise.

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
      newn = 0
      for i = 1 to n-1 inclusive do
         if A[i-1] > A[i] then
            swap(A[i-1], A[i])
            newn = i
         end if
      end for
      n = newn
   until n = 0
end procedure
```

Figure 1: Pseudo-code for bubble sort, taken from Wikipedia [1].

> *Assuming that you performed the bonus task of investigating the influence of input data distribution, you could write something like the following:*

For insertion sort and merge sort, we also investigated the dependence of runtime on the distribution of values in the input data. We achieved this by maintaining four data arrays in our benchmarking applications:

- `data_rnd` stores random data, just like the `data` array in the originally provided code.
- `data_asc` stores data which is sorted in ascending order. We initialize it from `data_rnd` and then pass it to our implementation of merge sort.
- `data_des` stores data which is sorted in descending order. We initialize it by copying `data_asc` in reverse order.
- `data_mix` stores a data which is chunkwise sorted, where each chunk may be ascending or descending. We initialize this data by duplicating `data_rnd` and the using the provided function `random_chunkwise_array`.

Inside the main loop, we then run the sorting algorithm on a duplicate of the first N items of each of these data arrays.

> *In case you hand in some partially solved tasks which do not compile, this should be clearly explained here in the implementation section. Which file does not compile, and why? What would need to be done in order to make it compile? Any compilation errors that are not covered by such an explanation are a serious failure.*

3 Results

> *In the results section, you have to present the obtained technical results. Thus, for each task, you have to produce one or more figures which clearly illustrate the point you are making. For example, if you found out that the performance of some algorithm gets much worse under some circumstance, then you have to include a figure which clearly shows just this effect, and mention it clearly in the text. It is also important to summarize your main findings in order to give a high-level understanding. This can be done at the end of the results section, or in a separate discussion or conclusion.*
>
> *The remainder of this section uses the provided benchmarks and example plot scripts, as well as some results that are part of the mandatory and bonus tasks, to illustrate this.*

Figures 2 and 3 show the basic running time plots for insertion sort and merge sort. They were prodcued with the `isort-example.plot` and `msort-example.plot` scripts. Comparing them,

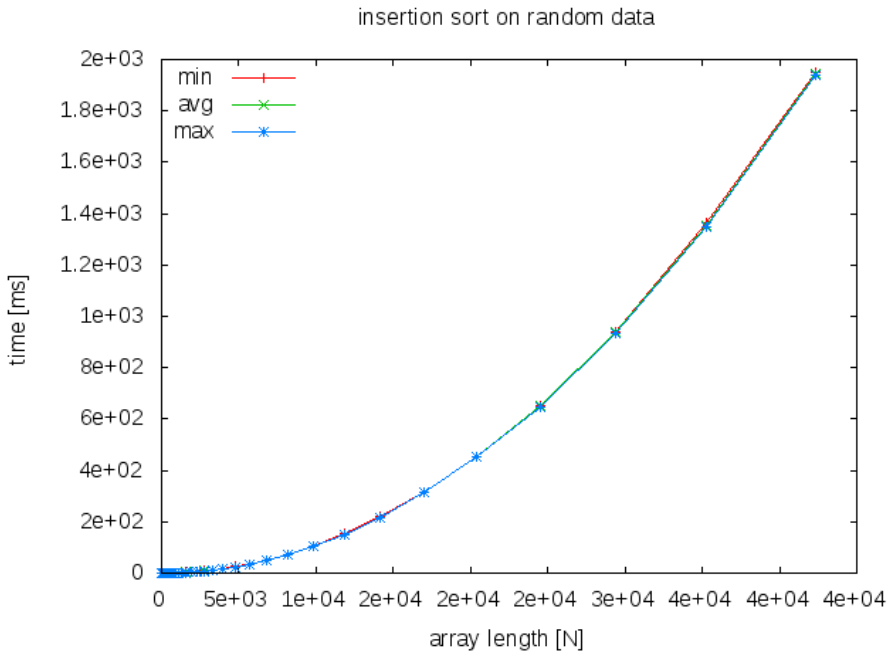

Figure 2: Runtimes of insertion sort on various arrays sizes.

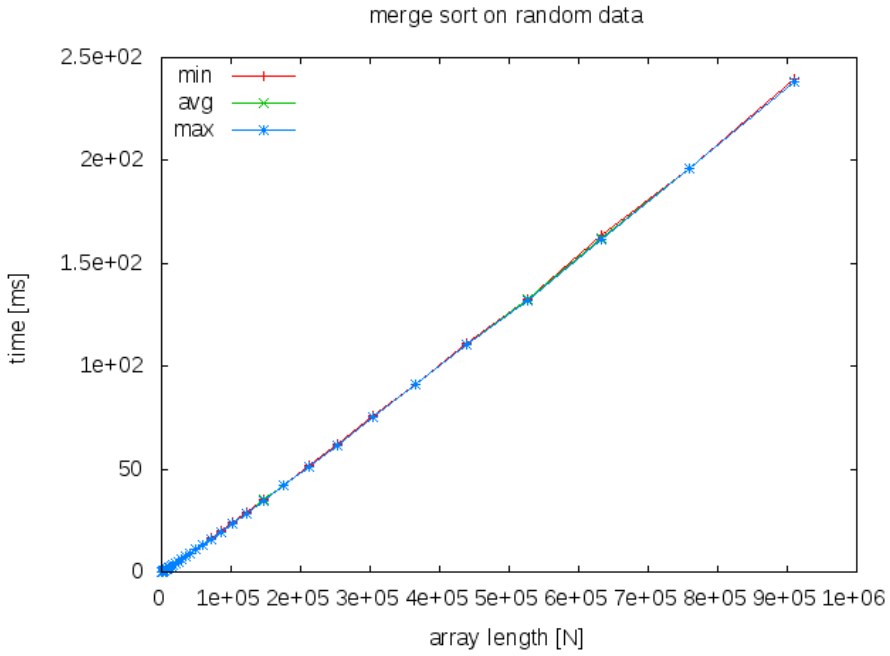

Figure 3: Runtimes of merge sort on various arrays sizes. Notice the much lower time values compared to the ones in figure 2.

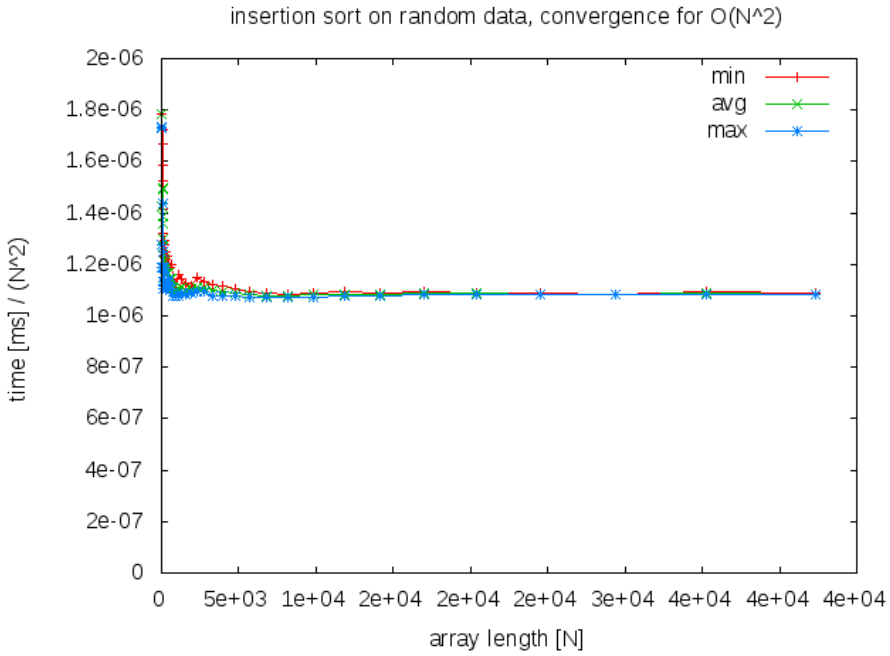

Figure 4: This plot shows how well the runtime measured by our insertion sort benchmark agrees with the theoretical complexity of $O(N^2)$.

the significant difference between these two algorithms is quite apparent. Insertion sort is much slower, and gets rapidly worse with increasing probelm size N. Indeed, insertion sort is known to be $O(N^2)$ [2] while merge sort has a complexity of $O(N \log N)$ [3]. To validate these theoretical complexities for the particular data we used in these benchmarks, we performed the procedure discussed in lecture 4.

Insertion sort has a runtime complexity of $O(N^2)$ [2]. Figure 4 shows that the empirical values measured by our benchmark on random data matches this quite nicely. Here we run 5 separate instances for each array length, log the minimum, maximum, and average runtimes, and then plot those times divided by N^2. For $N > 10^4$ this ratio converges to a non-zero value. To reproduce this figure, run the `main-insertion-sort` program and capture its output in a file called `isort-example.data`, and then use the provided `isort-example-N2.plot` script.

Similar results have been obtained for merge sort. Figure 5 follows the same procedure as for insertion sort, but this time we divide by the theoretical complexity of $O(N \log N)$ [3]. Again, we run 5 separate instances for each array length, log the minimum, maximum, and average runtimes. Here, those times are divided by $N \log N$ before plotting. For $N > 4 \cdot 10^5$ this ratio converges to a non-zero value. In order to reproduce this, run the `main-merge-sort` program and capture its output in a file called `msort-example.data`, and then use the provided `msort-example-NlogN.plot` script.

We also investigated the dependence of sorting times to the distribution of input data. The results are shown in figures 6 and 7 for insertion sort, and figure 8 for merge sort. These figures are based on the plot files `insertion-sort-idep.plot` and `merge-sort-idep.plot`. For insertion sort, it can be seen that the time required for reverse-sorted (descending) input data is twice as high as random or mixed data. And most interestingly, data that is already sorted (ascending) is much faster to sort, such that the graph practically coincides with the X-axis in figure 6. Figure 7 shows a the same measurements but with a significantly zoomed Y-axis.

> *In a real report, the results of the bonus task shown in figures 6, 7, and 8 would now have to be discussed: why is there such a strong dependency for insertion sort, but not for merge sort?*

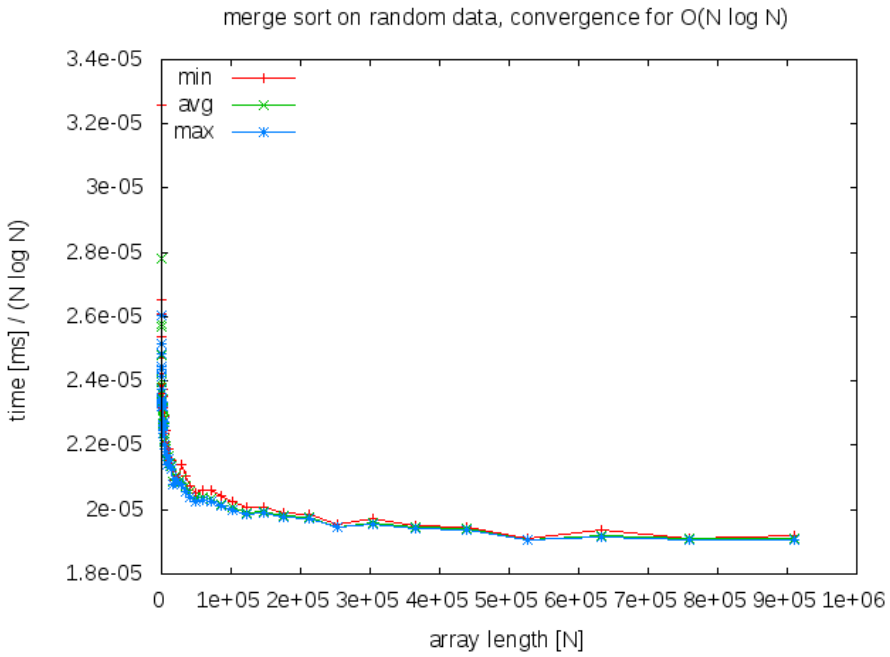

Figure 5: This plot shows that the runtime measured by our merge sort benchmark matches the theoretical complexity $O(N \log N)$.

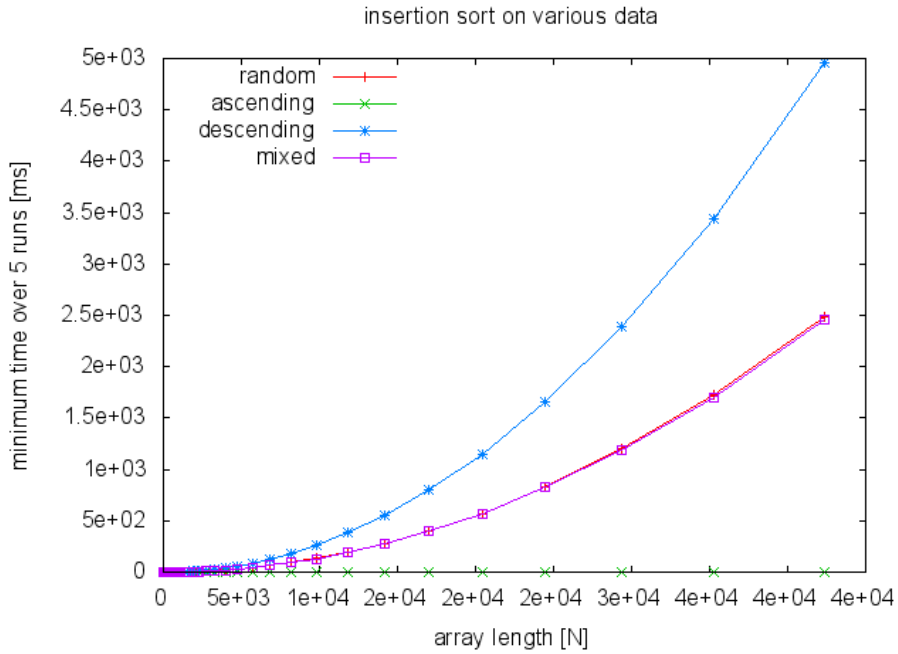

Figure 6: This plot shows that the runtime of insertion sort depends very strongly on the input data distribution.

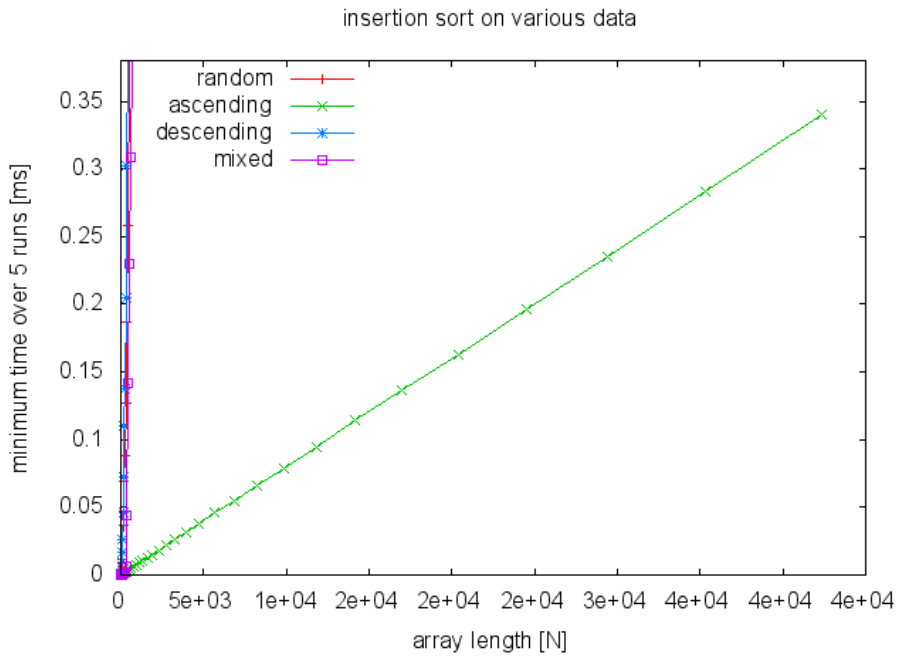

Figure 7: The same measurement as figure 6 but with a Y-axis range manually chosen to show the runtime for already-sorted input data.

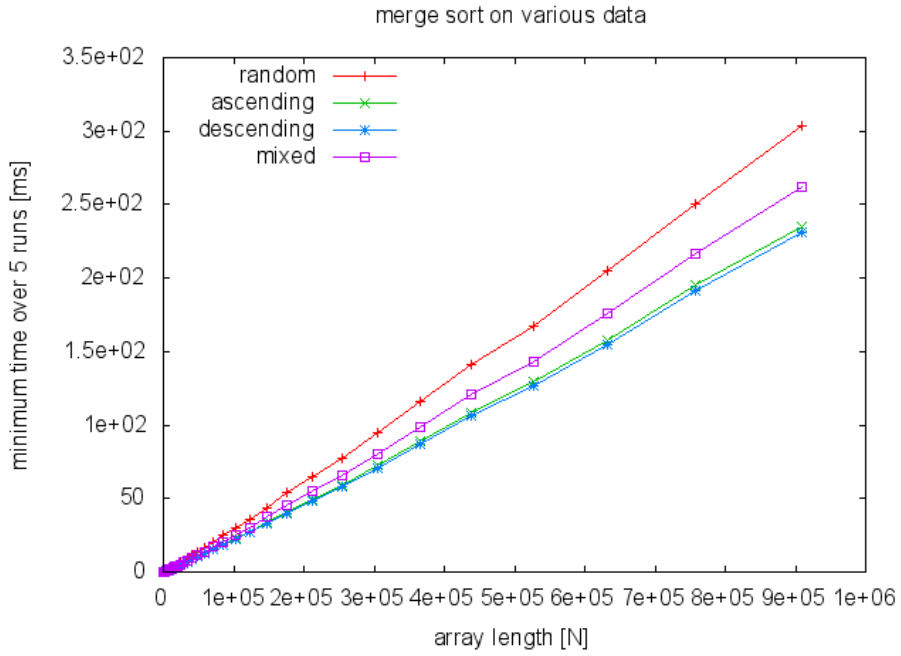

Figure 8: This plot shows that the runtime of merge sort does not significantly depend on the input data distribution.

4 Conclusion and Final Remarks

The most important points to remember about the report are:

- Mention course information: University name; course name, code, and date.
- Mention team information: names, email addresses, and study programme.
- Do not forget the project title and subtitle.
- The introduction section must summarize the work you have done.
- The implementation must present what you have added and changed, and how to run it
- The results must clearly show what you have achieved
- A separate discussion section is nice if you have many detailed results and want to give a high-level explanation.
- A conclusion is also a nice additional section, it allows you to make some overall comments and maybe point to future work.
- References must be given to all the web sites, books, and other material that you have used.

The ITADS course focuses on technical aspects of programming and problem solving. The quality of the written language in the report is secondary. As long as it remains comprehensible, English errors are not relevant. Verbatim quotes (copy-pasting) from other sources is acceptable, **if they are short, properly marked, and adequately cited**.

If language is a major obstacle for a team, it is possible to receive a deadline extension for the report only. Source code will still have to be handed in at the normal deadline. Notify the lecturer as early as possible in order to arrange a deadline extension for the report.

Halmstad University offers an **English Language Studio** for students who seek help with written English. The studio is available on Wednesdays between 9:00 and 12:00 in room F-208. The teacher is Nicholas Lloyd-Pugh `<nicholas.lloyd-pugh@hh.se>` (phone 167372).

References

[1] Wikipedia. Bubble sort. `http://en.wikipedia.org/wiki/Bubble_sort`, September 2011.

[2] Wikipedia. Insertion sort. `http://en.wikipedia.org/wiki/Insertion_sort`, September 2011.

[3] Wikipedia. Merge sort. `http://en.wikipedia.org/wiki/Merge_sort`, September 2011.

[4] Thomas Williams, Colin Kelley, and many others. Gnuplot. `http://www.gnuplot.info/`, September 2011.