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

Report Template for Project 1: Sorting Algorithm Benchmarks

Georg Michael Lexer <geolex13@student.hh.se>
Recep May <recmay13@student.hh.se>

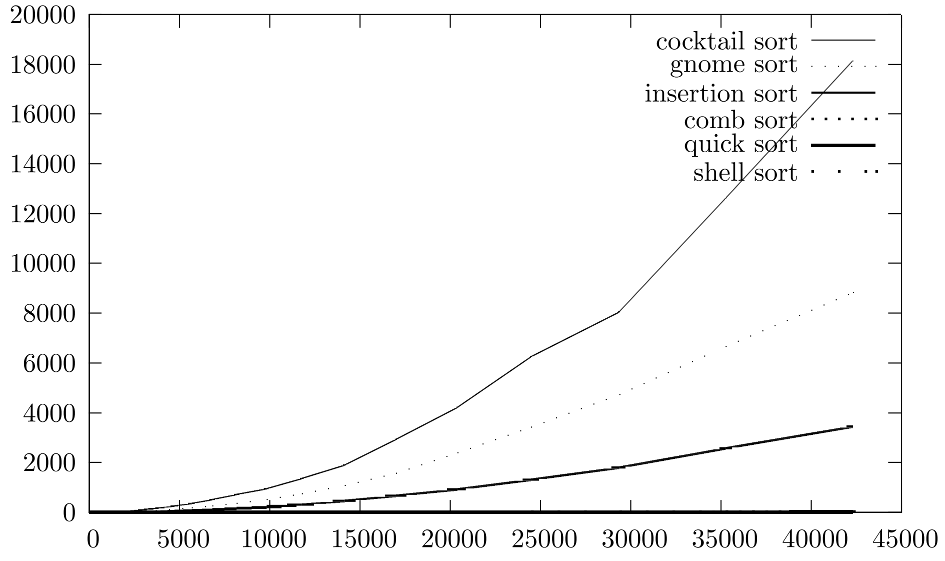
October 4, 2013

Summary

In this report you will find the results of these algorithms:

- cocktail sort
- gnome sort
- shell sort
- comb sort
- cycle sort
- quick sort

The results of each algorithm were plotted seperatly but also here we have an comparision of all the algorithms implementated.



Individual Algorithms

Cocktail Sort

The implementation of cocktail sort can be found in the source code `cocktail-sort--benchmark.c` in the `cocktail_sort()` function.

It is based on code examples readily available on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Cocktail_sort`.

Figure **??** shows the minimum, average, and maximum runtimes measured by running cocktail sort on ten different random arrays.

The figure can be reproduced with the following Gnuplot commands, assuming you have saved the benchmark output into a file called `cocktail`.

```
plot 'cocktail' u 1:2 w l t ' avgerage', '' u 1:3 w l t 'min', '' u 1:4 w l
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. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
plot 'cocktail' u 1:5 w l t ' convergence to =()
```

Gnome Sort

The implementation of merge sort can be found in the source code `gnome-sort-benchmark.c` in the functions `gnome_sort()`. Examples of gnome sort implementations are easily found on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Gnome_sort`.

The algorithm goes through the whole array and checks until it find a higher number then the one starting form, if it does the algorithm swaps those.

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.

```
plot 'gnome' u 1:2 w l t ' avgerage', '' u 1:3 w l t 'min', '' u 1:4 w l 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. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
plot 'gnome' u 1:5 w l t ' convergence to =()
```

Shell Sort

The implementation of shell sort can be found in the source code `shell-sort-benchmark.c` in the functions `shell_sort()`. Examples of merge sort implementations are easily found on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Shell_sort`.

Shell sort algorithm compares two positions in the array with each other and swaps them if the one with the lower index is smaller.

Figure 5 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.

```
plot 'shell' u 1:2 w l t ' avgerage', '' u 1:3 w l t 'min', '' u 1:4 w l t
'max'
```

According to Wikipedia, the theoretical complexity of insertion sort is $O(N^3/2)$. Figure 6 shows that, if the measured runtimes are divided by $N^3/2$. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
plot 'shell' u 1:5 w l t ' convergence to =()
```

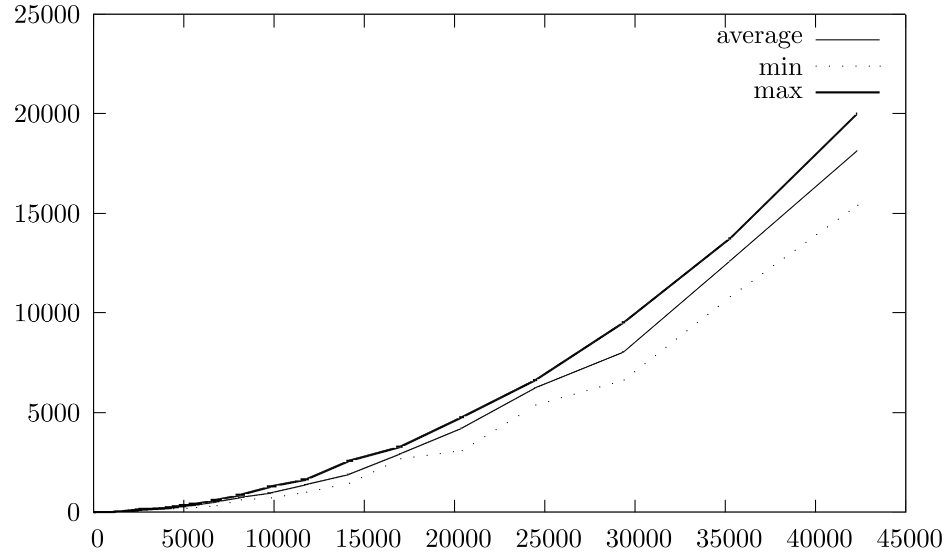


Figure 1: Runtimes of cocktail sort on various arrays sizes.

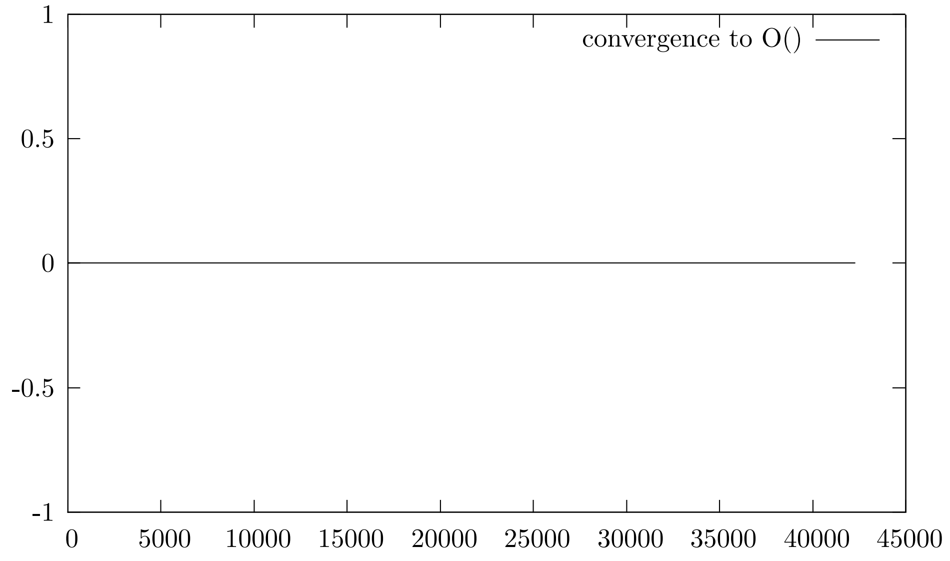


Figure 2: Dividing the measured runtimes of cocktail sort by N^2 shows a clear convergence to a non-zero value.

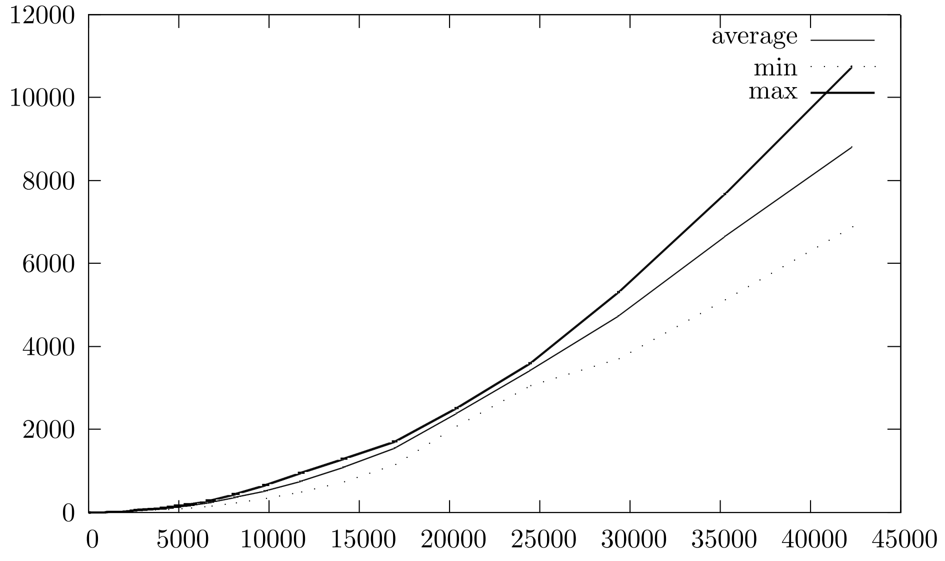


Figure 3: Runtimes of gnome sort on various arrays sizes.

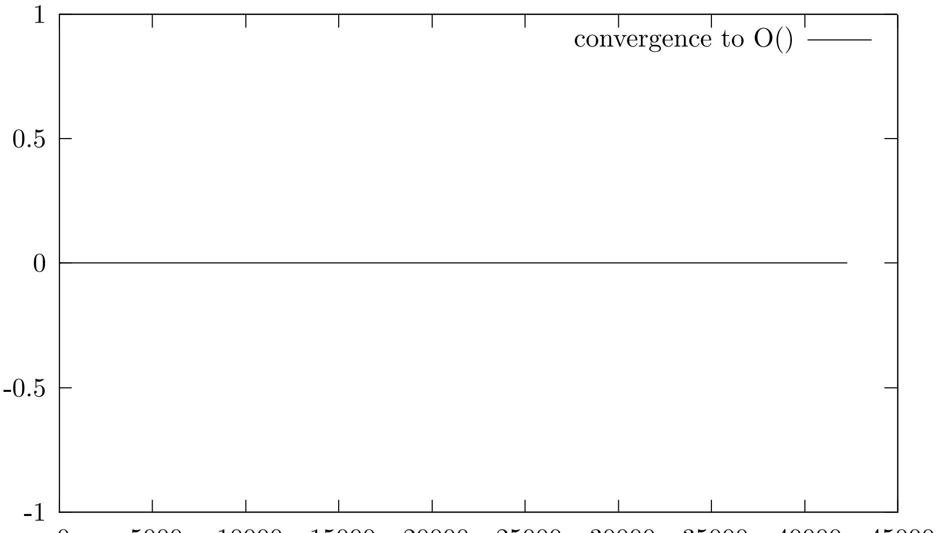


Figure 4: Dividing the measured runtimes of gnome sort by N^2.

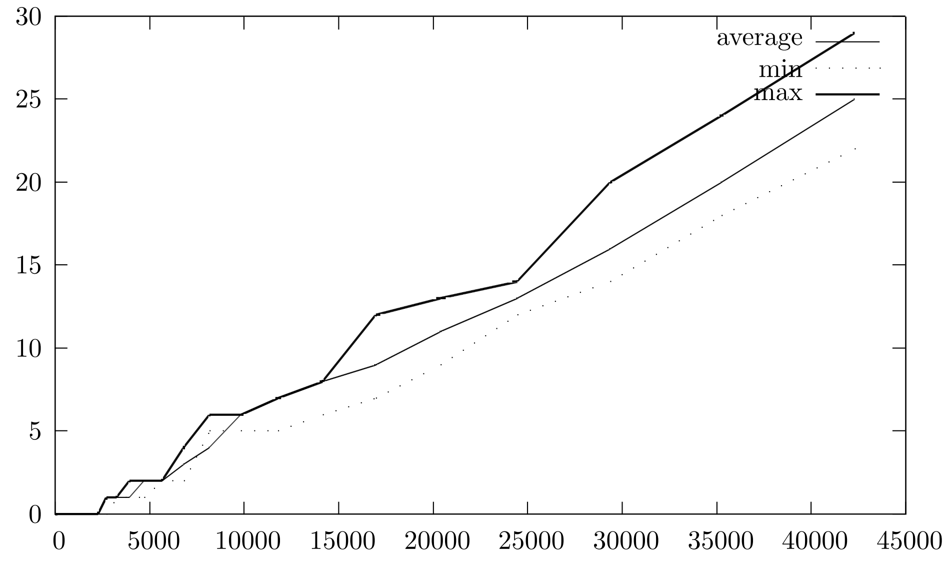


Figure 5: Runtimes of shell sort on various arrays sizes.

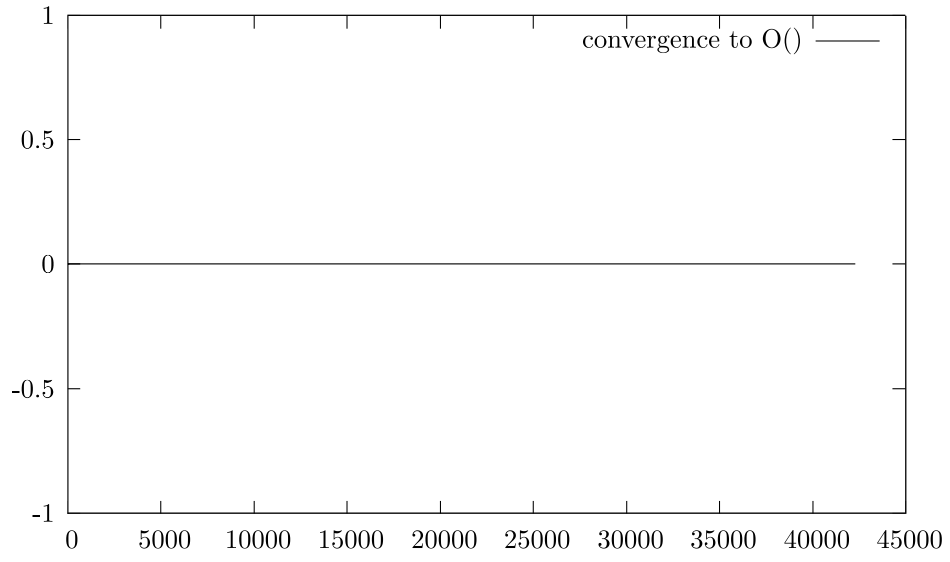


Figure 6: Dividing the measured runtimes of shell sort by $N^3/2$ shows a clear convergence to a non-zero value.

Comb Sort

The implementation of merge sort can be found in the source code `comb-sort-benchmark.c` in the functions `comb_sort()`. Examples of comb sort implementations are easily found on the internet, for example on Wikipedia at `http://en.wikipedia.org/wiki/Comb_sort`.

Figure 7 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.

```
plot 'comb' u 1:2 w l t ' avgerage', '' u 1:3 w l t 'min', '' u 1:4 w l t 'max'
```

According to Wikipedia, the theoretical complexity of insertion sort is $N \log N$. Figure 8 shows that, if the measured runtimes are divided by $N \log N$. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
plot 'comb' u 1:5 w l t ' convergence to =()
```

Quick 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/Quick_sort`.

Figure 9 shows minimum, average, and maximum runtimes over tem 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.

```
plot 'quick' u 1:2 w l t ' avgerage', '' u 1:3 w l t 'min', '' u 1:4 w l t
'max'
```

According to Wikipedia, the theoretical complexity of insertion sort is $N \log N$. Figure 10 shows that, if the measured runtimes are divided by $N \log N$. This illustrates that theory and practice match each other very closely in this case. This figure can be reproduced as follows:

```
plot 'quick' u 1:5 w l t ' convergence to =()
```

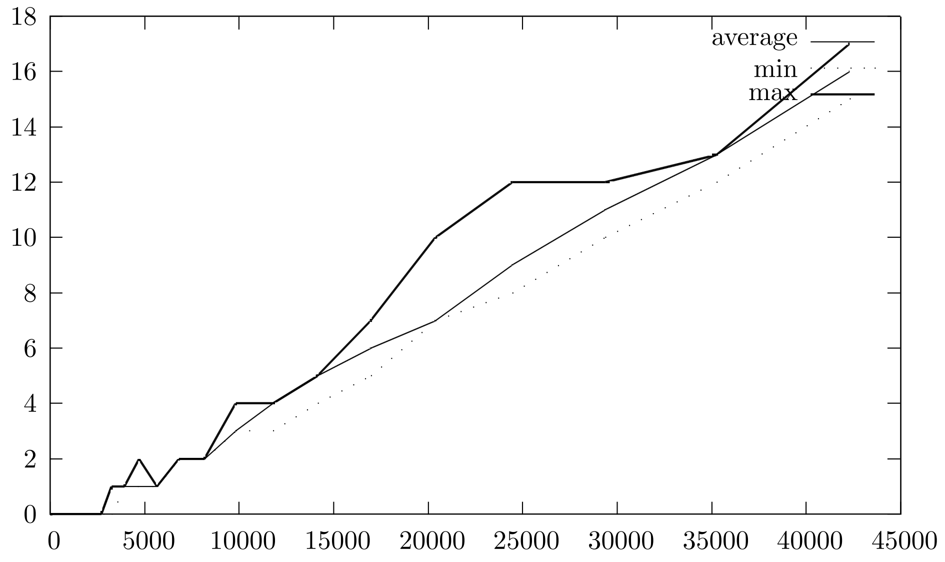


Figure 7: Runtimes of comb sort on various arrays sizes.

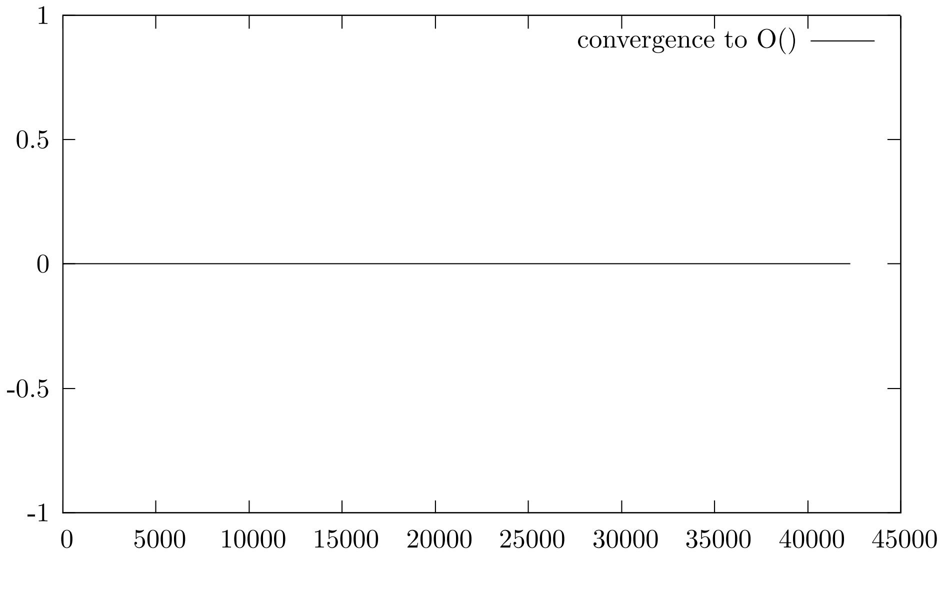


Figure 8: Dividing the measured runtimes of comb sort by $N \log N$ shows a clear convergence to a non-zero value.

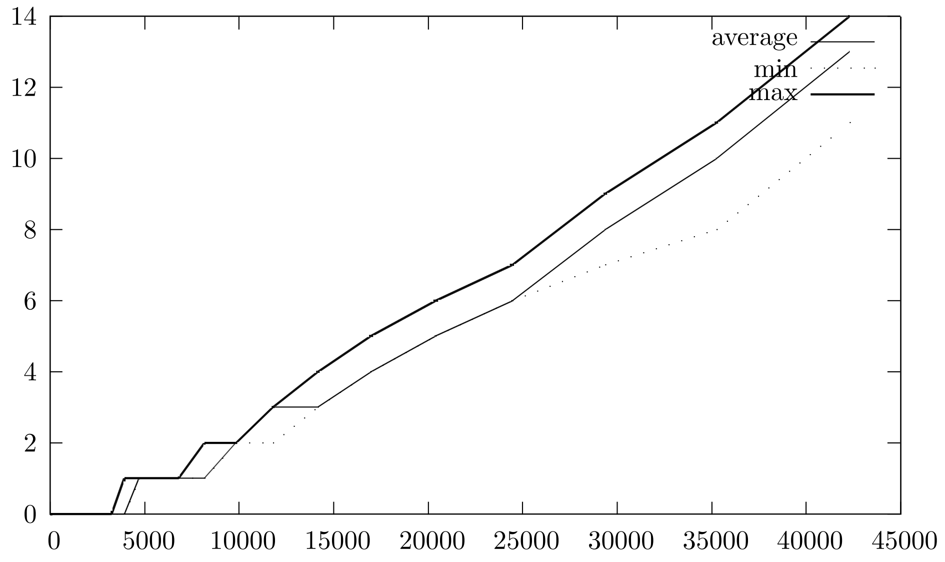


Figure 9: Runtimes of quick sort on various arrays sizes.

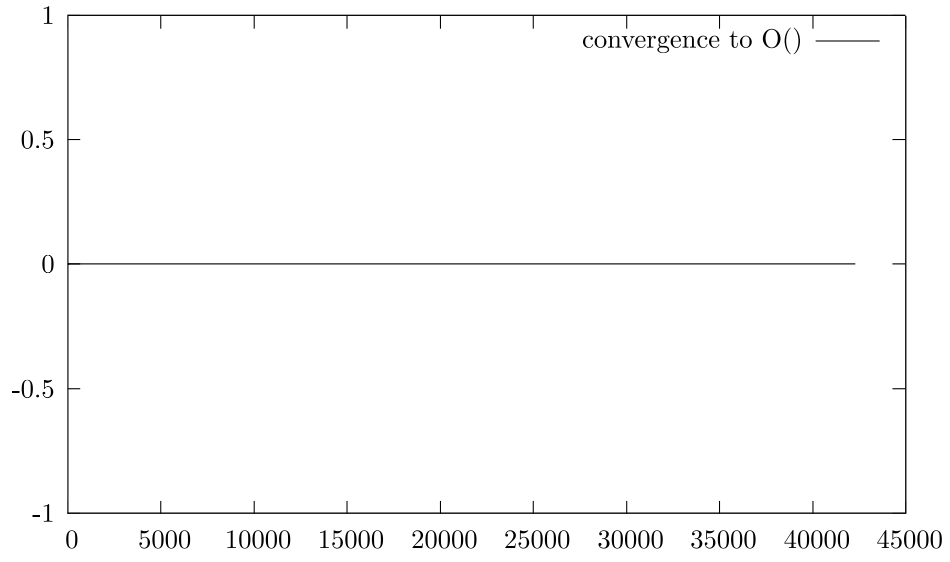


Figure 10: Dividing the measured runtimes of quick sort by $N \log N$ shows a clear convergence to a non-zero value.

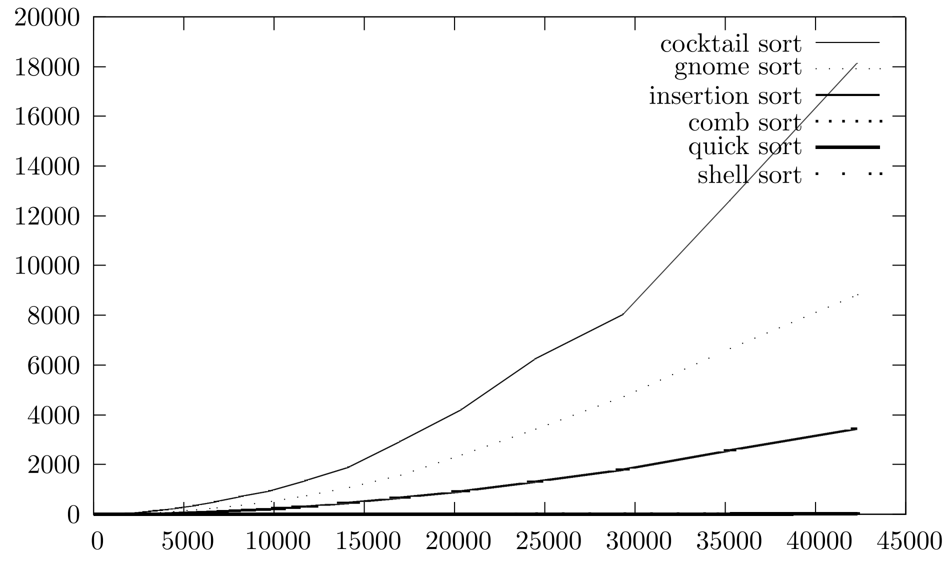


Figure 11: Average runtimes of all implemented sorting algorithms on various array sizes.

Algorithm Comparison

Figure 11 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`:

```
plot 'av' u 1:2 w l t 'cocktail sort', '' u 1:3 w l t 'gnome sort', '' u 1:4
w l t 'insertion sort', '' u 1:5 w l t 'comb sort', '' u 1:6 w l t 'quick sort',
'' u 1:7 w l t 'shell sort'
```

These figures clearly show that quick sort is orders of magnitude faster than any other sort algorithms on the all problem sizes. This is as expected, given that quick sort has a lower runtime complexity of $O(N \log N)$ whereas the other sort algorithms have $O(N^2)$ or $O(N^3/2)$which grows much faster. On the other hand, both algorithms take approximately the same time for $N \approx 100$ up to $N \approx 1.000$.

Conclusion

The runtime measurements performed with our benchmark programs for the sort algorithms show 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.

We can also see that algorithms like cocktail sort and gnome sort should not be used for larger sized arrays.