Benchmark Sorting Algorithms Project

Hongpeng Zhang
hongpeng.zhang@nmbu.no

Martin Nesse
martin.bergsholm.nesse@nmbu.no

Min Jeong Cheon
min.jeong.cheon@nmbu.no

ABSTRACT

In this paper, we analyse the performance of two well-known sorting algorithms, Insertion Sort and Merge Sort. We benchmark the performance of the different algorithms in the worst, average and best case. We compare the results from the benchmark to the theoretical performance of each algorithm for the three different cases: the worst, the average, the best.

1 INTRODUCTION

Sorting algorithms are a special kind of algorithms that rearranges the elements in a list into an order. Numerical order and lexicographical order, either ascending or descending, are most common, but any order is possible. Sorting algorithms are an important part of our digital world, such as in searching and database algorithms. Developing and benchmarking these algorithms are therefore an essential part of digital development. In this paper we benchmark two famous sorting algorithms by testing their performance, measured in time, on different sizes of data. By measuring the time the two algorithms takes to sort lists of different sizes we can compare our results with the expected theoretical results. The performance is tested on numerical lists that is to be sorted in ascending order. We will test both algorithms for three different types of lists. First we test the worst case, which is a list that is sorted in descending order. Then we run the sorting algorithms on randomized lists of fixed length to find the average performance. Lastly, the best case, which is a list that is already sorted. The logic behind the two sorting algorithms and the practical performance tests are explained in more detail in the Methods-part. In the Results-section we present the results of the benchmarking of the two algorithms. Lastly, in the Discussion-section we compare our results with the expected theoretical results.

2 METHODS

In this section, we go through how the experiment was carried out in detail. First we go through how we generated the data and which structure the test data has. Then we go through how we measured the runtimes and what hardware was used.

2.1 Generating data

Generating high quality data for testing the performance of the two sorting algorithms is essential. We used the Numpy Random generator [1] for generating lists of uniform distributed data with varied length. We chose to generate data with uniform distribution, which means that every number in the list only occurs once. Other distributions could have been used instead of the uniform distribution without affecting the results, but we chose the generate all the test data using the same distribution.

[1]Documentation for the Numpy Random Generator can be found here: https://numpy.org/doc/stable/reference/random/generator.html

2.2 Executing benchmarks

We chose to increase the length of the lists used for performance testing by 2^1, meaning that the length of the first list is 2^0, the second list 2^1, the third, 2^2 and so on until 2^{14}. Each sorting algorithm is then run on the worst case list (sorted in descending order) of length n, then the average list of length n and last the best case (sorted in ascending order) list of length n. For each run the time the algorithm uses is stored along with the length of the list. Then the process is repeated for lists of length $n + 1$ until the length of the lists are 2^{14}.

2.3 Measuring runtimes

In order to get precise measurements of the time the different algorithms spent on sorting lists of different sizes we used the built-in timer function in Python called "timeit" [2]. We built a function using timeit to record the amount of time spent sorting a list and used this function in a for-loop to loop over the lists of different length. Code for this setup is to be found in the appendix.

2.4 Hardware used

Table 1: Hardware

OS	Windows 10 21H2
Processor	Intel(R) Core(TM) i7-9750H CPU
Clock(GHZ)	2.60
Cores/CPU	6
Threads/CPU	12
RAM	16GB DDR4 2666MHZ

3 RESULTS

Table 2 shows the aggregate results of the performance tests. We see that Insertion sort performs better than Merge sort in the best case scenario, which is when all the objects in the list are already sorted in ascending order. In both the average and the worst case, Merge sort outperforms Insertion sort.

Table 2: Benchmark of algorithms

Algorithm	Best case	Average case	Worst case
Insertion Sort	n	n^2	n^2
Merge Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$

[2]Documentation for timeit can be found here: https://docs.python.org/3/library/timeit.html

3.1 Results of Insertion Sort

Figure 1 shows the performance of Insertion sort in the three different scenarios, best, average and worst case. The figure also shows the nonlinear function n^2 and the linear function n. The performance of Insertion sort on both the average and the worst case follows the n^2-function, while the performance in the best case follows the n-function. The performance in the average case is slightly better than the worst case in most of the different lengths of the lists.

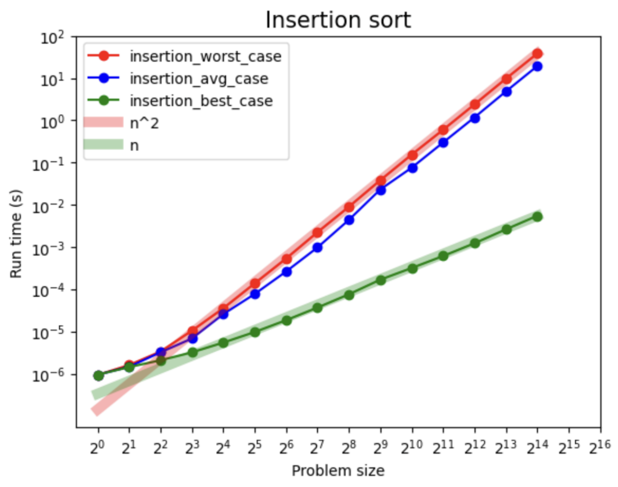

Figure 1: Benchmark results for Insertion sort

3.2 Results of Merge Sort

Figure 2 shows the performance of Merge sort on the same scenarios as the Insertion sort in figure 1. We see that the performance of the Merge sort aligns with the $n \log n$ function included in the plot. The performance of the Merge sort is not affected by whether a list is sorted or not. The slight deviation in performance from the $n \log n$ function is on lists of short length.

4 DISCUSSION

In this section we compare our results with the expectations from theory. Table 3 shows the results from our experiments in addition to the expected theoretical result. In order to limit the length of this paper, we will not explain the details of neither Insertion nor Merge sort. We assume that any reader of this report already has a basic knowledge of these algorithms.

Table 3: Time complexity

Algorithm	Best case	Average case	Worst case
Insertion Sort	n	n^2	n^2
Insertion Theoretical	n	n^2	n^2
Merge Sort	$n\log(n)$	$n\log(n)$	$n\log(n)$
Merge Theoretical	$n\log(n)$	$n\log(n)$	$n\log(n)$

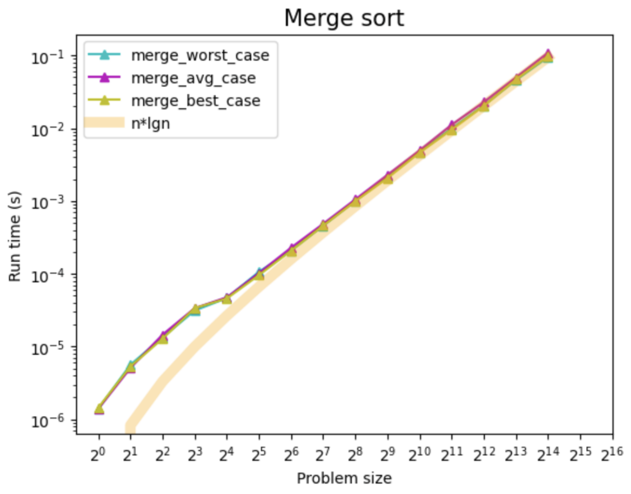

Figure 2: Benchmark results for Merge sort

4.1 Discussion of Insertion Sort

Since Insertion sort consists of a while-loop within a for-loop, the performance in the worst case is naturally n^2, since the while-loop then must run through the complete list for every repeat of the for-loop. From table 3 we see that we achieved the same result in our testing of the Insertion sort. In the best case, the while-loop is not executed at all, resulting in a performance of n. We see from table 3 that the result from our experiment is similar to the expected theoretical result. In the average case of Insertion Sort, the while-loop is assumed to run halfway through the list every time. From Figure 1 we see that the performance of Insertion sort in the average case is slightly better than the worst case, but the running time is still to be classified as n^2. Also this result is in line with the expected theoretical result.

4.2 Discussion of Merge Sort

Merge sort works by splitting the list that is to be sorted into $\log n$ pieces. Then all the pieces are assembled together, resulting in a run time of $n \log n$ regardless of best, worst or average case. We see from table 3 that we get similar results testing Merge sort. From figure 2 we have that the results from our performance test have a slightly longer run time than the theoretical expected result when the length of the lists are quite short. As soon as the lists increases in length, the run time of the Merge sort we benchmarked and the expected theoretical results coincide more and more. This is also the case for the Insertion sort in the best case.

5 CONCLUSION

We find that the run time of our experiments coincides with the expected theoretical results regarding both Insertion and Merge sort.