Benchmarking sorting algorithms in Python

INF221 Small Project, NMBU, Spring 2023

Peder Ørmen Bukaasen
peder.ormen.bukaasen@nmbu.no

Bård Tollef Pedersen
bard.tollef.pedersen@nmbu.no

Eivind Lid Trøen
eivind.lid.troen@nmbu.no

ABSTRACT

In this paper, we analyze the performance of three different sorting algorithms. The algorithms chosen for this paper are the Quick sort, Merge sort, and Insertion sort.

1 INTRODUCTION

Sorting is by many computer scientists considered one of the fundamental problems in the study of algorithms. For many applications sorting is a necessary step for the functionality of the respective application, an example of this can be rendering graphics where objects are placed on top of each other, these objects then have to be sorted by some "height" metric [Cormen et al. 2009]. In this study, three different sorting algorithms were tested and compared to their given time complexities. The respective sorting algorithms were Insertion sort, Merge Sort, and Quick sort.

2 METHODS

2.1 Theory

Measuring the performance of different sorting algorithms can be done by calculating the space and time complexity of the algorithm. For this paper, only the time complexity is considered. Time complexity can be denoted into three different categories, worst case denoted by O notation, average case denoted by Θ notation and best case denoted by Ω notation.

Table 1: In the table above the time complexities for Insertion sort, Merge sort and Quick sort is shown[Geeks for Geeks 2024c].

Algorithm	Best case	Average case	Worst case
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Quick sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$

Insertion sort is an efficient sorter for smaller datasets and is appropriate to use when datasets are partially sorted. This algorithm sorts in place and therefore has a constant space complexity. Table 1 shows that the best case for the insertion sort is when the list is already sorted, the worst case is when the list is sorted in reverse order. Table 1 shows that the average case has the same time complexity as the worst case[Geeks for Geeks 2024a]. Merge sort is an efficient sorter for larger datasets since it has the guaranteed worst-case time complexity of $O(n \log n)$. However Merge sort is not an in-place sorting algorithm, the algorithm uses more memory and therefore has a larger space complexity. For Merge sort there is no difference between best, worst, and average case since the algorithm executes the same steps regardless of the input[Geeks for Geeks 2024b]. Quick sort is efficient for large datasets and has a better space complexity than Merge sort. The best case for the Quick Sort algorithm is when the pivot is selected in a way that splits the list into equal halves. Table 1 shows that the average time complexity is the same as the best case. The worst case for Quick Sort is when the pivot is selected in a way that the partitions are highly unbalanced [Geeks for Geeks 2023].

2.2 Algorithms

All the different sorting algorithms were written in python based on the pseudo-code from the book *Introduction to Algorithms Third Edition* [Cormen et al. 2009]. The algorithms were tested with simple lists to check that the lists got sorted properly.

2.3 Data generation

In order to test the algorithms, appropriate testing data was generated using numpy. The sorted list was first created as an ordered ascending list. The reversed list was created by copying and reversing the sorted list. The average list was created by copying and randomly rearranging the sorted list. Each type of list was stored in its own CSV file, with a new line per list.

2.4 Benchmarking

The benchmarking was executed in Jupyter Notebook on a normal laptop computer using the lists stored in CSV files. The laptop was connected to a charger for the entire benchmarking process. For each algorithm, one type of list is sent to the benchmarking function at a time. All lists of that type are timed for that algorithm and the results are stored in a CSV file, before proceeding to the next type of list. When done with one algorithm, the program proceeds to the next.

The actual runtime of the algorithm was measured using the Timer class of the timeit module. This helps us record the precise runtime of the algorithms and avoid some common pitfalls during execution time measurement. It can automatically determine how many runs are needed to exceed a total runtime of more than 0.2 seconds[Python Software Foundation 2024]. This is used to adjust the timing setup to achieve accurate results for all list sizes. For the smaller lists, this can be as many as 500 000 runs. The execution time per run was found by dividing the total run time by the number of runs. Every list size was timed 7 times per algorithm to see how much the runtime varies. The results were later retrieved from the CSV files for analysis.

The following software was used for benchmarking:

- python 3.10.12
- numpy 1.26.3
- pandas 2.2.1

The laptop used for benchmarking has the following specs:

- Intel® Core™ i7-10510U CPU @ 1.80GHz × 8
- 16 Gb RAM

Table 2: Benchmark results for the three different algorithms, show the execution time in milliseconds for a list of 1024 elements for three different scenarios: sorted list, randomized list, and reversed sorted list.

Algorithm	Sorted list	Randomized list	Reversed list
Insertion sort	0.141	29.547	58.119
Merge sort	2.792	2.753	2.794
Quick sort	27.105	0.1433	44.571

- 2T SSD
- GeForce MX350

3 RESULTS

Table 2 shows that the Insertion sort uses 0.141 milliseconds for the best case 29.547 milliseconds and 29.547 milliseconds for the worst case. Merge sort executes in 2,792 milliseconds for the sorted list, 2,753 milliseconds for the randomized list, and 2,793 milliseconds for the reverse sorted list. The table also shows that Quick Sort executes in 27.105 milliseconds for the sorted list, 0.1433 milliseconds for the randomized list, and 44,571 for the reversed list.

3.1 Insertion sort

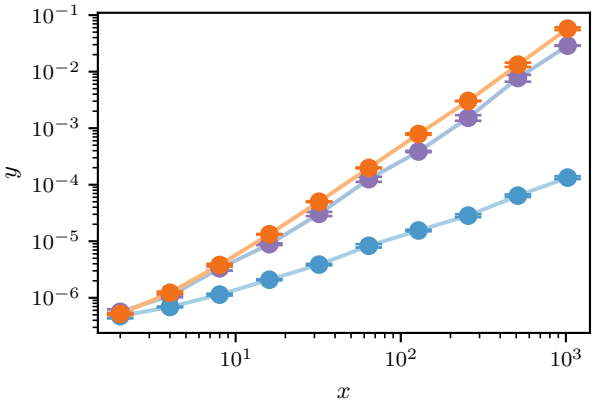

Figure 1: Benchmark results for insertion sort, where the y-axis represents the time in seconds and the x-axis represents the length of the list to be sorted. The orange color represents the scenario where the input list is in reverse order, blue represents the scenario where the list is sorted, and purple represents the scenario where the list is in random order. Each point in the graph displays the standard deviation as a whisker.

The graphs from figure 1 show a steady increase with the increase in list size. The graph also shows that the sorted list takes significantly less time than random and reverse sorted lists, while random and reverse sorted lists are somewhat close to each other. However, the algorithm takes consistently slightly shorter time to sort the randomized lists, than the reversed lists.

3.2 Merge sort

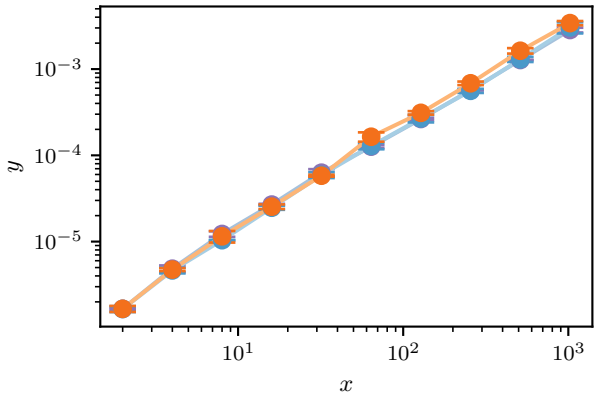

Figure 2: Benchmark results for merge sort, where the y-axis represents the time in seconds and the x-axis represents the length of the list to be sorted. The orange color represents the scenario where the input list is in reverse order, blue represents the scenario where the list is sorted, and purple represents the scenario where the list is in random order. Each point in the graph displays the standard deviation as a whisker.

The graphs from figure 2 show that all the different lists take approximately the same time. This corresponds with the result in table 2. In the table, the different times are separated by less than a tenth of a millisecond.

3.3 Quick sort

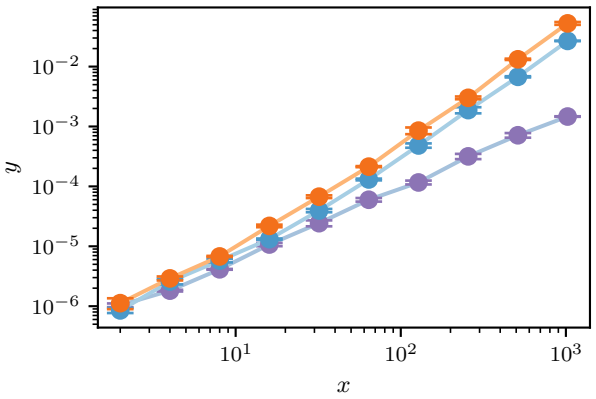

Figure 3: Benchmark results for quick sort, where the y-axis represents the time in seconds and the x-axis represents the length of the list to be sorted. The orange color represents the scenario where the input list is in reverse order, blue represents the scenario where the list is sorted, and purple represents the scenario where the list is in random order. Each point in the graph displays the standard deviation as a whisker.

The graphs from figure 3 show that random lists have the best performance, and the sorted and reverse sorted are close together and perform worst. The algorithm performed consistently worse on sorting reversed sorted lists than the randomized list.

4 DISCUSSION

4.1 Insertion sort

The plots in figure 1 have a good correlation with the theoretical time complexity for insertion sort from table 1. Here the best case has a significantly lower complexity than the worst and average. From the plot, the best case can clearly be seen as the sorted lists, and the worst and average have the same time complexity and that is the same case as for the reverse sorted and random lists.

This is because the insertion sort works by moving all the numbers that are in the wrong spot. If the list is already sorted it has nothing to move. But if the list is in reverse order it has to move all the elements in the list. This can be seen in table 2, where the time for the reversed list is nearly double the time as the random list and the sorted list is less than a hundredth of the time of the random list.

4.2 Merge sort

Merge sort has the same time complexity for all the different scenarios and this corresponds with what is shown in the plot of figure 2. Here all the different lists use the same time, this is because the algorithm works the same way regardless of the lists. It splits the lists in half until each list has a maximum of two elements, then compares and sorts each list, and then merges the sorted list back. So the arrangement of elements in the lists does not matter. This is also the reason why all the different time complexity have the same size. This can also be seen in table 2 where the time difference is less than a tenth of a millisecond.

4.3 Quick sort

The Quick Sort algorithm has the same time complexity for the best and average cases, with a slower worst case. The plot in figure 3 shows that sorted and reverse sorted take similar times and that random lists perform the best. One reason for this might be that sorted and reverse-sorted lists are both the worst-case scenario. The algorithm works by sorting the list with a pivot number. If this number is exactly half, the list will be evenly split and the algorithm will work better. But if the pivot point is the first or last element in a sorted or reverse sorted list it will split the list in an inefficient way. This will split the list into a list of one element and a list with the rest. Then repeat this. This can also be seen in the table 2 where a random list uses a tenth of a second and sorted and reverse sorted uses more than ten times the time.

From the pseudo-code retrieved from [Cormen et al. 2009], the pivot point is exactly this, the first element. So for a random list, the pivot is random and it works close to the best case, but when the list is already sorted or reverse sorted it splits in an inefficient way and gets a high time complexity. A way to work around this is to not use the first or last element but a random element in the list or the median value. This way would ensure a more stable way to get it to split evenly.

4.4 Overall

Overall the Merge sort algorithm is the fastest with Quick sort as the second fastest. The insert sort is the slowest. The time of quick sort would be closer to merge sort if the pivot point in the code was randomly selected. Another trend is that the reverse sorted list is the most time-consuming and often worst case. For merge sort and insertion sort, the best case is a sorted list, and because of the nonrandom pivot point, the best case for quick sort is a random list.

REFERENCES

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA.

Geeks for Geeks. 2023. *QuickSort – Data Structure and Algorithm Tutorials.* url={https://www.geeksforgeeks.org/quick-sort/}

Geeks for Geeks. 2024a. *Insertion Sort – Data Structure and Algorithm Tutorials.* url={https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/}

Geeks for Geeks. 2024b. *Merge Sort – Data Structure and Algorithms Tutorials.* url={https://www.geeksforgeeks.org/merge-sort/}

Geeks for Geeks. 2024c. *Time Complexities of all Sorting Algorithms.* https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

Python Software Foundation. 2024. *timeit — Measure execution time of small code snippets.* https://docs.python.org/3.10/library/timeit.html