Benchmarking Sorting Algorithms In Python

INF221 Term Paper, NMBU, Autumn 2020

Jon-Mikkel Korsvik
jonkors@nmbu.no

Yva Sandvik
ysandvik@nmbu.no

ABSTRACT

This paper analyses and compares the runtime of a selection of sorting algorithms with different input types. It covers theory and pseudocode, how data and plots were created, what challenges were encountered and their solutions. Finally, the superiority of hybrid sorting algorithms such as timsort, intro sort and merge sort combined with insertion sort, is discussed.

1 INTRODUCTION

“Sorting is a computational building block of fundamental importance” - [**?**]. The object of sorting is central in a wide range of applications from database systems to computer graphics.

During this investigation the performance of the sorting algorithms listed further down in the theory section (section **??**) have been assessed, under certain assumptions regarding their time complexity, and in some cases space complexity. Their performance has been timed using a timing function from the Python package time, and benchmarked while sorting different types of input arrays. In addition the time-development was assessed as the length of the arrays increased.

In the following subsections theory, pseudocode, as well as details surrounding the methods used when comparing seven different sorting algorithms is provided.

2 THEORY

All the algorithms compared in this paper have in common that they are comparison sorts. Meaning they read each element of an array through a single comparison operation, that determines which of two elements that should appear first in the sorted array.

The following terms have been frequently used throughout this paper; **??**, **??**, **??**, **??**, **??**, **??** and are therefore defined briefly before delving further.

2.0.1 Stable algorithms. An algorithm is stable when the inherent order of elements is kept intact, i.e: Two elements with the same value will keep the order from the initial input.

2.0.2 Runtime. The real time taken by an algorithm from start to end.

2.0.3 Efficiency. How efficiently a given algorithm performs, is based on the runtime on certain input types and sizes. This may vary as one will see with merge sort and insertion sort.

2.0.4 Consistent algorithms. The more even the runtime of the algorithm is across the different problem types and sizes, the more consistent the algorithms seems to be.

2.0.5 Input array. For a sorting algorithm to handle strings it needs keys, which often end up as integer data types. Therefore string was not used as one of the problem types. NumPy arrays were generated and used instead, which is what is defined by the term “input array”.

2.0.6 In-place algorithms. Only a constant number of elements of the input array are stored outside the array throughout the sorting period. **?**, II.Sorting and Order Statistics, p.148.

2.1 Algorithm 1 - Insertion Sort

Insertion sort is a simple and stable, in place, and comparison based sorting algorithm **?**, Ch. 2.1. Best case runtime for this algorithm is:

$$T(n) = \Omega(n) \ . \tag{1}$$

This is achieved when the input array is already sorted. The worst case runtime occurs if the input array is in reversed order. This gives a quadratic runtime of:

$$T(n) = O(n^2) \tag{2}$$

The average runtime is also quadratic $\Theta(n^2)$, making insertion sort a bad choice for sorting large arrays. However, it is one of the most consistent and quickest alternatives when it comes to sorting smaller arrays, due to it’s low overhead and simplicity. Exactly how small these arrays need to be in order for insertion sort to remain more efficient compared to other algorithms will be addressed later in the paper.

2.2 Algorithm 2 - Bubble Sort

Bubble sort (listing **??**) is know as a straightforward, stable and simple sorting algorithm. Both when it comes to implementation and understanding.

Although bubble sort and insertion sort grow asymptotically at the same rate $\Theta(n^2)$, the difference in bubble sort and insertion sort lies in the number of comparisons, as well as that plain bubble sort’s best-case time complexity is $\Omega(n^2)$ **?**, Ch. 2.3. In contrast to insertion sort, bubble sort could in the worst case have to make numerous comparisons that do not necessarily result in a swap, making it computationally slower.

In the modification of bubble sort used in this paper, a flag is included for keeping track of whether or not the last iteration exchanged any elements, if not; the algorithm ends. A round tracker is also included. The purpose of this is to increment after each iteration, on lines 4 to 10, shortening the ends of the arrays. These two modifications do not interfere with the loop invariant of plain bubble sort, and give the best case performance $T(n) = \Omega(n)$, which occurs when the input problem is already sorted in an ascending order.

2.3 Algorithm 3 - Merge Sort

Merge sort, found in listing **??**, is yet another comparison-based sorting algorithm that uses the divide-and-conquer approach. This

Listing 1 Bubble sort from [?], with slight modifications.

```
BUBBLE_SORT($A$)
$n = A.length$
$swapped = True$
$rounds = 0$
while $swapped$ :
    $swapped = False$
    for $i = 0$ to $n$-$rounds$-$1$
        if $A[i] > A[i+1]$ :
            $A[i], A[i+1] = A[i+1], A[i]$
            $swapped = True$
    $rounds + 1$
```

involves recursively splitting an array midway and merging together two and two presorted arrays such that the resulting array also is sorted. As with insertion sort and bubble sort, merge sort is a stable algorithm.

Listing 2 Merge sort from **?**, Ch. 2.3, p.31.

```
MERGE_SORT($A$)
 1  if $A.length > 1$:
 2      $mid = A.length/2$:
 3      $L\_arr = A[: mid]$
 4      $R\_arr = A[mid :]$
 5      MERGE_SORT($L\_arr$)
 6      MERGE_SORT($R\_arr$)
 7      $L\_ind = 0$
 8      $R\_ind = 0$
 9      $copy\_ind = 0$
10      while $L\_ind < L\_arr.length$ and $R\_ind < R\_arr.length$
11          if $L\_arr[L\_ind] < R\_arr[R\_ind]$
12              $L\_ind + 1$
13          else :
14              $A[copy\_ind] = R\_arr[R\_ind]$
15              $R\_ind + 1$
16      while $L\_ind < L\_arr.length$
17          $A[copy\_ind] = L\_arr[L\_ind]$
18          $L\_ind + 1$
19          $copy\_ind + 1$
20      while $R\_ind < R\_arr.length$
21          $A[copy\_ind] = R\_arr[R\_ind]$
22          $R\_ind + 1$
23          $copy\_ind + 1$
```

Unlike insertion sort and bubble sort it does not iterate through the entire list several times, but instead uses this recursive divide and conquer approach which gives a consistent average and worst case performance of:

$$T(n) = \Theta(n \lg(n)) \tag{3}$$

This is in fact the asymptotically optimal time complexity that can be achieved by a sorting algorithm **?**, Ch. 8.0. In addition, the fact that the input array is split into subsets, opens for the possibility to sort the two subsets simultaneously, also called to parallelize. This in turn decreases the constants not for computations, but for runtime in accordance to number of workers (threads) chosen.

However, one of merge sorts drawbacks stems from the time cost caused by the large overhead which effects its performance when sorting smaller arrays. In addition, the most common implementations of merge sort do not perform in-place sorting. Bringing us to a second drawback of merge sort; its memory requirement. The memory size of the input must be allocated for the sorted output to be sorted in, hence it uses more memory space than other in place sorting algorithms. It's space complexity thereby becoming $O(n)$ [?]. The actual "merging process" in the merge sort algorithm occurs from line 10 to 23 in listing **??**.

2.4 Algorithm 4 - Quick Sort

Quick sort is an efficient and important in-place, divide-and-conquer sorting algorithm.**?**, Ch. 7.1. It shares the same average time complexity as merge sort, and it is also straightforward to parallelize since it also uses the divide and conquer approach.

Listing 3 Quick sort from **?**, Ch. 7.2, p. 173 and [?] with modification.

```
MEDIAN_OF_THREE($A, low, high$)
mid = (low + high) // 2
if A[mid] < A[low]
    exchange A[low] with A[mid]
if A[high] < A[low]
    exchange A[low] with A[high]
if A[mid] < A[high]
    exchange A[mid] with A[high]

PARTITION($A, low, high$)
MEDIAN_OF_THREE($A, low, high$)
$pivot = A[high]$
$i = low - 1$
for $j = low$ to $high - 1$
    if $A[j] \leq pivot$
        $i = i + 1$
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return $i + 1$

QUICK_SORT($A, low, high$)
if $low < high$
    $pivot$ = PARTITION(A, LOW, HIGH)
    QUICK_SORT($A, low, pivot - 1$)
    QUICK_SORT($A, pivot + 1, high$)
```

However the worst case time complexity of quick sort is its main disadvantage, and is $O(n^2)$, i.e no better than insertion sort. Worst case behaviour is prompted by a non uniform distribution and by choosing an unfavorable pivot argument. The partitioning that occurs when quick sort splits the input array into two subsets is based on this pivot element. The selection of the pivot is therefore critical to the worst case performance of quick sort, as it determines the number of necessary recursion levels that will occur. If the algorithm consistently picks the median element as pivot, as opposed to the first, last or a random element, one can guarantee that one will get the best time complexity for the worst cases, ascending and descending inputs. Meaning, the worst time complexity can be avoided if the partitioning is balanced, making the expected runtime $\Theta(n \lg n)$, such as merge sort [?].

On the other hand the median only handles two of the worst cases; an array with all equal values or few unique values will still induce worst case behaviour. To handle these problems, multi pivot partitioning based on the dutch flag algorithm is used as a replacement for hybridization of the algorithm. When implemented as a three-way partitioning properly one can guarantee $\Theta(n \lg(n))$ worst case behaviour [?].

The worst space complexity from the partitioning is invariably $O(n)$ [?] auxiliary memory using Lomuta partionining, if Hoare partitioning is used the worst case space complexity is $O(\lg(n))$.

In listing **??** one can see how the partitioning in quick sort has been implemented in this paper. The Median_of_three function essentially chooses the median between the first, last and middle element of the array.

2.5 Algorithm 5 - Merge Sort Combined

Merge sort can be optimized to give merge sort combined by integrating insertion sort and making it a 'hybrid algorithm'. This takes advantage of the fact that insertion sort performs well on smaller arrays.

Which inputs are sorted by insertion sort and which inputs are sorted by merge sort is determined by a threshold, which is an input argument to the function shown in listing **??**. Being a hybrid algorithm means it is dependent on both the already mentioned in-place algorithms shown in **?**, Ch. 2.1 and listing **??**.

Merge sort combined therefore uses fewer comparisons in the worst case than both merge sort and insertion sort, potentially making it a very efficient sorting algorithm. [?].

Listing 4 Merge sort combined implementation.

```
Merge_sort_combined($A$, $threshold$)
if $A.length > threshold$
    $mid = A.length/2$:
    $L\_arr = A[: mid]$
    $R\_arr = A[mid :]$
    Merge_sort_combined($L\_arr$)
    Merge_sort_combined($R\_arr$)
    ...same implementation as in Listing ??
else
    Insertion_sort($A$)
```

2.6 Algorithm 6 - Python "sort()"

Timsort is the 'built-in' sorting function in Python [?]. In similarity to merge sort combined, timsort is a hybrid sorting algorithm derived from merge sort and insertion sort.

It is a stable sorting algorithm with the a time complexity of $\Theta(n \lg n)$ [?]. It starts by using insertion sort to boost the smaller naturally sorted elements found in the unsorted array to a sufficient size, after which they are merged using merge sort. By using insertion sort on the smaller arrays one reduces the overhead produced by using more complex algorithms on such small arrays. Hence, timsort efficiently sorts algorithms by taking advantage of ordering that is already present in the input array.

The small ordered subsets, also called 'Runs' or 'minruns', need to be of a minimum size, which is dependant on the length of the array that is to be sorted. Selection of minrun size is based on creating balanced merges. [?].

2.7 Algorithm 7 - NumPy "sort()"

NumPy sort() is the 'built-in' NumPy function which sorts an array in-place. NumPy sort() can alternate between different sorting algorithms, the default being intro sort, if specified one can also use merge sort, timsort or heap sort [?].

Intro sort is a quick sort algorithm with a limit for recursion depth before it switches to heap sort. In addition it switches to insertion sort when the number of elements is below a certain threshold. Meaning intro sort is a hybrid sorting algorithm, combining the best parts of three algorithms. It remains an unstable algorithm, such as 2 of it's algorithms: heap sort and quick sort.

2.8 Compiling with Numba

Numba is a "just-in-time" compiler that translates a subset of Python and NumPy code into fast machine code using the industry-standard LLVM compiler library [?].

As Python is known to have a decent amount of overhead for computation-intensive tasks such as sorting, we have chosen to apply a "numba.jit" decorator to our self implemented algorithms in order to speed up their performance.

3 METHODS

Below is a short summary of the steps used to generate data and how benchmarking was performed:

- Firstly the data used for benchmarking was produced. The test data was generated using the class methods of class ArrayGenerator found in the utility file.
- After that the benchmark function timed how long each algorithm used to sort given arrays with given lengths.
- The timer function timed each sorting algorithm in an array a given number of repetitions, and returned all the results in a dictionary as well as printing stats if verbose.
- Finally, the timing data was unpacked as a pandas DataFrame with columns 2^N, Time, TypeArray and Algorithm. Which was used for plotting results.

3.1 Data generation - Array Generator

A factor of 2^n was used to generate arrays, giving a logarithmic growth of problem sizes with logarithmic increment benchmarking.

3.1.1 Reasons behind using logarithmic scale. Using logarithmic scale gave a great range of arrays from 1 to arrays of 1,073,741,824 elements. In addition, the logarithmic scale reduced the size of results by a factor of. $\frac{n}{a \log_2(n)}$ where a is a scalar representing linear step sizes. The benchmarking process shaved of time usage by a factor of $\frac{\sum_{k=1}^{n/a} T(k \cdot a)}{\sum_{k=1}^{\log_2 n} T(2^k)}$, where T(x) is the $O(n)$ of an algorithm. This eased the computation time for generating data as well as for plotting, while retaining the information of the trends in runtimes for every algorithm.

3.1.2 Approach. Using an object oriented approach the seed of a random number generator from NumPy was assigned. The choice of NumPy gathered the benefits of large data, compared to the standard random library in Python. The ArrayGenerator class created arrays such as shown in figure ??. It can create random arrays of both float and integer type. Indicated in later plots as correspondingly **Random** and **Integers**.

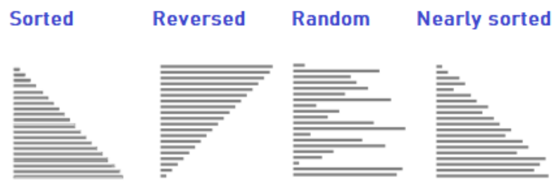

Figure 1: Different problem types

3.1.3 Creation of the Arrays. Sorted and Reversed were made using NumPy arrange method. Random and Integers were created using the random number generator of the ArrayGenerator class [?]. The nearly sorted array or the structured array was generated using sorted inpuut array and a shuffle range used for inplace shuffling of the indices given by equation ?? and gave an array as shown in figure ??.

$$S = \lfloor 2 * \frac{2^n}{n^2} \rfloor, \quad where \quad n \text{ is a power of } 2 \wedge S \text{ is shuffle range} \tag{4}$$

The Structured Array was shuffled for any array with number of elements > 0 since S has it's minimum at $n = 3 \implies$ ShuffleRange $= int(1.7778) = 2$

3.2 Runtime Benchmarking

The main bunch of work was done by the *repeating_timer* decorator in `src/utility.py`, which can be used with single or multiple algorithms (as a callable) several times on the same array. It handled in-place algorithms by assigning a new copy before every iteration, and thereafter saved the results from every iteration in a dictionary with algorithm name as the key and the list of timings as the value.

To generate results `notebook/generate_data_script.py` was used. Over time multiple results were generated, and merged together with `src/merge_csv.py`, which gave `data/merged_results.csv` as an output. All data points were stored with equality, which then was easy to manipulate when creating new variables and during plotting. See table ??.

3.2.1 Computer and software. OS: Windows 10, Python version 3.7.9, Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 Mhz, 4 cores, 8 logic threads. 8 GB of installed RAM. **Make and model:** Huawei MateBook X Pro i5 dGPU 8GB 256GB (2018)

Table 1: Five random samples from results

Results DataFrame			
Algorithm	2^N	TypeArray	Time
python_sort	15	Integers	0.003036
numpy_sort	21	Descending	0.028119
numpy_sort	1	Structured	0.000002
numpy_sort	15	Random	0.002030
merge sort	15	Structured	0.002981

Table 2: Versions of files used for this report; Team 19 gitlab repository.

File	Git hash
src/utility.py	04c0a468
src/* //algorithms and tests	04c0a468
notebooks/generate_data_script.py	3f422550
notebooks/plot_creation.ipynb	520dc68e
notebooks/opt_thresh_combined.ipynb	8ec07210f
data/merged_results.csv	88c28d55c
asymptotic_bounds/*	aa0b819e

3.2.2 GitLab. Git hashes for more details on the code used are provided in table ??.

3.3 Challenges and Solutions

3.3.1 Challenges: Various challenges that arouse during the benchmarking process are listed below.

(1) Time taken to benchmark algorithms.
(2) Quick sort was problematic with standard implementation. This caused recursion limits and made it difficult to plot against other algorithms. This was mainly due to worst-case partitioning and Python not optimizing tail recursion [?].
(3) Finding the threshold for merge sort combined.
(4) Verifying the functionality of the sorting algorithms.

3.3.2 Solutions: These implementations were made in order to solve the challenges listed in section ??.

(1) JIT-compiled sorting algorithms (effects negatively only on the first calls to the algorithms.) One could have compiled ahead of time, but since that caused complication with data types and what OS or Hardware one is running, we JIT-compiled every algorithm except NumPy sort.
(2) Iterative quick sort was implemented with median of three partitioning. This led to only iterative quick sort with median of three pivot optimization being used instead, for all benchmarking. There are other ways to handle the recursion depth issue, but they would stray far away from the standard quick sort. For example hybrid with heap or binary insertion sort, or multi-pivot partitioning.
(3) The largest $\frac{\delta(a,b)}{\delta(n)}$ where a is time taken by merge sort and b is insertion sort, was found. Which in this case was a threshold of 142. Hence, each split of merge sort with n < threshold was sorted using insertion sort. See figure ?? and

section ??. Another way to find the threshold dynamically could be with something similar to timsort's minrun size as described shortly in section ??.
(4) Unittests in tests/test_algos.py

4 RESULTS

The following section is dedicated to presenting visual results of the algorithms listed in section ??. The y-axis of all plots in the results section is runtime divided by N-length of arrays measured in microseconds, except for figure ?? which is time measured in seconds.

4.1 Optimal Threshold for Merge Sort Combined

This result is a dependency of the selected threshold in merge sort combined. If more details surrounding this is of interest, look up: `notebook/opt_thresh_combined.ipynb`, to see how the results were generated.

Figure ?? are increments of 1 from 25 to 200 showing the different types of input arrays for merge and insertion sort. The colored bands show the standard deviation for each algorithm.

Figure ?? shows the mean aggregated over algorithm and N, giving a single line representing the average-case. It also shows the threshold value for insertion sort. Below the array length of 300 elements, insertion sorts performs better than merge sort.

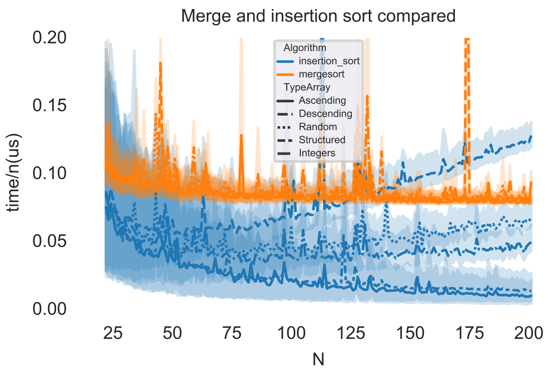

Figure 2: Comparing insertion sort with merge sort

4.2 Benchmarking Individual Algorithms

The results of the individual algorithms with different array sizes and types are presented in figures ??, ??, ??, ??, ??, ?? and ??.

Insertion sort ?? and bubble sort ??, displayed quadratic growth, and interestingly bubble sort performed around 100 seconds faster than insertion sort on descending arrays, when $N = 2^{20}$. Insertion sort performs well on ascending and structured data.

Python sort ?? performed in sub-quadratic time on structured and randomized input arrays. One can observe linear growth on ascending and descending input arrays, which with the selected y-axis is approximately a horizontal line.

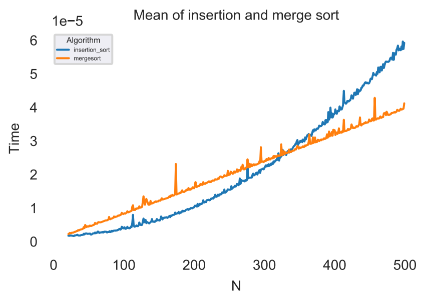

Figure 3: Comparing insertion sort with merge sort

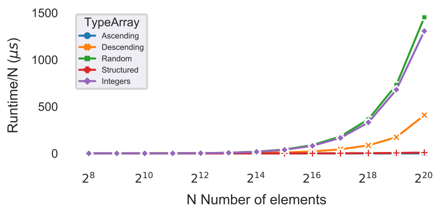

Figure 4: Bubble Sort

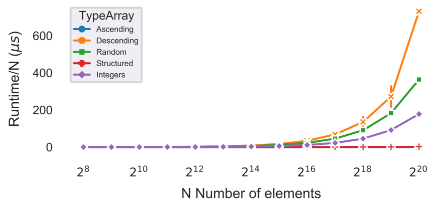

Figure 5: Insertion Sort

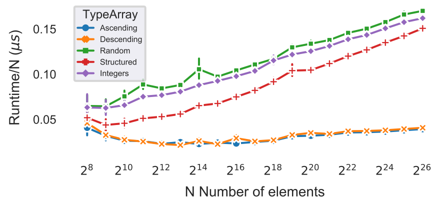

Figure 6: Python Sort

Iterative quick sort, shown in figure ??, produced similar behaviour to python sort, but descending arrays give sub-quadratic time complexity and the ascending input arrays looks almost linear. Keep in mind that this implementation is with median of three pivot.

In figure ?? one can see broad error bars for merge sort before it balances out after 2^{14} elements. Every array type induces $n \lg(n)$ growth with varying constants as with Python sort and iterative quick sort.

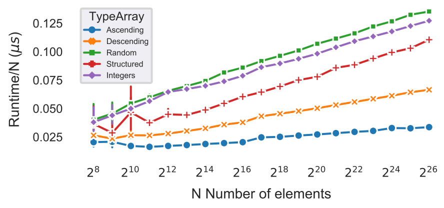

Figure 7: Iterative Quick Sort

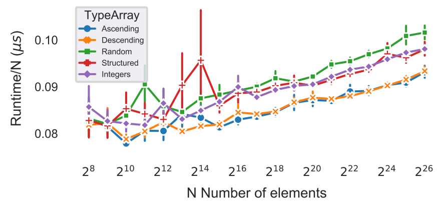

Figure 8: Merge Sort

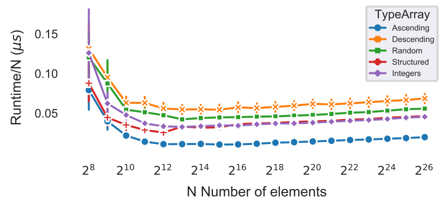

Figure 9: Merge Sort Combined

Merge sort combined looks deceivingly like it has linear growth, this is not the case. One can observe very small constants in figure **??**. It uses between 2 to 10 as much time per elements with arrays of size 2^8, compared to arrays larger than $2^{14} = 16,384$.

NumPy sort's results, shown in figure **??**, looks quite similar to python sort just with a lower y-axis.

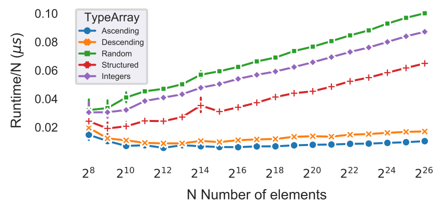

Figure 10: NumPy Sort

4.3 Comparing Algorithms

The first two algorithms compared were the quadratic sorting algorithms shown in figure **??**. Insertion sort clearly performs better for all types of arrays except for ascending arrays, and descending described in section **??** . It is possible to zoom in on the PDF to see the plot distinctions better.

Figure **??** shows a comparison of merge sort and insertion sort, as mentioned in section **??**. Seeing this it is evident that insertion sort is superior in sorting efficiently, compared to merge sort, given the array size. Merge sort however is more consistent in its performance across the different input arrays.

The next comparison plot, seen in figure **??**, is of the sub-quadratic sorting algorithms. Quick sort performs better than merge sort when sorting all arrays up to a length of 2^{15}. For ascending and descending arrays quick sorts performs the best across all array sizes. However, quick sorts performance also varies a lot depending on the type of the input array. Merge sort on the other hand, stays surprisingly consistent in performance when both the type and size of the input array varies. When the array size approaches a length of 2^{28} elements and greater, both the algorithms seemed to struggle. The error bars that occur on the lines early on the x-axis may have been disturbances that occurred while running the benchmarking.

Comparing merge sort with merge sort combined, seen in figure **??**, one can observe that both algorithms have a similar decrease in performance as the array size increases along the x-axis. However, merge sort combined clearly performs better than ordinary merge sort when it comes to all the array types. Another difference is that merge sort combined's performance is less consistent across the types of arrays. It sorts ascending arrays the best. Ordinary merge sort on the other hand performs quite similarly time wise on all the different input arrays. In this plot as well there seems to be a struggle when the array size exceeds a length of 2^{28}.

Finally the four best performers are compared; merge sort combined and iterative quick sort are compared to NumPy's and Python's in-place sorts, shown in figure **??**, **??** and **??**. Figure **??** shows an average-case representation of figures **??**, **??**. In these plots one can see that merge sort combined still is the most consistent both across different types of arrays and also as array size increases. In addition it beats the other algorithms in random, structured and integer arrays. However, NumPy sort is superior when it comes to ascending and descending arrays. In figure **??** we clearly see that NumPy sort and merge sort combined all round are the two best performers, beating both iterative quick sort and Python sort. Python sort is slightly more consistent than iterative quick sort (figure **??**). However, on average iterative quick sort clearly is the better performer of the two (figure **??**).

4.4 Asymptotic Bounds and Numerical Results

After analysis, table **??** are the asymptotic bounds for this paper's choice of algorithms. $f(n)$ with c_1 is lower bound, and $g(n)$ with c_2 is upper bound.

Presenting table **??** with $n_0 = 256$. Plots of results can be found in directory `asymptotic_bounds/*` in our gitlab repository in table **??**.

Denoted with the algorithm's abbreviations, table **??** displays the average runtime over all different array types and experiments conducted for this paper with the `notebooks/generate_data_script.py`.

5 DISCUSSION

As mentioned, the quadratic sorting algorithms are the slowest algorithms, which is as expected considering they both have the slowest average runtime of $\Theta(n^2)$. Which is confirmed when seeing

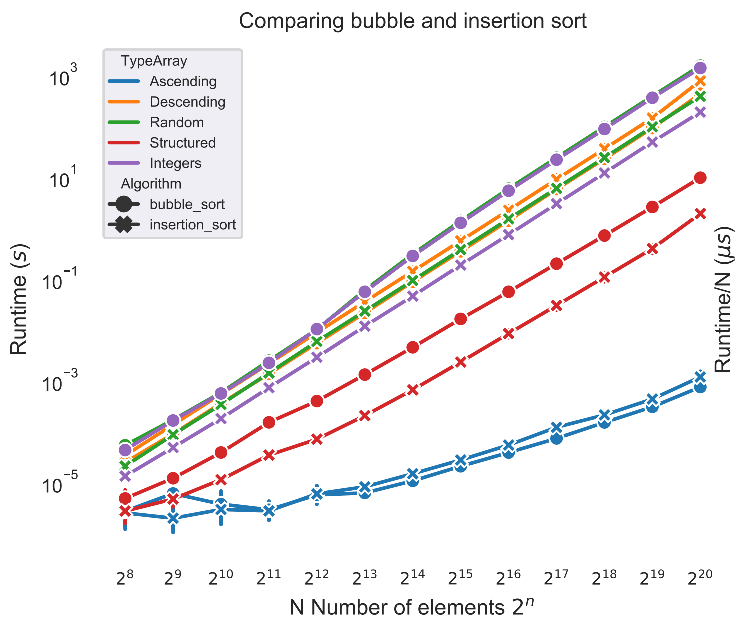

Figure 11: Insertion sort and bubble sort.

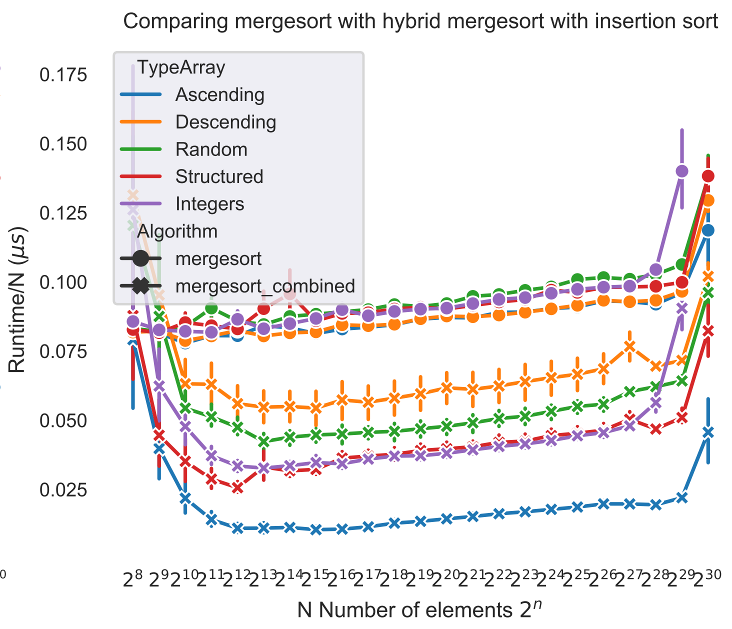

Figure 13: Merge sort and merge sort combined.

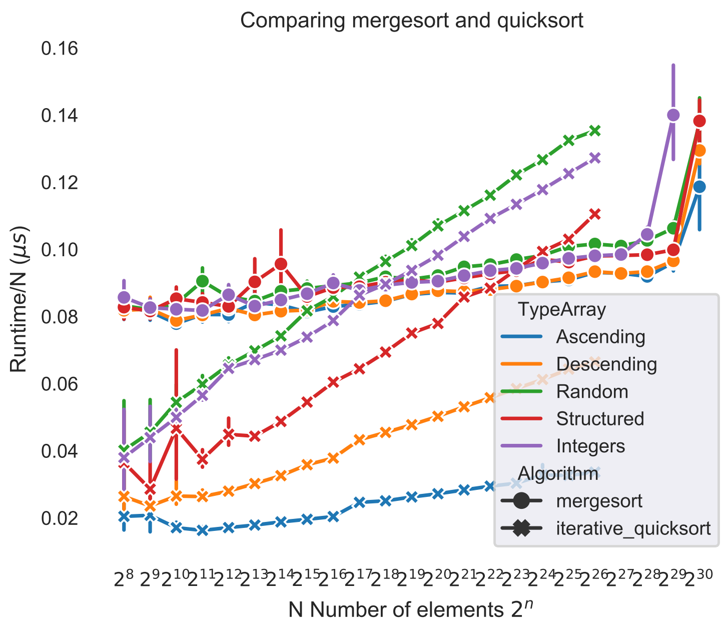

Figure 12: Merge sort and iterative quick sort.

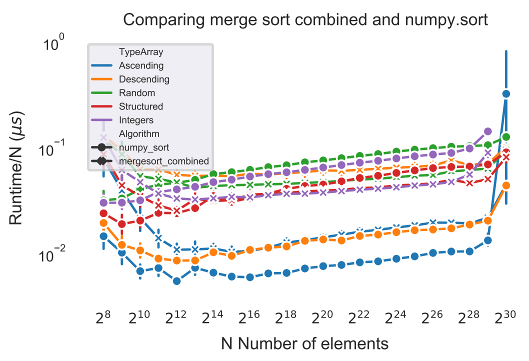

Figure 14: Merge sort combined and NumPy sort

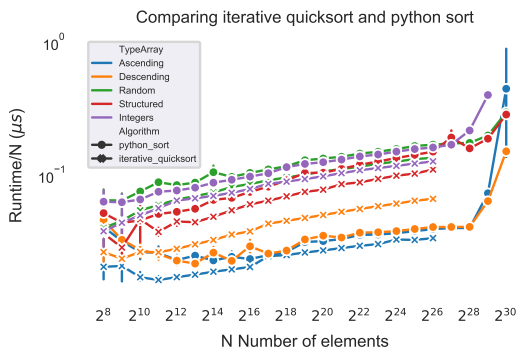

Figure 15: Iterative quick sort and python sort

their quadratic curves in the individual benchmark-plots (figure **??** and **??**).

Even though there are many more powerful algorithms than insertion sort, such as merge sort, the time cost caused by the large overhead results in them failing to beat insertion sort when sorting smaller arrays. In this case the threshold being around 300 elements.

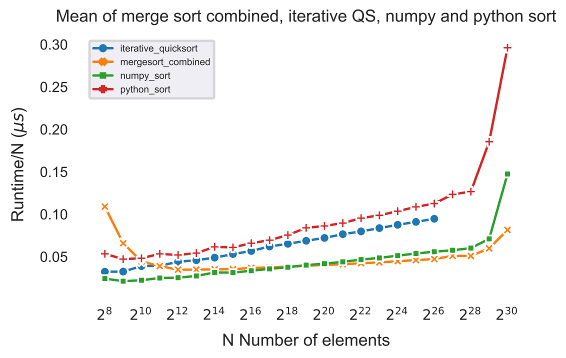

Figure 16: Mean of figures ?? and ??

Table 3: Asymptotic boundaries

Algorithm	*f(n)*	*g(n)*	c_1	c_2
Bubble sort	$\Omega(n)$	$O(n^2)$	$3.98 \cdot 10^{-10}$	$1.58 \cdot 10^{-9}$
Insertion sort	$\Omega(n)$	$O(n^2)$	$5.01 \cdot 10^{-10}$	$7.94 \cdot 10^{-10}$
Iterative QS	$\Omega(n \lg(n))$	$O(n \lg(n))$	$1.00 \cdot 10^{-9}$	$5.01 \cdot 10^{-9}$
Merge sort	$\Omega(n \lg(n))$	$O(n \lg(n))$	$2.51 \cdot 10^{-9}$	$6.31 \cdot 10^{-9}$
MS combined	$\Omega(n \lg(n))$	$O(n \lg(n))$	$5.01 \cdot 10^{-10}$	$3.16 \cdot 10^{-10}$
NumPy sort	$\Omega(n \lg(n))$	$O(n \lg(n))$	$3.98 \cdot 10^{-10}$	$6.31 \cdot 10^{-9}$
Python sort	$\Omega(n)$	$O(n \lg(n))$	$2.00 \cdot 10^{-8}$	$1.00 \cdot 10^{-8}$

Table 4: Average runtimes of algorithms (Time in (S))

Algorithm 2^N	BS	IS	IQS	MS	MSC	NS	PS
2	0.00000	0.00000	0.00000	0.00000	0.00002	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000	0.00002	0.00000	0.00001
6	0.00000	0.00000	0.00000	0.00001	0.00002	0.00000	0.00001
8	0.00003	0.00001	0.00001	0.00002	0.00003	0.00001	0.00001
10	0.00031	0.00021	0.00004	0.00008	0.00005	0.00002	0.00005
12	0.00521	0.00350	0.00018	0.00034	0.00014	0.00010	0.00021
14	0.13483	0.05552	0.00080	0.00142	0.00058	0.00051	0.00101
16	2.52257	0.88911	0.00372	0.00570	0.00241	0.00219	0.00432
18	41.11392	14.26344	0.01709	0.02310	0.01005	0.00985	0.01979
20	666.04966	267.50417	0.07563	0.09390	0.04230	0.04384	0.09039
22	nan	nan	0.33473	0.38491	0.17768	0.19543	0.40016
24	nan	nan	1.46942	1.58280	0.75032	0.86001	1.73815
26	nan	nan	6.35141	6.49816	3.17001	3.74989	7.55585
28	nan	nan	nan	26.35145	13.65878	16.14606	34.00469
30	nan	nan	nan	140.68718	87.54823	158.28938	317.99038

This is supported by the results in figure **??**, which coincides with the theory from section **??**. Bringing us to the reason why insertion sort is often used internally in other sorting algorithms, as it is highly efficient in sorting small portions of the input array, ultimately resulting in even better performing hybrid algorithms. Such as timsort or merge sort combined for instance.

In figure **??** we saw that quick sort performs the best on ascending and descending arrays. In theory section **??** we mentioned that these arrays were two of the worst case scenarios for quick sort with last pivot partitioning. The fact that quick sort performs this well when these arrays means that the pivot optimization and the iterative implementation of quick sort is successful in handling the recursion issues that normally would arise when sorting such problems. In fact, quick sort even outperforms Python sort for all input arrays except descending, as seen in figure **??**. The reason Python sort performs better for descending arrays is probably due to the ability of the minruns to detect smaller portions of the input array being in reverse order. Resulting in Python sort reversing the array in linear time.

Figure **??** and table **??** also shows that there are obvious performance differences between algorithms even though they are within the same complexity class, which is $\Theta(n \lg n)$ in this case.

As mentioned in the results, many of the algorithms 'struggled' towards the end of the plots. They increased in runtime by a factor between 2 and 10 as the array size reached a length of 2^{29} to 2^{30} on the x-axis. This is because the RAM of the simulation computer was filled, and the memory was then shifted between physical memory and virtual memory. This means we should take the results after 2^{28} with a grain of salt, since other factors as which order they are benchmarked might affect the results. It can also be favoring algorithms with non linear space complexity and/or small overhead.

Overall, the experimental results coincide well with the theoretical runtime assumptions. Although the self implemented algorithms perform much better than expected. Two out of the four best performing algorithms were as expected the built-in NumPy and Python sort (figure **??**). However, Python sort only performs the fourth best until a certain array length, after which it is beaten by ordinary merge sort. When the input array length exceeds 2^{20}, ordinary merge sort becomes the fourth best performing algorithm, as seen in table **??**. The fact that the implementations of both merge sort and merge sort combined performed nearly as well as NumPy sort, and in fact even beats NumPy sort at the end of the benchmarking (table **??**), is quite shocking. Although, one should be slightly apprehensive towards the benchmarking results passing an array length of 2^{28}.

This of course makes one wonder how the self-implemented algorithms perform so well in comparison to the NumPy and Python built-in-sort's. Using Numba, the slowdown of the algorithms implemented in pure Python code versus the built-in ones, is nearly eliminated. Which most probably is a large reason to why these results were obtained.

Finally, the two best performers; NumPy sort and merge sort combined, all have in common that they are hybrid algorithms. This goes to show that combining efficient sorting algorithms, and when implemented with suitable thresholds and compilation, results in superior performance.

Due to time and space constrictions we did not test the algorithms on arrays with many duplicate elements nor string. However, this could potentially be interesting to investigate in further explorations. Testing hybridization's of different algorithms with a greater number of array types, and specific worst-case "killer-problems", for the different algorithms, would be a worthwhile investigation.

ACKNOWLEDGMENTS

We are grateful to Khariton Gorbunov and Mathias Chunnoo for guidance. Bao Ngoc Huynh for good feedback on drafts. Fabian Nemazi for useful discussion and advice. Last but not least, we are thankful for the knowledge, interesting lectures and guidance provided by Hans Ekkehard Plesser.