Benchmarks of five popular comparison sort algorithms

INF221 Term Paper, NMBU, Autumn 2019

Adam Julius Olof Kviman
juliukvi@nmbu.no

Hemanth Babu Sana
hesa@nmbu.no

Mithunan Sivagnanam
misi@nmbu.no

ABSTRACT

In this paper, we perform benchmarks on five different comparison sort algorithms. The goal of the paper is to find out which algorithm performs the fastest as well as to find out what kind of ordering of data that goes the fastest to sort for each algorithm. We also compare the benchmarks for the theoretical worst case run time of each algorithm. The data being sorted are integer arrays of sizes ranging from 2^7 to 2^{19}. We tried to sort three different kinds of orderings of the data, already sorted arrays, reversely sorted arrays and randomly ordered arrays. The algorithms tested are Mergesort, Heapsort, Quicksort, Numpy_sort and Python_sorted. The sorting algorithms were run using Python 3.7.4. Our results show that Numpy_sort is the fastest algorithm for the three orderings of data using the array sizes mentioned above. For array sizes larger than 2^{25} one should consider using Python_sorted instead. All three orderings of data followed the theoretical worst case run-time for Heapsort. For Mergesort the random data did not follow the worst case run-time. For Quicksort and Numpy_sort none of the orderings followed the theoretical worst case run-time and for Python_sorted only the random data followed the worst case run-time.

1 INTRODUCTION

Benchmarking is a good way to measure the performance of algorithms. It is widely used in industry and research to ensure the quality of computer software and hardware. One can compare different algorithms on the same hardware to find out which software is the most efficient, alternatively one can compare the same software on different hardware to test the efficiency of the hardware. In this paper we have performed benchmarks on several sorting algorithms in Python using the same hardware. The algorithms tested were Heapsort, Mergesort, Quicksort, Numpy_sort (Numpy's builtin sorting algorithm) and Python_sorted (Pythons builtin sorting algorithm). The benchmarks were performed on integer arrays of different sizes and different orderings. The orderings used were, already sorted arrays, arrays with random data and reversely sorted arrays. The purpose of this paper is to compare theoretical expectations and to find out which algorithm that is the fastest for the different orderings of data as well as to determine what type of ordering of data that is the fastest to execute for each algorithm. We use plots to graphically analyze the algorithms and draw conclusion from our data. This paper is structured in the following way, section 2 consists of a brief introduction to each algorithm. The introduction includes the worst case run-time as well as code for each algorithm. In section 3 we present the methods used to perform the benchmarks, how the data was plotted, what hardware was used, what software versions that was used as well as the git hashes to the files used in this paper. In Section 4 we present the results as plots and make comments about what can be seen from them. In section 5 we discuss what conclusions we can draw from the data, how our benchmarks performed compared to the worst case run-times presented in section 2 and what type of improvements that can be made for future benchmarks of this type. Section 5 is followed by our acknowledgments to the people who has helped us with this paper and in the last part of the paper the references can be found.

2 THEORY

Here we provide a brief description of the algorithms that have been benchmarked, including pseudocode for the algorithms. The expected worst case run-time of algorithms in terms of problem size is also presented.

2.1 Quicksort

Quicksort(partition-exchange sort) is an efficient sorting algorithm. It was invented by Tony Hoare in 1961 [3]. It is a commonly used algorithm for sorting an array. When it is implemented on certain data it can be about two to three times faster than mergesort and heapsort [13]. Quicksort is not a stable sort, this means that elements that has the equal value may switch place during the sort. Quicksort can operate in-place on an array which means its memory footprint is small. It has a worst case space-complexity of $O(n)$. However, this can be decreased to $O(\log_2(n))$ by using methods described by Sedgewick [12]. Analysis of quicksort shows that the algorithm takes $O(n \log_2(n))$ comparisons to sort an array with n elements. In the worst case, it makes $0(n^2)$ comparisons. The steps of Quicksort are the following:

(1) Select an element from the array which we call it a pivot.
(2) Partitioning: Reorder the array such that elements with value less than the pivot are placed to the left of the pivot, and elements with values greater than the pivot are placed on the right side of the pivot.
(3) Apply the two above steps recursively to the sub-array of elements which are on the left-side of the pivot and to the right-side of the pivot.

The base case of the recursion is arrays of size one, and since they are a single value they are already sorted, so they never need to be sorted [3]. The partition and pivot selection can be done in several different ways; these choices can greatly affect the algorithm's performance. In listing 1 follows code for quicksort. In this project, a randomized partition scheme was used.

Listing 1 Quicksort algorithm from [2]

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j in range(p, r)
        if A[j] < x
                i += 1
                A[i], A[j] = A[j], A[i]
A[i+1], A[r] = A[r], A[i+1]
return i+1

QUICKSORT(A, p, r, random = False)
partition = Partition
if random
        partition = Randomized_partition
if p < r
        q = partition(A, p, r)
        quicksort(A, p, q-1, random)
        quicksort(A, q+1, r, random)
return A

RANDOMIZED_PARTITION(A, p, r)
m = random.randint(p, r+1)
A[r], A[m] = A[m], A[r]
return Partition(A, p, r)
```

Listing 2 Heapsort algorithm from [2]

```
HEAPSORT(A)
heap_size = Build_Max_Heap(A)
for i in range(len(A), 1, -1)
        A[0], A[i-1] = A[i-1], A[0]
        heap_size -= 1
        Max_Heapify(A, 1, heap_size)
return A

MAX_HEAPIFY(A, i, heap_size)
l = 2*i
r = 2*i+1
if l <= heap_size and A[l-1] > A[i-1]
        largest = l
else
        largest = i
if r <= heap_size and A[r-1] > A[largest-1]
        largest = r
if largest != i
        A[i-1], A[largest-1] = A[largest-1], A[i-1]
        Max_Heapify(A, largest, heap_size)

BUILD_MAX_HEAP(A)
heap_size = len(A)
for i in range(floor(len(A)/2), 0, -1)
        Max_Heapify(A, i, heap_size)
return heap_size
```

2.2 Heapsort

The heapsort algorithm was invented by J.W.J Williams in 1964 [16]. The heapsort algorithm starts by using BUILD_MAX_HEAP to turn the input array A[1..n] into a maxheap, Since A is a maxheap the maximum element of the array is stored at A[1], to put the maximum element in its right position we just switch A[1] and A[n]. We now ignore the last element of the array since its already sorted. The new array A[1..n-1] is also a maxheap except for the first element A[1] . All we need to do to make A[1..n-1] into a maxheap is to call MAX_HEAPIFY (A, 1). The heapsort algorithm repeats this until the array A has a size of one. Heapsort has an expected worst case run-time of $O(n \log_2(n))$. Below follows code for the Heapsort algorithm [2].

The Heapsort algorithm takes time $O(n \log_2 n)$, since the call to Build_Maxheap takes time $O(n)$ and each of the n-1 calls to Max_Heapify takes time $O(\log_2 n)$ [2].

2.3 Mergesort

The Mergesort algorithm was invented by John von Neumann in 1945 [5]. Mergesort follows a divide and conquer approach. Divide: Divide the n-element array into two smallar arrays of n/2 elements each. Conquer: Sort the two arrays recursively using merge sort. Combine: Merge the two arrays to produce the sorted array [2]. The recursion ends when the two created arrays have size one, those arrays are by definition already sorted. The mergesort then begins the combine stage when it starts to combine two already sorted arrays into a new single sorted array. First pairs of single elements are combined and sorted into arrays of two elements. Then pairs of these arrays are combined and sorted into arrays of four elements. This continues until there is a sorted array of size n. The Mergesort algorithm has an expected worst run-time of $O(n \log_2(n))$ [2]. Below follows code for the Mergesort algorithm.

Listing 3 Mergesort algorithm from [2]

```
MERGE(A, p, q, r)
 1  n1 = q - p + 1
 2  n2 = r - q
 3  L = [0]*n1
 4  R = [0]*n2
 5  for i in list(range(n1))
 6        L[i] = A[p+i-1]
 7  for j in list(range(n2))
 8        R[j] = A[q+j]
 9  L.append(float("inf"))
10  R.append(float("inf"))
11  i = 0
12  j = 0
13  for k in list(range(p-1, r))
14        if L[i] <= R[j]
15              A[k] = L[i]
16              i += 1
17        else
18              A[k] = R[j]
19              j += 1
```

```
MERGESORT(A, p, r)
1  if p < r
2        q = floor((p+r)/2)
3        Mergesort(A, p, q)
4        Mergesort(A, q+1, r)
5        Merge(A, p, q, r)
```

2.4 Python sorted

Pythons built in sorting algorithm is called Timsort after the Python developer Tim Peters and it was invented in 2002 [8]. Timsort is a hybrid that uses both merge sort and insertion sort. Insertion sort is used when the array has fewer than 32 elements. For larger arrays it will divide it into blocks known as run and sort those runs using insertion sort, it then uses mergesort to merge those runs together. Timsort first finds sequences of already ordered elements, these sequences exist in most real world datasets and in Timsort these are called "natural runs". The algorithm iterates over the array creating runs and at the same time merging those runs together into sorted subarrays [15]. Timsort is faster than most sorting algorithms and it can be optimised even more by using it in galloping mode. Galloping mode means that if we are merging A and B and Timsort notice that run A's values are larger than run B's values most of the times, then it assumes that run A's values will continue to be larger than run B's values [14]. The worst case run-time for Timsort is $O(n \log_2(n))$. Below follows code for timsort.

Listing 4 Python_sorted(Timsort) algorithm from [4]

```
1  RUN = 32
```

```
INSERTIONSORT(A, left, right)
1  for i in range(left+1, right+1)
2        temp = A[i]
3        j = i - 1
4        while A[j] > temp and j >= left
5              A[j+1] = A[j]
6              j -= 1
7        A[j+1] = temp
8  L[i] = A[p+i-1]
```

```
MERGE(A, 1, m, r)
 1  len1, len2 = m-l+1, r-m
 2  left, right = [], []
 3  for i in range(0, len1)
 4        left.append(A[l+i])
 5  for i in range(0, len2)
 6        right.append(A[m+1+i]
 7  i, j, k = 0, 0, l
 8  while i < len1 and j < len2
 9        if left[i] <= right[j]
10              A[k] = left[i]
11              i += 1
12        else
13              A[k] = right[j]
14              j += 1
15        k += 1
16  while i < len1
17        A[k] = left[i]
18        k += 1
19        i += 1
20  while j < len2
21        A[k] = right[j]
22        k += 1
23        j += 1
```

```
TIMSORT(A, n)
1  for i in range(0, n, RUN)
2        INSERTIONSORT(A, i, min((i+31), (n-1))
3  size = RUN
4  while size < n
5        for left in range(0, n, 2*size)
6              mid = left + size - 1
7              right = min((left + 2*size - 1), (n-1))
8              MERGE(A, left, mid, right)
9        size = 2*size
```

2.5 Numpy sort

Numpy's sort function uses as of 1.12.0 "introsort" as its standard sorting algorithm [7]. Introsort was invented by David Musser in 1997 and it is a hybrid sorting algorithm [6]. Introsort starts with using quicksort but it changes to heapsort when the recursion depth of the quicksort algorithm exceeds a certain limit. This combines

the good parts of both algorithms, with practical performance comparable to quicksort on standard data sets and a worst-case O(n log n) run-time if the algorithm needs to perform a heapsort. As both quicksort and heapsort are comparison sort algorithms, introsort is also a comparison sort [6]. Numpy sort can also use mergesort or heapsort. In this report introsort was used. Below follows the pseudocode for the introsort algorithm [6]. This algorithm assumes that there exists a heapsort algorithm and partitioning function. The partitioning function is described in more depth in listing 1.

Listing 5 Introsort algorithm from [6]

```
SORT(A)
1  maxdepth = ⌊log2(A.length)⌋ · 2
2  introsort(A, maxdepth)
INTROSORT(A, maxdepth)
1  n = length(A)
2  if n ≤ 1
3       return // base case
4  elseif maxdepth = 0
5       heapsort(A)
6  else
7       p = partition(A) // assume this function does
   // pivot selection, p is the final position of the pivot
8       introsort(A[0:p-1], maxdepth-1)
9       introsort(A[p+1:n], maxdepth - 1)
```

3 METHODS

In this section, we describe how the benchmarks were performed.

3.1 Test-data

The data used was integer arrays of different sizes. The integers were arranged in three different kinds of ways, already sorted, randomly and reversed sorted. We used Numpy to create the arrays. Below follows code for how the data was generated, Numpy has been renamed np.

Listing 6 Data generation using Numpy

```
1  SORTED data
2  test_data = np.arange(1, number_of_elements+1)
3  RANDOM data
4  test_data = np.random.random((number_of_elements,))
5  REVERSE sorted data
6  test_data = np.arange(1, number_of_elements+1)[::-1]
```

3.2 Benchmarks

A custom script benchmark.py was written in Python in which it could be chosen which algorithm to benchmark, how many elements to use, the number of executions of each benchmark, how to order the data and which seed to use for the random generator. The script then generated the chosen data and then created a timeit.Timer object with the chosen parameters. The Timer object then called its repeat method with 7 repeats and 15 executions. This means that a total of 95 executions of the algorithm was run for each benchmark. The average of each repeat was then saved to a pandas dataframe [10] which was saved as a pickle [11] file on the computer.

3.3 Data Analysis

To make the plots in this paper the custom scripts plot_multiple_algorithm and plot_single_algorithm were used. plot_single_algorithm was used to plot the run-times for a single algorithm using sorted, reversed and random data. plot_multiple_algorithm was used to plot the differences between the algorithms for either sorted, reversed or random data. Each script opened the saved pickle files and then calculated the average of the seven run-times as well as their standard deviation using Numpy. This data was saved into lists and then plotted using matplotlib.pyplot [9]. The expected worst run-time for each algorithm was also plotted.

3.4 Hardware and Software

The computer used was a Intel Core i7-4790 CPU @ 3.60Ghz octacore, having 32 gb of RAM and running on a Windows 10 64bit. The software used is listed below.

Table 1: Software used.

Software	Version
Python	3.7.4
pandas	0.25.2
numpy	1.16.5
matplotlib	3.1.1

3.5 Git-hashes

Here we present the files used for the report and the git commit hash of each file.

Table 2: Versions of files used for this report; Github repository https://github.com/juliukvi/inf221benchmarks.

File	Git hash
benchmarker.py	d24cafe
heapsort.py	d24cafe
mergesort.py	d24cafe
quicksort.py	d24cafe
number_of_executions_finder.py	d24cafe
plot_multiple_algorithm.py	d24cafe
plot_single_algorithm.py	d24cafe
plot_multiple_algorithm.py	d24cafe

4 RESULTS

Here we present and comment on the results of the benchmarks that was done.

4.1 Heapsort

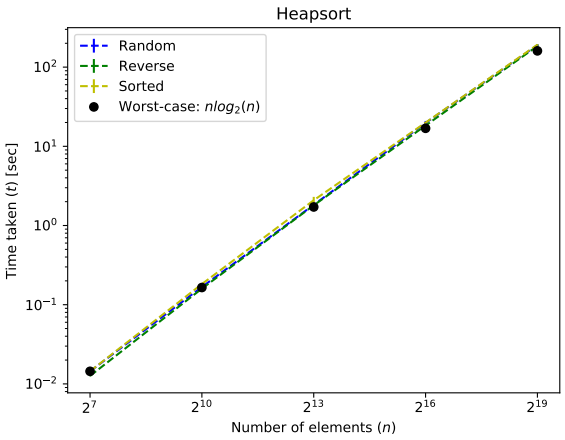

Figure 1: Benchmark results for Heapsort.

We see in figure 1 that there is almost no differences in the running times for the different orderings of the data. The benchmarks follows the theoretical worst run-time of the algorithm. The standard deviations are very small and can barely be seen.

4.2 Mergesort

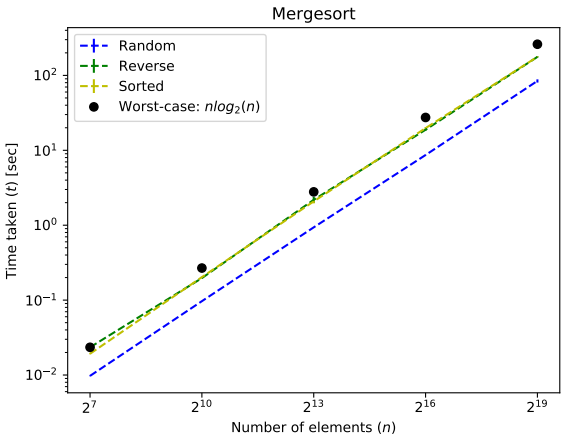

Figure 2: Benchmark results for Mergesort.

We see in figure 2 that the random data is sorted faster than the already sorted and reversely sorted data. All the different orderings of data follow the theoretical worst run-time quite well.. The standard deviations are so small that they can not be seen.

4.3 Quicksort

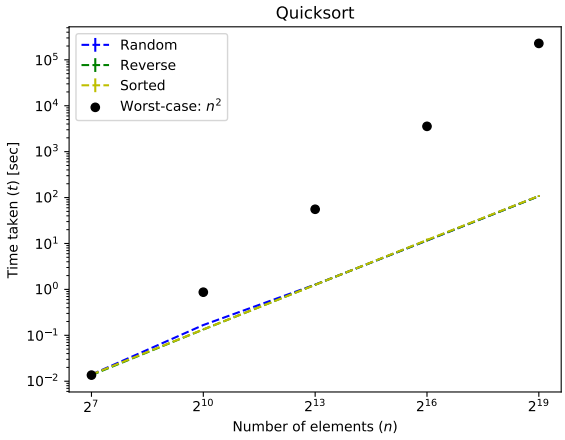

Figure 3: Benchmark results for Quicksort.

We see in figure 3 that there are almost no differences in the running times for the different orderings of the data. The benchmarks does not follow the theoretical worst run-time of the algorithm and the standard deviations are so small that they can not be seen.

4.4 Numpy_sort

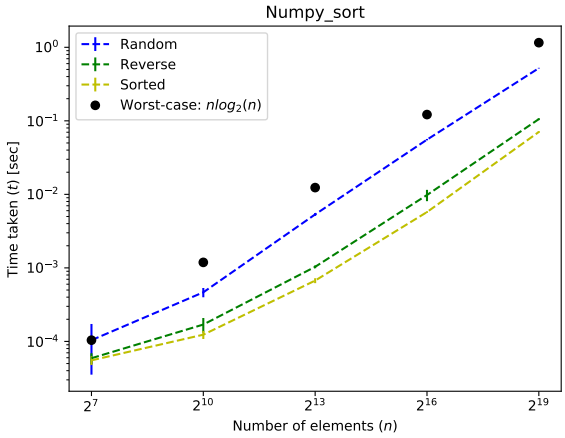

Figure 4: Benchmark results for Numpy_sort.

We see in figure 4 that the sorted ordering of the data is the fastest followed by the reversely sorted data. All of the benchmarks follow the the theoretical worst run-time and the random data is the closest followed by the reversely sorted data. The standard deviations are small except for the first data point of the random data.

4.5 Python_sorted

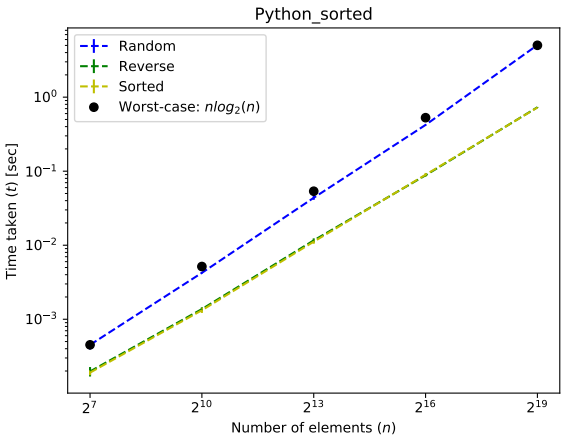

Figure 5: Benchmark results for Python_sorted.

We see in figure 5 that the sorted and the reversely sorted data are equally fast and that the random data is the slowest. The random data follow the theoretical worst run-time of the algorithm and the standard deviations are so small that they can not be seen.

4.6 Random data

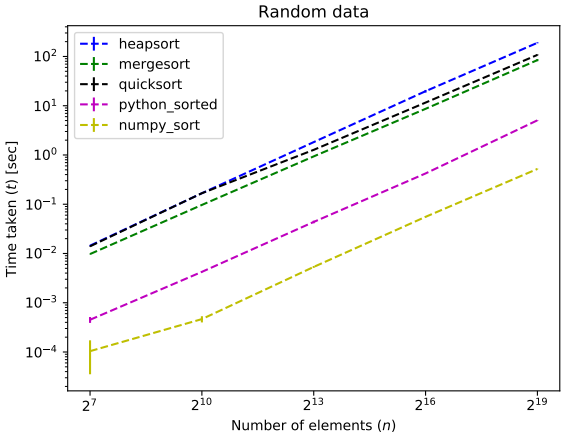

Figure 6: Benchmark results for random data

We see in figure 6 that for random data the order of the algorithms from fastest to slowest is the following, Numpy_sort, Python_sorted, Mergesort, Quicksort and Heapsort. This ordering is consistent for all different sizes of elements. Numpy_sort has a small slope for low element sizes but it increases and becomes similiar to the other slopes as the element size increases. The standard deviation are small except for the first data point of Numpy_sort.

4.7 Reverse data

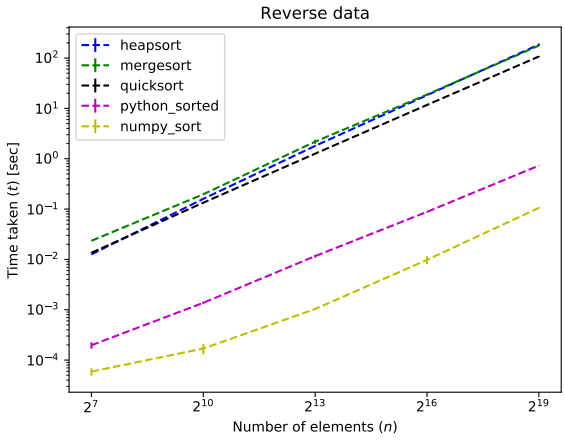

Figure 7: Benchmark results for reversely sorted data

We see in figure 7 that for reverse data the order of the algorithms from fastest to slowest is the following, Numpy_sort, Python_sorted, Quicksort, Heapsort and Mergesort. For a low number of elements Heapsort is similiar to Quicksort but as the number of elements increase it gets closer to Mergesort. At the last data point Mergesort and Heapsort take the same amount of time. The standard deviations are all small, but they are largest for Numpy_sort.

4.8 Sorted data

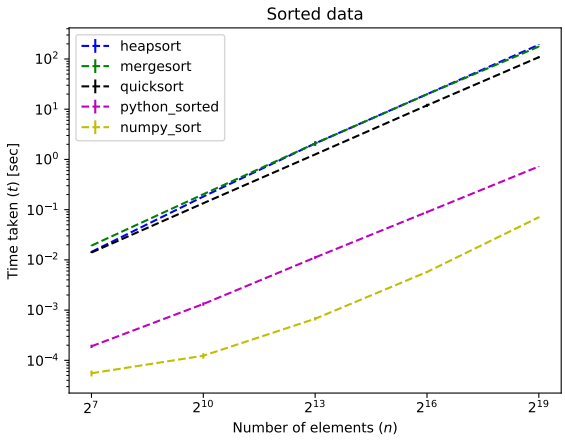

Figure 8: Benchmark results for sorted data

We see in figure 8 that the sorted results are almost identical to the reverse results. The differences are that it takes a smaller element size for Heapsort to take as long time as Mergesort and the standard deviations for Numpy_sort is smaller for the sorted data.

5 DISCUSSION

5.1 Comparison to theoretical worst case run-time

5.1.1 Heapsort. We can see in figure 1 that for all orderings of data the benchmarks for Heapsort follow the theoretical worst case run-time.

5.1.2 Mergesort. We can see in figure 2 that the random data did not follow the theoretical worst case run-time but the sorted and reversely sorted data did.

5.1.3 Quicksort. We can see in figure 3 that none of the orderings of data followed the theoretical worst case run-time for the Quicksort algorithm.

5.1.4 Numpy_sort. We can see in figure 4 that none of the orderings of data followed the theoretical worst case run-time but the random data followed it closer than the sorted and reversely sorted data.

5.1.5 Python_sorted. We can see in figure 4 that the random data followed the theoretical worst case run-time but the sorted and reversely sorted data did not.

5.2 Comparison of algorithms for different ordering of data

5.2.1 Random data. We can see in figure 6 that Numpy_sort is the fastest for all different array sizes. We therefore conclude that for random data and array sizes between 2^7 to 2^{19} one should choose Numpy_sort over the other algorithms if speed is important. The slope of Numpy_sort does not seem to be larger than the other algorithms at the largest array size, this suggests that Numpy_sort will continue to be faster than the other algorithms for larger array sizes than has been tested in this paper.

5.2.2 Reverse data. We can see in figure 7 that Numpy_sort is the fastest for all different array sizes. We therefore conclude that for reverse data and array sizes between 2^7 to 2^{19} one should choose Numpy_sort over the other algorithms if speed is important. The slope of Numpy_sort is larger than the other algorithms at the largest array size, this suggests that if we keep increasing the array sizes the other algorithms will outperform Numpy_sort in terms of speed. Assuming that the slopes stays the same as we increase the array size, we conclude that for array sizes above 2^{25} one should choose Python_sorted for the fastest execution.

5.2.3 Sorted data. We can see in figure 8 that the benchmarks of the sorted data looks almost identical to the reversed data. We therefore make the same conclusion for the sorted data as for the reversed data

5.3 Standard deviations

The standard deviations of the benchmarks were so small that they could be neglected in the analysis of the data. The only data point that showed a large standard deviation was the smallest array size of the Numpy_sort algorithm. This benchmark could be done again in order to see if the large standard deviation depends on the algorithm or if it was caused by other processes on the computer interfering with the benchmark.

5.4 How to improve the benchmarks

The first thing to do if one wants to increase the accuracy of the benchmarks is to increase the number of executions per repeat as well as the number of repeats per benchmark. Doing this will however increase the time the benchmark takes. If one wants to increase the executions and repeats without increasing the time the benchmark takes, a stronger computer has to be used. The benchmarks for the largest array sizes took around 25-30 minutes for the slowest algorithm in our case. Another way to improve this paper would be to run benchmarks on larger arrays, we decided to keep the largest array at 2^{19} because of the fact that we wanted to keep the total time of all benchmarks under six hours. Since the worst case run-time for Quicksort increases as $O(n^2)$ and the other algorithms as $O(n \log_2 n)$ we concluded that increasing the array sizes further could cause us to exceed six hours. One could also decrease the array sizes to see how the algorithms compare for very small arrays. This was not done because we assumed the differences between the algorithms would negligible for such small sizes. To decrease the likelihood of getting large standard deviations on the benchmarks one should make sure that no other demanding processes are started or stopped during the benchmark. If the benchmarks are taking a long time and one does not have access to a faster computer, running the benchmarks on a clean minimal Linux distribution will most likely go faster than using a Windows distribution [1]. Using a newly installed Linux distribution compared to an old Windows distribution also decreases the likelihood that unwanted background processes interfere with the benchmarks.

ACKNOWLEDGMENTS

We are grateful to our Professor Hans Ekkehard Plesser and teaching assistant Krista Marie Erickson Gilman for answering our questions.

REFERENCES

[1] *Benchmarking of different operating systems.* URL: https://www.phoronix.com/scan.php?page=article&item=win10-debian101-intel&num=1 (visited on 11/10/2019).
[2] Thomas H Cormen et al. *Introduction to algorithms.* MIT press, 2009.
[3] Charles Antony Richard Hoare. "Algorithm 64: quicksort". In: *Communications of the ACM* 4.7 (1961), p. 321.
[4] *Intro to Timsort.* URL: https://www.geeksforgeeks.org/timsort/ (visited on 11/10/2019).
[5] Donald Knuth. "Section 5.2. 4: Sorting by merging". In: *The Art of Computer Programming* 3 (1998), pp. 158–168.
[6] David R. Musser. *Introspective Sorting and Selection Algorithms.* 1997.
[7] *Numpy_sort documentation.* URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html (visited on 11/10/2019).
[8] Tim Peters. "Timsort Description". In: (). URL: http://svn.python.org/projects/python/trunk/Objects/listsort.txt.
[9] *Python matplotlib module documentation.* URL: https://matplotlib.org/api/pyplot_api.html (visited on 11/10/2019).
[10] *Python pandas module documentation.* URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html (visited on 11/10/2019).
[11] *Python pickle module documentation.* URL: https://docs.python.org/3/library/pickle.html (visited on 11/10/2019).
[12] Robert Sedgewick. "Implementing quicksort programs". In: *Communications of the ACM* 21.10 (1978), pp. 847–857.
[13] Steven S Skiena. "The Algorithm Design Manual,(2008)". In: *URl: http://dx. doi. org/10.1007/978-1-84800-070-4* ().

[14] *Timsort fastest sorting algorithm for real world problems.* URL: https://dev.to/s_awdesh/timsort-fastest-sorting-algorithm-for-real-world-problems--2jhd (visited on 11/10/2019).

[15] *Timsort the fastest sorting algorithm youve never heard of.* URL: https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399 (visited on 11/10/2019).

[16] John William Joseph Williams. "Algorithm 232: heapsort". In: *Commun. ACM* 7 (1964), pp. 347–348.