Benchmark sorting algorithms

INF221 Term Paper, NMBU, Autumn 2019

Elinor Brede Skårås
elskaras@nmbu.no

Stian Teien
steien@nmbu.no

ABSTRACT

In this paper we perform, analyse and compare different sorting algorithms by performing benchmark experiments with varied input data. The purpose is to compare the results with the theoretical expectations. We examine whether our results matches the theoretical boundaries by looking at the slopes and growth of the data.

Our results for the *Merge sort* and *Quick sort* algorithms are between the upper and lower bound. The run time of the *Quick sort* algorithm is affected by the form of the input data. When it comes to the *Heap sort* algorithm, the results is under the upper bound. We can not make a statement about the lower bound of this algorithm because of lack of information.

Our conclusion is that the algorithms has the expected theoretically behavior and are surrounded by the boundaries.

1 INTRODUCTION

In this paper we will perform a benchmark of a set of sorting algorithms. All algorithms will be tested with a various set of data to see how the algorithms performs. The performance of the algorithms will be analysed and visualized graphically. The results will be present with necessary theoretical background and compared with expected results.

The goal of this paper is to compare the algorithms real-life behavior with the theoretical expectation. To run the benchmark in Python, we need correct syntax and precise code. The test data to be sorted needs to be carefully selected sizes and reset for every new benchmark.

The sorting algorithms is a set of frequently used algorithms. We will perform benchmark on:

- Merge sort
- Heap sort
- Quick sort

In addition to these algorithms we will benchmark the built in sorting functions provided by Python and NumPy. In the benchmarks we will test three different scenarios that is all with integers:

(1) Worst case behavior
(2) Best case behavior
(3) Average case behavior

To evoke this behavior we use four kinds of data sets: Sorted data, nearly sorted data, reverse sorted data and random sorted data. The limit of the largest problem size will be benchmark execution no bigger than six hours.

The paper is set up first with a theory part were all three algorithms are shown in psuedocode and theoretically explained. Then the method is described, followed by the results. The paper ends with a discussion and conclusion of the results.

2 THEORY

We will provide a brief description of the algorithms and investigate these sections:

- Pseudocode of the algorithms
- Expected run time of the algorithms in terms of problem size

2.1 Algorithm 1: Merge sort

The first sorting algorithm we will look at is *Merge sort*. This algorithm is an efficient, general-purpose and uses comparison-based approach meaning it compares and iterates each element. *Merge sort* is a divide-and-conquer algorithm meaning it divides the problem into sub problems then when merging the sub problem back together the algorithm sorts by comparing. The algorithm does not sort in-place, which means that extra space is used during the sorting. [Cormen et al. 2009, Ch. 2.3]

Listing 1 Merge sort algorithm from Cormen et al. [2009, Ch. 2.3].

```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

Pseudocode for the *Merge sort* algorithm is shown in Listing. 1.

Listing 2 Merge procedure from Cormen et al. [2009, Ch. 2.3].

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Pseudocode for the *Merge* procedure is shown in Listing. 2.

The run time for the *Merge sort* algorithm is

$$T(n) = \Theta(n \lg n) \; . \tag{1}$$

The run time of the algorithm is independent of the form of the input. The procedure consists of two equally sized sub problems, each half takes $T(n/2)$ time. Total time is therefore $2T(n/2)$. Combining the sub problems takes $\Theta(n)$ time. The recurrence is:

$$T(n) = 2T(n/2) + \Theta(n) \; . \tag{2}$$

The master method is the procedure to determine recurrence of this form. By the master theorem, the run time of the *Merge sort* algorithm is: $T(n) = \Theta(n \lg n)$.

[Cormen et al. 2009, Ch. 2.3]

2.2 Algorithm 2: Heap sort

The second sorting algorithm is *Heap sort*, which is build on using a max heap to sort the array. The array to be sorted is represented as a binary tree. The *Max heap* property is that the largest element is the root, and every node is less than or equal to its parent.

The *Heap sort* algorithm divides the input into to groups; one sorted and one unsorted. The sorted group increases incremental by moving the largest element from the unsorted group until it is empty. *Heap sort* sorts in-place, and therefore needs less memory usage than *Merge sort*. [Cormen et al. 2009, Ch. 6]

In the execution of the benchmarking, the *Heap sort* Python code is taken from brilliant.org [nd]

Listing 3 Heap sort algorithm from Cormen et al. [2009, Ch. 6.4].

```
HEAP-SORT(A)
BUILD-MAX-HEAP(A)
for $i = A.length$ downto 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    MAX-HEAPIFY(A, 1)
```

Pseudocode for the *Heap sort* algorithm is shown in Listing. 3.

Listing 4 Build max heap procedure from Cormen et al. [2009, Ch. 6.3].

```
BUILD-MAX-HEAP(A)
$A.heap\text{-}size = A.length$
for $i = \lfloor (A.length)/2 \rfloor$ downto 2
    MAX-HEAPIFY(A, i)
```

Pseudocode for the *Build max heap* procedure is shown in Listing. 4.

Listing 5 Max heapify procedure from Cormen et al. [2009, Ch. 6.2].

```
MAX-HEAPIFY(A, i)
$l$ = LEFT(i)
$r$ = RIGHT(i)
if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY(A, largest)
```

Pseudocode for the *Max heapify* procedure is shown in Listing. 5.

The upper bound for the run time of the *Heap sort* algorithm is

$$T(n) = O(n \lg n) \; . \tag{3}$$

We did not have good, available information about the lower bound, the best case run time. The run time of the algorithm is independent of the form of the input. The *Heap sort* call to the *Build max heap* takes the time $O(n)$. The *Build max heap* function call to the *Max heapify* function $n - 1$ times. Each time, *Max heapify* runs $O(\lg n)$ times. The run time of the *Heap sort* algorithm is therefore $O(n \lg n)$. [Cormen et al. 2009, Ch. 6.4]

2.3 Algorithm 3: Quick sort

The third algorithm we will look at is *Quick sort*. The algorithm uses the divide-and-conquer approach, like *Merge sort*. The key to the *Quick sort* algorithm is the *Partition* procedure. The array is divided into to sub arrays, and both are then sorted. In the end the sub arrays are combined. [Cormen et al. 2009, Ch. 7.1]

Listing 6 Quick sort algorithm from Cormen et al. [2009, Ch. 7.1].

```
QUICK-SORT(A, p, r)
if $p < r$
    $q$ = PARTITION(A, p, r)
    QUICK-SORT(A, p, q - 1)
    QUICK-SORT(A, q + 1, r)
```

Pseudocode for the *Quick sort* algorithm is shown in Listing. 6.

Listing 7 Partition procedure from Cormen et al. [2009, Ch. 7.1].

```
PARTITION(A, p, r)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$
```

Pseudocode for the *Partition* procedure is shown in Listing. 7.

The run time of *Quick sort* depends on the balance of the partitioning.

(1) Worst case run time
The worst case run time for the *Quick sort* algorithm is
$$T(n) = \Theta(n^2)\ . \tag{4}$$
It occurs when the partitioning is the most unbalanced. The *Partition* produces two sub problems, one with n - 1 elements and the other 0 elements. This will be the case when the data is already sorted (also reversed order), and if all of the the elements in the data are equal.

(2) Best case run time
The best case run time for the *Quick sort* algorithm is
$$T(n) = \Theta(n \lg n)\ . \tag{5}$$
It is achieved when it is the most even partitioning. The *Partition* produces two sub problems, each with maximum n/2 elements.

(3) Average case run time
The average case run time for the *Quick sort* algorithm is
$$T(n) = O(n \lg n)\ . \tag{6}$$
[Cormen et al. 2009, Ch. 7.2]

3 METHODS

This section describes the methods behind how to make the data and execute the benchmarks.

3.1 Data sets

The data that will be sorted in the tests will vary in size and form. The key essentials is the increasement of the data and form of the content. The size of the problem will increase binary. The problem will increase by a factor of 2^n. Starting at $n = 1$.

With binary growth the problem size will quickly increase in size and computation time. Here is an illustration on how the problem size will quickly grow. The most interesting result will we get from big and small arrays. We also have a few ones in between.

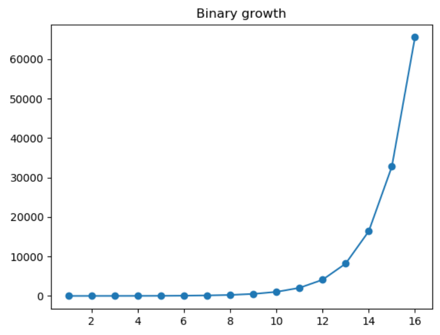


Figure 1: Binary Growth

The problem form of the data will be four types:

- Sorted
- Random
- Nearly sorted
- Reversed

And can be illustrated like this: [topdal.com nd]

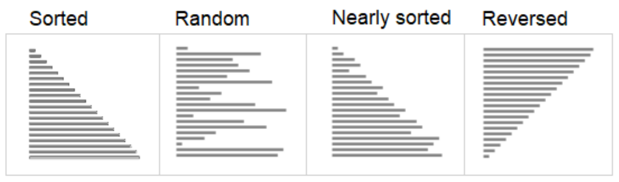


Figure 2: Data problem forms

3.2 How data was created

There will be created new sets of data for every time we will sort. By recreating the problem for each time the algorithms will sort it we will use a seeded random function. This is because we want each algorithm to be comparable when solving problem of same size.

In the same time there will be enough problem sizes to create a variety of problem shapes. This is how we created the four kinds of problems. The two first is straight forward and simple. The Sorted array creator will make an array from 0 to 2^n. The Random array creator will use this function and just shuffle it:

Listing 8 Sorted array creator

```
problem-generator(n)
return [i for i in range($2^n$)]
```

Listing 9 Random array creator

```
random-array(n)
random.seed(12345)
return random.shuffle(problem-generator(n))
```

The reversed sorted problem in Listing. 10 will be made by loop starting on n going down to 1, and appending every i.

Listing 10 Reverse sorted problem

```
reversed-array(n)
return [i **for** i = $2^n$ downto 1]
```

The one most advanced sorting problem to make is the nearly sorted array. This we made our self and to make it work we need to split the array up in sections and inside the section it could be randomly distributed. The array will always be split in n^2 parts.

Listing 11 Nearly sorted problem

```
nearly-sorted-array(n)
random.seed(12345)
array = range(1,$2^n$)
nearly-sorted-array = []
split = math.ceil($2^n/n^2$)
for i = range(0, len(array), split):
    temp-array = list(array[i:i+split])
    random.shuffle(temp-array)
    nearly-sorted-array += temp-array

return nearly-sorted-array
```

To get the data nearly sorted, the problem size needs to be at least $2^5 = 32$. This is because the rule we set up will split sections with only 1 integer until n is bigger than 4.

$$\begin{aligned} 2^n &> n^2 \\ n &> 4 \end{aligned} \tag{7}$$

3.3 Execute benchmark and measuring

To measure the time an algorithm takes to execute a sorting problem we need to take note of the time the algorithm stars and the time it stops. By subtracting the end time with start time we get the run time of the problem.

$$runtime = endtime - starttime \tag{8}$$

Luckily python has a own package called *timeit* that will collect the time of an execution. The timer will give back the time it takes in milliseconds. In our timer we use a function from INF200 called time-sort.

This function will run a timer on the sorting algorithms and if it takes to short of a time (>0.2s) the timer runs the algorithm again. Returning the time divided by the amount of runs it did, meaning it returns the average time used. We can also decide how many times we want to test the same data. Usually it would be just one time. In Listing. 12 the $make - problem(...)$ will create the array depending on which one we actually want.

Listing 12 Draft benchmark setup.

```
times = []
for k in range(1, 20):
    n = make-problem(2**k)
    times.append(time_sort(func, n, num-reps=1))
```

3.4 Hardware and software

Every computer has different qualities and characteristics. These differences will effect how long the run time is and how long the benchmark will take. To make the benchmarks a little more standardized we only had Spyder and Google Chrome open when the tests ran.

3.4.1 *Hardware.*

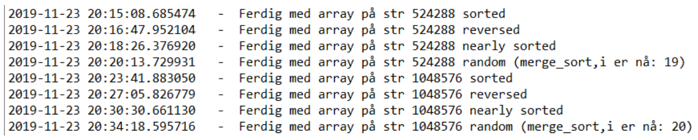
The model of the computer the test ran on is Lenovo Yoga 910. This has a CPU called Intel Core i5-7200U at a clock speed of 2.50 GHz, but can have a "boost" modus or something at 2.71GHz. The RAM of the computer is 8.00GB where 7.84GB can actually be used. It also have the possibility to be flipped into tablet if we want to do a simulation in tablet mode.

3.4.2 *Software.*

The simulation was written in Python 3.6 and ran with Spyder's python terminal. The numpy version we used was 1.17.4.

3.5 Problems and challenges

When we created the simulation and the size of the problem increased the simulation took long time and did everything behind the scenes. This made us not in control on how long the simulation have sorted or if it actually have stopped. Therefore we implemented a logging function to log every time an algorithm was done sorting and adding a date stamp in front. This gave us the possibility to run several simulation up to an hour long in control on of long it have lasted and the possibility to observe if something went wrong. Here is a sneak peak of how it looked like:



```
2019-11-23 20:15:08.685474   -  Ferdig med array på str 524288 sorted
2019-11-23 20:16:47.952104   -  Ferdig med array på str 524288 reversed
2019-11-23 20:18:26.376920   -  Ferdig med array på str 524288 nearly sorted
2019-11-23 20:20:13.729931   -  Ferdig med array på str 524288 random (merge_sort,i er nå: 19)
2019-11-23 20:23:41.883050   -  Ferdig med array på str 1048576 sorted
2019-11-23 20:27:05.826779   -  Ferdig med array på str 1048576 reversed
2019-11-23 20:30:30.661130   -  Ferdig med array på str 1048576 nearly sorted
2019-11-23 20:34:18.595716   -  Ferdig med array på str 1048576 random (merge_sort,i er nå: 20)
```

Figure 3: Logging

Table 1: Versions of files used for this report; Github repository https://github.com/stianteien/Term_Paper_INF221_H19.

File	Git hash
`main.py`	4dccbcc
`make_problem.py`	9c71e1a
`time_sort.py`	b51ed78
`merge_sort.py`	b51ed78
`heap_sort.py`	be656b7
`quick_sort.py`	b51ed78

4 RESULTS

The results are shown in graphs where the x axis is the size of the problem. The y axis is the response variable to x, showing how long our function took measured in seconds. The four lines represent the four kinds of problems. The dots is real measurement. The dotted lines are the lower and upper bounds.

4.1 Merge sort results

For the *Merge sort* algorithm, the biggest problem with a size of little over 1 million elements took 12 seconds to be completely sorted. All four problems took about the same time.

The results of the tests are shown graphically in Figure. 4

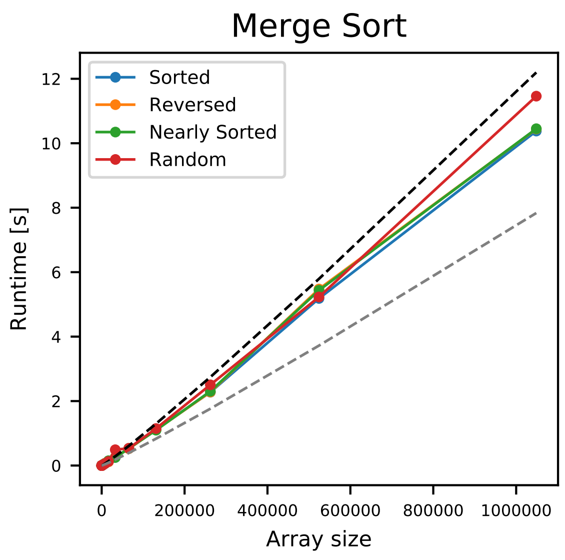


Figure 4: Merge Sort results

4.2 Heap sort results

The *Heap sort* uses the heap structure in an array. The heap structure and sorting doesn't separate between the different problem, and does the sorting approximately in the same amount of seconds.

The results of the tests are shown graphically in Figure. 5

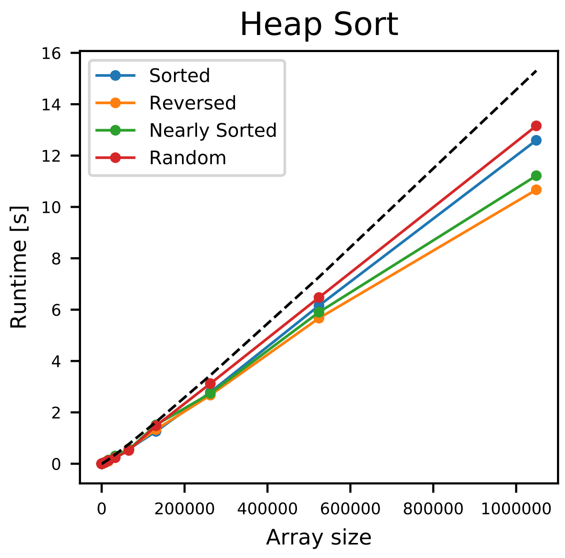


Figure 5: Heap Sort results

4.3 Quick sort results

Quick sort is a rather fast sorting algorithm as long as the data is random. As we see in the results random data took by far the least time to sort. The already sorted array was the hardest one and took the longest time. A problem with testing *Quick sort* is that Spyder raised an RecursionError if the problem size became to big. It said: maximum recursion depth exceeded in comparison. This happen when the problem size was larger than $2^{10} = 1042$. Therefore the array size is smaller for the *Quick sort* results.

The results of the tests are shown graphically in Figure. 6

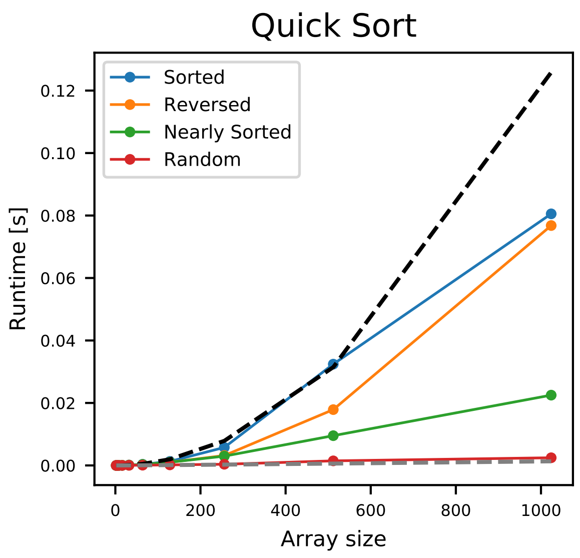


Figure 6: Quick Sort results

4.4 Numpy sort results

Numpy sort is usually used to just sort numpy arrays, and numpy arrays always consist of numbers. This makes the sorting process much faster because Python doesn't need to check if the element is an integer or string (or something else for that matter). The random array took longer time to sort compared to the other arrays.

The results of the tests are shown graphically in Figure: 7

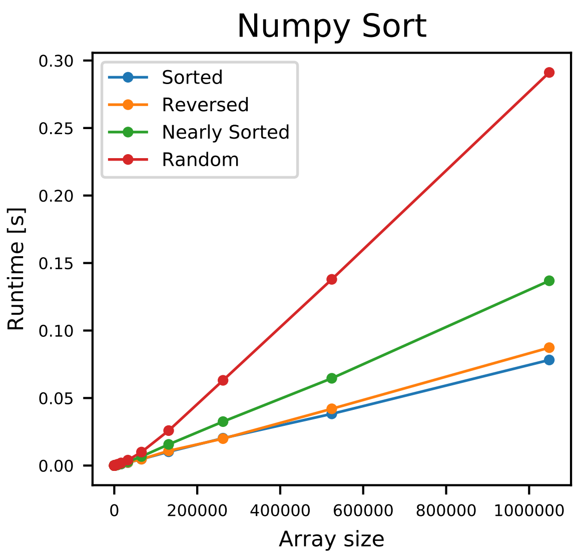


Figure 7: Numpy Sort results

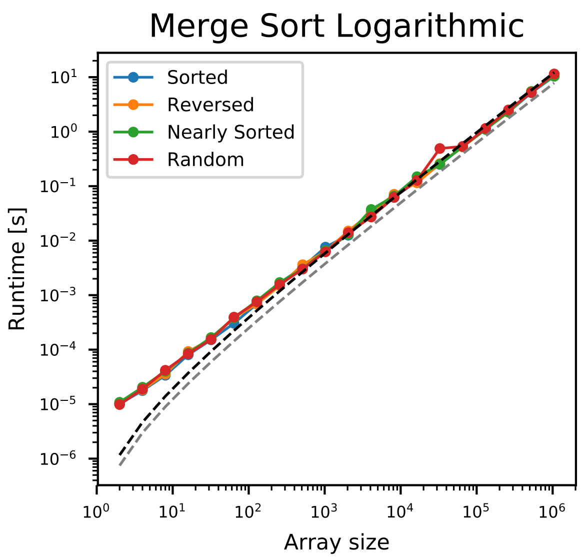


Figure 9: Merge Sort logarithmic

4.5 Sorted sort results

The *Sorted sort* algorithm is the one integrated standard in Python.

The results of the tests are shown graphically in Figure: 8

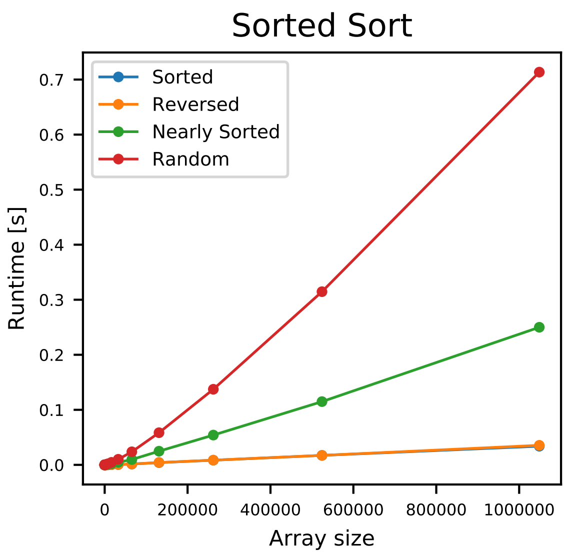


Figure 8: Sorted Sort results

4.6 Logarithmic figures

Here is a logarithmic figure of all five results. The axes are now logarithmic to get a better presentation of the smallest results. The Runtime[s] and Array size are logarithmic and only comparable to the other logarithmic graphs.

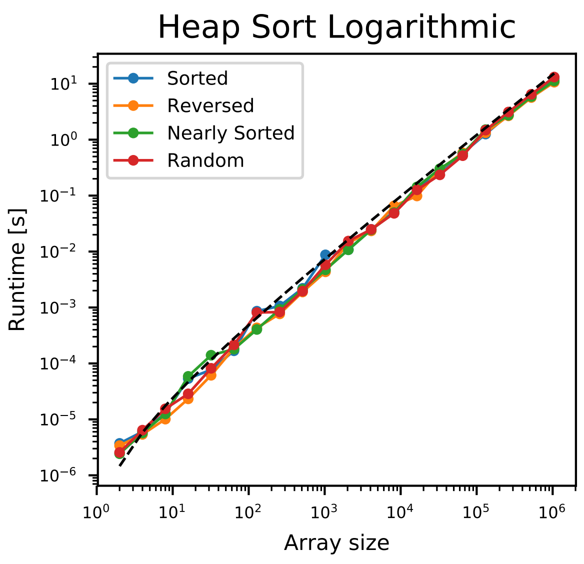


Figure 10: Heap Sort logarithmic

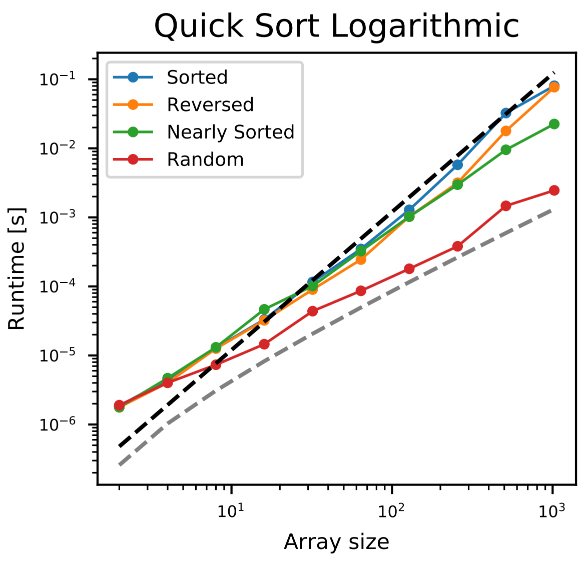


Figure 11: Quick Sort logarithmic

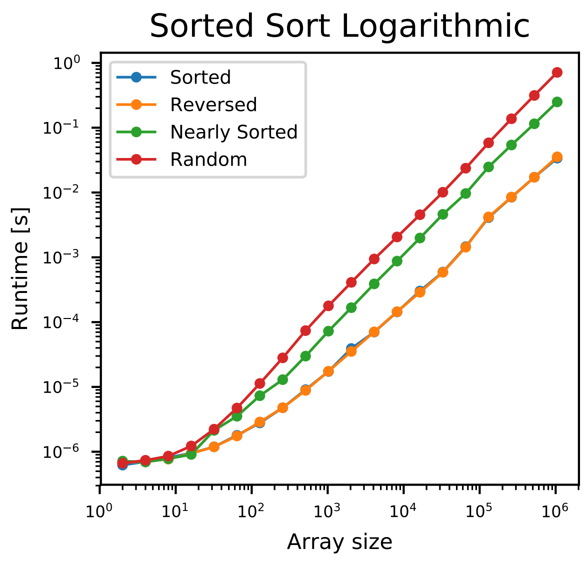


Figure 13: Sorted Sort logarithmic

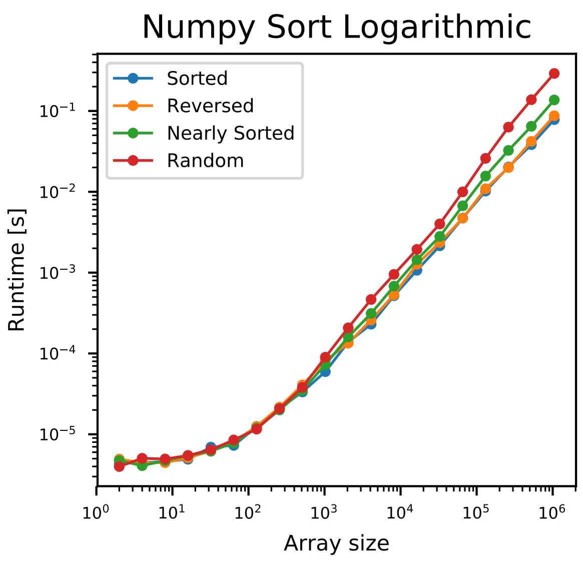


Figure 12: Numpy Sort logarithmic

5 DISCUSSION

5.1 Summarized results

Table. 2 shows the results of sorting the problem with random shuffled elements. This was the array that the algorithms used the longest time sorting, except the *Quick Sort* algorithm who did the random array the fastest. Therefore the table is good for comparing and see the results for the different algorithms.

Table 2: Seconds to solve random array

Size	Merge	Heap	Quick	Numpy	Sorted
2	9.46e-06	2.74e-06	2.01e-06	4.42e-06	7.53e-07
4	2.01e-05	6.17e-06	3.09e-06	4.51e-06	7.57e-07
8	3.66e-05	1.36e-05	6.40e-06	5.09e-06	9.56e-07
16	7.39e-05	3.07e-05	1.58e-05	4.90e-06	1.26e-06
32	0.00015	7.26e-05	4.54e-05	5.64e-06	2.29e-06
64	0.00035	0.00018	8.18e-05	7.76e-06	4.68e-06
128	0.00076	0.00036	0.00022	1.25e-05	1.09e-05
256	0.00161	0.00098	0.00044	2.00e-05	2.72e-05
512	0.00298	0.00240	0.00121	3.70e-05	7.39e-05
1024	0.00631	0.00450	0.00190	9.10e-05	0.00018
2048	0.01588	0.01147	-	0.00021	0.00040
4096	0.03000	0.02534	-	0.00046	0.00091
8192	0.06010	0.06141	-	0.00091	0.00211
16384	0.12008	0.12132	-	0.00184	0.00453
32768	0.26705	0.26083	-	0.00478	0.01018
65536	0.54124	0.58792	-	0.00975	0.02553
131072	1.63811	1.36199	-	0.02801	0.05810
262144	3.02663	2.93190	-	0.06199	0.13748
524288	6.94837	6.35870	-	0.13671	0.31482
1048576	11.3000	13.7010	-	0.29027	0.71515

5.2 Finding the constants c_1 and c_2

To find the constants c_1 and c_2 for *Merge-* and *Heap sort*, we used the theory from sec. 2. $f(n)$ is how long the sorting algorithm took:

$$\begin{aligned} c_1 n \lg n \leq f(n) &\leq c_2 n \lg n \\ c_1 &\leq \frac{f(n)}{n \lg n} \\ c_2 &\geq \frac{f(n)}{n \lg n} \end{aligned} \tag{9}$$

and for *Quick sort*:

$$\begin{aligned} c_1 n^2 \leq f(n) &\leq c_2 n \lg n \\ c_1 &\leq \frac{f(n)}{n^2} \\ c_2 &\geq \frac{f(n)}{n \lg n} \end{aligned} \tag{10}$$

This gave us the results from using table 2 as reference:

$$\begin{aligned} Mergesort: & c_1 \leq 8.39 * 10^{-7} \\ & c_2 \geq 5.39 * 10^{-7} \\ Heapsort: & c_1 \leq 10.53 * 10^{-7} \\ Quicksort: & c_1 \leq 1.20 * 10^{-7} \\ & c_2 \geq 1.86 * 10^{-7} \end{aligned} \tag{11}$$

As we see in the results the bounders really apply when n is getting over a size of 100 to 200.

5.3 Results versus theory

We examine whether our results matches the theoretical boundaries by looking at the slopes and growth of the data.

From the theory presented in Sec. 2, the run time of *Merge sort* is independent of the form of the input data. The run time has an upper and lower bound, $c_1(n \lg n)$ and $c_2(n \lg n)$, respectively. This is supported by the data shown in Figure 4, with the suitable constants $c_1 = 8.39 * 10^{-7}$ and $c_2 = 5.39 * 10^{-7}$. Our results from the *Merge sort* shows that the results is between the upper and lower bound, and we can conclude that the theoretical boundaries are correct.

According to the theory, the run time of *Heap sort* algorithm has an upper bound $c_1(n \lg n)$, and is also independent of the form of the input data. The theory about the upper bound is supported by the data shown in Figure 5, with the suitable constant $c_1 = 10.53 * 10^{-7}$. Our results are below the upper bound. Due to the lack of information about the lower bound of the *Heap sort*, we can not make a statement about this. We can not conclude that our results are over the lower bound, but we see that all the results are under the theoretical upper bound. Therefore we conclude that the theoretical upper bound is correct.

The run time of *Quick sort* algorithm depends on the form of the input. The theoretical run time has the upper and lower bounds, $c_1(n^2)$ and $c_2(n \lg n)$, respectively. This is supported by the data shown in Figure 6, with the suitable constants $c_1 = 1.20 * 10^{-7}$ and $c_2 = 1.86*10^{-7}$. Our results are between the upper and lower bound. We therefore conclude that the theoretical boundaries are correct. The results show that the random data gives the best case run time. When the data is random sorted, the partition is balanced and the best case run time occurs. The results with the Sorted data shows the worst case run time. When the data is sorted, the partition is unbalanced and the worst run time occurs.

We can conclude that the algorithms has the expected theoretically behavior and are surrounded by the boundaries explained above.

ACKNOWLEDGMENTS

We are grateful to Prof. Hans Ekkehard Plesser for great guidance and feedback. We also would like to thank teacher assistant Krista Marie Erickson Gilman for good advice.

REFERENCES

brilliant.org. n.d.. Heap Sort. https://brilliant.org/wiki/heap-sort/
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.
topdal.com. n.d.. Sorting Algorithms Animations. https://www.toptal.com/developers/sorting-algorithms