Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2019

Bishnu Poudel
bishnu.poudel@nmbu.no
MS in Data Science, NMBU

Mohamed Radwan
mohamed.radwan@nmbu.no
MS in Data Science, NMBU

ABSTRACT

In this paper, we measure the run-times for three sorting algorithms, quicksort and mergesort which use divide-and-conquer approach and heapsort which uses max-heap approach. In addition to these three, the NumPy built-in sort and Python sorted functions are also timed. Then we compare these experimental run-times with the expected theoretical run-times. We use plots and descriptive statistics to analyse our observations and to draw conclusions. We are working with list of floats ranging from just ten elements to 10.5 million. As expected, all the sorting algorithms have run-times that follow the theoretical order of $\Theta\left(n \log\left(n\right)\right)$, but the constant factor is different, the ration being 500 between the fastest and the slowest algorithm.

1 INTRODUCTION

Sorting and searching are very basic and routine operations for computers of any type. Much research has already been done on the subject, and software today use the most robust algorithms that are available in order to meet their particular requirements. Many applications use a combination of two or more algorithms depending on the size of input data. However, it would be interesting to see how the sorting algorithms behave in a machine.

The main questions we are trying to address here are: Do the algorithms show their theoretical average case behavior in real situations? Which of the merge, quick and heapsort is more efficient? Do these three sorts have a chance to do a perform better than the NumPy built-in sort and Python sorted functions? How are the run-times of each sort distributed, is there any statistical significance in the distribution?

In the Theory section, we describe the pseudo-code of the algorithms together with their theoretical run-times. We discuss the best, average and the worst cases. In the Methods section, we describe the Python implementation of the algorithms, and also the Python-functions we wrote to extract the run-time information. We also discuss the type and amount of data we collected. Results section has facts and figures from our analysis. In the Discussion section, we summarize our findings and compare them to the theoretical expectations.

2 THEORY

We briefly discuss the sorts we are benchmarking here. By following the given pseudo-codes, we wrote Python implementations for heap, merge and quicksort. NumPy's **sort** function and Python's **sorted** function were used in their default form without additional parameters. The three permutations of data we're using in our analysis are random data, data sorted in ascending order, and data sorted in descending order. The output is sorted in ascending order.

2.1 Heapsort

Heapsort uses the max-heap (or min heap property) to sort elements in an array. First we build a max heap (BUILD-MAX-HEAP) out of the given array. Then inside a loop, we isolate the root of the heap from the array, storing it at the end of the array. Then we call MAX-HEAPIFY on the rest of the array, storing away the root each time. The heap size decreases by 1 in each of the MAX-HEAPIFY calls. heapsort is not stable i.e. the keys having exact value might be interchanged. Therefore, it might not be useful while sorting with multiple keys, for instance in a database table.

Heapsort does the sorting in place, so no extra memory is required. Theoretically, The three cases of heapsort (worst, best and average) follow the run-time notation $\Theta\left(n \log\left(n\right)\right)$. It will be interesting to see which of the three sets of data (random, ordered, reversely ordered) data performs better for heapsort.

Listing 1 Heapsort algorithm pseudo-code from Cormen et al. [2009, Ch. 6.4].

```
1 HEAP-SORT(A)
2     BUILD-MAX-HEAP(A)
3     for i = A.length downto 2
4         swap A[1] and A[i]
5         A.heap_size = A.heap_size - 1
6         MAX-HEAPIFY(A, 1)
7     return A
```

2.2 Mergesort

The function mergesort divides the list of keys repeatedly until the list has 1 element each in which case the MERGE function gets called to work for the first time. It first merges a set of two lists with single element each. Then it starts merging two lists with two elements each, both of which are already sorted and so on.

Mergesort has the run-time is $\Theta\left(n \log\left(n\right)\right)$ in all of its best, average and worst cases. It will be interesting to see if the already sorted data is the best case for a mergesort. It does not sort the keys in place, but it is a stable sorting algorithm.

2.3 Quicksort

In case of quicksort, most of the work is done by the PARTITION function. It does the in-place swapping of the list

Listing 2 Mergesort algorithm pseudo-code (modified after Cormen et al. [2009, Ch. 2.3]).

```
MERGE-SORT(A,p,r):
    if p = r:
        return A[p]
    q = floor( (p+r) / 2)
    B = MERGE-SORT(A,p,q)
    C = MERGE-SORT(A,q+1,r)
    D = MERGE(B,C)
    return D
MERGE(A, L, R)
    i=1
    j=1
    k=1
    While i <= L.Length and j <= R.Length
        if L[i] < R[j]
            A[k] = L[i]
            i=i+1
        else A[k]=R[j]
            j = j+1
        k=k+1
    while $i <= L.Length
        A[k] = L[i]
        i=i+1
        k=k+1
    while j <= R.Length
        A[k] = R[j]
        j=j+1
        k=k+1
    return A
```

Listing 3 Quicksort algorithm pseudo-code 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)
PARTITION(A, p, r)
  x = A[ floor(p+r)/2 ]
  i = p-1
  for j= p to r-1
    if A[j] <= x
      i = i + 1
      exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i+1
```

elements. PARTITION function also returns the position of the pivot element to the quicksort function. Unlike mergesort, Recursion happens after the PARTITION function, so we could change the recursive function to a loop as well. Recursive approach has been used in this paper.
Quicksort has a average and best case run-time of the order of $\Theta\left(n \log\left(n\right)\right)$, the best case has a smaller constant of course. The worst case occurs when each subsequent partition has only one less element than the last partition and will have a theoretical run-time of $\Theta\left(n^2\right)$. It will be interesting to see if we face this scenario!

2.4 NumPy's NumPy.sort

The NumPy sort function uses quicksort by default and a combination of other sorts depending on the input data. Quicksort, mergesort, radixsort, timsort and heapsort are used by NumPy sort based on the input data and the current state of the sort process.
Theoretically, the order of run-time for NumPy sort is also $\Theta\left(n \log\left(n\right)\right)$. More details about NumPy sort can be found here SciPycommunity [2019]

2.5 Python's default sorted

Python uses another kind of sort called timsort Peters [2019]. It is a better version of mergesort, where the algorithm first analyzes the input data and isolates portions that are already sorted. That might explain why Python sorted is faster for sorted and reverse-sorted data compared to random data.
Theoretically, the order of run-time for Python sorted is also $\Theta\left(n \log\left(n\right)\right)$.

3 METHODS

We worked with data sizes ranging from just 10 elements to 10485760 elements. The lists were randomly generated floating point numbers. We generated them as NumPy arrays, but converted them to list before feeding into the algorithms we were benchmarking. The data is generated using for-loop to give data sizes from 10 to 10485760.

3.1 Python implementation of algorithms

Python implementations for heapsort and quicksort were written following the pseudo-code from Cormen et al. [2009]. Algorithm for mergesort was slightly different from Cormen et al. [2009]. All the Python codes, data generated and figures can be found at Github
The MERGE-SORT function in our algorithm returns a merged list (merged from two sub lists), while the one in Cormen et al. [2009] does not return anything. In Cormen et al. [2009], It just modifies the array in the MERGE function. Also, he MERGE function in Cormen et al. [2009], takes the index of the start, end and middle of a list and creates two sub-lists. Then it merges the two lists by comparing them element-wise. In our algorithm, MERGE function takes two lists as input and starts comparing them element-wise. We also return the merged list back to the MERGE-SORT function.

3.2 Timing function

We used the time-it library to record run-times of the algorithms for a range of data-sizes. The skeleton of the function was provided by professor H.E. Plesser, to which we added parameters. Since the process can slow down due to other processes in the computer we repeated the time-it experiment seven times. We use the fastest time among the seven readings in most of our analysis. We use the same seed at all times, to ensure the data is exactly same as shown in Listing 4.

Listing 4 Time it function used with parameters

```
import NumPy as np
import timeit
import copy

def timing_function(number_of_data_points
, sort_type,randomization_type, seed_number=12235):
    np.random.seed (seed_number)
    test_data = np.random.random(
                number_of_data_points,)
    test_data = list(test_data)

    if randomization_type=='reverse':
        test_data= sorted(test_data, reverse =True)
    elif randomization_type=='sorted':
        test_data= sorted(test_data)

    clock=timeit.Timer(stmt='sort_func(copy(data))',
            globals ={'sort_func': sort_type,
            'data': test_data,
            'copy': copy.copy })
    n_ar , t_ar = clock.autorange ()
    n_ar , t_ar = clock.autorange()
    t = [tm / n_ar for tm in clock.repeat(repeat=7,
     number=n_ar)]
    return t, n_ar
```

Then we defined a helper function displayed in Listing 5 in order to call timing function with the range of sizes from 10 to 10485760. The time information is stored in a pandas dataframe and saved into .csv format files.

3.3 Data Analysis

We imported the .csv files exported from the timing step and combined them to a single dataframe. Then we compared the run-times of the five algorithms for all data sizes in a number of line graphs. Data-sizes ranging from 80000 and above were plotted as we're interested in the behavior of algorithms for large data size.

Additionally, we plotted box-plots of the seven run-times we got for each of the algorithms to see if they hold any statistical significance. We also compared the run-time for reverse sorted and already sorted data and see if it drastically different than the random data for any of the algorithm. The line plots were plotted in seconds, so that the constants for the run-time, for instance the c_1 in $c_1 n \log\left(n\right)$ remains within two decimal points.

Python implementation for plotting have been included in the GitHub hashes section 3.5. Also, the colour scheme for the line graphs was taken from Rivera-Thorsen [2013].

3.4 Hardware and software Specifications

The run-times were obtained from a machine with the following configuration.

- Processor: Intel Xeon CPU E5-1607 v4- 3.10 GHz
- Memory : DDR4 32 GB, 2133 MHz
- OS : Windows 10 Enterprise, 64 bit

Following are the versions of packages used in Python 3.7.4:

Listing 5 Call timeit function and save data to .csv

```
import pandas as pd
import NumPy as np

def get_time_and_write_to_dataframe(Algorithm,
     randomize_type):
    time_data_points= []

    for i in (10,20,40,80,160,320,640,1280,2560,5120
     ,10240,20480,40960,81920,163840,327680,655360,1310720
                ,2621440,5242880,10485760):
        time , n_ar =   timing_function(i, Algorithm ,
      randomize_type)
        for each_time in time:
#             print (each_time)
            time_data_points.append( {'Sort_Type':
     Algorithm.__name__ , 'Data_Type_or_List_type':
     randomize_type
           ,'List_length':i, 'Runtimes': each_time ,
     'Number_of_repeatitions':n_ar
           ,'Datetime':pd.Timestamp.now() } )
    return time_data_points

# Call the get_time.. function and store each file to
     csv

list_of_sorts = [np.sort, sorted ,quick_sort,
     merge_sort,heap_sort]
list_of_randomize = ['random', 'reverse_sorted', '
     sorted']
path ="C:\\Users\\bipo\\OneDrive - Norwegian
     University of Life Sciences\\termpaper01\\
     plots_and_data\\csvs\\_20191120\\"

for sort in list_of_sorts:
    for permut in list_of_randomize:
        columns = ['Sort_Type','Data_Type_or_List_type
     ','List_length','Runtimes','
     Number_of_repeatitions','Datetime']
        df_ = pd.DataFrame( columns=columns)
        df_
        list_of_dict = get_time_and_write_to_dataframe
     (sort, permut)
        df_= df_.append(list_of_dict)
        df_.to_csv (path+sort.__name__+'_'+permut+'.
     csv', index = None, header=True)
        del df_
```

- Pandas: 0.25.1
- Pandasql: 0.7.3
- NumPy : 1.16.5
- matplotlib : 3.1.1
- nbimporter : 0.3.1
- ipywidgets: 7.5.1

3.5 Benchmark data and Python codes

The final version of appended .csv file, the Python notebooks, the plots exported, and the source .tex file are in github as listed in table 1.

4 RESULTS

From the data collected we try to answer the questions as listed in the introduction section 1.

Table 1: Github repository details for files used https://github.com/vsnupoudel/termpaper01.

File	Git hash
plots_and_data/dataset_concat.csv	fb2689b
plots_and../Line_plots_interactive-LogLog	fb2689b
plots_and_data/Box_plots.ipynb	fb2689b
Time_it.../time_it_function.ipynb	fb2689b
plots_and_data/Statistical_analysis.ipynb	0762c23
Time_it.../heap_sort_heap_size.ipynb	9856f1c
Time_it.../merge_sort.ipynb	9856f1c
Time_it.../quick_sort.ipynb	fb2689b

4.1 Box plot of run-times for random data

To begin, we plotted the box plot of all the twenty one time-points (for all three orderings) to check if we observe any anomalies or outliers. In the analysis that follows, we use only the minimum time among the seven observations.

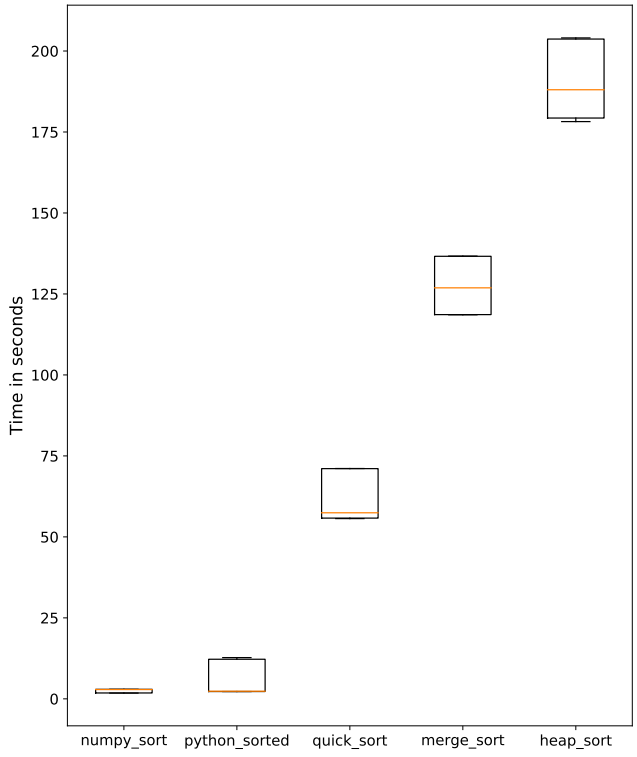

Figure 1: run-times for data size: 10485760

In figure 1 the run-times have quite a distinct separation, from which we can quickly decide the fastest and slowest algorithm. There is overlap between numpy-sort and python-sorted which we will investigate further.

4.2 All sorts for a particular permutation

Figures 2-3 that follow present the run-time of all algorithms against the same randomized data. We have split the data sizes from 10-40960 in Figure 2 and from 81920 to 10485760 in Figure 3. In the rest of the figures we will only analyse the second interval as we're interested in the asymptotic behavior of the algorithms, so our $\mathbf{n_0 = 81920}$.
We can see that NumPy **sort** is the most efficient for large list size. Python's **sorted** function follows NumPy **sort**.

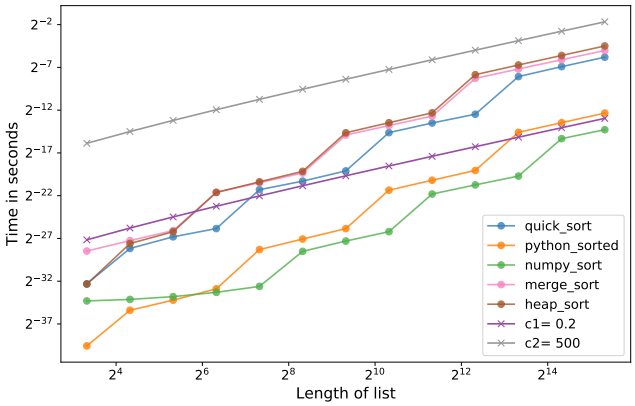

Figure 2: Randomized data of size 10-40960

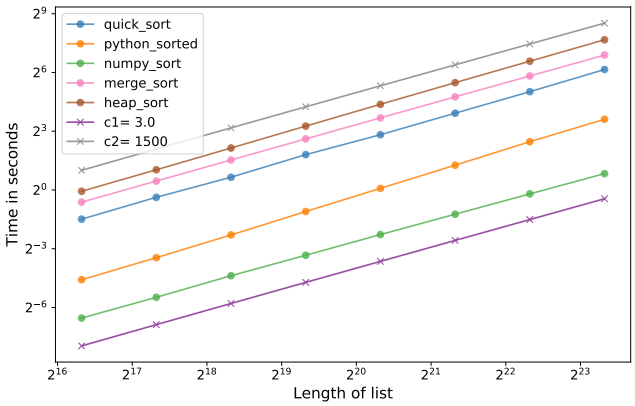

Figure 3: Randomized data of size 81920-10485760

We also have plotted a lower boundary, just below NumPy sort and a upper boundary just above heapsort in order to prove that our algorithms do run asymptotically in a run-time of the order of $n \log(n)$. Both X and Y axis are in a log base 2 scale. NumPy **sort** is over the manually chosen limit of $c_1 = 3 * 10 - 9$ while heap sort is just below $c_2 = 1500 * 10 - 9$.

To sum it up, the conclusion that can be drawn from figure 3 is that all of our algorithms show a asymptotic behavior of $\Theta(n \log(n))$, since they lie between $c_1 \log(n)$ and $c_2 \log(n)$, where $c_1 = 3 * 10 - 9$ and $c_2 = 1500 * 10 - 9$.

We also compared the algorithms for sorted and reverse sorted data. Figure 4 shows the asymptotic nature of the algorithms for sorted data while Figure 5 shows it for reverse

Table 2: run-times for sorts for size 10485760

Sort Type	List type	List length	run-time (sec)
NumPy sort	random	10485760	1.784178
Python sorted	random	10485760	12.224272
Quicksort	random	10485760	71.071689
Mergesort	random	10485760	118.565381
Heapsort	random	10485760	203.664645

sorted data.
For reverse sorted and sorted data the Python sorted function is faster than NumPy's sort function. Like the randomized data, the run-times are asymptotically bounded between $c_1 n \log(n)$ and $c_2 n \log(n)$ for the chosen values of c_1 and c_2.

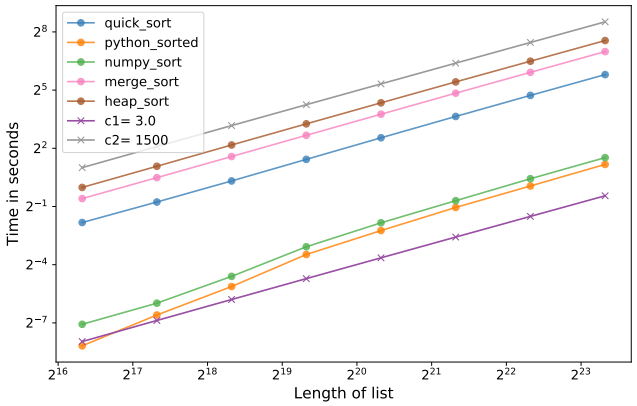

Figure 4: Sorted data of size 81920-10485760

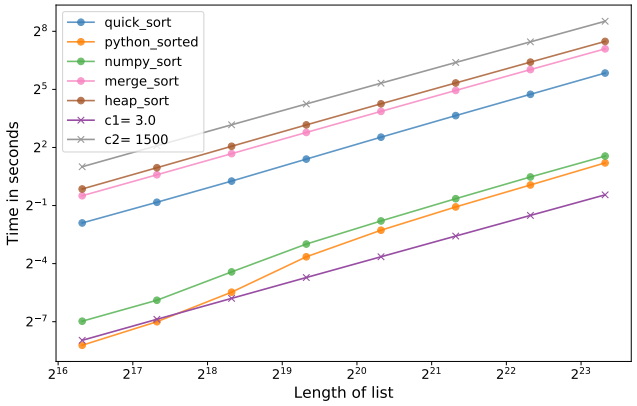

Figure 5: Reverse sorted data of size 81920-10485760

4.3 All permutations for a particular sort

Next, All permutations have been plotted and c_1 and c_2 are shown in these graphs. The x axis scale is still logarithm of the original list size, while the y axis is in seconds.

4.3.1 NumPy sort. It shows that it's the fastest for sorted data followed by random and reverse sorted data. The difference between sorted and reverse sorted data is significant as shown in the Figure 6.

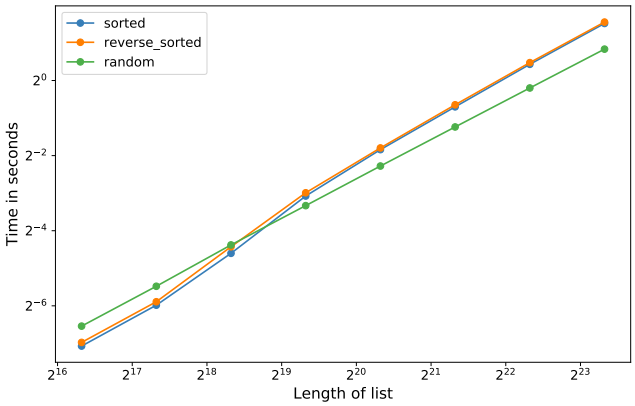

Figure 6: NumPy sort for data of size 81920-10485760

4.3.2 Python sorted. Contrary to NumPy sort, Python sorted is fastest for reverse sorted data, which is interesting as shown in Figure 7. Also, If we look closely in Figure 5 , we can see that Python sorted was the ultimate winner among other sorts for reverse sorted data, which is a notable observation.

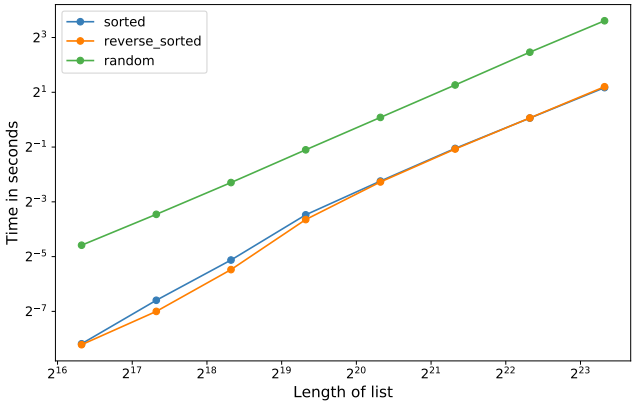

Figure 7: Python sorted for data of size 81920-10485760

4.3.3 Quicksort. Random data takes more time than the sorted and reverse sorted variant in case of quicksort, Figure 8

4.3.4 Heapsort. Sorted and reverse sorted data are doing better than random data as shown in Figure 9.

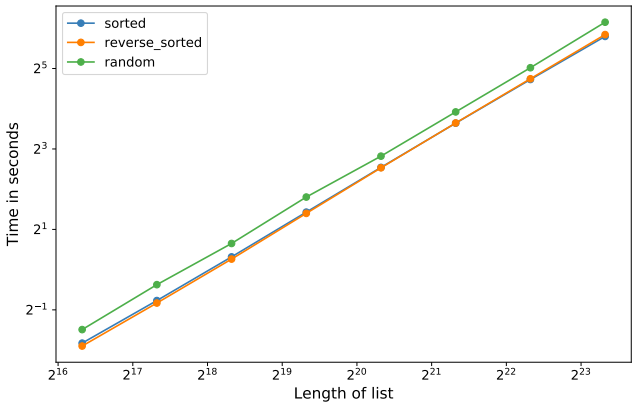

Figure 8: quicksort for data of size 81920-10485760

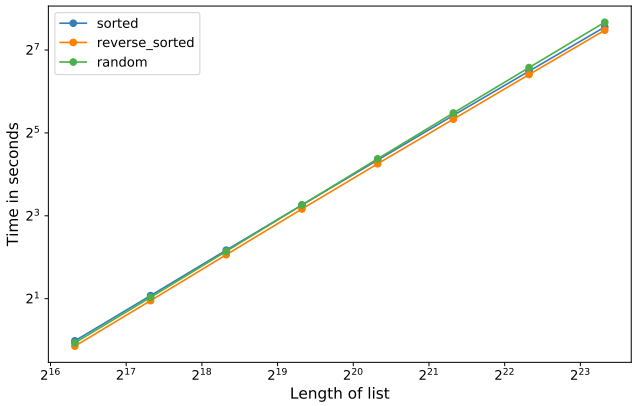

Figure 9: heapsort for data of size 81920-10485760

4.3.5 Mergesort. In case of mergesort, the times for all 3 permutations of data are giving mixed results. So the permutation of input data does not matter for mergesort.

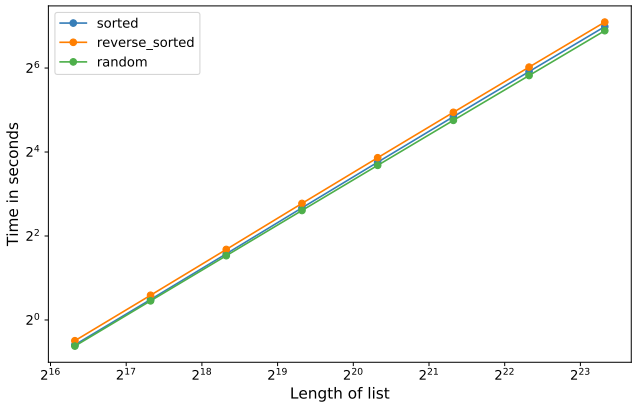

Figure 10: mergesort for data of size 81920-10485760

4.4 Comparison between the fastest and slowest sort

Here we compare the increase in times between the fastest sort and the slowest and prove that they have indeed a run-time increase of $\Theta\left(n \log\left(n\right)\right)$. Heapsort increases more rapidly on random case data. In table 3, we can see that, the data-size in our experiment is increased by two folds each time. In such a scenario, if the run-time was quadratic n^2, the run-time should have increased by 4. If it was of the order $n\sqrt{n}$ it should have increased by a factor or 2.828. And, if it is $n \log\left(n\right)$, the factor of should be between 2 and 2.828 and decrease towards 2 as the data size increases. The average factor for heapsort is 2.154, and the factor is also decreasing as well. So the time must be increasing in $n \log\left(n\right)$ asymptotically.

Table 3: heapsort for random permutation

Listlength	Singlerun-time	factor
40960	0.443787	-
81920	0.955577	2.153
163840	2.044499	2.139
327680	4.416153	2.160
655360	9.58716	2.170
1310720	20.747912	2.164
2621440	44.638867	2.151
5242880	95.394334	2.137
10485760	203.664646	2.134
	Average	2.1514

The average factor for numpy is 2.08, and the factor is decreasing as well. So the time must be increasing in $n \log\left(n\right)$ asymptotically.

Table 4: NumPy sort for random permutation

Listlength	Single run-time	factor
40960	0.005008	-
81920	0.010742	2.14
163840	0.022419	2.08
327680	0.048081	2.14
655360	0.099308	2.06
1310720	0.206241	2.07
2621440	0.424289	2.05
5242880	0.871039	2.05
10485760	1.784178	2.04
	Average	2.08

5 DISCUSSION

We have put out the following discussion points from our results and observations.

- Among the pure sort algorithms, quicksort was the quickest followed by mergesort and heapsort.

- NumPy sort was fastest followed by Python's sorted in case of random data, while Python's sorted narrowly finished first in case of sorted and reverse sorted data
- Run-times for all the sorts do increase by $\Theta\left(n \log\left(n\right)\right)$ asymptotically. However, the default sort algorithms have quite small constants compared to the pure algorithms. The ratio being $\frac{1500}{3} = 500$ between heapsort and NumPy sort.
- Mergesort in our case is insensitive to the initial ordering of the list. This might be due to the MERGE algorithm we used, which compares every element even though if the remainder of the list is already sorted.
- In our observation, Heapsort, Quicksort and Python sorted sort the random list slower than the other lists. The difference being bigger in case of Quicksort and Python sorted. Numpy sort and Mergesort on the other hand sorted the random list faster than the ordered lists.
- All the algorithms have runtimes follows the order $\Theta\left(n \log\left(n\right)\right)$ asymptotically.

6 ACKNOWLEDGEMENTS

We would like to thank Professor H.E. Plesser for his guidance, and for his valuable and detailed feedback on the initial draft. Also would like to thank our TA Krista Gilman for her guidance and for answering our questions related to the paper.

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.

Tim Peters. 2019. cpython/Objects/listsort.txt. https://github.com/python/cpython/blob/master/Objects/listsort.txt

Thøger Rivera-Thorsen. 2013. A color blind/friendly color cycle for Matplotlib line plots. https://gist.github.com/thriveth/8560036

The SciPycommunity. 2019. NumpySortDocumentation. https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html