# 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 generate the run-times for three pure sorting algorithms that use the divide-and-conquer approach to sort data. In addition to the three, the inbuilt numpy sort and the python sorted 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. Unlike our expectations, even for lists larger than 1 million the run-times do not strictly follow the order of $n \log n$.

## 1 INTRODUCTION

Sorting and searching are very basic and routine operations for computers of any type. Much research has already been done in the subject, and computers/soft-wares today use the most robust algorithms that they can lay their hands on in order to meet their particular requirements. Many applications use a blend of 2 or more algorithms depending on the nature input data or the application. However, it would be interesting to see how the sorting algorithms behave in real-machine situation!
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 stand-alone sort is more efficient? Do these standalone sorts have a chance against the built-in sorts of numpy and python? How are the run-times of each run 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. Acknowledgements and References conclude the paper.

## 2 THEORY

We briefly discuss the sorts we are bench-marking here. By following the given pseudo-codes, we wrote python implementations for heap, merge and quick sort. 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 Heap Sort

**Listing 1** Pseudo code for Heap sort algorithm from Cormen et al. [2009, Ch. 6.4].

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

Heap sort 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. Heap sort 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.

Also heap sort does the sorting in place, so no extra memory is required. Theoretically, all 3 cases of heap sort (worst, best and average) are of the order $n \log n$. It will be interesting to see which of the three sets of data ( random, ordered, reversely ordered) data performs the best on heap sort.

### 2.2 Merge Sort

**Listing 2** Pseudo code for Merge sort algorithm used

```
def 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
```

The function MERGE-SORT 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 2 lists with 2 elements each, both of which are already sorted and so on. Merge sort probably got its name from this sub-process of the algorithm.

Merge sort has the run-time of the order of $n \log n$ 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 merge sort. It cannot sort the keys in place, but it is a stable sorting algorithm.

### 2.3 Quick Sort

**Listing 3** Quick sort Algorithm pseudo-code from Cormen et al. [2009, Ch. 2.3.1].

```
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)
```

In case of quick-sort, most of the work is done by the PARTITION function. It does the in-place swapping of the list elements. PARTITION function also returns the position of the pivot element to the QUICK-SORT function. Unlike merge sort, here the recursion happens after the PARTITION function (tail recursion), so we could change the recursive function to a loop as well. We have used the recursive approach for now.

Also, quick-sort has a average and best case run-time of the order of $n \log n$, 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 $n^2$. It will be interesting to see if we indeed run into this scenario!

### 2.4 Numpy's numpy.sort

The numpy sort function uses quick-sort by default and a combination of other sorts depending on the input data. Quick-sort, merge-sort, radix-sort, tim-sort and even heap sort 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 $n \log n$. More details can be found here SciPycommunity [2019] numpy.sort documentation.

### 2.5 Python's default sorted

Python uses a now popularly used sort algorithm in many programming languages, called tim-sort which is named after its inventor. It is a better version of merge sort, where the algorithm first analyzes the input data to isolate 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 $n \log n$. More details can be found here Peters [2019] tim-sort explained from implementer himself!

## 3 METHODS

We worked with data-sizes ranging from just ten elements to 10.5 million 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 bench-marking.

### 3.1 Python implementation of algorithms

Python implementations for heap sort and quick sort were written following the pseudo-code from Cormen et al. [2009]. Algorithm for merge sort was slightly different from Cormen et al. [2009]. All the python codes, data generated and figures can be found at Github

It was interesting that we could implement quick-sort in a few lines. Below in Listing 4 we have included the partition function for quick-sort. Style and colours were set according to Overleaf [2019].

**Listing 4** Quick sort Partition Function

```python
def quick_comparison_and_swap(input_list
                              , start, end):
    pivot_last = input_list[end]
    l_id = start - 1

    for c_id in range(start, end):
        if input_list[c_id] <= pivot_last:
            l_id += 1
            input_list[l_id], input_list[c_id] =\
                input_list[c_id], input_list[l_id]

    input_list[end], input_list[l_id + 1] =\
        input_list[l_id + 1], input_list[end]

    return l_id + 1
```

Also, the merge function for merge sort can be found at Listing 5

### 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 7 times. We use the fastest time among the 7 readings in most of our analysis. We use the same seed at all times, to ensure the data is exactly same. Timing function is in Listing 6.

Then we defined a helper function displayed in Listing 7 in order to call timing function with the range of sizes from 10 to approximately 10.5 million. The time information is stored in a pandas dataframe and exported individual .csv files.

### 3.3 Analysis, tables and plots

We imported the .csv files exported from the timing step and combined them to a single dataframe. Then we compared the

**Listing 5** Merge sort's Merge function

```python
def sort_two_sorted_lists(A,B):
    len_total= len(A)+len(B)
    C= [None]*len_total
    index_total= len_total-1
    a= A.pop()
    b= B.pop()

    while True:
        if a >= b:
            C[index_total]= a
            if len(A) >0 :
                a= A.pop()
            else:
                a= -math.inf
            index_total -= 1
        else:
            C[index_total]= b
            if len(B) >0 :
                b= B.pop()
            else:
                b= - math.inf
            index_total -= 1

        if a == - math.inf and\
                b == - math.inf:
            break
    return C
```

**Listing 6** Time it function used with parameters

```python
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 ()
    t = clock.repeat ( repeat =7, number=n_ar)
    return t
```

run-times of the 5 algorithms for all data sizes in a number of line graphs. Data-sizes ranging from 80 thousand and above were plotted as we're interested in the behavior of algorithms for large data size.

Additionally, we plotted box-plots of the 7 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

**Listing 7** Call timeit function and save data to .csv

```python
import pandas as pd
import numpy as np

def get_time_and_write_to_dataframe(Algorithm):
    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 =   timing_function(i, Algorithm )
        for each_time in time:
            time_data_points.append(
            {'Sort_Type':Algorithm.__name__ ,
            'Data_Type_or_List_type':'random'
           ,'List_length':i
           , 'Runtimes': each_time
           ,  'Number_of_repeatitions':'7'
           ,'Datetime':pd.Timestamp.now() } )
    return time_data_points

# Call the get_time... funtion

columns = ['Sort_Type','Data_Type_or_List_type'
            ,'List_length','Runtimes'
            ,'Number_of_repeatitions','Datetime']

df_ = pd.DataFrame( columns=columns)

# Recording times for numpy sort for instance

list_of_dict = get_time_and_write_to_dataframe(np.sort
    )
df_= df_.append(list_of_dict)
df_.to_csv (r'export_dataframe.csv', index = None
, header=True)
```

were plotted in **nanoseconds**, so that the constants for the run-time, for instance the $c1$ in $c1 * n * logn$ remains within 2 decimal points.

Python implementation for one set of plots is given in Listing 8 and Listing 9 , while others have been included in the GitHub hashes section 3.6. Also, the colour for the line graphs was taken from Rivera-Thorsen [2013].

## 3.4 Code used for curve fitting

Another way prove that the data-points are following $n \log n$ behavior, is to fit a curve on the set of points. We've tried this on the mean readings for quick-sort's random permutation of data. Code is given below in 10 and the figure 14 is in section 4.5.

## 3.5 Hard-wares and soft-wares used

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

- Processor: AMD A4-3330MX APU 2.30 GHz
- Caches: L1D-64 KB*2 , L1I-64KB*2, L2 -512KB*2
- Memory : DDR3 4 GB, 800 MHz
- OS : Microsoft Windows 7 Professional, 64 bit

**Listing 8** Example code to generate plots

```python

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import pandasql as ps
import matplotlib as mpl

plt.rcParams['axes.titlesize'] = 15
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['xtick.labelsize'] = 10
plt.rcParams['ytick.labelsize'] = 10
plt.rcParams['legend.fontsize'] = 10
plt.rcParams['text.usetex'] = False
# Color cycle for color blind Source:
# https://gist.github.com/thriveth/8560036
CB_color_cycle = ['#377eb8', '#ff7f00', '#4daf4a',
                  '#f781bf', '#a65628', '#984ea3',
                  '#999999', '#e41a1c', '#dede00']

# Set the default color cycle
mpl.rcParams['axes.prop_cycle'] = mpl.cycler(
                            color=CB_color_cycle)

# Function for a consistent length of figures of 84 mm
#, converting to inches
def new_figure(height=55):
    "Return figure with width 84mm and given height in mm."

    return plt.figure(figsize=(84/10.16, height/10.16))

# Read Dataset

dataset = pd.read_csv("dataset_concat.csv")
# Extract the minimum from 7 observations

min_query = """SELECT Sort_Type,
Data_Type_or_List_type,
List_length,
min( Runtimes/Number_of_repeatitions) as Single_runtime
FROM dataset
GROUP BY Sort_Type,
Data_Type_or_List_type,
List_length """

df_min = pd.DataFrame( ps.sqldf(min_query) )
```

Following are the versions of packages used in Python 3.7.4:

- Pandas: '0.25.1'
- Pandasql: '0.7.3'
- Numpy : '1.16.5'
- matplotlib : '3.1.1'
- nbimporter : '0.3.1'
- ipywidgets: '7.5.1'
- timeit :
- pickle :

## 3.6 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.

**Listing 9** Example code to generate plots continued

```python
def plot_minimum_times(input_type='sorted'
, lower_limit=81920, upper_limit=10485760
, c1=0.2, c2=2):
    filter01 =  (df_min['Data_Type_or_List_type']
    ==input_type)
    plot_data = df_min[filter01]

    filter02 = (plot_data['List_length']<=
                        upper_limit)
    &  (plot_data['List_length']>=lower_limit)
    plot_data = plot_data[filter02]

    fig = new_figure()
    list_of_sorts= sorted(np.unique(
    plot_data['Sort_Type']).tolist(),reverse=True)
    for sort_type in list_of_sorts:
        filter03 = (plot_data['Sort_Type']==sort_type)
        plot_this = plot_data[filter03]
        plt.plot( np.log2( plot_this['List_length'])
        , plot_this['Single_runtime']*1000000000,'-o'
        ,alpha=0.7, label=sort_type) #nanoseconds

    n_log_n_small= c1*plot_this['List_length'] *
    np.log2( plot_this['List_length'] )
    n_log_n_large= c2*plot_this['List_length'] *
    np.log2( plot_this['List_length'] )
    plt.plot ( np.log2( plot_this['List_length'])
    , n_log_n_small,'-x', label="c1= "+str(c1) )
    plt.plot ( np.log2( plot_this['List_length'] )
    , n_log_n_large,'-x', label='c2= '+str(c2))
    plt.xlabel('Log 2 of the size of the numeric
                            array')
    plt.ylabel('Time in nanoseconds')
    plt.title("Runtimes of sort Algorithms for
    -"+input_type+" data")
    plt.legend()
    plt.tight_layout()
# plt.savefig("log_plots\For -"+input_type+" from-"
# +str(lower_limit)+" to-"+str(upper_limit)+".pdf"
# , bbox_inches='tight')
    plt.show()

## Interact to get most relevant plots
from ipywidgets import interact
pickle_object= interact(plot_minimum_times,
          input_type=['random', 'reverse_sorted'
          , 'sorted']
    ,lower_limit=  np.unique(df_min['List_length']
         ).tolist()
     ,upper_limit=  np.unique(df_min['List_length']
         ).tolist()
        , c1=(0,3,0.1), c2=(200,601,50) )
```

## 4 RESULTS

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

### 4.1 Box plot of run-times for random data

To begin, we plotted the box plot of all the 7 time-points to check if we observe any anomalies or outliers. In the analysis that follows, we use only the minimum time among the 7.

In figure 1 and figure 2 the run-times had a larger variation in numpy **sort** and merge sort respectively. The variation in

**Listing 10** Code for curve fitting quick sort data points

```
def time(xdata, c1 ):
    return  c1*xdata *np.log2( xdata )

# Guess the constant of 145
def guess(c):
    return [
    time(i,c) for i in  plot_this['List_length']
    ]
c_guess = 145

fig=new_figure()
plt.plot(plot_this['List_length']
, plot_this['Runtimes'], '+' )
plt.plot(plot_this['List_length']
, guess(c_guess), label='Guess- c=145' )

from scipy.optimize import curve_fit

t = plot_this['Runtimes']
s = plot_this['List_length']
g = np.asarray( guess(c_guess) )
c, cov = curve_fit(time, xdata= s, ydata=t
, p0= 145 )
c =  float("{0:.2f}".format(float(c)))

plt.plot(plot_this['List_length'], guess(c)
, label='Curve fit-c='+str(c) )

plt.xlabel('Size of the list from 10240-10485760')
plt.ylabel('Time in Nanoseconds')
plt.title("Curve fitting for mean run-time of quick-
    sort")
plt.legend()
plt.tight_layout()
plt.show()
```

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

| File | Git hash |
|---|---|
| `plots_and_data/dataset_concat.csv` | a9ddf5d |
| `plots_and_data/Line_plots_interactive.ipynb` | 0e8e551 |
| `plots_and_data/Norm_for_numpy_python.ipynb` | 0e8e551 |
| `plots_and_data/Box_plots.ipynb` | 0e8e551 |
| `Time_it.../time_it_function.ipynb` | 0e8e551 |
| `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` | 9856f1c |

time, we believe is due to other applications or background processes running on the machine.

## 4.2 All sorts for a particular permutation

Figures 3-4 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 3 and from 81920 until 10.5 million in Figure 4. 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 $\mathbf{n_0 = 81920}$.

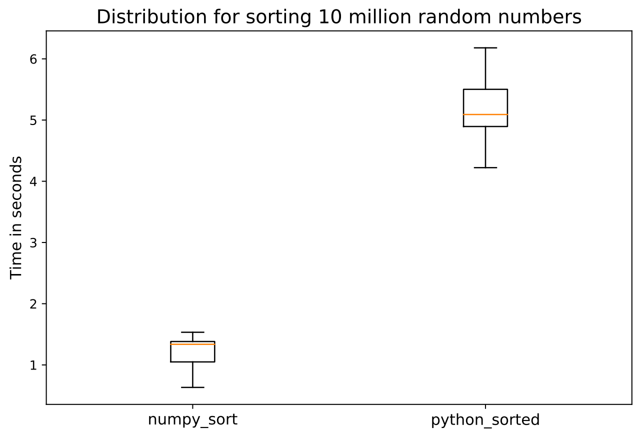

**Figure 1: run-times for data size: 10485760**

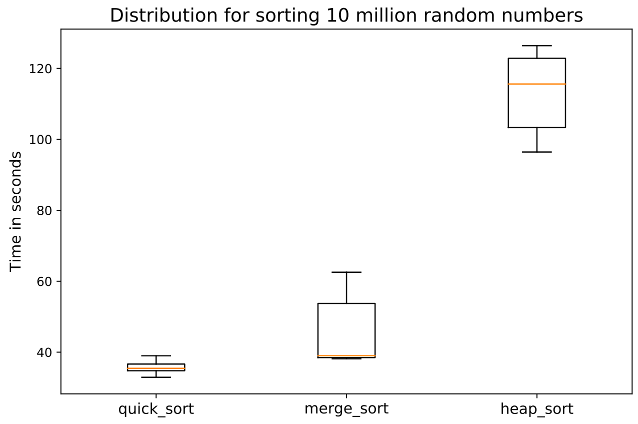

**Figure 2: run-times for data size: 10485760**

We can see that numpy **sort** is the most efficient for large list size. Python's **sorted** function follows numpy **sort** closely, until numpy finally clinches victory around the 1 million mark.

In figure 3, it is worth noting that no clear winner can be judged from the graph in the beginning. The run-times do not show any linear behaviour either. In many cases larger data sizes are being sorted faster. This may also be due to other background processes running on our computer. Plot starts from around 3.5 which is Log base 2 of 10-our smallest list size. At each interval the list-size (data-size) is doubling. Only after the list-size reaches 20,000 i.e. 14.5 in the log base2 measurement, we see a distinct separation of run-times.

We also have plotted a lower boundary, just below numpy sort and a upper boundary just above heap sort in order to prove that our algorithms do run asymptotically in a run-time of the order of $n \log n$. Note that the x axis of the Figures are actually a $logbase2$ of the size of the input numeric array,while the y axis is in order of **nanoseconds**. We Due to the nanosecond scale in y axis, we can choose intuitive

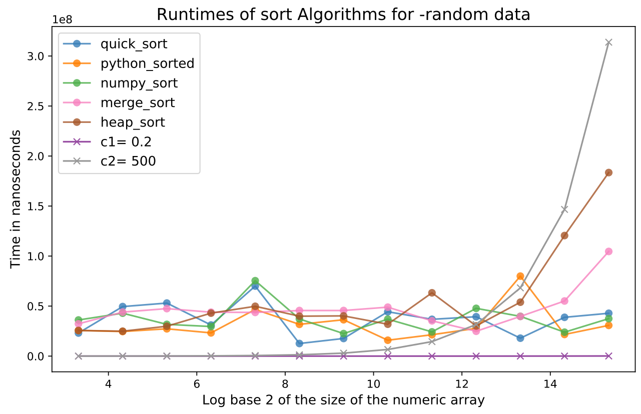

**Figure 3: Randomized data of size 10-40960**

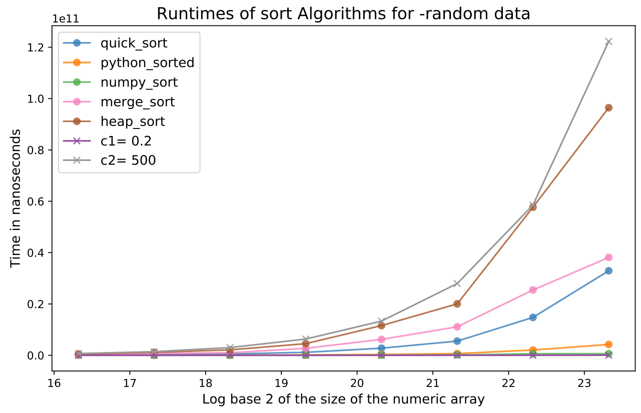

**Figure 4: Randomized data of size 81920-10485760**

values for the constants $c1$ and $c2$ in our interactive plots. Numpy **sort** is over the manually chosen limit of $c1 = 0.2$ when time is measured in nanoseconds. If the y axis (time) had been measured in microseconds the optimal constant would be $c1 = 0.00002$. It would get worse if we measure in seconds. Putting the time in nanoseconds, in a way makes the scales of x-axis and y-axis similar!

**Table 2: run-times for sorts for size 10485760**

| Sort Type | List type | List length | run-time (sec) |
|---|---|---|---|
| numpy sort | random | 10485760 | 0.632434 |
| python sorted | random | 10485760 | 4.222582 |
| quick sort | random | 10485760 | 32.919218 |
| merge sort | random | 10485760 | 38.122927 |
| heap sort | random | 10485760 | 96.435979 |

To sum it up, the conclusion that can be drawn from figure 4 is that all of our algorithms show a asymptotic behavior of $n * log * n$, since they lie between $c1 * log * n$ and $c2 * log * n$, where c1=0.2 and c2=500.

Additionally, we can note that, In the graph, numpy **sort** and python **sorted** look awfully close to the lower limit . To take a closer look, we plot the run-times separately for these 2 in figure 5. We also have plotted the lower bound of $c1$ which is just below numpy. Also, Heap sort is the slowest asymptotically, quick sort being the quickest of the 3 stand-alone sorts.

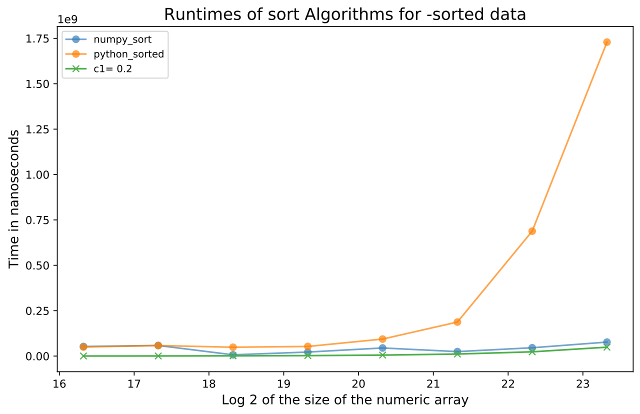

**Figure 5: Numpy vs Python with lower bound; array 81920-10485760**

We also compared the algorithms for sorted and reverse sorted data. We could not get the run-time for quick sort for reverse sorted and already sorted data, as it hits the maximum recursion error after the data size 2560. Figure 6 shows the asymptotic nature of the algorithms for sorted data while Figure 7 shows it for reverse sorted data.
Interestingly, for reverse sorted data the python sorted function is faster than numpy's sort function. Like the randomized data, the run-times are asymptotically bounded between $c1 * n * logn$ and $c2 * n * logn$ for the chosen values of $c1$ and $c2$.

Also, notable is the fact that the line for sorts are farther away from constant $c2$ when compared to the plot for random data. This indicates that the sorting algorithms are in general faster for already sorted and reverse-sorted list. Python's sorted function is an exception here, which we will discuss in detail later.

## 4.3 All permutations for a particular sort

Next, we wanted to plot each sort in one graph with the 3 permutations: randomized, sorted and reverse sorted. We have chosen not to show $c1$ and $c2$ in these graphs. The x axis scale is still log base 2 of the original list size, while the y axis is in microseconds.

*4.3.1 Numpy sort.* Numpy sort is 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 8.

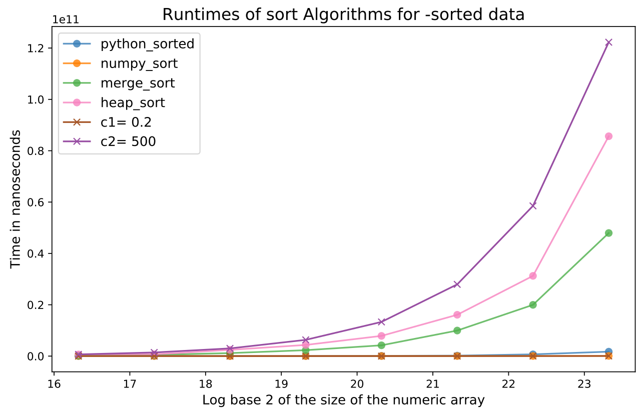

**Figure 6: Sorted data of size 81920-10485760**

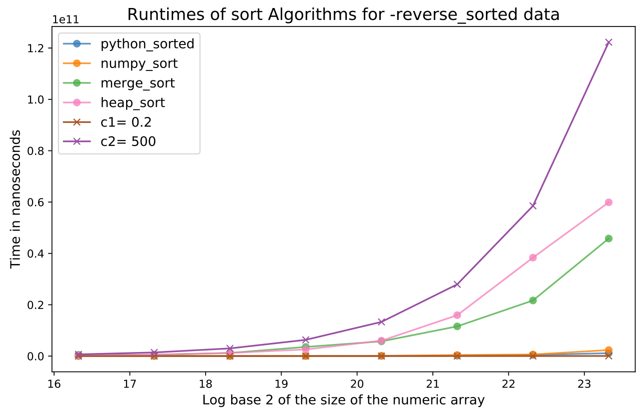

**Figure 7: Reverse sorted data of size 81920-10485760**

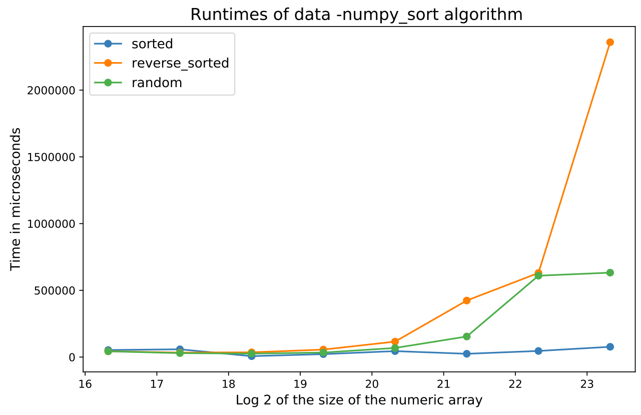

**Figure 8: Numpy sort for data of size 81920-10485760**

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

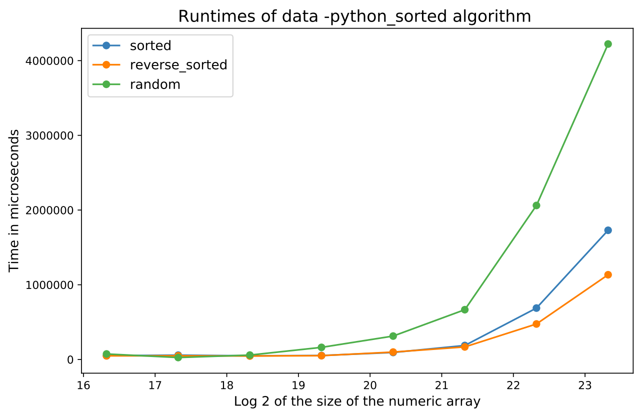

**Figure 9: Python sorted for data of size 81920-10485760**

*4.3.3 Quick sort.* We don't have run-times for reverse sorted and sorted data for quick-sort yet, as we hit the maximum recursion limit, when we chose the last element as pivot. The plots will be added in the final version of the term-paper. Figure 10

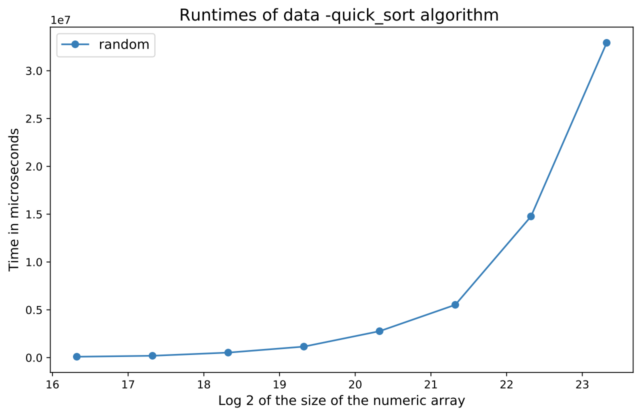

**Figure 10: Quick sort for data of size 81920-10485760**

*4.3.4 Heap sort.* In case of heap sort, the sorted and reverse sorted data are doing better than random data as shown in Figure 11.

*4.3.5 Merge sort.* In case of merge sort, the times for all 3 permutations of data are giving mixed results. So the permutation of input data does not matter for merge sort.

## 4.4 Comparison between the fastest and slowest sort

Here we try to compute the difference between the fastest sort and the slowest and give them appropriate equations. Heap

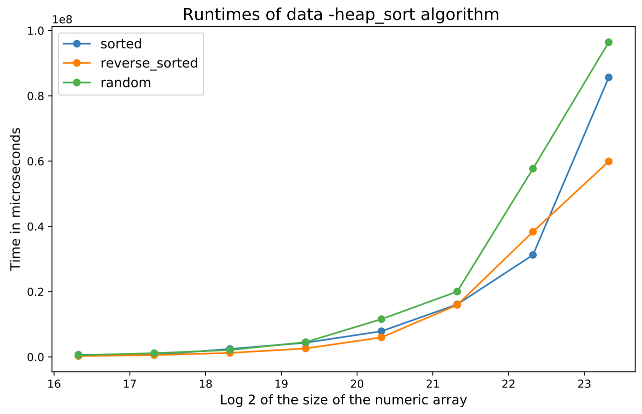

**Figure 11: Heap sort for data of size 81920-10485760**

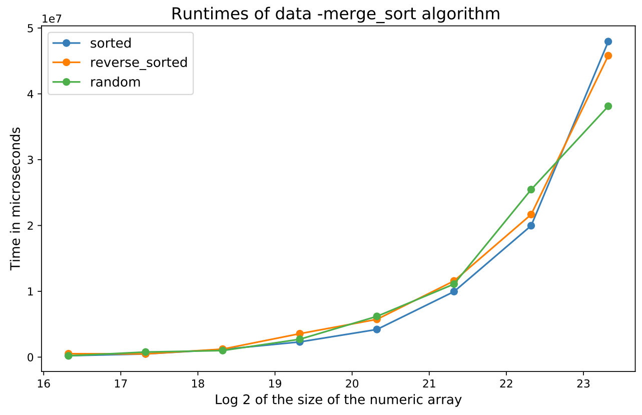

**Figure 12: Merge sort for data of size 81920-10485760**

sort will takes the most time and also increases more rapidly on **average**, as size of data increases. In table 3, we can see that, the data-size in our experiment is increased by 2 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 $n \log n$, the factor of should be between 2 and 2.828 and decrease towards 2 as the data size increases. The average factor for Heap sort is 2.23 from 3, so the function must be increasing in $n \log n$ asymptotically.

Surprisingly, the average factor for numpy sort is 1.66 when calculated from the data points in table 4, that is less than linear increase! Figure 13 confirms our doubts. Numpy sort does it in less than linear time for the data we tested. One possible explanation could be that the larger size data generated by our random number generator produced a easily sortable list than the preceding list of half its size.

### 4.5 Curve fit in quick sort data points

The curve does indeed fit on a $n \log n$ curve with constant c=143.64, which is another way of proving that the algorithms do follow a run-time behavior of $n \log n$. We picked quick sort data points for this, as it had the most consistent behavior.

**Table 3: Heap sort for random permutation**

| Listlength | Singlerun-time | factor |
|---|---|---|
| 40960 | 0.18348 | – |
| 81920 | 0.525849 | 2.86 |
| 163840 | 1.154753 | 2.19 |
| 327680 | 2.116844 | 1.83 |
| 655360 | 4.535283 | 2.14 |
| 1310720 | 11.578997 | 2.55 |
| 2621440 | 20.032117 | 1.73 |
| 5242880 | 57.685499 | 2.87 |
| 10485760 | 96.435979 | 1.67 |

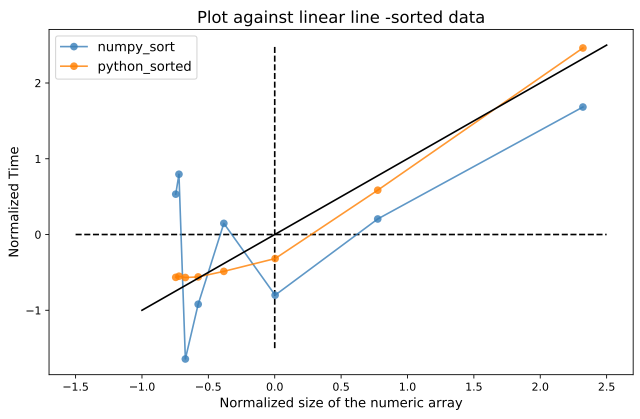

**Figure 13: Numpy sort, normalized and plotted against line of slope 1**

**Table 4: Numpy sort for random permutation**

| Listlength | Single run-time | factor |
|---|---|---|
| 40960 | 0.037461 | – |
| 81920 | 0.043112 | 1.15 |
| 163840 | 0.029619 | 0.68 |
| 327680 | 0.026161 | 0.88 |
| 655360 | 0.033746 | 1.28 |
| 1310720 | 0.068165 | 2.01 |
| 2621440 | 0.154086 | 2.26 |
| 5242880 | 0.609677 | 3.95 |
| 10485760 | 0.632434 | 1.03 |

## 5 DISCUSSION

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

- Among the stand-alone sorts, quick sort was the quickest followed by merge sort and heap sort.

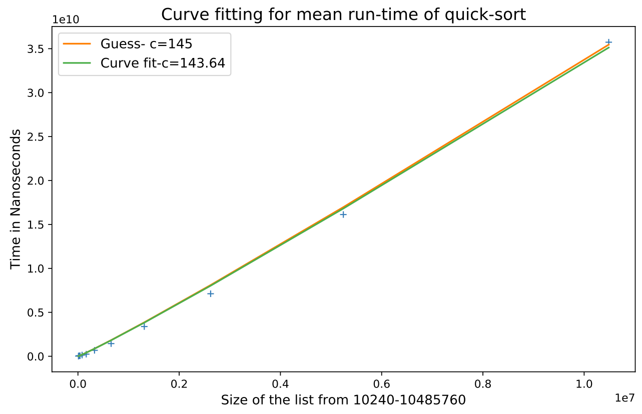

**Figure 14: Curve follows n*log(n) with c= 143.64 for quick sort**

- Numpy sort was fastest followed by python's sorted in case of random and sorted data, while python's sorted narrowly finished first in case of reverse sorted data
- Run-times for all the sorts do increase by a factor of $n \log n$ asymptotically. However, the default sort algorithms have quite small constants compared to the standalone algorithms. The ratio being $\frac{500}{0.2} = 2500$ between heap sort and numpy sort.
- Merge sort is insensitive to the initial permutation of data, when compare to other sorts.
- We found considerable variance in the experimental run-times we collected ( for the same sort and same data size), possibly due to other applications running in the computer.
- The sorts might not follow a clear $n \log n$ order of increase in its run-time for every increase in the list size, because the type and order of elements in the list influences the final run-time. In our case we plotted a case where numpy sort showed a less than linear behavior on average!

## 6 ACKNOWLEDGEMENTS

We would like to thank our Professor H.E Plesser and TA Krista Gilman for their 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.

Overleaf. 2019. Code listing - Overleaf, Online LaTeX Editor. https://www.overleaf.com/learn/latex/Code_listing

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