Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2020

Mohamed Radwan
mohamed.radwan@nmbu.no

Nasibeh Mohammadi
nasibeh.mohammadi@nmbu.no

Abstract

The objective of this paper is to show the results of performing benchmark experiments on sorting algorithms. We implemented several sorting algorithms and compared them in terms of their run-time. The algorithms were investigated into four categories: quadratic (Insertion-Sort and Bubble-Sort), sub-quadratic (Merge-Sort and Quick-Sort), combined Merge-Insertion-Sort and built-in (Python-Sort and Numpy-Sort) algorithms. The input data were lists of float values with sizes from 2^4 to 2^{14} in the form of 3 different ordering permutations: sorted, reversed and random ordering. The study was conducted using Python programming language and several libraries like Numpy, Scipy, Pandas and Matplotlib. Quick-Sort showed very similar run-time in sorting reversed and sorted data. Merge-Sort also showed the same observation. For finding an explanation for these behaviours, we investigated these observations further by measuring the number of comparisons that Merge-Sort and Quick-Sort executed in sorting the data. The study also shows that Numpy-Sort and Python-Sort are faster than all other implemented algorithms. Bubble-Sort is the slowest sorting algorithm among all these algorithms. Additionally, we implemented a combined Merge-Insertion-Sort which performed slightly faster than Merge-Sort. A challenging question is left open that is why Python-Sort is 10 times faster than our implementation of Merge-Insertion-Sort.

1 Introduction

Sorting algorithms play an essential role in implementing and optimizing efficient algorithms that solve our problems. In this study, we aimed to examine whether the sorting algorithms follow the theoretical run-time. We wanted to compare the theory with real performance of the algorithms and explain the results. Theoretical details about each type of sorting algorithms are explained in the "Theory" section where we give description on the theoretical expectations of the algorithms' execution times. In third section, "Methods", we explain the methods utilised in this research for executing the benchmarks and the hardware and software specifications. In the "Result" section, the numerical results and observations are explained in details supported by figures. In the "Discussion" section, we use our findings and combine our observed results with the theoretical expectations to understand the behaviours of these sorting algorithms.

2 Theory

2.1 Quadratic Algorithms

Insertion-Sort and Bubble-Sort belong to this family of sorting algorithms where the run-time follows a quadratic big theta time complexity $\Theta(n^2)$.

2.1.1 Insertion-Sort

According to Cormen et al. [2009], "Insertion-Sort is an efficient algorithm for sorting a small number of elements". Insertion-Sort does the sorting of the input data in-place. This algorithm, as shown in the pseudo-code in listing 1, starts by taking one non-sorted element at a time and compares it with already sorted elements and inserts it in correct place among sorted elements. Comparison is done in a way that if the element is smaller than the preceding element, they will be swapped. These swaps are done following inner condition, as shown in listing 1, which is repeated until j reaches -1 (each time the while loop is performed, j is decremented by 1). Those swaps are executed until all elements before and including the element at index i are in sorted order. This process is repeated until all the elements are in sorted order.
This algorithm at worst and average cases will go through the two loops, so the worst and average case scenario of Insertion-Sort follow the quadratic form as shown in equation 1.

$$T(n) = \Theta(n^2) \tag{1}$$

Insertion-Sort, in the best case scenario, has a linear run-time of $\Theta(n)$ where the data are already sorted and the condition of while-loop in line 4 in the algorithm is not satisfied.

Listing 1 Insertion-Sort algorithm pseudo-code (modified after from Cormen et al. [2009, Ch. 2.1]).

```
INSERTION-SORT($A$)
1  for $i = 2$ to $A.length$
2      $Key = A[i]$
3      $j = i - 1$
4      while $j > 0$ and $key < A[j]$
5          $A[j+1] = A[j]$
6          $j = j - 1$
7      $A[j+1] = key$
```

2.1.2 Bubble-Sort

According to Cormen et al. [2009], Bubble-Sort is in-place sorting algorithm. The algorithm, as shown in listing 2, works from the leftmost element and goes to the rightmost and swaps the pairs which are not sorted. It swaps every pairs of elements if the condition in line 3 of the algorithm in the listing 2 is satisfied. In the first run, the algorithm iterates through all elements and pushes the greatest element to the rightmost place. In the second run, the algorithm iterates over all elements except the last element and swaps pairs of elements. These iterations are repeated until i reaches the end of the array.
The algorithm contains one nested loop thus the run-time is as shown in the equation 2. The study made by Astrachan [2003]

shows that Bubble-Sort, in general, is 3 times slower than Insertion-Sort.

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

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

```
BUBBLE-SORT($A$)
for $i = 1 \rightarrow A.length$
    for $j = 1 \rightarrow A.length - 1$
        if $A[j] > A[j+1]$
            exchange $A[j]$ with $A[j+1]$
```

2.2 Sub-Quadratic Algorithms

This family of algorithms contains Quick-Sort and Merge-Sort. Those algorithms follow a logarithmic run-time $\Theta(n \cdot \lg n)$. These algorithms use the divide-and-conquer approach where the problem is broken into sub-problems that are solved recursively and the solutions are combined to form a solution to the original problem.

2.2.1 Quick-Sort

According to Cormen et al. [2009], Quick-Sort algorithm is in-place algorithm that follows divide and conquer approach where it splits the data into two partitions based on the pivot. Elements in the left partition are smaller than the pivot and elements in the right partition are larger than the pivot. Recursive calls will be done on left and right sub-arrays till all arrays are sorted. Pseudo-code of the Quick-Sort algorithm is shown in listing 3. Assuming that partitioning is done in the median of the data, time complexity for the algorithm is as shown in equation 3.

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

The equation 3 gives the best case run-time of the algorithm where the pivot is the median. If the pivot is at the beginning of the data, that would result into uneven split and that is a big challenge in terms of run-time. In case of an already sorted data, all non-pivot elements will end up in right sub-array which gives run-time as in the equation 4.

$$T(n) = \Theta(n^2) \ . \tag{4}$$

Both Quick-Sort and Merge-Sort follow the logarithmic run-time average case. However, Skiena [2008] stated that properly implemented Quick-Sort is typically 2 − 3 times faster than Merge-Sort or Heap-Sort. This claim was supported by the study conducted by Rajput et al. [2012].

2.2.2 Merge-Sort

Merge-Sort divides n elements sequence, where n is greater than 1, into two sub-sequences and sort the sub-sequences recursively. Then, it combines the two sorted sub-sequences using the merge function. Pseudo-code for the Merge-Sort algorithm is shown in listing 4. According to Cormen et al. [2009, Ch. 2.3], time complexity for all the cases (best, worst, average) is shown in the equation 5.

$$T(n) = \Theta(n \cdot \lg n) \ . \tag{5}$$

Listing 3 Quick-Sort algorithm pseudo-code (modified after from Cormen et al. [2009, Ch. 2.1]).

```
QUICK-SORT($A$)
if $A.length > 1$
    $p = A[\lfloor A.length/2 \rfloor]$
    for $i = 1$ to $A.length$
        if $A[i] < p$
            $l.insert(A[i])$
        elseif $A[i] = p$
            $e.insert(A[i])$
        elseif $A[i] > p$
            $r.insert(A[i])$
    $A = $ QUICK-SORT($l$) $+ e +$ QUICK-SORT($r$)
```

Listing 4 Merge-Sort algorithm pseudo-code (modified after Cormen et al. [2009, Ch. 2.3]). The parameter S in the merge function is the start index of the left array to be sorted.

```
MERGE-SORT($A$)
$r = A.Length$
$p = 1$
if $r > p$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A[p \dots q]$)
    MERGE-SORT($A[q + 1 \dots r]$)
    MERGE($A, A[p \dots q], A[q + 1 \dots r], 0$)
return A

MERGE($A, L, R, S$)
$i = 1$
$j = 1$
$k = S$
while $i \le L.Length$ and $j \le R.Length$
    if $L[i] < R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
    $k = k + 1$
while $i \le L.Length$
    $A[k] = L[i]$
    $i = i + 1$
    $k = k + 1$
while $j \le R.Length$
    $A[k] = R[j]$
    $j = j + 1$
    $k = k + 1$
```

2.3 Combined Sort

We combined the Merge-Sort with Binary-Insertion-Sort as shown in the listing 5. As stated above, Insertion-Sort follows $\Theta(n^2)$ run-time at worst and average cases and $\Theta(n)$ at the best case. Merge-sort follows $\Theta(n \cdot \lg n)$ run-time in all cases. According to Cormen et al. [2009], Merge-Sort is generally faster than Insertion-Sort. For

the mathematical proof, see Cormen et al. [2009, Ch. 2.3]. On the other hand, Insertion-Sort typically runs faster than Merge-Sort for small input sizes. According to Cormen et al. [2009], "there will always be a crossover point beyond which Merge-Sort is faster".
The proposed algorithm is a modified implementation of the Python Software Foundation [2020] implementation of Tim-Sort. The algorithm, shown in listing 5, consists of two steps. The first step is dividing the data into 'runs' and applying the Binary-Insertion-Sort (McIlroy [1993]) on each run. This step is used to sort each run in-place by providing the start and end of each run in the passed parameters to the Binary-Insertion-Sort. The second step uses the merge function that we built earlier in the listing 4 to merge the sorted runs.
Python Software Foundation [2020] used a function to optimally calculate the run length. The run length is used mainly to optimize the Merge-Sort balance. The function that is used to calculate the run sizes is as shown in listing 5. In this function, line 5 performs a bitwise right shift where it shifts the bits of the left operand l by the right operand 1. As the while-loop finishes, the binary value of number l will be the most significant bits of the size of the input data. Line 4 performs bitwise OR operator that only result in 1 if any 1 remaining bit are set (Python Software Foundation [2020]). The sum of l and r is a number between 32 and 64 inclusive and it is a power of 2 or slightly less than a power of 2 where Merge-Sort is most efficient (Wikipedia contributors [2020]).
Binary-Insertion-Sort, as shown in listing 6, is an extension of the Insertion-Sort where it uses the binary search. The binary search function is used to subdivide a sorted input into two halves and search for the target item in the half of the data where it belongs. Binary search is used here to get the position where to insert the element in the sub-array that was sorted by the earlier swaps of the Insertion-Sort. Binary search executes $\lg n$ number of comparisons to search through n number of elements which is an improvement over linear search that executes n number of comparisons. In terms of run-time, Auger et al. [2018] provided a proof that Tim-Sort run-time is in $O(n \cdot \lg n)$.

2.4 Built-in Algorithms

2.4.1 Python-Sort

Python-Sort uses a complicated implementation of Tim-Sort. However, the main building blocks for Python-Sort are the same as our implemented Binary-Merge-Insertion-Sort (see listings. 5 and 6). A full implementation of Tim-Sort can be accessed from the source code of the Python programming language by Python Software Foundation [2020]. According to Auger et al. [2018], Python-Sort is in $O(n \cdot \lg n)$ or even $O(n + n \cdot \lg \rho)$ where ρ is the number of runs.

2.4.2 Numpy-Sort

Numpy-Sort originally uses the algorithms (Quick-Sort, Merge-Sort, Heap-Sort). Quick-Sort is the default algorithm in the latest versions of Numpy as described in Harris et al. [2020]. Tim-Sort and Introspective-Sort were also included in Numpy-Sort. Introspective-Sort is a hybrid sorting algorithm that takes the advantages of both Quick-Sort and Heap-Sort. That is because Quick-Sort has a worst case scenario of $\Theta(n^2)$ and Heap-Sort has a worst case scenario of $\Theta(n \cdot \lg n)$. So, Introspective-Sort starts with Quick-Sort and

Listing 5 Merge-Insertion-Sort algorithm pseudo-code (merge function is modified after Cormen et al. [2009, Ch. 2.3] as shown in listing 4. The function calculate-run is modified following Python Software Foundation [2020]) implementation of Tim-Sort.

```
CALCULATE-RUN($n$)
1  $minrun = 32$
2  $r = 0$
3  $l = n$
4  while $l \geq minrun$
5      $r = r \mid (l \,\&\, 1)$
6      $l = l >> 1$
7  return $l + r$

MERGE-INSERTION-SORT($A$)
1   $run$ = CALCULATE-RUN($A.length$)
2   for $start = 1$ to $A.length$ by $run$
3       $end = \min(start + run - 1, n - 1)$
4       BINARY-INSERTION-SORT($A, start, end$)
5   $size = run$
6   while $size < n$
7       for $start = 1$ to $A.length$ by $run$
8           $middle = \min(n - 1, start + size - 1)$
9           $end = \min(n - 1, start + 2 \cdot size - 1)$
10          $left = A[start : middle + 1]$
11          $right = A[middle + 1 : end + 1]$
12          $A$ = MERGE($A, left, right, start$)
13      $size = size * 2$
14  return $A$
```

Listing 6 Binary-Insertion-Sort algorithm pseudo-code. Binary search pseudo-code is modified after Kuk et al. [2019].

```
BINARY-INSERTION-SORT($A, start, end$)
for $i = start + 1$ to $end + 1$
    $temp = A[i]$
    $pos$ = BINARY-SEARCH($A, temp, start, i$) $+ 1$
    for $k = i$ to $pos$ by $-1$
        $A[k] = A[k - 1]$
    $A[pos] = temp$

BINARY-SEARCH($A, key, start, end$)
if $end = start$
    if $key < A[start]$
        return $start - 1$
    else
        return $start$
$mid = \lfloor (start + end)/2 \rfloor$
if $A[mid] < key$
    return BINARY-SEARCH($A, key, mid, end$)
elseif $A[mid] > key$
    return BINARY-SEARCH($A, key, start, mid$)
else
    return $mid$
```

Table 1: Versions of relevant files used for this article on GitLab repository: https://gitlab.com/nmbu.no/emner/inf221/h2020/student-term-papers/team_45/inf221-term-paper-team-45.

File	Git hash
unitest.py	a5c6d64d
run_benchmarks.py	60352eea
algorithms.py	a4f5cab3
visualize.ipynb	5e82f684
nr_comparisons.ipynb	d096ff47
all_data.csv	3254ddae
clean_data.csv	3254ddae

Table 2: Machine Specifications

Memory	3,6 GB
Processor	Intel® Core™ i7 CPU M 620 @ 2.67GHz × 4
Graphics	Intel® Ironlake Mobile
OS	Linux Fedora 31

switches to Heap-Sort when the recursion exceeds a certain depth which is lg n where n is the number of elements.

3 Methods

We used Python unit testing framework to guarantee the correctness of the implemented algorithms. In these unit tests, we examined whether the algorithms succeeded in sorting a random shuffled list. The version for the source code of the unit tests is as shown in table 1. All benchmark data were saved into csv file named all_data.csv (see table 1). We extracted the minimum of run-time for each case and saved it into another csv file named clean_data.csv that was ready for plotting.

3.1 Benchmark Data

The data for the benchmark testing consisted of random lists of floats generated by using the NumPy random interface with the seed of 12235. The sorted and reversed data were generated by applying Python-Sort and reversed Python-Sort on the random generated data. We used list sizes from 16 to 16384 by taking the exponent of 2 to record a noticeable results.

3.2 Benchmark Setup

We used calculate_run_time function, as in the listing 7, which used python *timeit* function (Python Documentation [2020]). We used 3 repetitions for each combination of permutation, algorithm and list length. Benchmark data were saved on csv format using the function save_run_data.

3.3 Hardware and Software

The hardware and software specifications used for benchmark testing are shown in table 2 and table 3.

Listing 7 Benchmark Setup (Plesser [2020])

```
def calculate_run_time(algorithm, test_data,
                       len_list, case):
    clock = timeit.Timer(
        stmt = 'sort_func(copy(data))' ,
        globals = {'sort_func': algorithm,
                   'data': test_data,
                   'copy': copy.copy})
    n_ar, t_ar = clock.autorange()
    run_list = clock.repeat(repeat=3, n_ar)
    for rep, run_time in enumerate(run_list):
        save_run_time(algorithm, len_list,
                      case, run_time, rep)
```

Table 3: Software Specifications

Python	3.7.4
Pandas	1.1.3
Numpy	1.18.5
Scipy	1.5.2
matplotlib	3.3.2

4 Results

In this section, the visualisations of the execution of all the sorting algorithm in the different permutations are presented. First, the performances of all of the sorting algorithms are compared. Then, the results for each group of algorithm (quadratic, sub-quadratic, combined and built-in) are discussed in details. Figure 1 shows all the performances of all sorting algorithms using the different different permutations. Hence, we used log-scaled axes to be able to visualize all sorts and compare them in one graph. From figure 1, Python-Sort and Numpy-Sort show the best performances. Merge-Insertion-Sort gives similar results to Merge-Sort. In general, quadratic sorts perform the worst among all algorithms. The exception is having a sorted input data where Insertion-Sort run-time decreases dramatically.
Box-plots were created to check the variation in observations among the 3 repetitions. The interquartile range for Bubble-Sort is 1.7 seconds, while the interquartile range for Insertion-Sort 0.35 seconds. Also, the interquartile range for Merge-Insertion-Sort is 0.00004 seconds (see figure 2).

4.1 Quadratic Sorts

Figure 3 shows a comparison between Bubble-Sort and Insertion-Sort. Both sorting algorithms follow quadratic run-time except in best case of Insertion-Sort. Insertion-Sort follows linear run-time in sorting already sorted data. Overall, Insertion-Sort is 4 times faster than Bubble-Sort in sorting random data. Insertion-Sort also is 2 times faster than Bubble-Sort in sorting reversed data. It is also noticed that Insertion-Sort and Bubble-Sort take longer run-time to sort random data than reversed data. Bubble-Sort is 2 times faster in sorting random data than sorting reversed data.

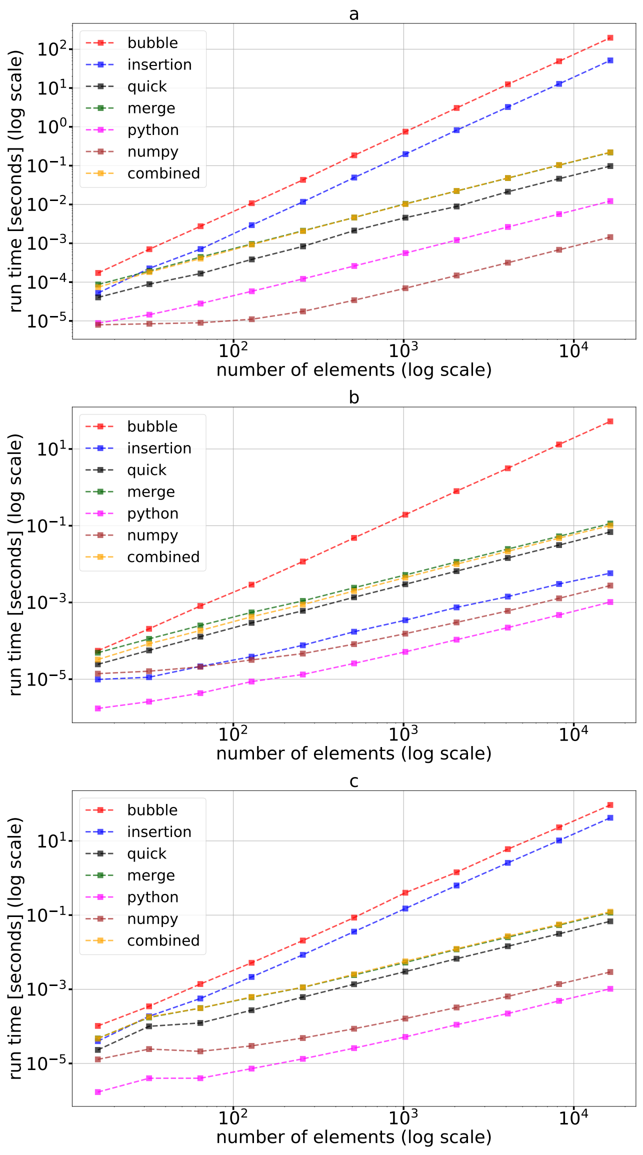

Figure 1: Comparison between all algorithms using a)random, b)sorted and c)reversed data. Dashed lines are depicted between the points for better illustration of trends.

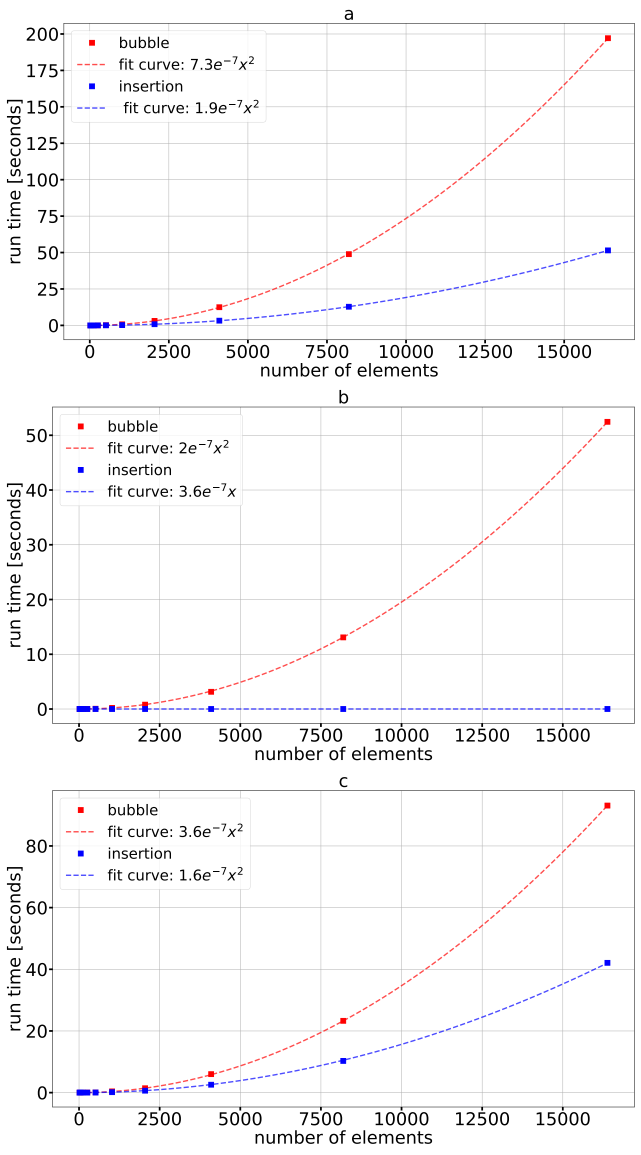

Figure 3: Comparison between Bubble-Sort and Insertion-Sort using a)random, b)sorted and c)reversed data. Bubble-Sort shows quadratic run-time on the different permutations. Insertion-Sort shows a linear run-time for sorting the sorted data.

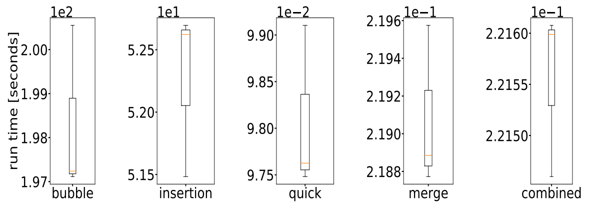

Figure 2: Box-plots for Bubble-Sort, Insertion-Sort, Quick-Sort, Merge-Sort and Merge-Insertion-Sort (combined sort).

4.2 Sub-Quadratic and Combined Sorts

Figure 4 shows the comparison between Quick-Sort, Merge-Sort and Merge-Insertion-Sort for the different data permutation cases. Quick-Sort is approximately 2 times faster than Merge-Sort. Merge-Insertion-Sort and Merge-Sort have very similar results in sorting the random data. For the data size 16384, Merge-Insertion-Sort is approximately 15% faster than Merge-Sort except in the case of reversed data where Merge-Sort is 6% faster than the Merge-Insertion-Sort.

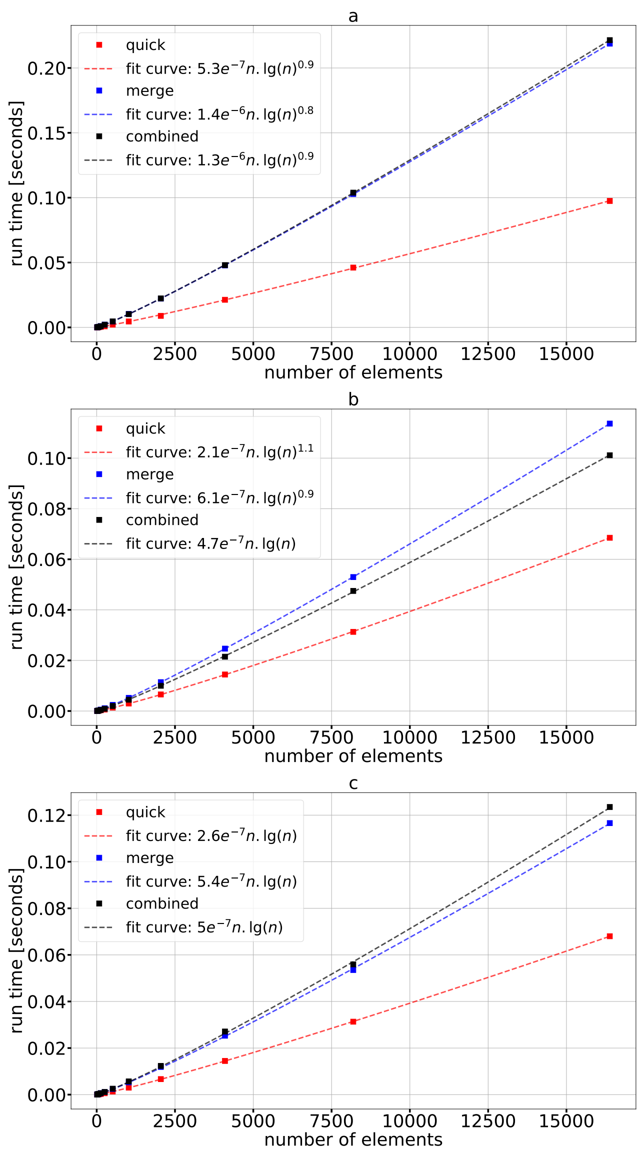

Figure 4: Comparison between Quick-Sort, Merge-Sort and Merge-Insertion-Sort using a)random, b)sorted and c)reversed data. Notice that Merge-Sort and Merge-Insertion-Sort have identical run-time in sorting random data.

4.2.1 Quick-Sort

From figure 5, sorting random data is only 0.6 as fast as sorting sorted or reversed data. Figure 5 show almost very similar results for reversed and sorted data. To further understand these results, we calculated number of executed comparisons within the algorithm for all data permutations and sizes. The results are shown in section.4.4.

4.2.2 Merge-Sort

In Merge-Sort, it is clear that sorting already sorted data takes very similar run-time as sorting reversed data while random data takes longer time as shown in figure 6. From the results, sorting already sorted or reversed data is 2 times faster than sorting random data.

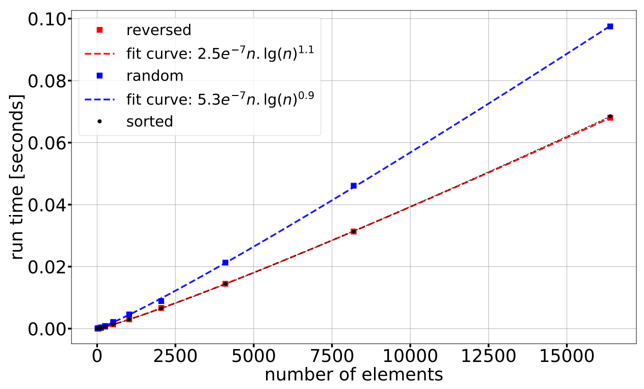

Figure 5: Benchmark results for Quick-Sort. Notice the similar run-time for sorting reversed and sorted data. Grey points circles represent the run-time using sorted data which overlap with the run-time using reversed data (red squares). We used the same fit curve for both the reversed and sorted data points.

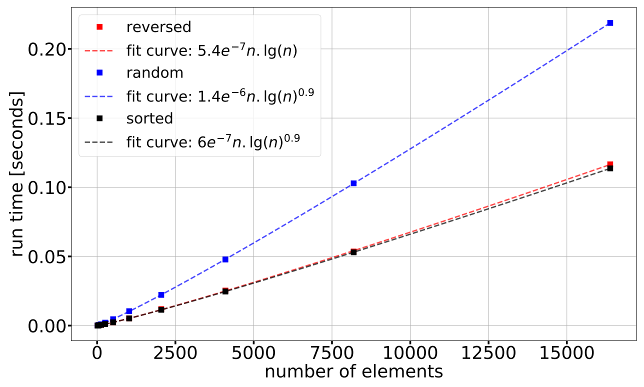

Figure 6: Benchmark results for Merge-Sort. Notice the similar run-time for sorting reversed and sorted data.

4.2.3 Merge-Insertion-Sort

Figure 7 shows that sorting reversed data is 2 times faster than sorting random data. Merge-Insertion-Sort does not show similar results for sorting sorted data and sorting reversed data unlike Merge-Sort and Quick-Sort.

4.3 Built-in Sorts

Python-Sort also shows the noticed observation of Merge-Sort that performances are almost identical for sorted and reversed data which is noticed from figure 8. Using Python-Sort to sort sorted and reversed data is almost 12 times faster than sorting random data. Also, Python-Sort is 10 times faster than our implementation Merge-Insertion-Sort.

Application of Numpy-Sort on random data is approximately 2 times faster than application of Numpy-Sort on sorted data as

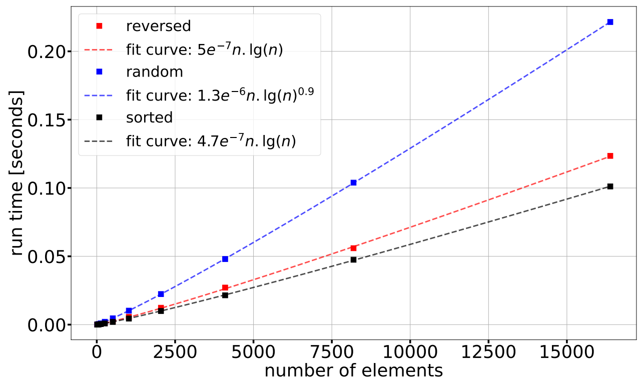

Figure 7: Benchmark results for Merge-Insertion-Sort

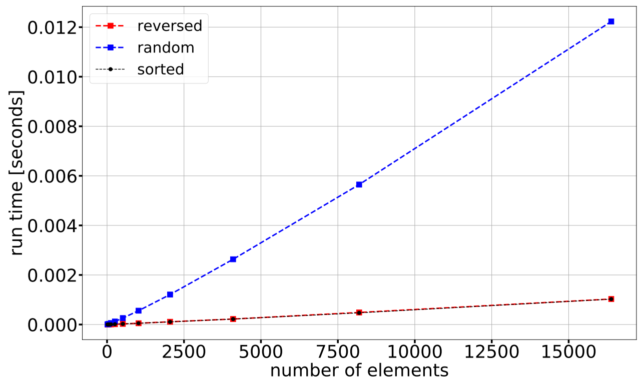

Figure 8: Benchmark results for Python built-in sort on sorted and reversed data. Notice the identical run-time for sorting reversed and sorted data. Grey points circles represent the run-time benchmark using sorted data which overlap with the run-time benchmark using reversed data.

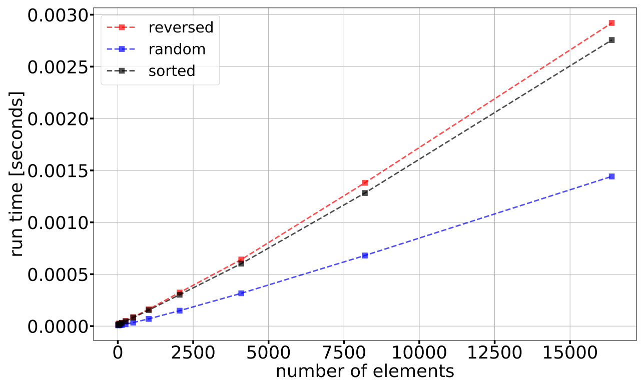

Figure 9: Benchmark results for Numpy-Sort

shown in figure 9. Also, it is noticed that Numpy-Sort is almost 10 times faster in sorting the random data than Python-Sort in sorting the random data.

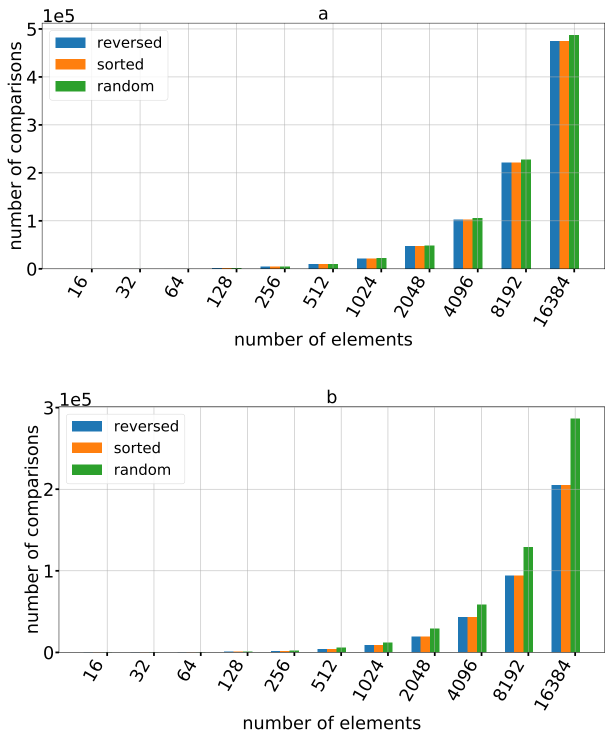

Figure 10: Number of comparisons for a)Merge-Sort and b)Quick-Sort using different data permutations and different data sizes

4.4 Number of Comparisons

For further analysis of the observations of Merge-Sort and Quick-Sort algorithms, we used a counter to measure number of executed comparisons by the algorithms using the different data permutations. Figure 10 shows the recorded number of comparisons. Merge-Sort and Quick-Sort have the exact same number of comparisons when sorting reversed and sorted data. The number of comparisons differs in the case of random data. Taking an example of a list with the size 1024, Merge-Sort execute 21503 comparisons to sort the sorted or the reversed data. On the other hand, it executes 22289 comparisons to sort the random data. For Quick-Sort, it executes 8716 comparisons to sort the reversed data or the sorted data while it executes 11682 comparisons to sort the random data.

5 Discussion

5.1 Quadratic Sorts

Both Bubble-Sort and Insertion-Sort proved to follow quadratic run-time in general except the case of using Insertion-Sort on sorted data which followed a linear run-time. Theoretically, the worst case scenario for Bubble-Sort and Insertion-Sort happens when the used data is in reversed order. The recorded run-time showed that it did not follow the theory as sorting random data was worse than sorting reversed data. Insertion-Sort proved to be 4 times faster than Bubble-Sort using the random data. Also, Insertion-Sort shows that it is more than 2 times faster than Bubble-Sort on sorting reversed data which follows the results measured by Astrachan [2003].

5.2 Sub-Quadratic Sorts

Asymptotically, Merge-Sort and Quick-Sort always take $\Theta(n \cdot \lg n)$ time ignoring the worst case of Quick-Sort. But we argue here that the cases which require more comparisons would take more time. So, if different list permutations have the same number of comparisons, that would lead to a similar run-time. Merge-Sort showed similar run-time for sorting sorted and reversed data which was attributed to the implemented algorithm in listing 4 that executed the same number of comparisons in sorting sorted and reversed data. Performance got worse on sorting the random data as the number of comparisons increased as shown in figure 10. Similarly, Quick-Sort showed similar results from sorted and reversed data because the number of comparisons were identical for sorted and reversed data. Quick-Sort is 2 to 3 times faster than Merge-Sort and these observations support the claims made by Skiena [2008].

5.3 Merge-Insertion-Sort

The combined Merge-Insertion-Sort did not have a dramatic change in performance in comparison with Merge-Sort as explained in details in the results section. The slight improvement in performance on sorting the already sorted data is attributed to the introduction of the Binary-Insertion-Sort. Binary-Insertion-Sort follows a linear function to sort sorted data. Additionally, the decrease in performance on sorting reversed ordered data was because Binary-Insertion-Sort follows a quadratic run-time in sorting reversed ordered data.

5.4 Built-in Sorts

Since Python-Sort uses Tim-Sort, we expected that the run-time for sorting the already sorted data and reversed data should not be similar and the results did not follow this argument. That is because our implementation of Tim-Sort is still not the same as the Python Software Foundation [2020] implementation of Tim-Sort. This introduced a challenge to study the Python Software Foundation [2020] implementation of Tim-Sort in details to understand what made it 10 times faster than our implementation of Merge-Insertion-Sort. Further work in this area is needed to give an explanation of the performance of Python-Sort. Numpy-Sort also followed the same phenomena of Quick-Sort where the run-time of sorting reversed or sorted data is relatively similar. That is because Numpy-Sort uses Quick-Sort by default. For data length 16384, the run-times were 0.002756 and 0.002920 seconds for sorting the already sorted and reversed data respectively. This is because the algorithm is using Quick-Sort that has identical number of comparisons for sorting sorted and reversed data. However, using Numpy-Sort to sort random data was 2 times faster than using Numpy-Sort to sort the already sorted data. This observation differed from the observation of Quick-Sort. This introduced another challenge to study Harris et al. [2020] implementation of Numpy-Sort in details.

6 Conclusion

In this study, we implemented benchmark testing on several sorting algorithms using several input data permutations. Bubble-Sort and Insertion-Sort proved to follow quadratic run-time. Insertion-Sort proved best case scenario on sorted data as it was expected. Also, Insertion-Sort proved to be more efficient than Bubble-Sort which supported Astrachan [2003] claims.
Quick-Sort and Merge-Sort shared an interesting phenomena that they showed the same run-time for sorting the already sorted and reversed sorted data which was analysed further by studying the number of executed comparisons. Quick-Sort was 2 times faster than Merge-Sort which supported the claims of Skiena [2008]. Combined Merge-Insertion-Sort gave slightly better results than Merge-Sort but it was still inferior to Quick-Sort.
Python-Sort uses Merge-Sort at larger data which means the execution time should be similar for sorting the already sorted and reversed sorted data which is recorded in the results. Sorting random data was recorded to have poorer performance than sorting the already sorted and reversed data which follows Merge-Sort. Python-Sort showed that it is 10 times faster than our implementation of Tim-Sort. Numpy-Sort showed interesting results that was not as predicted where it showed that Numpy-Sort is 2 times faster when sorting random data than sorting sorted or reversed data. Further work is needed to understand the performance of the Python Software Foundation [2020] implementation of Tim-Sort and Harris et al. [2020] implementation of Numpy-Sort in details.

Acknowledgments

We want to express our gratitude to Prof. Hans Plesser for the guidance through this study. We would also like to thank PhD candidate Bao Ngoc Huynh for her suggestions and comments.

References

Owen Astrachan. 2003. Bubble Sort: An Archaeological Algorithmic Analysis. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)* (03 2003). https://doi.org/10.1145/611892.611918

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. 2018. On the Worst-Case Complexity of TimSort. (05 2018).

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.

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kristijan Kuk, Petar Milic, Petar Spalević, and Milan Gocic. 2019. Algorithm design in Python for cybersecurity.

Peter M. McIlroy. 1993. Optimistic Sorting and Information Theoretic Complexity. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, Vijaya Ramachandran (Ed.). ACM/SIAM, 467–474. http://dl.acm.org/citation.cfm?id=313559.313859

Hans Ekkehard Plesser. 2020. Benchmarking sorting algorithms in Python. *Data Science Section, Faculty of Science and Technology, Norwegian University of Life Sciences* (2020).

Python Documentation. 2020. *timeit.* https://docs.python.org/2/library/timeit.html [Online; accessed 13-November-2020].

Python Software Foundation. 2020. *CPython Source Code.* https://github.com/python/cpython/blob/master/Objects/listobject.c [Online; accessed 27-November-2020].

I. S. Rajput, Bhawnesh Kumar, and Tinku Singh. 2012. Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms. *International Journal of Computer Applications* 57 (2012), 14–22.

Steven S. Skiena. 2008. *The Algorithm Design Manual.* Springer, London. https://doi.org/10.1007/978-1-84800-070-4

Wikipedia contributors. 2020. Timsort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Timsort&oldid=990880207 [Online; accessed 1-December-2020].