Benchmarking Sorting Algorithms in Python

INF221 Term Paper, NMBU, Autumn 2020

Krishna Mohan Shah
krishna.mohan.shah@nmbu.no

Phong Hai Nguyen
phong.hai.nguyen@nmbu.no

ABSTRACT

Most of the time, we need to rearrange the data to efficaciously use it. Over the years, computer scientists have created many sorting algorithms to sort data. In this paper, We examine and evaluate the speed of sorting algorithms. We study different sorting algorithms on differently ordered data of input size up to ten thousand. We generate three differently ordered data set to simulate best-case, worst-case, and average-case scenarios. The motivation for this paper was to analyze the runtimes of different sorting algorithms presented in the theories and practice.

Among Insertionsort, Bubblesort, Mergesort, Quicksort, Timsort, Built-in Python sort, and NumPy sort that we tested, Built-in Python was the fastest in sorting sorted and reversed data whereas NumPy sort execution was better for large randomized data. Though Bubblesort performed well with sorted data, it lagged in sorting random data by almost 25000 times for ten thousand input data as per the result from the benchmark. The built-in sorting function in Python and NumPy sort appears to triumph over the alternative sorting technique if we focus on the time performance only

1 INTRODUCTION

"An algorithm is a sequence of computational steps that transform the input into the output" [Cormen et al. 2009, Ch. 1]. Algorithms define a series of procedures to process the desired input to the desired output. There are many types of algorithms namely, searching algorithms like Binary search, sorting algorithms like Mergesort, and compression algorithms like Data compression.

Sorting means putting elements in an ordered sequence. Several sorting algorithms can be used to sort a set of elements. An efficient sorting algorithm will contribute to a better application [Edelkamp and Weiß 2019]. In this paper, we work on some common sorting algorithms. We test different sorting algorithms with three different ordered data sets to put them in best-case, worst-case and average-case scenarios. These different data sets are sorted, reversed, and randomly ordered. We benchmark these algorithms with above mentioned data sets of input size from ten up to ten thousand.

This paper also presents brief theories of different sorting algorithms and analyses the benchmark performance on quadratic algorithms (insertion and bubble sort), sub-quadratic algorithms (merge, and quick sort), combined algorithm (merge sort switching to insertion sort for small data), and built-in sorting functions (Python “sort()”, NumPy “sort()”) to compare performance amongst them.

2 THEORY

2.1 Insertion Sort

Insertion sort is an efficient algorithm that works the way people sort a hand of playing cards[Cormen et al. 2009, Ch. 2]. Small process description: It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array.

If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead. Listing 1 is the pseudocode of the Insertion sort. This algorithm is efficient for sorting a small number of elements. The worst case of insertion sort is O(n2) [GeeksforGeeks 2020a; Wikipedia 2020a] and it happens when the list to be sorted is in the reverse order [GeeksforGeeks 2020a]. A downside of this algorithm is that it is impractical for sorting large arrays as the average case is also quadratic. [Wandy 2020; Wikipedia 2020a]. For N elements to be sorted, it requires N square step to do the sorting[Wandy 2020].

Listing 1 Insertion sort algorithm from [Cormen et al. 2009, Ch. 2].

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

Best case runtime for this algorithm is:

$$T\left(n\right) = \Theta\left(n\right) \ . \tag{1}$$

It is achieved for correctly sorted input data.

2.2 Bubble sort

Bubble sort is a comparison sort that basically just runs through an array and swaps adjacent elements if they are out of order until the list is sorted. Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$ [Wikipedia 2020].

It requires only O(1) space, as the only extra memory it requires is that of the temporary value created when swapping adjacent elements. It is not efficient in the case of a reverse-ordered collection and does not work well with large collections. The optimised algorithm was taken from [Zovko 2020] and it was tested separately before applying it.

Listing 2 Bubble sort algorithm with modification from [Zovko 2020]

```
BubbleSort(A)
 1  n = A.length
 2  update = True
 3  j = 1
 4  while update == True and n > 1
 5      update = False
 6      for i = 1 to n − j
 7          if A[i − 1] > A[i]
 8              exchange A[i − 1] with A[i]
 9              update = True
10      n = n − 1
11      j = j − 1
12  return A
```

2.3 Merge sort

Merge sort is a divide-and-conquer algorithm and the idea behind the merge sort is that it divides the initial list into sub-lists until there is one element in each sub-list. Then it merges those sublists in the manner at its results into a sorted list. The worst and the average-case performance of the merge sort is O($nlgn$) [Bunse et al. 2009; Wikipedia 2020b]. It can be used to implement a stable sort and is highly parallelizable. The drawback of mergesort is that it requires extra space for storing data temporary.

Listing 3 Merge sort algorithm from [Cormen et al. 2009, Ch. 2]

```
Merge − Sort(A,p,r)
1  if p < r
2      q = (p + r)/2
3      Merge − Sort(A,p,q)
4      Merge − Sort(A,q+1,r)
5      Merge(A,p,q,r)
```

2.4 Tim sort

Timsort is a data sorting algorithm that implements the idea that the real-world data sets almost always contain already ordered subsequences, so the sorting strategy is to identify them and sort them further using both merge and insert methods. It was implemented by Tim Peters in 2002 for use in the Python programming language. Timsort performs exceptionally well on already-sorted or close-to-sorted lists, leading to a best-case scenario of O(n). the worst case for Timsort is also O($nlog2n$) [Valdarrama 2020; Wikipedia 2020b].

Here, Minrun is a size that is determined based on the size of the array. Minrun is chosen from the range 32 to 64 inclusive, such that the size of the data, divided by minrun, is equal to, or slightly less than, a power of two [Wikipedia 2020b]. The algorithm selects it so that most runs in a random array are, or become minrun, in length. Here, if the array we are trying to sort has fewer than 64 elements in it, Timsort will execute an insertion sort. Its execution time is accurately fast.

Listing 4 Tim sort algorithm [GeeksforGeeks 2020b].

```
Timsort(A)
 1  n = A.length
 2  minRun = cMinRun(n)
 3  tart = 1 to n 1; step = minRun
 4      end = min ( start, minRun, n)
 5      timinsertion(A, start, end)
 6  size = minRun
 7  while size < n
 8      for left = 1 to n Step = (2 ∗ size) + 1
 9          mid = min(n, left+size)
10          right = min((left+2, size),n)
11          timmerge(A, left, mid, right)
12      size= 2∗ size
```

2.5 Quick sort

Quicksort uses the divide-and-conquer technique. It uses a pivot element to partition the array into two subarrays according to whether they are less than or greater than the pivot and after sorting subarrays recursively, the elements on the left subarrays are less or equal to the elements placed in the right sub-arrays. Quicksort's best-case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other.

The best and the average-case performance of the quick sort is O($n \lg n$). The worst-case performance of the quick sort is O(n^2) when the pivot element is the smallest or the largest in the list or when all the elements are equal [Bunse et al. 2009; Wikipedia 2020a]. It has an efficient average-case compared to any aforementioned sort algorithm.

Listing 5 Quick sort algorithm from [Cormen et al. 2009, Ch. 7.1]

```
Quicksort(A,p,r)
1  if p < r
2      q = Partition(A,p,r)
3      Quicksort(A,p,q − 1)
4      Quicksort(A,q 1,r)
```

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

```
Partition(A,p,r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i =i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i 1
```

2.6 Python sort and Numpy sort

Python sort is a built-in function that is available in a standard installation of Python. It is a method for the list data type and only works on lists [Python Software Foundation 2020]. While using sorted(), the original list is unchanged as it sorts any list by copying the element in a new list. Numpy sort returns a sorted copy of an array. It sort arrays of strings, or any other data type. By default, NumPy sort is based on Quicksort. However, the user may specify if it will use e.g NumPy based on merge-sort or heap-sort instead [The SciPy Community 2020].Based on the datatype and by considering the stable, the average speed and the space, the NumPy-sort will automatically switch from default quick sort to Heapsort , mergesort or Timsort[The SciPy Community 2020]

3 METHODS

Test-data for the benchmarks was generated by the random function in NumPy. Three tested data cases were used: the random case (created by random function), the sorted (ascending order: sorted the random case by built-in sorted function), the reverse case (descending order, reversing of the sorted case).

We implemented the benchmarking of algorithms in Python (ver. 3.8) using Jetbrains Pycharm 2020.2.3. We tested the correctness of the algorithms by generating small data sizes and verifying them with self-examination. We provided a fresh copy of data to be sorted on every call to the *sort_func* parameter. We used the minimum time value to plot the graphs after executing 7 repetitions to sort the same amount of data. We have also saved the results so that we don't have to run the tests over again.

We generated random data using the Numpy library in Python. Pandas and matplotlib library in python were used to plot and analyze different kinds of input data which were sorted, reverse, and random data.

The benchmarks were run on a computer with processor specifications: Intel Core i5 – 7200U @ 2.50Ghz (2 Cores) with 8 GBytes of DDR4 RAM. Jetbrains Pycharm (ver. 2020.2.3), Python (ver. 3.8) with Numpy (ver 1.19.3), and Pandas (ver. 1.1.4) were used. The implementations are in Python file which is stored at our Gitlab repository https://gitlab.com/nmbu.no/emner/inf221/h2020/student-term-papers/team_09/inf-221-term-paper-team-09

Listing 7 Script for timinig with `timeit.Timer` [Plesser 2020].

```
import numpy as np
import timeit
import copy

rng = np.random.default_rng(12235)
test_data = np.random.uniform(size=100)

clock = timeit.Timer(
          stmt='sort_func(copy(data))',
          globals={'sort_func': sorted,
                   'data': test_data,
                   'copy': copy.copy})

n_ar, t_ar = clock.autorange()

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

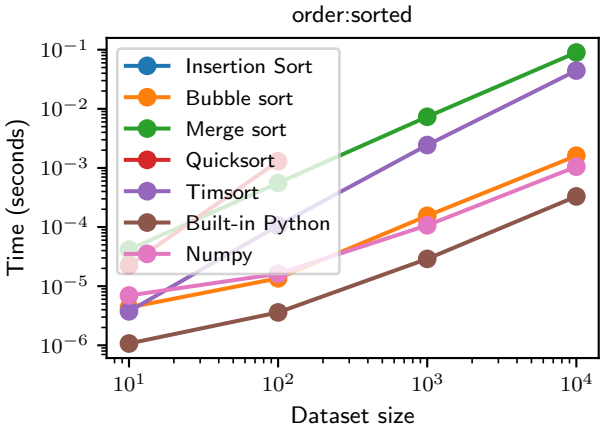

Figure 1: Benchmark results for sorted data of all algorithms.

4 RESULTS

	1000	Insertion	Bubble	Merge	Quick	Timsort	Python	NumPy
0	sorted	0.000296	0.000155	0.007337	NaN	0.002441	0.000029	0.000107
1	reverse	0.126016	0.188451	0.007214	NaN	0.009707	0.000030	0.000111
2	random	0.167915	0.410999	0.009422	0.007331	0.013799	0.000390	0.000041

Table 1: Benchmark results for a thousand of data, minimum time

From Table 1, for sorted data, the benchmarking results show that python sort is the fastest for sorting in $29\mu s$ followed NumPy sort in $107\mu s$ then Bubblesort in $155\mu s$ and Insertionsort and Timsort in $296\mu s$ and $2.4ms$ respectively.

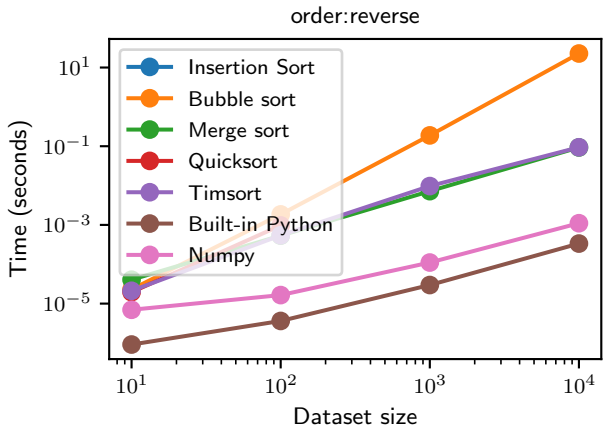

Figure 2: Benchmark results for reversed data of all algorithms.

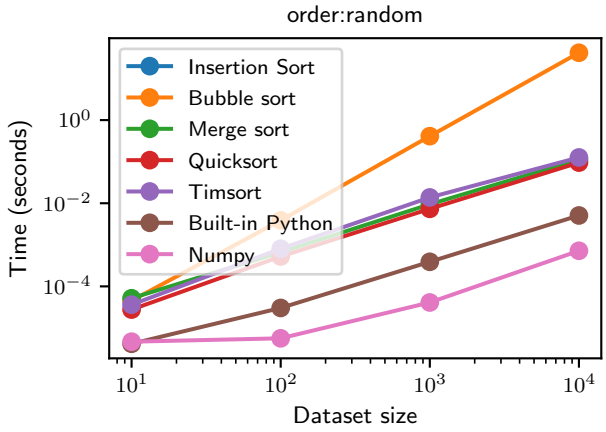

Figure 3: Benchmark results for random data of all algorithms.

	10000	Insertion	Bubble	Merge	Quick	Timsort	Python	NumPy
0	sorted	0.003343	0.001610	0.089832	NaN	0.044378	0.000330	0.001051
1	reverse	14.789248	22.706867	0.093134	NaN	0.093957	0.000336	0.001103
2	random	17.108207	41.722382	0.111818	0.094978	0.126398	0.005078	0.000718

Table 2: Benchmark results for thousands of data, minimum time

Quicksort is excluded because of the recursion limit of a thousand. For reversed sorted data of a thousand elements, Python sort is the quickest among all sorting in $3\mu s$ and NumPy sort being the second sorting in $0.1ms$ followed by Mergesort, Timsort, Insertion sort, and Bubblesort.

For random data of a thousand elements, NumPy sort finishes in $41ms$ faster than Python sort that finishes sorting in $0.39ms$. Bubblesort completes in $0.41s$ which is the worst of them all followed by Insertionsort that sorts in $0.18s$. We

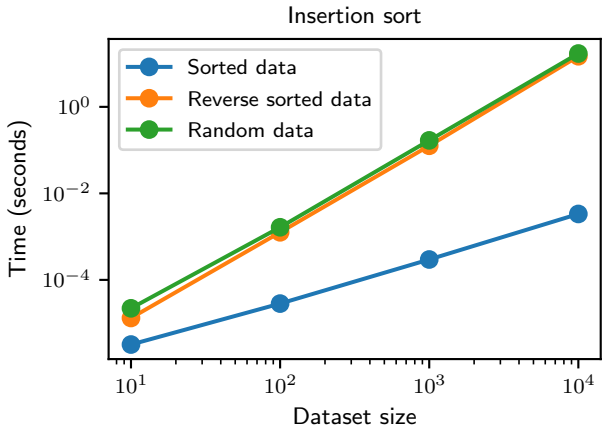

Figure 4: Benchmark results for Inserion sort.

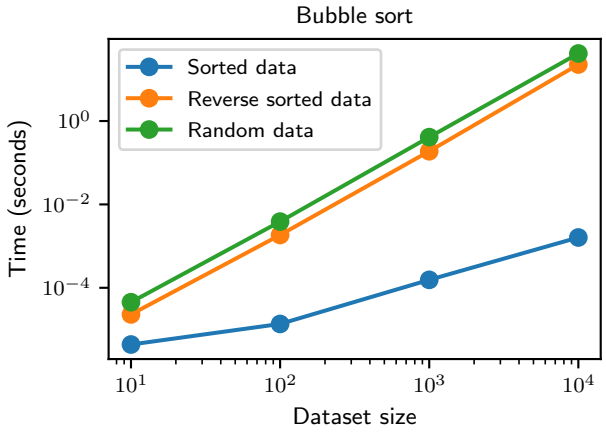

Figure 5: Benchmark results for Bubble sort.

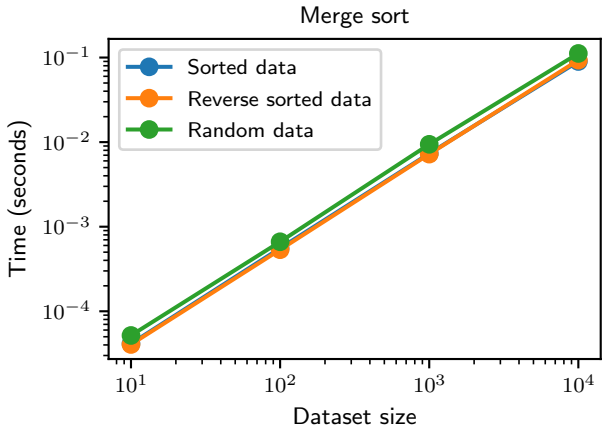

Figure 6: Benchmark results for Merge sort.

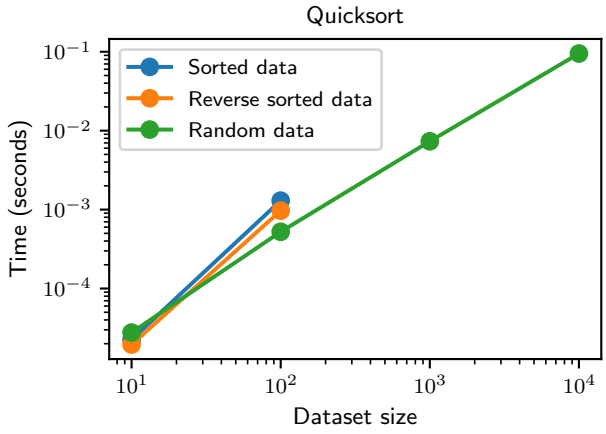

Figure 7: Benchmark results for Quick sort.

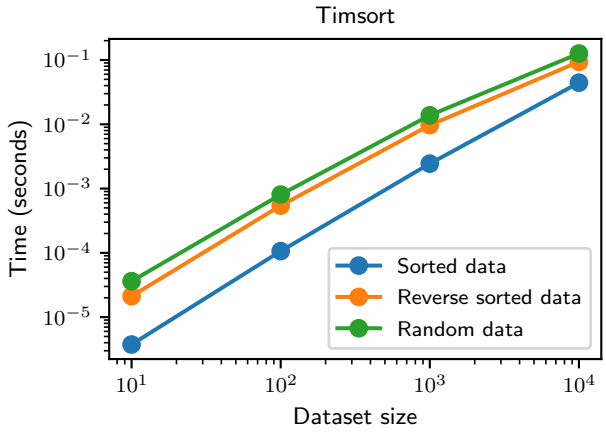

Figure 8: Benchmark results for Tim sort.

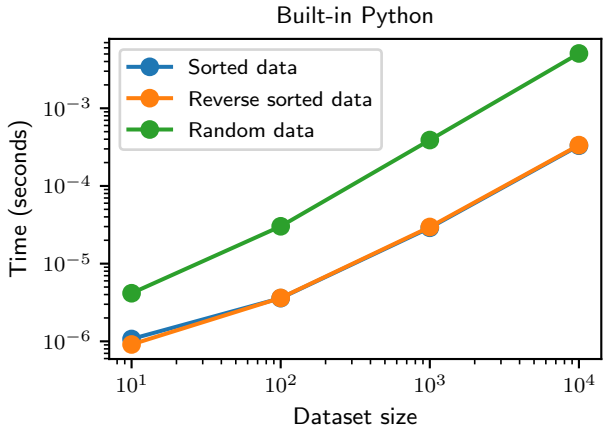

Figure 9: Benchmark results for Built-in Python sort.

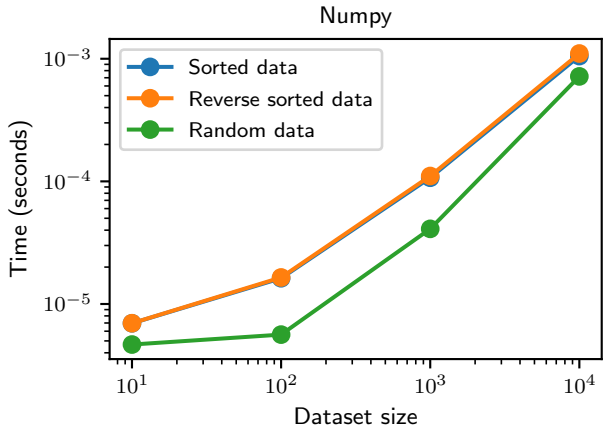

Figure 10: Benchmark results for NumPy sort.

can see from Table 2, Python sort has the lowest sorting time in sorted and reverse data. NumPy sort is the quickest in sorting random data. Python sort and Numpy are performing relatively better than Python implemented sorting algorithms. Bubblesort speed decreases as it goes from sorted to reverse to random just as Insertion sort.

In figure 1, for sorted data Built-in Python sort is the fastest followed by NumPy sort. NumPy sort performs better while sorting random data. Bubblesort performs quicker with sorted data than with random and reverse data. The duration for sorting random and reverse data is almost identical for Timsort.

5 DISCUSSION

The ordering of data did not seem to largely affect the sorting time in Mergesort. Mergesort took the same time for sorting sorted data, reverse data as well as random data (Fig.6). This proves that the Mergesort is a stable sorting algorithm for large data set (if we see the time complexity). Mergesort requires more memory space than some other simple sorting algorithms(i.e. bubble sort and insertion sort)[Valdarrama 2020]. This memory requirement may become a limitation of this sorting algorithm. Depending on the requirement/ priority on time or space, this Mergesort will be chosen or not.

The sorting time of random and reverse data is the same for Insertionsort in spite of the size of the input data (Fig. 4). Insertion sort was relatively faster for sorting a sorted data. Here, the inner loop is never executed, resulting in an O(n) runtime complexity, just like the best case of bubble sort.

Bubble sort is worst for random data (Fig. 3) because Bubble sort repeatedly passes through the unsorted part of the list, it has a worst-case complexity of $O(n^2)$. This leads to the conclusion that the impractical selection of Bubble sort for sorting large data sets.

Table 3: Versions of files used for this report; GitLab repository https://gitlab.com/nmbu.no/emner/inf221/h2020/student-term-papers/team_09/inf-221-term-paper-team-09/.

File	Git hash
`Insertion sort_c.pdf`	`de36bff`
`Bubble sort_c.pdf`	`de36bff`
`Timsort_c.pdf`	`de36bff`
`Quicksort_c.pdf`	`de36bff`
`Merge sort_c.pdf`	`de36bff`
`Numpy_c.pdf`	`de36bff`
`Built-in Python_c.pdf`	`de36bff`
`sorted.pdf`	`de36bff`
`reverse.pdf`	`de36bff`
`random.pdf`	`de36bff`

When sorting a small array, the Timsort algorithm turns into a single insertion-sort. It results in an improvement in the efficiency of Timsort for a small array.

Bubble sort's sorting curve is quite similar to the Insertion sort. Python sort is the most efficient in runtime when looking at the sorted data and reverse data while NumPy is the fastest for random data for ten thousand data (Fig. 1,2 and 3). Among the Python implemented sorting methods, Quicksort is the fastest with random data whereas it lags behind in sorted and reversed data.

Figure [4-9] shows the time complexity of the tested sorting-algorithms. Different data types (sorted data set, reverse sorted data set, or random data set) affect the time efficiency of the sorting algorithms. It is also very important to understand the characteristics of the data set as well as the efficiency in using the computer's memory of the sorting algorithms before choosing a suitable sorting algorithm. In addition, further study may improve the time efficiency of the sorting algorithms.

ACKNOWLEDGMENTS

We are grateful to Professor Hans Ekkehard Plesser who is responsible for the course, Computer Science for Data Scientists, and also for providing us with necessary materials.

We would like to thank the teaching assistant Ashesh Gnawali for his continuous support and supervision. We would also like to thank Bao Ngoc Huynh for reading our report and giving us invaluable feedbacks.

REFERENCES

Christian Bunse, Hagen Höpfner, Suman Roychoudhury, and Essam Mansour. 2009. Choosing the "Best" Sorting Algorithm for Optimal Energy Consumption. In *ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 2, Sofia, Bulgaria, July 26-29, 2009*, Boris Shishkov, José Cordeiro, and Alpesh Ranchordas (Eds.). INSTICC Press, 199–206. https://dblp.org/rec/conf/icsoft/BunseHRM09.bib

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.

Stefan Edelkamp and Armin Weiß. 2019. BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. *ACM J. Exp. Algorithmics* 24, Article 1.4 (2019), 22 pages. https://doi.org/10.1145/3274660

GeeksforGeeks. 2020a. Time complexity of insertion sort when there are O(n) inversions. https://www.geeksforgeeks.org/time-complexity-insertion-sort-inversions/ Online; Retrieved 2020-11-20.

GeeksforGeeks. 2020b. TimSort. https://www.geeksforgeeks.org/timsort/ Online; Retrieved 2020-11-20.

Hans Ekkehard Plesser. 2020. Benchmarking sorting algorithms in Python, Sample Term Paper. Retrieved 2020-10-15.

Python Software Foundation. 2020. Sorting HOW TO. https://docs.python.org/3/howto/sorting.html Online; Retrieved 2020-11-15.

The SciPy Community. 2020. numpy.sort. https://numpy.org/doc/stable/reference/generated/numpy.sort.html Online; Retrieved 2020-11-15.

Santiago Valdarrama. 2020. Sorting Algorithms in Python. https://realpython.com/sorting-algorithms-python Online; Retrieved 2020-11-15.

Joe Wandy. 2020. The Advantages and Disadvantages of Sorting Algorithms. https://sciencing.com/the-advantages-disadvantages-of-sorting-algorithms-12749529.html Online; Retrieved 2020-11-20.

Wikipedia. 2020. Bubble Sort. https://en.wikipedia.org/wiki/Bubble_sort Online; Retrieved 2020-11-15.

Wikipedia. 2020a. Insertion Sort. https://en.wikipedia.org/wiki/Insertion_sort Online; Retrieved 2020-11-15.

Wikipedia. 2020b. Merge Sort. https://en.wikipedia.org/wiki/Merge_sort Online; Retrieved 2020-11-15.

Wikipedia. 2020a. Quick Sort. https://en.wikipedia.org/wiki/Quick_sort Online; Retrieved 2020-11-15.

Wikipedia. 2020b. Timsort. https://en.wikipedia.org/wiki/Timsort Online; Retrieved 2020-11-15.

Ante Zovko. 2020. Bubble Sort and Optimized Bubble Sort Algorithms in Python. https://gist.github.com/piotechno/8710199#gistcomment-3110086 Online; Retrieved 2020-11-15.