Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2021

Christianie Torres
christianie.torres.nmbu.no
NMBU
Norway

Jorid Holmen
jorid.holmen@nmbu.no
NMBU
Norway

Mathilde Haglund
mathilde.haglund@nmbu.no
NMBU
Norway

ABSTRACT

In this paper, we performed and analysed different sorting algorithms in Python. The motivation for this project was to discover the most effective sorting algorithm to use on different data sizes and orders. Sorting algorithms are essential to the performance of numerous operations such as search and database operations. Thus, there will be instances where it is necessary to know which sorting algorithm is the most efficient. Sorted data can also be helpful for selecting or duplicating items in an array and for analysing frequency distribution. The goal of this term paper was to compare real-life behaviour of sorting algorithms with theoretical expectations. We benchmarked the runtime of the data using a timer in Python. The benchmark data were later saved in a data frame, which was used for plotting the results. The results showed that the built-in Python algorithms were by far the most effective. The quadratic algorithms, especially Bubble Sort, was the least effective, except for Insertion Sort on sorted data. Some algorithms did not perform exactly as we expected, but the results mostly coincided with the time complexity of the algorithms.

1 INTRODUCTION

Benchmark testing sorting algorithms can help us understand how the varying input sizes and input orders affect the running time of each sorting algorithm. In addition, the benchmark testing can help us identify areas for improvement in instances where there are any performance gaps. In this paper we wanted to compare real-life behaviour of sorting algorithms with theoretical expectations. The performances of the different algorithms were also compared with each other. The specific theory on each sorting algorithm is discussed in section 2. In section 3, we explained the methods we used to benchmark as well as the hardware and software we used. Furthermore, the numerical results and observations are represented in section 4 and discussed in section 5. Section 5 also uses the results to compare with the theoretical expectations and to better understand the behaviour of the sorting algorithms.

2 THEORY

2.1 Quadratic algorithms

2.1.1 *Insertion Sort*.

Insertion Sort is an efficient algorithm for sorting a small number of elements (Cormen et al. [2009, Ch. 2.2]). This algorithm starts with one sorted array and one non-sorted array. Then the algorithm inserts the non-sorted elements one by one into the correct place among the sorted elements. Comparison is made by checking if an element is smaller than the preceding element. If the statement is fulfilled, the two elements will be swapped.

The algorithm's worst-case scenario happens when the data is sorted in reverse order (Cormen et al. [2009, Ch. 2.2]). In this case, each element from the array will be compared with all the other array elements, meaning the algorithm must go through the while-loop until $i = 0$ in line 3 listing 1 for each iteration (Cormen et al. [2009, Ch. 2.2]). Both the worst-case and average case running time are therefore quadratic, shown in equation (1).

Insertion Sort's best-case is when the data is sorted in advance, which means that the algorithm does not perform any swap operation. Thus, the algorithm only goes through a for-loop and never enters the while-loop in listing 1 line 4, as the condition is not satisfied (Cormen et al. [2009, Ch. 2.2]). The best-case scenario has a linear runtime, shown in equation (2).

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

$$T(n) = \Omega(n) \tag{2}$$

Listing 1 Insertion Sort algorithm from Cormen et al. [2009, Ch. 2.1].

```
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
```

2.1.2 *Bubble Sort*.

Bubble Sort is an algorithm that iterates from the leftmost element, goes to the rightmost element and swaps the pairs if the element to the left is bigger than the element to the right. In the first run in listing 2, the algorithm iterates through all the array elements and pushes the greatest element to the rightmost place. In the next run, the algorithm does the same process, but pushes the next greatest element to the next rightmost place. The algorithm follows this procedure for the rest of the iterations as well.

The worst-case scenario for Bubble Sort occurs when the data is in descending order, while the best-case occurs when the data is sorted (GeeksforGeeks contributors [2021a]). In the worst-case the swapping of the pairs will always happen, which means that the condition of the if-statement in listing 2 line 3 is satisfied for each iteration. Bubble Sort's time complexity for all the cases is shown in equation (3).

$$T(n) = \Theta(n^2) \tag{3}$$

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

```
BUBBLE-SORT(A)
1  for i = 1 → A.length
2      for j = 1 → A.length − 1
3          if A[j] > A[j + 1]
4              exchange A[j] with A[j + 1]
```

2.2 Sub-quadratic algorithms

2.2.1 *Merge Sort*.

Merge Sort is a sorting algorithm that recursively splits the input array in half. Afterwards the algorithm merges the halves into a sorted array. A disadvantage of Merge Sort is that it always follows the whole procedure, even on an input array sorted in advance (GeeksforGeeks contributors [2021b]).The pseudo-code for the Merge Sort algorithm is shown in listing 4, and it implements the merge algorithm, shown in listing 3.

Merge Sort's worst-case time complexity occurs when both the left and right subarrays have alternate elements of the sorted array, every time we implement the merge algorithm (Baeldung author [2021]). This results in the algorithm comparing each element from both subarrays at least once (Baeldung author [2021]), which is most likely to happen on random data. In spite of this being Merge Sort's worst-case, the time complexity, shown in equation (4), is the same for all cases.

$$T(n) = \Theta(n \cdot \lg n) \tag{4}$$

Listing 3 Merge algorithm (Modified after GeeksforGeeks contributors [2021b]).

```
MERGE(A, LEFT, RIGHT, START)
 1  i, j, k = 0, 0, START
 2  while i <= LEFT.length and j <= RIGHT.length
 3      if LEFT[i] <= RIGHT[j]:
 4          A[k] = LEFT[i]
 5          i = i + 1
 6      else :
 7          A[k] = RIGHT[j]
 8          j = j + 1
 9      k = k + 1
10  while i < LEFT.length):
11      A[k] = LEFT[i]
12      k = k + 1
13      i = i + 1
14  while j < RIGHT.length):
15      A[k] = RIGHT[j]
16      k = k + 1
17      j = j + 1
```

2.2.2 *Quicksort*.

Quicksort is an algorithm where the main idea is that an element is chosen as pivot and the given array is divided into portions around that pivot. In this paper, we have implemented a version where the middle element is chosen. The elements in the array that are smaller than the pivot are placed in the left subarray before the pivot, and the elements greater than the pivot are placed in the right subarray after the pivot. If there are some elements that are equal to the pivot, they are added to an equal subarray. The algorithm then recursively calls on the Quicksort algorithm again, so that each subarray undergoes the same procedure as the one described above. The recursion continues until the whole array is sorted. The pseudo-code for Quicksort is shown in listing 5.

Quicksort's worst-case arises when the chosen pivot from the partition function is the greatest or smallest element of the input array every time the function is called upon (Cormen et al. [2009, Ch. 7]). In this case all the elements are placed either above the pivot or under the pivot, resulting in uneven subarrays. As we have chosen the middle element as pivot, this scenario will most likely happen on a randomly sorted array. The time complexity for Quicksort's worst-case is quadratic, which is shown in equation (5).

Quicksort's best-case arises when the chosen pivot from the partition process is the middle value of the input array every time the function is called upon (Cormen et al. [2009, Ch. 7]). This scenario will happen when the array is sorted or reversed sorted. The best-case and average running time is shown in equation (6).

$$T(n) = O(n^2) \tag{5}$$

$$T(n) = \Theta(n \cdot lgn) \tag{6}$$

Listing 4 Merge_sort algorithm (modified from Cormen et al. [2009, Ch. 2.3]).

```
MERGE_SORT(A)
1  if A.length > 1:
2      q = A.length/2
3      R = A[q... A.length]
4      L = A[1... q]
5      merge_sort(R)
6      merge_sort(L)
7      A = merge(A, L, R)
```

Listing 5 Quicksort algorithm (modified after Mailund [2021, Ch. 9].

```
QUICKSORT(A)
 1  if A.length > 1
 2      p = A[A.length//2]
 3      for i = 1 to A.length
 4          if A[i] < p
 5              l.insert(A[i])
 6          elseif A[i] = p
 7              e.insert(A[i])
 8          elseif A[i] = p
 9              r.insert(A[i])
10      A = Quicksort(l) + e + Quicksort(r)
```

2.3 Combined algorithms

2.3.1 *Combined Merge Sort*.

Combined Merge Sort is an algorithm using Merge Sort with Insertion Sort. Merge Sort is generally faster than Insertion Sort, but Insertion Sort is faster for small input sizes. For proof, see Cormen et al. [2009, Ch. 2.3]. As mentioned in section 2.2.1, Merge Sort divides the original array into several subarrays, and afterwards merge the subarrays into a sorted array. In Combined Merge Sort the smaller subarrays will be sorted using Insertion Sort. When Merge Sort merges in the second part of the algorithm, it merges previously sorted arrays, which leads to fewer numbers of comparisons needed to sort the data.

The pseudo-code for Combined Merge Sort is shown in listing 7. The algorithm defines Insertion_sort_for_combined shown in listing 6, which is similar to the original Insertion Sort. The difference is that Insertion_sort_for_combined adds the inputs 'left' and 'right' to know which side of the array we are about to sort. Therefore, Merge_insertion first creates slices of the array with size min_run and sorts them using Insertion_sort_for_combined. Afterwards the algorithm merges the slices into bigger slices, and the slice size doubles for each iteration through the loop. This implementation is iterative, and not recursive like Merge Sort.

Choosing a correct min_run is important, which is the minimum length of a subarray. If min_run is too large, it will defeat the purpose of Combined Merge Sort since these subarrays will be sorted by Insertion Sort. On the other hand, a min_run value that is too small, will lead to too many runs being merged in the following steps (Volodymyr Korniichuk [2015]). Combined Merge Sort is however most effective when the number of runs are a power of two, due to the recursive slicing in half. It is common to choose a value of min_run between 32 and 64 (Santiago Valdarrama [2020]). In this paper we chose min_run = 32.

The average and worst-case time complexity is shown in equation (7). This case will happen when the arrays are large enough to make the benefit of Insertion Sort unnoticeable, and therefore the algorithm has the same worst-case time complexity as Merge Sort. When the array is short, about 128 or shorter, the algorithm will almost explicitly use Insertion Sort. The best-case is therefore on previously sorted arrays with these lengths, and the time complexity is shown in equation (8).

$$T(n) = O(n \cdot lgn) \tag{7}$$

$$T(n) = \Omega(n) \tag{8}$$

2.3.2 *Hybrid Quicksort*.

Hybrid Quicksort is a hybrid algorithm, where we implement the Insertion Sort algorithm to the Quicksort algorithm. It is known that the Quicksort algorithm works efficiently on arrays with large input sizes (Naina Gupta [2021]). The Insertion Sort algorithm, however, is more efficient on arrays with small inputs, as discussed in 2.3.1. We have therefore included this hybrid algorithm since its sorting is done by the most efficient algorithm for that particular array size. This algorithm uses recursion where either Hybrid Quicksort or Insertion Sort is called depending on the array size. Insertion Sort is called when the array size is smaller than the threshold. Partition, shown in listing 8, and Hybrid Quicksort, , shown in listing 9, are called when the array size is larger (Naina Gupta [2021]). The choice of threshold is therefore as important as it is choosing min_run in Combined Merge Sort in section 2.3.1. The threshold value can be calculated using the average number of comparisons in two partitions, but we have instead chosen 10 as a fixed threshold value in the algorithm, as seen in Naina Gupta [2021]. The time complexity for all cases is shown in equation (9).

$$T(n) = O(n^2) \tag{9}$$

Listing 6 Insertion Sort for use with Merge Sort and Hybrid Quicksort from (Santiago Valdarrama [2020])

```
INSERTION-SORT-FOR-COMBINED(A, left = 0, right = None)
1  if right is not otherwise defined
2      right = A.length - 1
3  for i = left to right
4      key = A[i]
5      j = i - 1
6      while j ≥ left and A[j] > key
7          A[j + 1] = A[j]
8          j = j - 1
9      A[j + 1] = key
```

Listing 7 Merge Sort with Insertion Sort (modified after Santiago Valdarrama [2020])

```
MERGE_INSERTION(A)
 1  min_run = 32
 2  n = A.length
 3  for i = [0 to n with min_run steps]
 4      insertions_sort_for_merge(A, i, min(i+min_run-1), n-1))
 5  size = min_run
 6  while size < n
 7      For i = [0 to n with (size *2) steps]
 8          midpoint = start + size - 1
 9          end = min((start + size * 2 - 1), (n - 1))
10          left = A(start to midpoint + 1)
11          right = A[midpoint + 1 to end +1]
12          merge(A, left, right, start)
13      size = size * 2
```

Listing 8 Partition algorithm from Cormen et al. [2009, Ch. 7.1], used in Hybrid Quicksort algorithm

```
PARTITION(A, p, r)
1  x = A[r//2]
2  j = p - 1
3  for i = p to r-1
4      if A[i] ≤ x
5          exchange A[i] with A[j]
6          j = j + 1
7  exchange A[i+1] with A[r]
```

Listing 9 Hybrid Quicksort algorithm from (Naina Gupta [2021])

```
HYBRID-QUICKSORT(A, p, r)
while p<r:
    if r-p + 1<10:
        insertion_sort_hybrid(A, p, r)
        break
    else :
        pivot = Partition(A, p, r)
        if pivot-p < r-pivot:
            Hybrid Quicksort(A, p, pivot-1)
            p = pivot + 1
        else :
            Hybrid Quicksort(A, pivot + 1, r)
            r = pivot-1
```

2.4 Built-in sorting functions

2.4.1 *Python Sort*.

Python's sort() is a built-in sorting function from Python, which uses a complex implementation of the Combined Merge Sort algorithm (Mailund [2021, Ch. 5]). A full implementation of the algorithm can be accessed through the source code of the Python programming language by Python Software Foundation [2021]. The time complexity is the same as Combined Merge Sort, with a worst and average case in equation (10), and best-case in equation (11).

$$T(n) = O(n \cdot lgn) \tag{10}$$

$$T(n) = \Omega(n) \tag{11}$$

2.4.2 *NumPy Sort*.

NumPy Sort is another built-in sorting function from Python, which uses Quicksort as the default sorting algorithm (Numpy contibutors [2021]). However, it is possible to select the sorting algorithm of your choosing. The other alternatives are Merge Sort, Heapsort and stable (Numpy contibutors [2021], but we used the default settings. The full implementation of the NumPy Sort algorithm is found from the source code of the Numpy fundamental package [2021] in Python. The worst-case and best-case running time is shown in equation (12) and (13), which are the same as the ones for Quicksort.

$$T(n) = O(n^2) \tag{12}$$

$$T(n) = \Omega(n \cdot lgn) \tag{13}$$

3 METHODS

3.1 Data Generation

The benchmark data used for testing was created using the NumPy random interface with seeding 12235. The data are arrays that are all exponents of 2, from 2^4 to 2^{13}. We used three different variations of data for the benchmark testing: arrays in random order, sorted arrays and sorted arrays in reversed order. For creating the sorted data we used Python's built-in sorted(), while for creating the reversed, we first sorted and then reversed with Python's reversed(). The data was generated in a function called generate_data and saved as an output-value.

3.2 Benchmark execution

To benchmark the sorting algorithms, we used the timer algorithm from listing 10. Timeit was used to take the time and we used 7 repetitions for each sorting algorithm and array length. All the benchmark data were saved in a data frame, and the data frame was saved to a pickle file. The pickle files were later read and used for plots and analysis. When analysing the result, we used the minimum value of all 7 executions. For plotting we used the common logarithmic value of the minimum time.

Listing 10 Timer for benchmark (Plesser [2021])

```
def timer(sort_function,test_data,list_type):
  clock=timeit.Timer(stmt='sort_func(copy(data))',
    globals={'sort_func':sort_function,
    'data': test_data,
    'copy': copy.copy})
  n,t=clock.autorange()
  run_list=clock.repeat(repeat=7,number=n)/n
  for t in run_list:
    stats = [sort_function,len(test_data),
            list_type, t]
    save_time(stats)
```

3.3 Software- and Hardware Specifications

The source code and data generated can be found here: https://gitlab.com/nmbu.no/emner/inf221/INF221_2021/student-term-papers_21/group1650/term_paper_2021/-/tree/main The git hashes to the relevant files given in figure 1.

all_algorithms.py	ee7d00d0
benchmark_testing.py	6adf8c65
benchmark_all.py	6e94125b
plots.py	b770005f
df_results.py	76cf4246

Figure 1: Hardware specifications

3.3.1 *Hardware*.

The hardware used for benchmark testing are shown in figure 2.

Processor	1,6 GHz two-core Intel Core i5
Memory	8GB
Graphic	Intel UHD Graphics 617 1536 MB

Figure 2: Hardware specifications

3.3.2 *Software*.

The software used for benchmark testing are shown in figure 3.

Python	3.8.5
NumPy	1.19.1
Pandas	1.1.3
Matplotlib	3.3.2

Figure 3: Software specifications

4 RESULTS

We observed that the built-in sorting functions are the most effective in all cases. For random and reversed sorted arrays the quadratic

algorithms are the least effective. However, for previously sorted arrays, Insertion Sort is faster than all the other algorithms, except for the built-in ones. Bubble Sort is still the slowest. For random data with size 8192 Bubble Sort is 64 948 times slower than the built-in Python Sort. The legend plot with the colours used for each algorithm is shown in figure 4 and the runtime for all the sorting algorithms are shown in figure 5.

Figure 4: The color codes for the different sorting algorithms

4.1 Quadratic algorithms

Figure 6 shows a comparison between the runtime results of Insertion Sort and Bubble Sort. We observe that they have the same time complexity except for when the data is sorted in advance. In this case Bubble Sort shows a steeper curve. Another observation is that Insertion Sort is the fastest of the two algorithms, regardless of the order of the data. We can especially see this when the data is sorted. When the data size is 8192 for example, we see that Bubble Sort is 4605 times slower than Insertion Sort.

4.1.1 *Insertion Sort*.

The runtime performance of Insertion Sort varies across the three different data orders. In figure 5 we also see that Insertion Sort is the third fastest of all the algorithms when the data is sorted in advance. Insertion Sort's fastest runtime is found when the data is sorted, while the slowest runtime is found when the data is randomized. On arrays of length 8192, the algorithm were 1.12 times faster on the reversed array, than it was on the random array.

4.1.2 *Bubble Sort*.

Figure 6 shows that Bubble Sort follows the same time complexity for all the order permutations. The fastest runtime is however found on the sorted data, while the slowest runtime is found on the random data. The runtime for reversed data were on average 2.04 times faster than the runtime for random data.

4.2 Sub-quadratic and combined algorithms

We compare Merge Sort, Quicksort, Combined Merge Sort and Hybrid Quicksort in figure 7. Quicksort is the fastest algorithm in all cases, except for sorted data with a length of 1024 or less. In this case, Combined Merge Sort is the fastest. On all data longer or equal to a size of 256, the least effective algorithm is Hybrid Quicksort. On random data smaller than 256, Merge Sort and Combined Merge Sort are the slowest, though close to Hybrid Quicksort. Merge Sort

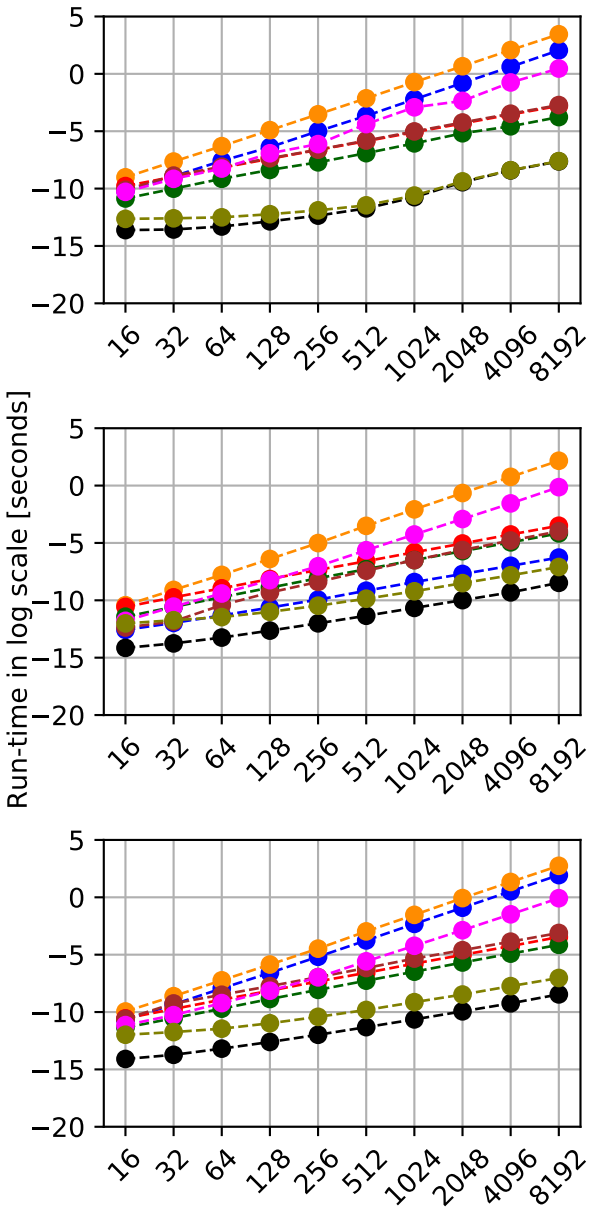

Figure 5: The runtime compared between all the algorithms. The x-axis is the lengths of the data, and the y-axis is the time in log scale seconds. The first subplot is random data (top), the second sorted data (middle), the third reversed data (bottom).

is the slowest on sorted data, while Combined Merge Sort is the slowest on reversed data.

4.2.1 *Merge Sort*.

The runtime for Merge Sort on sorted or reversed arrays were almost the same on all data sizes. The runtime for random data were nearly twice as long in all instances. On data with size 8192 the runtime for random arrays were 2.02 times slower than the sorted array of same size.

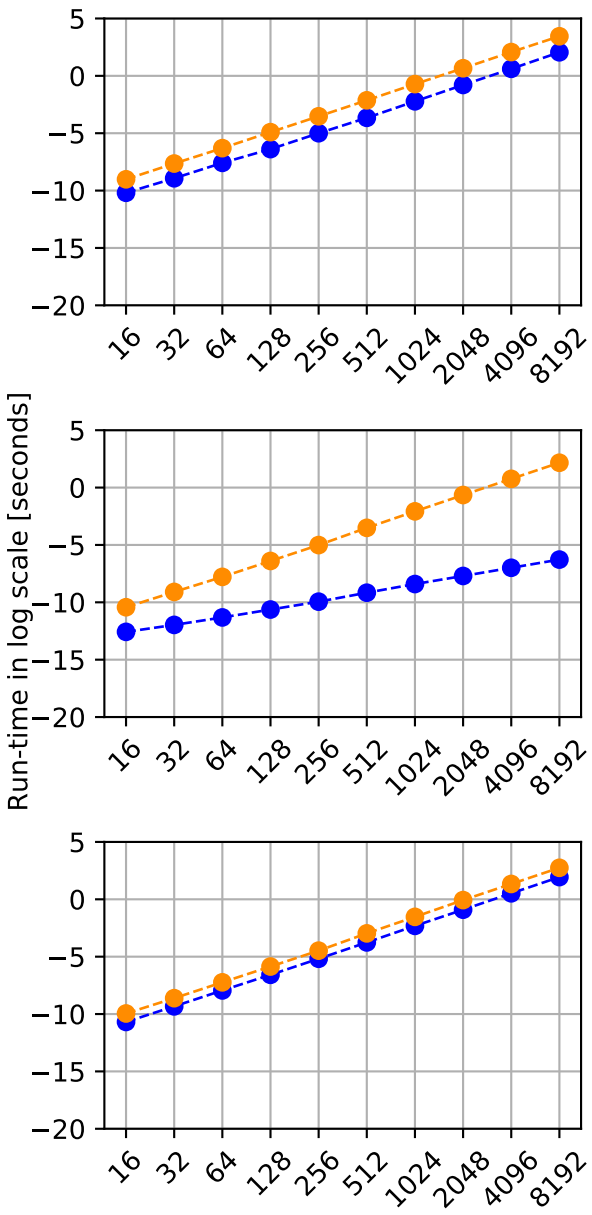

Figure 6: The runtime compared between the quadratic algorithms. The x-axis is the lengths of the data, and the y-axis is the time in log scale seconds. The first subplot is random data (top), the second sorted data (middle), the third reversed data (bottom).

4.2.2 *Quicksort*.

Quicksort has almost the exact same runtime for sorted and reversed data. The runtime for random arrays were for all instances about 1.5 times slower. On the array of length 8192 the runtime was exactly 1.49 times slower on a random array, than it was on a sorted array.

4.2.3 *Combined Merge Sort*.

Combined Merge Sort is fastest on a sorted array and slowest on random array, for all instances. On sorted arrays the algorithm is especially fast on arrays with a length of 128 or less. On reversed arrays the algorithm is slow on these lengths compared to the rest of the runtime on reversed arrays. On arrays of length 8192 the runtime was 3.54 times slower on a random array, than on a sorted array. Combined Merge Sort is always faster than Merge Sort, except for on reversed data smaller than 256.

4.2.4 *Hybrid Quicksort*.

For Hybrid Quicksort, sorted data is always the fastest, but for arrays with lengths of 128 or longer, the runtime for reversed data is similar to the runtime of sorted data. For data with size 1024, the algorithm were 1,04 times slower on reversed data, than on sorted data. On arrays of length 8192 the runtime was 1.72 times slower on a random array, than on a sorted array. Hybrid Quicksort is always slower than Quicksort, except for sorted data of size 16.

4.3 Built-in sorting functions

In figure 8 we compare running time between the built-in sorting algorithm in Python, and the built-in sorting algorithm in the Python library NumPy. We see that the Python Sorting algorithm is the fastest of the two with both sorted data and reversed data. It is also the fastest algorithm for random data, up to where the length of the array reaches about 1024. There, the different built-in sorting functions have almost the exact same runtime.

4.3.1 *Python Sort*.

The Python Sorting algorithm works best with sorted and reversed arrays, and it uses slightly longer time on random arrays. On average, about twice as long time. With sorted and reversed arrays, the Python Sort function uses approximately the same amount of time.

4.3.2 *NumPy Sort*.

The NumPy Sorting algorithm uses longer time than the Python algorithm to sort sorted and reversed arrays. NumPy Sort uses almost the same amount of time to sort these arrays, but it uses shorter time to sort an array with random data. The runtime for random data is on average 1.8 times as fast for random data than the runtime for reversed data.

5 DISCUSSION

5.1 Quadratic sort

In theory, the best-case for both Insertion Sort and Bubble Sort arises when the order of the data is sorted, while the worst-case occurs when the data order is reversed. The two algorithms have the same time complexity for all cases except for best-case, where insertion sort's time complexity is of a lower growth order. We observed that Bubble Sort only showed a different growth from Insertion sort when the data was sorted. Here Insertion Sort showed a less steep graph than all the other cases from quadratic algorithms. Theoretically, this makes sense since this is when the best-case for Insertion Sort occurs. This is coincidentally the best time-complexity out of all the other algorithms.

The performance of Insertion Sort mostly follows the theory, as the algorithm had its fastest runtime when it used sorted data. The slowest runtime was however recorded on random data. On random data the probability of best-case is greater, which does not

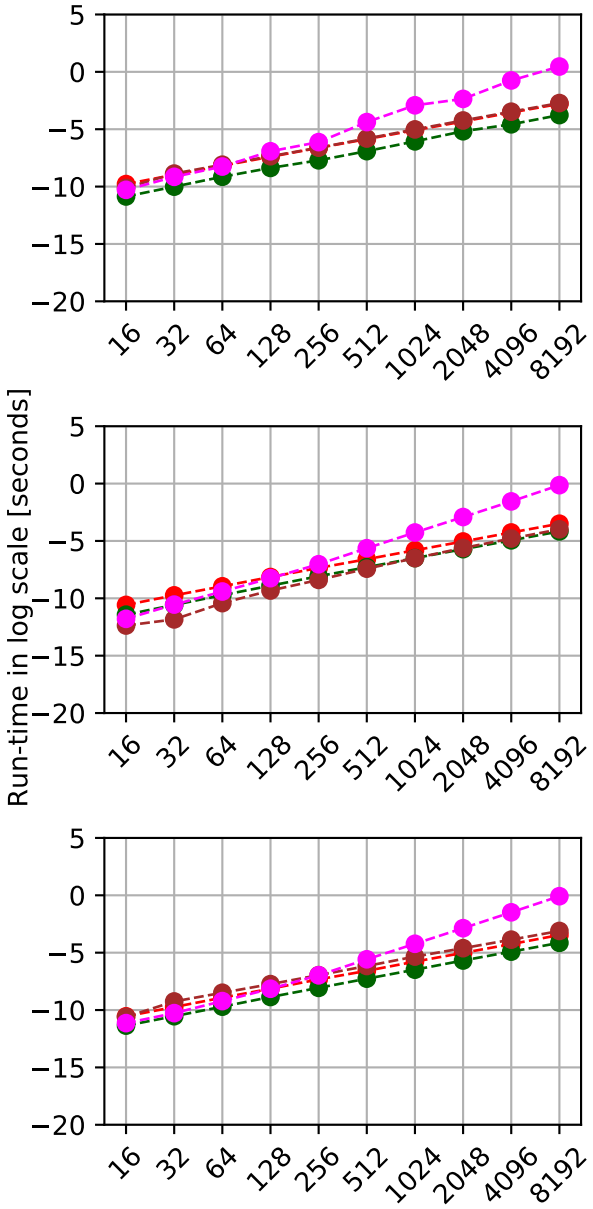

Figure 7: The runtime compared between the sub-quadratic algorithms. The x-axis is the lengths of the data, and the y-axis is the time in log scale seconds. The first subplot is random data (top), the second sorted data (middle), the third reversed data (bottom).

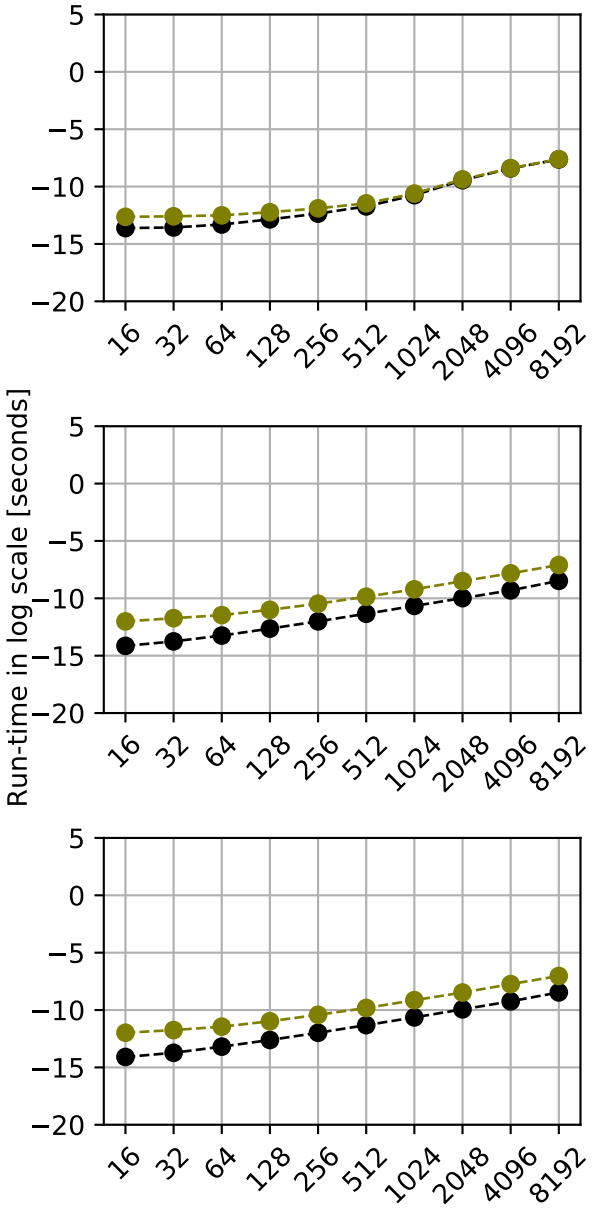

Figure 8: The runtime compared between the built-in algorithms. The x-axis is the lengths of the data, and the y-axis is the time in log scale seconds. The first subplot is random data (top), the second sorted data (middle), the third reversed data (bottom).

fit the theory that the algorithm works slightly slower on random data, than it does on reversed data.

Bubble Sort recorded runtime follows the same time complexity for all the data order types, which follows the theory since Bubble Sort theoretically has the same time complexity for all cases. Bubble Sort's best-case scenario is supposed to occur when the data is previously sorted, which coincides with the results. However, the performance did not support the theory concerning the worst-case. Similar to Insertion Sort, Bubble Sort's runtime was slowest on randomized data and not on reversed data, as the theory suggests it should be.

The unexpected results from the random data in Bubble Sort and Insertion Sort should be looked into further if one would resume the work.

5.2 Sub-quadratic sort and combined sort

Merge Sort, Quicksort, Combined Merge Sort and Hybrid Quicksort coincides with the theory, when looked at on its own.

Merge Sort has the same time complexity in all cases and the runtime is almost the same in all three cases. On random data Merge Sort takes twice as long to sort, but this difference is not significant since the runtime is fast. The worst-case for Merge Sort is also most likely to occur on random data.

Quicksort's theory suggests that it is most effective on sorted arrays and arrays sorted in reverse, and the worst-case is most likely to happen on a random array. Since Quicksort is slightly slower on a random list, the theory and results add up.

Combined Merge Sort is a version of Merge Sort made to be more effective on smaller arrays, than what Merge Sort is. The purpose is that Combined Merge Sort should be more effective than Merge Sort. The results proves that Combined Merge Sort is faster, except for on reversed data smaller than 256. When the data is this small, most of the sorting is done by Insertion Sort. Since Insertion Sort works poorly on reversed data and best on sorted data, we see the effects of that in Combined Merge Sort. The cause of the algorithm being especially fast on sorted data with a size of 128 or smaller, is also because most of the sorting is done by insertion sort, which is one of the most effective algorithms on sorted data. The reason for it using Insertion Sort on 128 is due to the choice of min_run. When min_run is 32 and the array size is 128, Combined Merge Sort will only slice the array into 4 subarrays, leading to only 3 merge-operations.

Hybrid Quicksort is a version of Quicksort made to be more effective on smaller arrays than what Quicksort is. The time complexity for all order permutations is the same, which coincides with the runtime for all cases being similar. The runtime is slightly slower on random data, but due to the choice of pivot in the theory section (2), Quicksort, and therefore also Hybrid Quicksort, works worse on random data.

One would assume that Hybrid Quicksort is a more effective algorithm than Quicksort. However, Hybrid Quicksort is always slower than Quicksort, except for sorted data of size 16. The reason for Hybrid Quicksort being more effective in this case, is due to the almost exclusive use of insertion sort, which is effective on small, sorted data. Why Hybrid Quicksort is slower than expected is a question we need to investigate. A possible explanation could be the choice of threshold in listing 9 line 2. We have chosen threshold to be 10, but a better implementation would be to calculate the threshold using the input data, instead of using a constant value.

5.3 Built-in sort

The Python Sorting algorithm is a hybrid sorting algorithm that uses a complex version of Combined Merge Sort and as we saw in section 4.3, the runtime is effective. It is least effective on random arrays, similarly to Combined Merge Sort. Python Sort has the same runtime for sorted arrays and reversed arrays, which also happens on our implementation of Combined Merge Sort.

The NumPy Sorting algorithm uses Quicksort to sort arrays. NumPy Sort is most effective on random arrays, meaning the default settings of the algorithm chooses the last element as pivot, unlike our implementation of Quicksort, which chooses the middle element.

In our results Quicksort is faster than Combined Merge Sort. Since Python Sort uses Combined Merge Sort and NumPy Sort uses Quicksort, NumPy Sort should be more effective than Python Sort. This is not the case. This leads us to believe that Python Sort indeed uses a complex, more effective version of Combined Merge Sort, as stated in the source code. We also suspect that the built-in algorithms are written in an optimized way, possibly in a different, more efficient language. This would explain why Python Sort and NumPy Sort is more efficient than Combined Merge Sort and Quicksort, respectfully.

5.4 Summary

In conclusion, we have discovered when it is the most efficient to utilize the different sorting algorithms. When we need to sort data, it is the most efficient to use the built-in algorithms, specifically Python Sort. Bubble Sort is always the slowest and therefore there are no cases we are familiar with where this algorithm is needed. If one does not have the opportunity to utilize the built-in algorithms, one should use Quicksort for random and reversed data, and Insertion Sort for sorted data. However, there will be instances where the data permutation order will be unknown. In those cases, we would recommend always using Quicksort.

For further work we would look more into why the quadratic algorithms' worst-case was not as we expected. We would start by looking at our implementation. We would also investigate how the built-in sorting algorithms were written, as this might change our conclusion regarding which algorithm to choose.

6 ACKNOWLEDGEMENTS

We would like to thank Bao Ngoc Huynh for giving us helpful feedback on the previous drafts. In addition, we would like to express our gratitude to Fadi Al Machot for the guidance through the project.

REFERENCES

Baeldung author. 2021. When Will the Worst Case of Merge Sort Occur? https://www.baeldung.com/cs/merge-sort-time-complexity [Online; accessed 11-oktober-2021].

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.

GeeksforGeeks contributors. 2021a. Bubble Sort. https://www.geeksforgeeks.org/bubble-sort/ [Online; accessed 11-november-2021].

GeeksforGeeks contributors. 2021b. Merge sort. https://www.geeksforgeeks.org/merge-sort/#_=_ [Online; accessed 18-oktober-2021].

Thomas Mailund. 2021. *Introduction to Computational Thinking.* Apress, 1 New York Plaza, New York, NY 10004, U.S.A.

Naina Gupta. 2021. Advanced Quick Sort (Hybrid Algorithm). https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/ [Online; accessed 18-oktober-2021].

Numpy contibutors. 2021. numpy.sort. https://numpy.org/doc/stable/reference/generated/numpy.sort.html [Online; accessed 06-november-2021].

Numpy fundamental package. 2021. numpy source code. https://github.com/numpy/numpy/blob/v1.21.0/numpy/core/fromnumeric.py#L846-L999 [Online; accessed 28-november-2021].

Hans Ekkehard Plesser. 2021. Benchmarking sorting algorithms in Python. *Data Science Section, Faculty of Science and Technology, Norwegian University of Life Sciences* 1, 1 (2021), 1–4.

Python Software Foundation. 2021. Cpython Source Code. https://github.com/python/cpython/blob/main/Objects/listobject.c [Online; accessed 07-november-2021].

Santiago Valdarrama. 2020. Sorting Algorithms in Python. https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python [Online; accessed 18-oktober-2020].

Volodymyr Korniichuk. 2015. TIMSORT SORTING ALGORITHM. https://www.infopulse.com/blog/timsort-sorting-algorithm/ [Online; accessed 06-november-2021].