Benchmarking of various sorting algorithms

INF221 Term Paper, Group 3, NMBU, Autumn 2021

Jisoo Park
jisoo.park@nmbu.no

Jorge Eduardo Hermoso Valle
jorge.eduardo.hermoso.valle@nmbu.no

Aria (Ian) McKenney
ian.mckenney@nmbu.no

ABSTRACT

In this paper, we analyse the performance of several sorting algorithms. We examine the quadratic algorithms, insertion sort and bubble sort, the sub-quadratic equations, merge sort and quick sort, and Python and NumPy's built in sorting functions. We also examine the performance of merge sort and quick sort when insertion sort is used instead for small amounts of data. Algorithms are provided as pseudo code in this paper and were implemented in Python. Our results show that overall Python's **`sorted()`** algorithm was fastest, and `numpy.sort` second fastest.

1 INTRODUCTION

This paper analyzes the performance of eight sorting algorithms, insertion sort, bubble sort, merge sort, quick sort, a hybrid of merge sort and insertion sort, a hybrid of quick sort and insertion sort, and Python's **`sorted()`** function, and NumPy's `sort()` function. We have generated performance data on our selected sorting algorithms using non-specialized computers so that we can see if our theoretical expectations of our algorithms' performance matches data taken from non-specialized machines. The next section will discuss the theory behind the sorting algorithms and following sections will discuss our benchmarking implementation and results respectively.

2 THEORY

In this section we define our sorting algorithms with pseudocode and examine the time complexity that we expect from our algorithms. First we define the terms which we use to explain our algorithms:

- *Time complexity* - The theoretical expectation of how long an algorithm will take to run. Expressed in terms of the largest factor impacting the time taken. e.g. if we have an algorithm that performs $n + 3$ actions we would say it has a time complexity of $O(n)$. The 3 is dropped as for large n it will not have a negligible impact on how long the algorithm will take.
- *Running time* - Same as *Time complexity*.
- *Runtime* - The actual amount of time an algorithm takes expressed in seconds.
- *Worst-case (running time)* - The running time of an algorithm with the maximal number of actions the algorithm will have to perform. Expressed as O.
- *Average-case (running time)* - The running time of an algorithm with the average number of actions the algorithm will have to perform. Expressed as Θ.
- *Best-case (running time)* - The running time of an algorithm with the minimal number of actions the algorithm will have to perform. Expressed as Ω

2.1 Insertion sort

Insertion sort is "an efficient algorithm for sorting small number of elements" Cormen et al. [2009, Pg. 17]. In insertion sort, each element is compared with every prior element in order to find the location where it should be inserted in our list. For an arbitrary element e if it is larger than the prior element no change is made. If e is smaller than the prior element, e is compared against each prior element until it reaches the index, i, that has an element that is smaller than e. The index of e is set to $i + 1$ and the index of all elements between $i + 1$ and e are incremented by 1 as well.

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      $A[i + 1] = key$
```

Pseudocode for insertion sort is shown in Listing 1. The best case running time for this algorithm is:

$$T(n) = \Omega(n) \ . \tag{1}$$

This best case occurs when we use the insertion sort algorithm on an list that is already sorted. The algorithm iterates over each of the n elements in our list and makes no changes. The algorithm performs n number of $O(1)$ operations which yields a best case of $\Omega(n)$.

The average case and worst cases occur when we use the insertion sort algorithm on ran unsorted list. The time complexity of these cases can be expressed as $an^2 + bn + c$ where a, b, and c are constants Cormen et al. [2009, Pg. 27,28]. Thus the worst and average case running time is:

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

2.2 Bubble sort

Bubble sort works by iterating over the elements in a list in pairs, swapping the elements based on whichever is greater. This process is repeated until every pair has been checked or swapped.

Pseudocode for bubble sort is provided in Listing 2. The average or worst case scenario for bubble sort is list that has some number of elements that are out of order. In an average or worst case scenario the inner loop of bubble sort makes $\frac{n}{2}$ swaps and is performed n

times, giving us an $O(n^2)$ complexity. In a best case scenario where we perform bubble sort on an already sorted list, the inner loop makes $\frac{n}{2}$ checks and is performed n times also giving us an $O(n^2)$ complexity. Thus the time complexity of bubble sort is:

Listing 2 Bubble sort algorithm from Cormen et al. [2009, Ch. 2.2].

```
Bubble-Sort($A$)
for $i = 1$ to $A.length - 1$
    for $j = A.length$ down to $i + 1$
        if $A[j] < A[j-1]$
            swap $A[j] <-> A[j-1]$
```

$$T(n) = \Omega(n^2) = \Theta(n^2) = O(n^2)\ . \qquad (3)$$

It is possible to modify bubble sort to keep track of whether or not the algorithm has made any swaps when iterating over the $\frac{n}{2}$ pairs of elements. If no swaps are made over the whole loop, the algorithm terminates. With this modification the best-case time complexity for bubble sort becomes $\Omega(n)$.

2.3 Merge sort

Merge sort is a recursive algorithm that follows a divide and conquer approach. It divides a list equally into two sub-lists, then sorts the sub-lists using recursive calls, and finally merges the sorted sub-lists into one. Thus, the actual sorting occurs when the two sub-lists are merged together. Pseudo code for merge sort is provided in Listing 3.

The amount of merges the algorithm makes follows a lgn running time and the algorithm makes n comparisons of elements for each merge. Therefore merge sort has a time complexity of $O(nlgn)$. Because the merge sort algorithm does not change its behavior based on the list that is input, $O(nlgn)$ is the time complexity of its best, worst, and average case. Hence the time complexity of merge sort is:

$$T(n) = \Omega(nlgn) = \Theta(nlgn) = O(nlgn)\ . \qquad (4)$$

2.4 Quick sort

Quick sort algorithm is another recursive algorithm with a divide-and-conquer approach like merge sort. Pseudo code for quick sort is provided in Listing 4. The most significant factor in the divide-and-conquer approach for quick sort occurs in the divide steps.

$$T(n) = \Omega(nlgn) = \Theta(nlgn) = O(n^2)\ . \qquad (5)$$

When the list is sorted or almost sorted and the pivot value selected is the smallest or largest value, the time complexity of quick sort has as $O(n^2)$ due to an unbalanced partition. However, in the average case, it has a $O(nlgn)$ time complexity on n elements. This depends on whether the pivot is in the first, the last, or middle of a list. To prevent poor performance from quick sort, a randomized pivot selection could be used.

Listing 3 Merge sort algorithm from Cormen et al. [2009, Ch. 2.3.1].

```
Merge($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
Let $L[1 : n_1 + 1]$ and $R[1 : n_2 + 1]$
for $i = 1$ to $n_1$
    $L[i] = A[p + 1 - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else
        $A[k] = R[j]$
        $j = j + 1$

Merge-Sort($A, p, r$)
if $p < r$
    $q = floor(p + r/2)$
    Merge-Sort($A, p, q$)
    Merge-Sort($A, q + 1, r$)
    Merge($A, p, q, r$)
```

Listing 4 Quick sort algorithm adapted from Cormen et al. [2009, Ch. 7.1].

```
Quick-Sort($A, p = None, r = None$)
if $p < r$
    if $p$ is $None$
        $p = 0$
    if $r$ is $None$
        $r = A.length - 1$

    $q = PARTITION(p, r)$
    $QUICKSORT(A, p, q - 1)$
    $QUICKSORT(A, q, r)$

PARTITION($A, p, r$)
$pivot = A[\lfloor (p + r)/2 \rfloor]$
while $p \leq r$
    while $A[p] < pivot$
        $p = p + 1$
    while $A[r] > pivot$
        $r = r - 1$
    if $p \leq r$
        exchange $A[p]$ with $A[r]$
        $p = p + 1$
        $r = r - 1$
return $p$
```

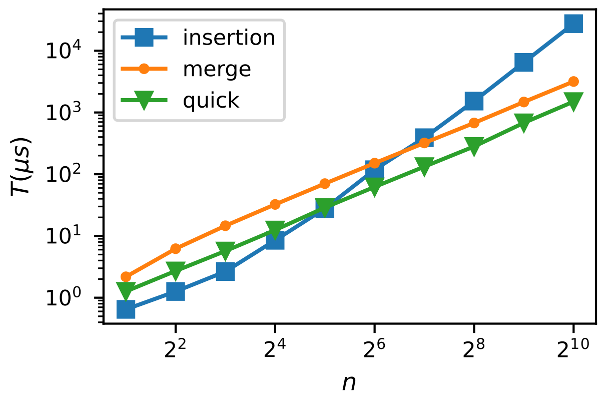

Figure 1: Benchmark results for insertion sort, merge sort, and quick sort on random lists.

2.5 Hybrid Merge-Insertion sort

In our analysis of our algorithms benchmarks in Figure 1 we discovered that for lists with less than 2^6 elements, insertion sort is faster than merge sort. We have implemented a branching algorithm that will perform insertion sort for lists that have less than 2^6 elements and merge sort otherwise. The time complexity of our hybrid merge sort algorithm is analogous to the time complexity for regular merge sort as for any n over 2^6 the merge sort algorithm will be used. Hence the time complexity is:

$$T(n) = \Omega(nlgn) = \Theta(nlgn) = O(nlgn) \ . \tag{6}$$

2.6 Hybrid Quick-Insertion sort

From Figure 1 we see that our quick sort algorithm, also shows faster performance than insertion sort for lists with more than 2^4 elements. This is because pivots, once decided, are removed on the next operation and reduce unnecessary moving elements, exchange with a long distance of elements through the partition. Furthermore, insertion sort shows good performance when the list is small. We have implemented a hybrid algorithm that applies insertion sort if a list has less than 2^4 elements and quick sort otherwise. The worst case time complexity of our hybrid algorithm is still $O(n^2)$ as both insertion sort and quick sort have a worst case time complexity of $O(n^2)$. Both insertion sort and quick sort have a worst case time complexity of $O(n^2)$, so the worst case time complexity of our hybrid quick sort algorithm is:

$$T(n) = \Omega(nlgn) = \Theta(nlgn) = O(n^2) \ . \tag{7}$$

2.7 Python's sorted() function

Python has a built-in function called sort; this sort function uses a Timsort algorithm. [Foundation 2021] Python's Timsort implementation has "a running time of $O(n + n \lg \rho$ where ρ is the number of runs (i.e. maximal monotonic sequences)" [Auger et al. 2018]. The pseudocode for Timsort is shown in Listing 5:

Listing 5 Timsort algorithm from Auger et al. [2018, pg.3].

```
TIMSORT($S$)
    Input: A sequence $S$ to sort.
    Result: The sequence S is sorted
    into a single run, which remains on the stack
    Note: The function merge_force_collapse repeatedly
    pops the last two runs on the stack R, merges them
    and pushes the resulting run back on the stack

runs $\leftarrow$ a run decomposition of $S$
$R \leftarrow$ an empty stack
while runs $\neq \emptyset$
    remove a run $r$ from runs and push $r$ onto $R$
    merge_collapse($R$)
if height($R$) $\neq 1$
    merge_force_collapse($R$)
```

```
merge_collapse
    Input: A stack of runs $R$
    Result: The invariant of $r_{i+2} > r_{i+1} + r_i$
    and $r_{i+1} > r_i$ is established.
    Note: The runs on the stack are denoted by
    $R[1] \ldots R[\text{height}(R)]$, from top to bottom. The length of run
    $R[i]$ is denoted by $r_i$. The blue highlight indicates that the
    condition was not present in the original version of TimSort.

while height($R$) > 1 do
    $n \leftarrow$ height($R$) $- 2$
    if $(n > 0$ and $r_3 \leq r_2 + r_1)$ or $(n > 1$ and $r_4 \leq r_3 + r_2)$ then
        if $r_3 < r_1$ then
            merge runs $R[2]$ and $R[3]$ on the stack
        else merge runs $R[1]$ and $R[2]$ on the stack
    elseif $r_2 \leq r_1$ then
        merge runs $R[1]$ and $R[2]$ on the stack
else break
```

2.8 Numpy's sort() function

The Numpy sort function uses an algorithm called introsort as a default setting [Community 2021]. Introsort is a hybrid algorithm that uses a quick sort algorithm until a maximum recursion depth is reached, then switches to heapsort. Additionally, if introsort is used on a list with a suitably low amount of elements it will use insertion sort instead of quick sort or heapsort [Musser 1997]. By switching to heapsort when the maximum recursion threshold is reached, introsort improves the worst case time complexity of quick sort from $O(n^2)$ to $O(nlgn)$.

3 METHODS

3.1 Test data generation

Data were generated using NumPy's np.random.uniform function with a preset seed so that the randomization is reproducible between executions. Python lists were generated using increasing powers of 2, i.e. $2^1, 2^2, 2^3, \cdots, 2^{10}$. We have an upper bound in place of 2^{10} to avoid stack overflow errors from our recursive algorithms.

The data is then added to a pandas DataFrame and pickled to a results file so that the data can be reused later.

3.2 Algorithm testing

The algorithms that we implemented were tested using a test suite utilizing Python's `unittest` library. Each algorithm is performed on the list, `[3, 2, 4, 1, 5]` and must output `[1, 2, 3, 4, 5]`. Additionally each algorithm is performed on a randomly generated list and must output the same result as `numpy.sort()`.

3.3 Benchmarking

To benchmark our data we used Python's timeit module, which performs a preset number of repetitions to ensure we get accurate measurements of our algorithms runtimes. The timeit autorange function repeats the execution of a function until a certain amount of time has passed, in our case at least 0.2 seconds must pass. It then returns the number of executions, n_E, of our algorithm required to take at least 0.2 seconds. We then run our algorithm $7n_e$ times to ensure that we are getting an accurate view of our data. From these executions we use the minimum runtime of our timeit executions divided by n_E to calculate the runtime of our algorithm. In the last line of this snippet we used the min(t) instead of mean(t) because usually higher values in the results vector are not caused by variability in Python's speed, but other background processes interfering with the time measurement. So in this case the min() function gives a lower bound for how fast the machine can execute the sorting algorithm. Code of our benchmarking system can be found in Listing 6.

Listing 6 Draft benchmark setup.

```
import timeit
import copy

rng = np.random.default_rng(seed=42)
data = np.random.uniform(size = n)

clock = timeit.Timer(stmt='algo(copy(data))',
            globals={
                'algo': insertion,
                'data': data,
                'copy': copy.copy,
        })

n_ar, t_ar = clock.autorange()
t = clock.repeat(repeat=7, number=n_ar)
result = min(t) / n_ar
```

3.4 Hardware and software

The Table 1 and Table 2 describes our hardware and software. We use Python 3.8 version in Intel I7-8550U.

Table 1: Software used for this paper

Software	Version
Jupyter	1.0.0
Jupyter Lab	3.1.7
NumPy	1.20.3
Pandas	1.3.2
Python	3.8

Table 2: Hardware used for this paper

Computer	CPU Model	Clock speed	Number of Cores	Number of Logical Processors	RAM
Dell XPS 13 (2018)	Intel i7-8550U	1.80 GHz	4 Core(s)	8	16 GB

Table 3: Files used for this report; From GitLab repository [Group3 2021]

File	Git hash
`benchmarking.py`	`e9e1beae`
`data_generation.ipynb`	`6ecd5c53`
`plot.py`	`6ecd5c53`
`results.xz`	`75c40f2d`
`sorting.py`	`48bac3b3`
`testing.py`	`886f1a31`

3.5 Git hashes

Below Table 3 contains the list of Git hashes that used in this paper. These include the algorithms implementation, data generating and its benchmarking with the plots.

4 RESULTS

Here we present the results from benchmarking tests in the form of tables and figures.

4.1 Benchmarking results of each list; random, sorted, and reversed lists

The results obtained from random, sorted, and reversed lists are shown in Tables 4, 5, and 6 respectively. Additionally Figures 2, 3, and 4 plot the figures from these tables.

These tables and figures show how one algorithm can perform better than another depending on the size of the list. For example, insertion sort does better for smaller lists of sizes up to 2^4 and merge sort is quicker for sizes bigger than that.

Note that for clarity we have opted to not include our hybrid merge sort and quick sort algorithms in our figures as they are effectively a duplicate of insertion sort when dealing with list sizes below our threshold and a duplicate of quick or merge sort when above the threshold.

Table 4: Benchmarking results for random data

	List size									
Algorithm	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Bubble Sort	0.78	1.85	4.65	15.35	52.46	210.68	747.78	2,945.83	12,878.49	56,733.84
Insertion Sort	0.65	1.26	2.66	8.51	27.79	118.18	390.00	1,529.36	6,479.71	27,446.27
Merge Sort	2.18	6.21	14.62	32.42	70.37	151.18	320.88	676.20	1,478.40	3,172.87
Hybrid Merge Sort	0.80	1.38	2.80	8.63	27.90	150.70	321.88	678.36	1,476.14	3,179.09
NumPy's sort()	3.02	3.13	3.32	3.66	4.29	5.58	8.31	13.49	24.12	45.31
Python's sorted()	0.33	0.35	0.41	0.58	0.99	1.84	3.61	7.24	14.73	35.15
Quick Sort	1.26	2.71	5.70	12.47	28.79	62.26	132.66	281.56	679.20	1,499.04
Hybrid Quick Sort	0.76	1.38	2.81	8.66	28.84	62.57	133.21	283.58	680.33	1,487.28

Table 5: Benchmarking results for sorted data

	List size									
Algorithm	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Bubble Sort	0.80	1.56	3.73	10.74	35.65	128.00	485.91	1,882.66	7,994.74	34,460.39
Insertion Sort	0.66	0.91	1.35	2.24	4.01	7.53	14.53	28.53	63.51	133.95
Merge Sort	2.24	6.30	14.85	33.66	73.02	157.39	335.91	709.69	1,562.21	3,362.33
Hybrid Merge Sort	0.78	1.03	1.50	2.40	4.17	157.76	338.06	712.88	1,560.24	3,384.45
NumPy's sort()	2.99	3.05	3.21	3.44	3.96	4.82	6.76	10.46	17.76	32.64
Python's sorted()	0.34	0.38	0.45	0.57	0.79	1.31	2.21	3.99	7.38	14.96
Quick Sort	1.31	2.74	5.58	11.91	25.26	53.17	112.93	240.40	540.33	1,172.02
Hybrid Quick Sort	0.78	1.03	1.49	2.39	25.34	53.33	112.70	241.03	537.95	1,165.02

Table 6: Benchmarking results for reverse data

	List size									
Algorithm	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Bubble Sort	0.91	2.15	6.41	22.01	81.86	314.63	1,233.78	4,900.91	20,644.14	87,950.90
Insertion Sort	0.74	1.45	4.11	14.20	53.68	209.60	828.09	3,297.49	13,677.73	57,617.48
Merge Sort	2.25	6.31	15.00	33.56	73.54	157.56	335.50	712.27	1,553.05	3,345.84
Hybrid Merge Sort	0.87	1.60	4.23	14.36	53.58	158.57	339.40	713.24	1,561.41	3,365.87
NumPy's sort()	2.98	3.06	3.24	3.50	3.93	4.95	6.98	10.85	18.71	34.59
Python's sorted()	0.34	0.38	0.44	0.57	0.81	1.30	2.22	3.96	7.51	15.05
Quick Sort	1.24	2.80	5.89	12.75	26.98	57.33	120.73	256.18	573.68	1,228.14
Hybrid Quick Sort	0.87	1.61	4.23	14.36	27.14	57.47	120.35	256.72	568.88	1,235.73

4.2 Best and worst case scenario per algorithm

We analyzed our algorithms by comparing the runtime of each algorithm's performance against random, sorted, and reverse sorted lists as shown in Figure 5 to Figure 10.

Figure 5 shows our results for insertion sort. The figure depicts similar runtime growth for random and reversed lists, and shows that insertion sort performs best on already sorted data. Insertion sort has the best time complexity of $\Omega(n)$ and average time complexity of $O(n^2)$. We can see that the random and reversed lists grow at a similar rate as n^2 when the length of a list is over 2^5.

Figure 5 shows the runtime growth of Insertion Sort. The best case for this algorithm occurs when used on a sorted list and is the second-fastest overall, after Python's **`sorted()`** function, for small lists up to a size of 2^4 as shown in Table 5. In the worst case for the insertion sort algorithm, reverse lists, it is still the fastest of the algorithms that we implemented for lists up to a size of 2^3.

Figure 6 shows the runtime growth of Bubble Sort, the best case for this algorithm occurs for already sorted data and the worst case when the data is given in reverse order. Bubble sort performance for smaller lists up to 2^3 is average in comparison with other algorithms. Bubble sort is faster than Quick Sort and Merge Sort algorithms for smaller lists but when the size of the list grows larger the performance goes down. Bubble sort is the absolute worst algorithm when reaching a list size of 2^6 regardless of the pre-sorting of the list.

Figure 7 shows the runtime growth of Merge Sort. The 3 cases for this algorithm are very similar, fastest was against random data and the worst case is tight between sorted and reverse data. For example given a list of size 2^{10} it's running times are 1478.40μs, 1562.21μs and 1553.05μs for random, sorted and reverse respectively as shown in Tables 4, 5 and 6. Merge sort is the slowest algorithm for small list sizes up to 2^4. For larger lists this algorithm runs faster than Insertion and Bubble Sort on random and reverse sorted lists. However on sorted data, merge sort is slower than insertion sort as sorted data is insertion sort's best case scenario.

Figure 8 shows the runtime growth of our quick sort algorithm. The random, sorted, and reverse sorted results for a list of size 2^{10} are 1,499.04μs, 1,172.02μs, and 1,228.14μs respectively. Our quick sort algorithm was the fastest among algorithms that we implemented, but was still slower than both the Numpy sort and the Python sort.

Figure 9 shows the runtime growth of the Numpy `sort` function. The random, sorted, and reverse sorted results for a list of size

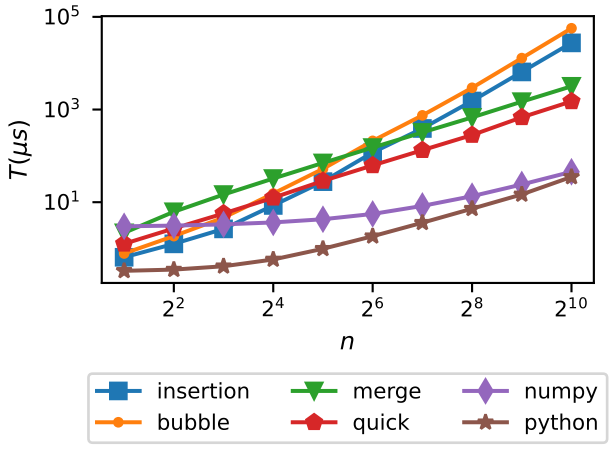

Figure 2: Benchmark results when inputs are random lists.

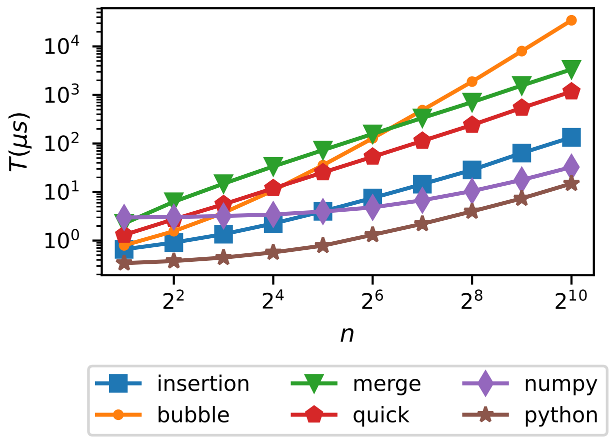

Figure 3: Benchmark results when inputs are lists that are already sorted.

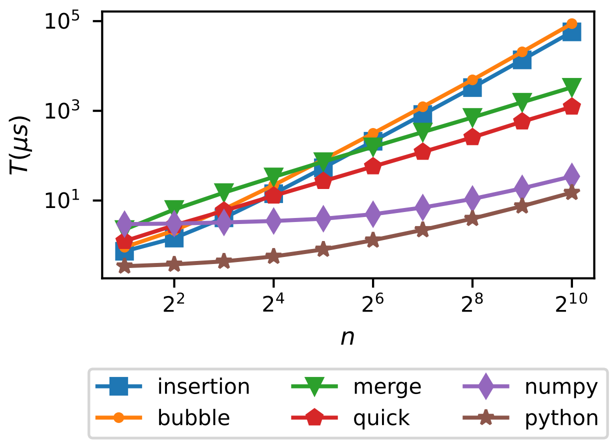

Figure 4: Benchmark results when inputs are lists sorted in reverse order.

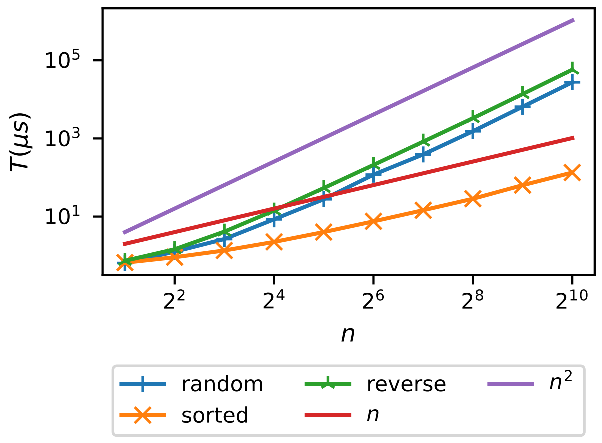

Figure 5: Benchmark results for Insertion Sort

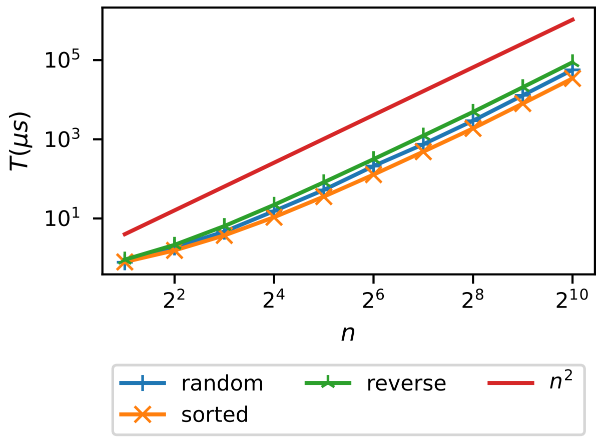

Figure 6: Benchmark results for Bubble Sort

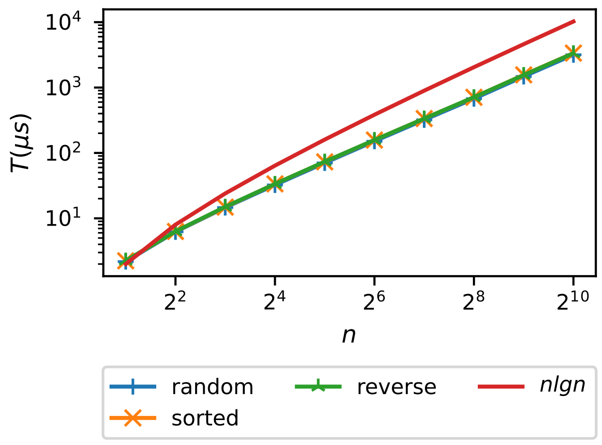

Figure 7: Benchmark results for Merge Sort

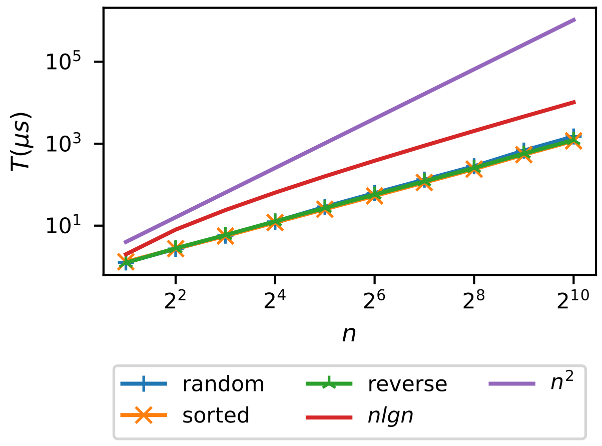

Figure 8: Benchmark results for Quick Sort

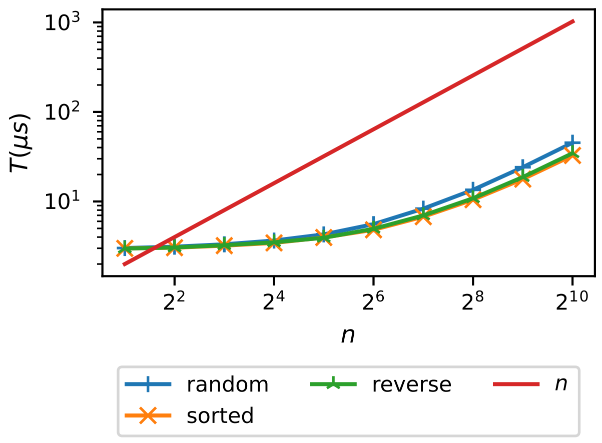

Figure 9: Benchmark results for NumPy's sort() function

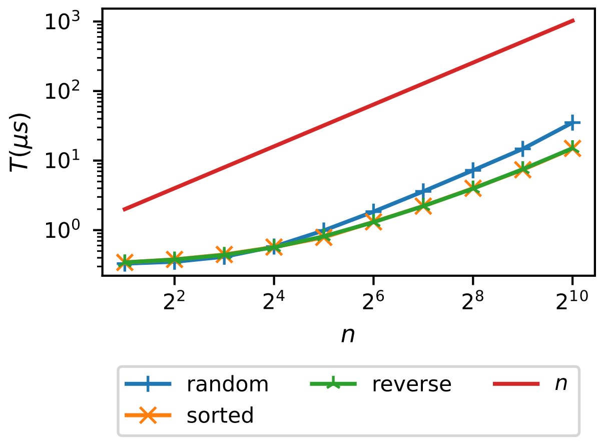

Figure 10: Benchmark results for Python's sorted() function

2^{10} are 45.31μs, 32.64μs, and 34.59μs respectively. The best case for `numpy.sort()` is against sorted lists, whereas the worst case is against random lists. The `numpy.sort()` algorithm was the overall second fastest algorithm that we examined, though was slower than the other algorithms for list sizes less than or equal to 2^2.

Figure 10 shows the runtime growth of the Python **`sorted`**`()` which utilizes a TimSort algorithm. We expect the TimSort algorithm to be faster on already sorted and reverse sorted data as it has optimizations in place for both cases. Our results match this expectation as the times for the **`sorted`**`()` function are 35.15μs, 14.96μs, and 15.05μs for random, sorted and reverse sorted data respectively. The Python **`sorted`**`()` function was the overall fastest algorithm of the sorting algorithms examined in this paper.

5 DISCUSSION

Our results are consistent with the theory stating our quadratic algorithms, insertion sort and bubble sort, are faster than our merge sort or quick sort algorithms for sorting smaller lists. However since the time complexity of quadratic algorithms grows quickly in proportion to the list size, we see that after a size of 2^5 the runtime of our quadratic algorithms dramatically increases.

Our results also show that our sub-quadratic algorithms, merge sort, quick sort, hybrid merge sort, and hybrid quick sort, of the order *nlgn* are faster than quadratic algorithms when the size of the list grows larger. Our merge sort results were similar for random, sorted, and reversed lists. Those results are expected as our merge sort algorithm performs the same operations regardless of whether or not the list is sorted. For our quick sort implementation, we see that sorted and reverse sorted data were actually faster than random data. This is because our algorithm chooses the pivot element based on the midpoint of the list. Because our lists were randomly generated using `numpy.random.uniform`, choosing the midpoint of a sorted or reverse sorted list makes it more likely that we select the optimal pivot element that is the median of the list. For truly random data, the pivot selection was more likely to be unoptimal, resulting in slower runtimes which we see reflected in the results.

The Python **`sorted`**`()` and `numpy.sort` algorithms take the best existing sorting algorithms and adapt their behavior to optimize their performance for different list sizes. These functions also have optimizations in a lower-level programming language which is why they are the most effective for these tasks. For `numpy.sort()`, we would expect the Numpy introsort algorithm to be faster on sorted and reverse sorted data as the function has optimizations that speed up the algorithm in those cases. Our results match this expectation, both sorted and reverse data were faster than random data for `numpy.sort()`. We see a similar trend in the results from Python's **`sorted`**`()` function for similar reasons as `numpy.sort()`. The **`sorted`**`()` function has optimizations to help speed it up when dealing with sorted or reversed data.

ACKNOWLEDGMENTS

We are grateful to each of our team members for their contribution and dedication to this paper, as well to our teachers Fadi al Machot, and the TA's Bao Ngoc Huynh, Jon-Mikkel Ryen Korsvik and Filip Kristoff Ørn.

REFERENCES

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. 2018. On the Worst-Case Complexity of TimSort. In *26th Annual European Symposium on Algorithms (ESA 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 112)*, Yossi Azar, Hannah Bast, and Grzegorz Herman (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:13. https://doi.org/10.4230/LIPIcs.ESA.2018.4

The NumPy Community. 2021. numpy.sort. https://numpy.org/doc/stable/reference/generated/numpy.sort.html#numpy-sort Last updated on Jun 22, 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.

Python Software Foundation. 2021. Python.sort.built-in. https://docs.python.org/3/howto/sorting.html#sortinghowto version 3.10.

INF221 Group3. 2021. Gitlab Repository. https://gitlab.com/nmbu.no/emner/inf221/INF221_2021/student-term-papers_21/group3/inf221-term-paper-group3 NMBU H2021.

David R. Musser. 1997. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* 27 (1997), 983–993.