Benchmarking sorting algorithms in Python

INF221 Term Paper - Team 29, NMBU, Autumn 2020

Nablis Ogubamichael Gebrehiwet
nablis.ogubamichael.gebrehiwet@nmbu.no

Jørgen Rekdal Mangelrød
jorgen.rekdal.mangelrod@nmbu.no

ABSTRACT

In this paper, we analyse and perform benchmarks of six sorting algorithms implemented in Python. The algorithms have been compared to their theoretical expectations. The purpose of this paper is to determine which of the sorting algorithms performs the best on a randomly generated, sorted data set. The benchmarks are compared against the expected worst case run time of each individual algorithm.Our benchmarks have been performed on random integer arrays. These arrays vary in size from 10^1 elements to 10^5 elements. One ordering of data has been tested, randomly ordered arrays sorted by the sorting algorithms. The algorithms tested are Insertion sort, Bubble sort, Merge sort, Quick sort, Python sort and Numpy sort. Jupyter Notebook running Python version 3.7.4 is the software that has been used for the benchmarks. Python sort and Numpy sort are the two algorithms that perform the best in our benchmarks. This is due to them being hybrid sorting algorithms. Bubble sort and Insertion sort grow exponentially as the array size increase, thus following their worst case theoretical run time of $\Theta(n^2)$. Merge Sort and Quick sort are also very close in their performance, but testing on lager array sizes shows that Merge sort will be slightly faster for larger array sizes.

1 INTRODUCTION

A good method to analyze the performance of algorithms is benchmarking. To evaluate the software and hardware quality of a computer, benchmarking is used in both the industry and in research. Software can be compared by testing the various algorithms on the same hardware, and hardware can be tested by using the same algorithm on various hardware. Sorting algorithms takes specific elements and place them into an ordered sequence. By timing how long each algorithm takes to sort an array of a certain size, one can evaluate the algorithm against their best case run time, worst case run time and each other.

In this paper the following algorithms have been tested: Insertion sort, Bubble sort, Merge sort, Quick sort, Python sort and Numpy sort. To analyze the different algorithms arrays of different sizes have been randomly generated, sorted and timed. In the benchmarks random integer arrays of different sizes have been randomly generated through Numpy. The goal of this paper is to determine which algorithm is the fastest for sorting arrays of a given size and comparing it to the theoretical run time expectation of the algorithms. Plots will be provided for each algorithm to present the results.

The theory section provides a brief introduction to each algorithm, pseudocode for the algorithm and their best/worst case run times. In the third section the methods used to perform the benchmarks are presented, how they were plotted, the software used and the hardware used for the analysis. The results are then presented before what conclusions can be drawn based on the data presented. This section is followed by our acknowledgements and references at the end of the paper.

2 THEORY

Section 2 provides a description of each of the algorithms used in our benchmarks. Pseudo-codes have been provided for each algorithm. Each algorithm has the expected run time presented. A table has also been added that provides an overview of the expected run times of all the algorithms.

2.1 Bubble Sort

Bubble sorting algorithm is one of the simplest and easily understood algorithm by programmers. It is a straight forward algorithm, where the algorithm iterates through an array or list by comparing the value at the present and the value at the next index. If the value at the index is greater than the value at the next index, then it swaps the values and repeats this step until all elements are sorted in the desired order. By the desired order, it means that it can be either descending or ascending order[Kumar [n.d.]].
Even though it is short, easy to understand and is preferable due to its simplicity, it can be quite slow [Cormen et al. 2009, Ch. 2]. It is usable only for small sets of data. If the data that is going to be sorted is too large, then the drawback of this algorithm quite noticeable. Bubble sort algorithm has a time complexity of $\Theta(n^2)$.This can be a highly inefficient and time consuming algorithm to be used in real life scenarios. Pseudo-code for Bubble sort can be seen in Listing 1

Listing 1 Bubble Sort Algorithm from [Cormen et al. 2009, Ch. 2.3]

```
BUBBLE-SORT(A)
1  For i from to A − 1
2      For j from to A − 1
3          IF A[j] > A[j + 1]
4              Swap A[j]←→A[j+1]
```

- Let us see how bubble sort works

Table 1: Bubble sort

A	[1,4,3,2]			
A	1	4	3	2
A	1	3	4	2
A	1	2	3	4

2.2 Insertion Sort

Like bubble sort, insertion sort is also one of the simplest algorithms to write and understand. Compared to bubble sort, it can be more efficient. This sorting algorithm chunks lists into sorted and unsorted element then compares the rest of the unsorted elements one by one with the sorted ones[Sodhi et al. 2013]. A good example of insertion sort will be playing off a deck of cards. Insertion sort can be faster than Bubble sort for small sizes of data. This being said,insertion sort is inefficient for large sets of data as it has$\Theta(n^2)$ a as worst case time complexly [Cormen et al. 2009, Ch. 3]. The best case time complexity for Insertion sort is $\Theta(n)$. Pseudo-code for the first algorithm is shown in Listing 2.

Listing 2 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
```

Let us see how Insertion sort works:

Table 2: Insertion sort

A	[8,5,6,3]			
A	8	5	6	3
A	5	8	6	3
A	5	6	8	3
A	3	5	6	8

2.3 Merge sort

Merge sort is one of the most popular and effective sorting algorithms. Merge sort is widely used to sort a very big and complex data sets. It is also known as a divide and conquer algorithm where it divides the given array in to sub arrays, and then sorts all the following sub arrays [Sedgewick 1978, Ch 2.2]. After the sub arrays are sorted, it merges all of the sub arrays in to a single array or a list. Even though it's quicker when compared with both insertion and bubble sort, it also has its drawbacks. Merge sort can be quite efficient for large set of data. When it comes to smaller data sets, the divide and conquer method can be quite punishing. This will only cost more time and create a higher complexity. Merge sort has $\Theta(n \log n)$ as it's worst case run-time [Cormen et al. 2009, Ch. 3]. Listing 3 shows the pseudo-code for the Merge sort algorithm.

Listing 3 Merge Sort Algorithm from [Cormen et al. 2009, Ch. 2.3.1]

```
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)
```

```
1  MERGE (A, p, q, r)
2  n1 = q − p + 1
3  n2 = r − q
4  let L(1..n1 + 1] and R[1..n2 + 1] be new arrays
5  for i = 1 to n1
6      L[i) = A[p + i − 1]
7  for j = 1 to n2
8      R[j] = A[q + j
9  L[n1 + 1] = ∞
10 R[n2 + 1] = ∞
11 i = 1
12 j = 1
13 for k = p to r
14     if L[i] ≤ R[j]
15         A[k] = L[i]
16         i = i + 1
17     else A[k] = R[j]
18             j = j + 1
```

2.4 Quick sort

Like Merge sort, the Quick sort is a popular and efficient algorithm. It also uses the divide and conquer method of sorting arrays. It takes one element from the array and makes it a pivot point. Then the algorithm partitions the rest of the elements around the pivot point. The elements are placed into sub divided arrays, depending if they are greater or smaller than the pivot point where they will be sorted effectively [Wikipedia [n.d.]b]. With time complexity of $\Theta(n \log n)$, the Quick sort algorithm is the fastest of all algorithms with the best expected performance. It can be twice as fast as Merge sort if the algorithm is implemented correctly. Unlike merge sort, it does not require extra memory. The down side of this algorithm is that it's not stable. In Listing 4, pseudo-code for Quick sort can be found.

Listing 4 Quick Sort Algorithm from [Cormen et al. 2009, Ch. 7.1]

```
QUICK-SORT(A, p, r)
if p < r
    q =PARTITION(A, P, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤x
        i = i + 1
        exchange A[i] with A[j]
    exchange A[i + 1] with A[j]
    return i + 1
```

2.5 Merge sort switching to insertion sort for small data

Depending on the size of the data, one has to decide which type of algorithm should use. All sorting algorithms can used to sort data, but when it comes to small size of data, one has to choose either bubble or insertion sort. Insertion sort is quicker and has best performance than any algorithm when sorting small size data. Even though Merge sort is effective, in Big - O notation, we can conclude that a run time of $\Theta(n^2)$ goes faster than $\Theta(n \log n)$ in small size data.

Table 3: comparison in Run time complexity

	Best case	Average case	worst case
Bubble	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quick	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Python	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Numpy		$\Theta(n \log n)$	$\Theta(n \log n)$

2.6 Python 'sort()'

Python has a built in sorting algorithm called Timsort. This algorithm was created by Tim Peters in 2002 with the intention to be used as the standard Python sorting algorithm[Wikipedia [n.d.]c]. Timsort is a hybrid sorting algorithm, combining the best of both merge sort and insertion sort [Heumann 2018]. For arrays with a size of 64 elements or less, insertion sort is used. For arrays larger than 64 elements the algorithm looks for sub sequences of data that are already in order (runs) and uses these runs to sort the rest of the elements in the array. The array is divided into the runs and then merged two at a time. When all the data is run through, one sorted run remains. Timsort for looks elements in the array that are already in order. In real world data sets, these sequences usually exist and Timsort calls these sequences for natural runs [Wikipedia [n.d.]c]. Timsort has a worst case run time complexity of $\Theta(n \log n)$. Pseudo-code for Python Sort (Timsort) can be found in Listing 5.

Listing 5 Python_sorted(Timsort) algorithm from [GeeksforGeeks 2020]

```
MINMERGE = 32

INSERTIONSORT(A, left, right)
for i in range(left + 1, right + 1)
    temp = A[i]
    j = i − 1
    while A[j] > temp and j ≥left
        A[j + 1] = A[j]
        j− = 1
    A[j + 1] = temp
L[i] = A[p + i − 1]

MERGE(A, 1, m, r)
len1, len2 = m − l + 1, r − m
left, right = [], []
for i in range(0, len1)
    left.append(A[l + i])
for i in range(0, len2)
    right.append(A[m + 1 + i]
i, j, k = 0, 0, l
while i < len1and j < len2
    if left[i] ≤right[j]
        A[k] = left[i]
        i+ = 1
else
        A[k] = right[j]
        j+ = 1
    k+ = 1
while i < len1
    A[k] = left[i]
    k+ = 1
    i+ = 1
while j < len2
    A[k] = right[j]
    k+ = 1
    j+ = 1

TIMSORT(A, n)
for i in range(0, n, MINMERGE)
    INSERTIONSORT(A, i, min((i + 31), (n − 1))
size = RUN
while size < n
    for left in range(0, n, 2 ∗ size)
        mid = left + size − 1
        right = min((left + 2 ∗ size − 1), (n − 1))
        MERGE(A, left, mid, right)
    size = 2 ∗ size
```

2.7 Numpy 'sort()'

Numpy sort returns us a sorted copy of a given array. As of version 1.12.0 This function has introsort as its default sorting algorithm[NumPy 2020]. Introsort is a hybrid sorting algorithm that starts with quicksort, but switches to heapsort when the recursion depth exceeds a certain threshold [Wikipedia [n.d.]a]. By doing

this, Numpy sort combines the best of both algorithms. This results in a performance similar to quicksort on standard data sets and a worst case $\Theta(n \log n)$ run time when heapsort is used. Since the algorithms used are comparison sorts, Introsort is also a comparison sort [Wikipedia [n.d.]a]. Listing 2 describes the pseudo-code for Numpy Sort (Introsort).

Listing 6 Introsort algorithm from [?]

```
SORT(A)
1  maxdepth = ⌊log2(A.length)⌋ · 2
2  introsort(A, maxdepth)
INTROSORT(A, maxdepth)
1  n = length(A)
2  if n ≤ 1
3       return // base case
4  elseif maxdepth = 0
5       heapsort(A)
6  else
7       p = partition(A) // assume this function does
   // pivot selection, p is the final position of the pivot
8       introsort(A[0 : p − 1], maxdepth − 1)
9       introsort (A[p + 1 : n], maxdepth − 1)
```

3 METHODS

The following section includes an explanation to how the test data was generated. Explanations to the execution of the benchmarks and plotting the results are also provided. The hardware and software used has also been explained.

3.1 Test Data

The test data has been generated randomly through Numpy. The creation of the array has been put into a nested loop, where the array size increases each time the function passes through the loop. The size of the arrays vary depending on which algorithms are being tested. For insertion sort and bubble sort the maximum array size has been set to 10^4. Maximum array size for the for the theoretically quicker algorithms have been set to 10^5. To plot the results of the benchmarks, Matplotlib has been used.

3.2 Benchmarks

Listing 7 Benchmark setup.

```
for sort in sorts:
    times = list()
    start_all = time.clock()
    for i in range(1, 5):
        measurements=list()

        for k in range(1,repetitions):
            random_array =
            np.random.randint(1000, size=10**i)
            start = time.clock()
            sort["sort"](random_array)
            end = time.clock()
            measurements.append(end-start)
            print(sort["name"], "Sort", 10**i,
            "Elements in", end - start,
              "seconds")

        times.append(average(measurements))


    end_all = time.clock()
    print(sort["name"], "Sorted Elements in",
    end_all - start_all, "seconds")
```

The benchmarks have been executed in two different files. One file contains the benchmarks for all the algorithms for a given array size. The second file contains the benchmark for only the quickest algorithms, where the array size has been increased. To time the benchmarks the timeit.Timer function in python was used. Included in the Timer function were the array sizes, the sorting algorithm, the number of repeats and the number of executions. All the algorithms have been repeated 10 times and executed once. The results were then stored in pandas data frames and the mean run time of the benchmarks have been plotted to evaluate the performance of each algorithm. These data frames are presented in the results section.

3.3 System

The benchmarks have been executed on a Lenovo Legion T530 desktop computer with 8gb RAM. It runs a Windows 10 operating system version 1909. The computer has a CPU speed of 3.89 GHz, along with 1.5 MB L2 cache and 9.0 MB L3 cache.

Table 4: Software used.

Software	Version
Python	3.7.6
pandas	1.0.1
numpy	1.0.1
matplotlib	3.1.3

Table 5: Versions of files used for this report; GitLab repository https://gitlab.com/nmbu.no/emner/inf221/h2020/student-term-papers/team_29/inf221-term-paper-team29.

File	Git hash
`Benchmark_sorted_large.ipynb`	`3667f88`
`benchmarks_all.ipynb`	`3667f88`

4 RESULTS

Results of the benchmarks are summarized in this section through tables and plots.

4.1 Benchmark Rankings

The result of the different sorting algorithms shows from the fastest to the slowest are as follows:
1. Numpy and Python sort
2. Merge and Quick sort
3. Insertion sort
4. Bubble sort

There are huge differences when comparing time complexity of different algorithms. From the results that we collected, we can see that out of all the algorithms, Numpy sort and Python sort are the fastest sorting algorithms with the best performance.This is because they are derived from a hybrid sorting algorithms. For example, the python sort uses both merge and insertion sort which makes it preferable when dealing with both small and big size data. On the other side, we find that both bubble and insertion sort are the slowest sorting algorithms of them all. One has to make sure not to use either bubble or insertion sort when dealing with a huge sets of data. The result also shows that when sorting small sizes of data, both Bubble and Insertion sort beats Merge sort by quite a substantial amount. This is due to the overhead of recursion function call.

Table 6: Average time taken to sort a random set of data in seconds. This includes the smallest arrays.

	Different size of lists		
	10	100	1000
Bubble	3.29778e-05	0.0022974	0.246711
Insertion	3.1922e-05	0.00092	0.94667
Merge	3.32889e-05	0.000355	0.00500
Quick	3.31111e-05	0.00027	0.00447
Numpy	1.92333e-05	6.41444e-05	8.58333e-05
Python	7.85556e-06	1.89556e-05	0.00010

Table 7: Average time taken to sort a random set of data in seconds. This includes the largest arrays.

	10000	100000
Bubble	0.24671	24.2176
Insertion	0.09466	9.66965
Merge	0.06754	0.0860119
Quick	0.081645	0.078104
Numpy	0.000384	0.0005299
Python	0.00378	0.0005230

4.2 Benchmark of sorted data for all algorithms with smaller arrays

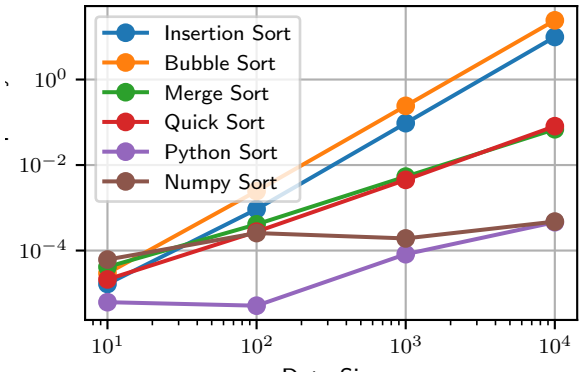

Figure 1: Benchmark results for all the sorting algorithms with array sizes from 10 - 10 000. The x-axis displays the array size and the y-axis displays the run time.

Figure 1 shows us that insertion sort and bubble sort are the slowest algorithms for the sorted data for a given array size. Based on the slope of the lines for insertion sort and bubble sort in the graph, it will make little sense to use these algorithms for large data sets. The two algorithms perform when the array size is sufficiently small. Merge sort and Quick sort follow each other for the various array sizes. The same seems to be the case for Numpy Sort and Python Sort. Based on the slope of the lines of these algorithms it seems that they will also be the fastest for large data sets.

4.3 Benchmark of sorted data for the fastest algorithms with larger arrays

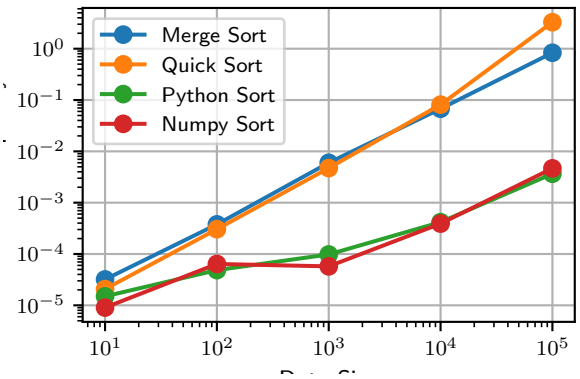

Figure 2: Benchmark results for all the sorting algorithms with array sizes from 10 - 100 000. The x-axis displays the array size and the y-axis displays the run time.

In Figure 2 the benchmarks for the quickest algorithms are plotted. Based on the slope of Quick sort, we can see that when the array size increases, this will be the slowest algorithm of the four. This contradicts the expected behavior of Quick Sort being faster than Merge sort when the array size increases. Python Sort and Numpy Sort are still close to each other, with little to no differences between the two algorithms in regards to run time. As stated above, it was correct that these two algorithms were the fastest when the data size was increased. Based on the slope of the two algorithms in the figure, it looks like they will continue to be the fastest algorithms when the data size increases.

5 DISCUSSION

In this section, you should summarize your results and compare them to expectations from theory presented in Sec. 2.

5.1 Comparison of results to worst case run time

5.1.1 Bubble Sort. We can see from the figure above that the slope for Bubble sort growths exponentially which means that it follows the theoretical theory for its worst case of $\Theta(n^2)$.

5.1.2 Insertion Sort. The same as bubble sort, the slope for insertion sort growths also rapidly by following the theoretical theory for its worst case of $\Theta(n^2)$.

5.1.3 Merge Sort. Theoretically, Merge sort has $\Theta(n \log n)$ at its best and worst case. Based on the two figures, we can see it follows its theoretical expected run time.

5.1.4 Quick Sort. Even though quick sort has same time complexity as merge sort, for the worst case, it has the same as bubble and insertion sort time complexity. A smart choice for the pivot could make the algorithm faster and in figure above we see following the theoretical theory of run time.

5.1.5 Python Sort. From the figure above we can see that python sort's slope decreases down words. With the worst time complexity of $\Theta(n \log n)$ it follows the theoretically theory of the run time

5.1.6 Numpy Sort. The same as python sort, the slope bends down indicating that it follows its theoretically theory.

5.2 Standard Deviations

Standard deviations of the benchmarks have been evaluated during the bench-marking process. These standard deviations are so small that they could be regarded in this case. None of the algorithms showed any sign of large standard deviations when performing the benchmarks. If a large standard deviation should occur, one can run the benchmark again to see if it was an ongoing process interfering with the benchmark or correlated to the algorithm it self.

5.3 Improvents to the benchmark

Several improvements can be made to the benchmark. Firstly, we have only used one datatype. We could improve accuracy of the benchmark by using random data, sorted data and reverse sorted data. Doing this will increase the time required to perform the benchmarks, but it should still be manageable by the computer. The second thing that can be done is to increase the amount of executions performed in the benchmarks. The time required to complete the benchmarks will increase here as well. If we wanted to increase the amount of executions while keeping the time the same, a stronger computer would be needed. We could also increase the array sizes, but decided to set the maximum number of elements in an array to 10^5. When trying to increase the array size further than 10^5 elements in the arrays, Quick sort started to struggled with sorting the arrays. This is also the reason for 10^4 being the largest arrays for Bubble sort and Merge sort. When the array sizes were increased passed this, both algorithms started to struggle. On the other side, the array sizes can also be decreased into very small arrays, to look at the performance variations here. This has been deemed unnecessary as we expect the differences not to vary as much for arrays of a very small size. Another thing one can do to reduce the standard deviations in the benchmarks is to reduce the amount on ongoing, demanding processes while performing the benchmarks.

ACKNOWLEDGMENTS

We would like to thank our Professor Hans Ekkhard Plesser for always answering our questions quickly whenever we had any issues. We would also like to thank Bao Ngoc Huynh for the feedback on our drafts and guiding us how to write a scientific paper. This has been a new experience for the both of us.

REFERENCES

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA.

GeeksforGeeks. 2020. Timsort. https://www.geeksforgeeks.org/timsort/ Visited on 28.11.2020.

Angela Heumann. 2018. The magic behind the sort algorithm in Python. https://medium.com/ub-women-data-scholars/the-magic-behind-the-sort-algorithm-in-python-1cb9515294b5 Visited on 26.11.2020.

Manish Kumar. [n.d.]. ENHANCING PERFORMANCE OF BUBBLE SORT ALGORITHM. ([n. d.]).

NumPy. 2020. numpy.sort. https://numpy.org/doc/stable/reference/generated/numpy.sort.html Visited on 27.11.2020.

R. Sedgewick. 1978. Implementing Quicksort programs. *Commun. ACM* 21 (1978), 847–857.

Tarundeep Singh Sodhi, Surmeet Kaur, and Snehdeep Kaur. 2013. Enhanced insertion sort algorithm. *International journal of Computer applications* 64, 21 (2013).

Wikipedia. [n.d.]a. Introsort. https://en.wikipedia.org/wiki/Introsort Visited on 27.11.2020.

Wikipedia. [n.d.]b. Quicksort. https://en.wikipedia.org/wiki/Quicksort Visited on 27.11.2020.

Wikipedia. [n.d.]c. Timsort. https://en.wikipedia.org/wiki/Timsort Visited on 28.11.2020.