Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2020

Anders Mølmen Høst and Muhammad Fahad Ijaz
Data Science Section, Faculty of Science and Technology, Norwegian University of Life Sciences
anders.molmen.host@nmbu.no,muhammad.fahad.ijaz@nmbu.no

ABSTRACT

ABSTRACT TEXT

1 INTRODUCTION

The purpose of this paper is to perform benchmarks on various sorting algorithms. Benchmarking is to be done on test data of different size and structure to analyse the behaviors of the algorithms in each case.Those differences are to be visualised graphically and the results are analysed. The results are important to find out the real life behavior of the algorithm as compared to their theoretically expected behaviour.The following sorting algorithms were analysed based on the psuedocode presented in the course.

- Quadratic algorithms
  - Insertion Sort
  - Bubble Sort
- Sub-quadratic algorithms
  - Merge Sort
  - Quick Sort
- Combined Algorithm
  - Merge Sort switching to insertion sort for small data
- Built-in sorting functions
  - Python 'sort()'
  - NumPy 'sort()'

2 THEORY

2.1 Quadratic algorithms

2.1.1 insertion sort. Insertion sort uses in incremental approach to sorting. It builds the sorted array by taking only one item at a time. It starts by comparing the first two elements of the array. If they are already sorted, it will move onto the next set. Otherwise, it will swap them. Then it will compare the second element with the third one and swap them if necessary. If swapped, it'll then compare the new second element with the first and swap them if required. In the next step, it'll compare the the third and the fourth elements and repeat the same process for each element until all the sorted. Insertion sort will give best performance on an array which is already sorted. The worst case would be an array which is sorted in reverse order. Insertion sort is an in-place sorting algorithm.

The best case expected run time of the algorithm has an upper bound of $O(n)$. The average case and worst case both have a run time of $O(n^2)$.

2.1.2 Bubble sort. Bubble sort sorts in the following way. It starts with the last positioned element of the array. Then compares the last element with the second last element. If the last element is smaller, the elements swap positions in the array. Continuing with the second last element. This element is then compared to the third last element and the process is repeated until all adjacent elements are compared. Now the smallest element of the array are placed in the first position. Then during the next round we start with the last element again and comparing adjacent elements and stop when the second element is reached. Now we have the two smallest numbers in their correct positions. The process continues until all elements are sorted.

The run time of the algorithm has an upper bound of $O(n^2)$ in the worst case. Bubble sort will need to swap all elements while iterating through the inner loop. Then after each iteration through the outer loop the remaining elements to sort will only go down by one. In the best case the run time is $O(n)$ since the algorithm then only iterates through the list one time and will not make any swaps. However the average case is $O(n^2)$. Because of the double for-loops the algorithm will need to compare and possible swap elements, sorting one element at each iteration. The algorithm sorts in place as Insertion sort, so that the elements are not copied in a new array requiring additional storage space.

Listing 1: Bubble sort algorithm from Cormen et al. [2009, Ch. 2. p 40].

```
BUBBLE-SORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 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 uses the divide-and-conquer approach which is a recursive approach to sorting. The approach is to divide the problem into several sub problems and then solve them recursively. Then it combines the results of the sub problems into the solution for the original problem. Cormen et al. [2009, Ch. 2.3] That approach is used by merge-sort by dividing the given array recursively until there is only one element in each of the sub arrays. Then it merges the sub arrays in a sorted order. So, merge sort needs additional space to perform the sorting operation,i.e. it is not an in-place sorting algorithm, opposed to Insertion sort.

[Cormen et al. 2009]

Listing 2: Merge sort algorithm from Cormen et al. [2009, Ch. 2.3. p 31].

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Listing 3: Merge algorithm from Cormen et al. [2009, Ch. 2.3. p 34].

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17               j = j + 1
```

2.2.2 Quick sort. quick sort like Merge sort applies the divide-and-conquer paradiqm. quick sort sorts by first choosing a pivot element. Then through the partition algorithm in Listing 5 it compares the remaining elements in the array to the pivot element. Elements are then divided into two sub-arrays with the elements less than the pivot placed in a sub-arrays to the left of the pivot, and elements greater placed in a sub-array to the right. The two sub-arrays are then themselves sorted by recursive calls to quick sort, Listing 4. When the pivot element partitions sub-arrays consisting of at most one element, all sub-arrays are combined, and the combined array consisting of all elements is then sorted. [Cormen et al. 2009, p. 170-172]

The worst-case run time of quick sort is $O(n^2)$. This is when sorting an already sorted array. In this case only one element will be partitioned and thus sorted for each recursive call. The average and best case run time is $O(nlogn)$. In the average case one considers that the pivot element splits a combination of balanced splits and unbalanced splits. [Cormen et al. 2009, p. 174-176]

Listing 4: quick sort algorithm from Cormen et al. [2009, Ch. 7.1. p 171].

```
QUICK SORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICK SORT(A, p, q − 1)
4      QUICK SORT(A, q + 1, r)
```

Listing 5: Partition algorithm from Cormen et al. [2009, Ch. 7.1. p 171].

```
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]withA[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

2.3 Combined algorithm and built-in sorting functions

The combined algorithm uses insertion sort when sorting data with a size less than a specified threshold. Then it switches to merge sort if the size of the data is at the threshold or above. Python uses timsort for sorting. Timsort is a hybrid sorting derived from merge sort and insertion sort. The algorithm uses subesets of data that are already sorted and uses these to sort the rest. Best case performance of timsort has an upper bound of $O(n)$ whereas average and worst case performance have upper bounds of $O(nlogn)$. [Wik [n.d.]] NumPy implements four different algorithms for sorting, consisting of quick sort, heap sort, merge sort and timsort. By default, NumPy uses a hybrid sort called introsort. This algorithm uses quick sort by default but is able to switch to heap sort if the algorithm does not make enough progress. ["NumPy" 2020]

Listing 6: Cobined algorithm

```
COMBINED-SORT(A, p, t)
1  if A.length < t
2      INSERTION SORT(A)
3  else MERGE-SORT(A, p, A.length)
```

3 METHODS

3.1 Data Generation

The test data used for the purpose of this paper was of base 10 with increasing in power from 1 to 6. So

Random Generator from NumPy was used to generate the datasets for benchmarking. The seed used for generating the dataset was 12. The datatype of the generated data was float64. Three variations of the data were used for running the benchmark. For sorted data, Python built-in 'sorted' funciton was used. For reverse sorted data, 'reversed' was used after sorting the data using 'sorted' on the randomly generated data using the random generator.

3.2 Benchmark Execution

'timeit' module from Python was used to measure the time for each sorting run of the funciton.

3.3 Specifications - Software & Hardware

The benchmark was executed in Jupyter Notebook with Python 3.7.9 and Numpy 1.18.5. The Machine was HP Spectre x360 2-i-1 13-ap0805no with an 8th generation Intel(R) Core(TM) i7-8565U Quad Core x64-based Processor with 8 MB cache, base frequency of 1.80 GHz (with max turbo frequency 4.60 GHz) and 16.0 GB DDR4 Ram on a Windows 10 Home (Version 20H2) operating system.

```
import numpy as np
import timeit
import copy

np.random.seed(12235)
test_data = np.random.random((1000,))

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=5, number=n_ar)
```

Listing 7: Timer function from [?]

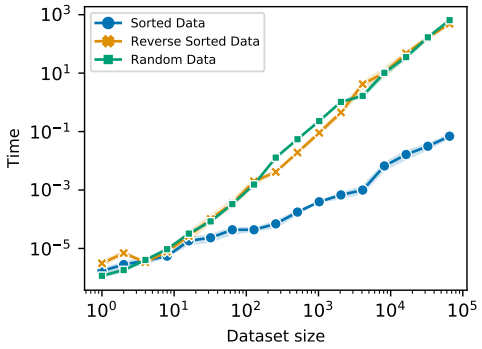

Figure 1: Insertion sort

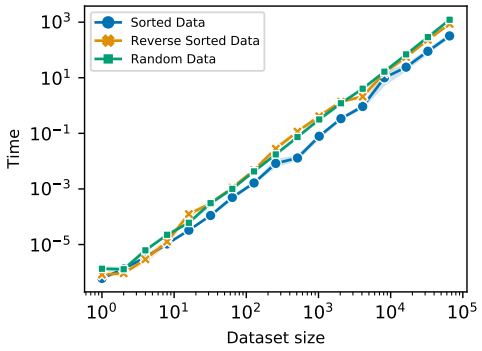

Figure 2: Bubble sort

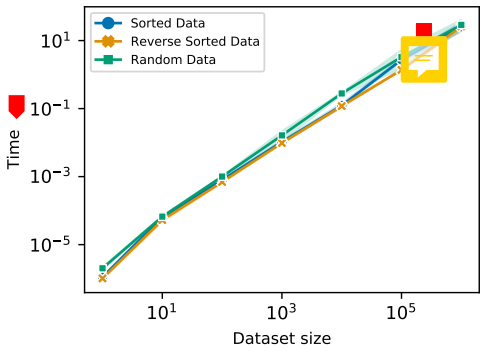

Figure 3: Merge sort

4 RESULTS

We benchmark our algorithms on sorted data, reverse sorted data and random sorted data. For insertion sort in Figure 1, we can see that its clearly sorts already sorted data faster than reverse sorted data and random data. The difference becomes more significant when the size of the data is above 10^2.

Bubble sort, as shown in Figure 2, is sorting the three different cases at almost the same speed for all data sizes.

In Figure 3, we see the same plot for merge sort. We can see that the time it takes to sort already sorted data and reverse sorted data increases faster for smaller data set sizes. Then sorting follows almost the same path for all three cases.

Sorting with quick sort in 4, we see in the plot that both reverse sorted data and sorted data increases f We see a clear divergence from random sorted data when the data set size is exceeding approximately a size of 100. Continuing with larger data set sizes a recursion limit is hit at a size of around 1000 for sorted and reverse sorted data. Making the algorithm fail.

Figure 5 shows the plot for our combined algorithm. At the specified threshold of data size equals 100 the sorting time makes a jump when switching to merge-sort.

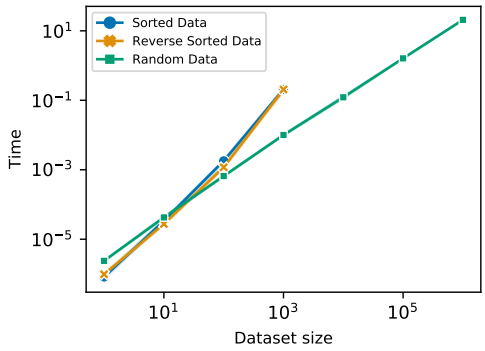

Figure 4: Quick sort

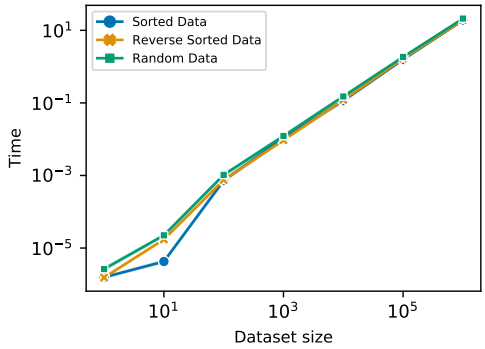

Figure 5: Combined sort

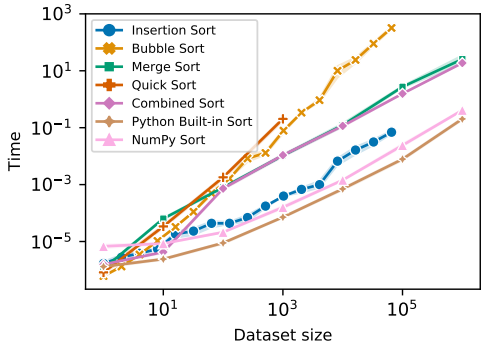

Figure 6: Sorted data

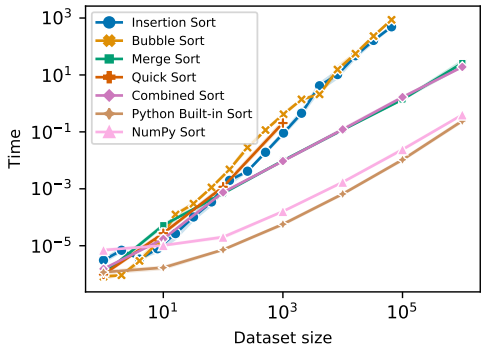

Figure 7: Reverse sorted data

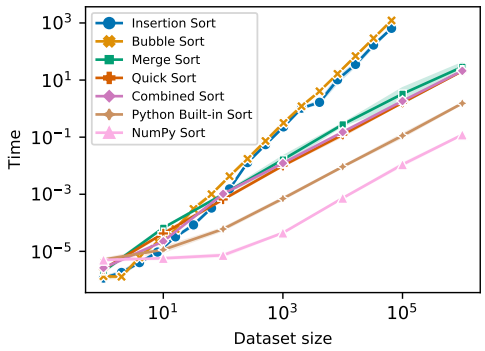

Figure 8: Random sorted data

In Figure 6, the plot showing sorted data for all algorithms including the Combined algorithm and the sorting functions implemented in Python and NumPy. We see here that bubble sort and quick sort performs worst as the size of the data increases. Then follows merge sort and the combined algorithm. The points of those algorithms are close as expected since the combined algorithm uses merge sort for data sizes above 100 elements. Insertion sort performs well on sorted data, only sorting NumPy and Python performs better og which Python is the best.

Looking at reverse sorted data in Figure 7 we see a similar picture as with the sorted data in Figure 6. However, insertion sort which was among the third fastest sorting algorithms for sorted data, is now almost as slow as the slowest algorithm, bubble sort when sorting reverse sorted data.

In Figure 8 all algorithms are compared for random sorted data. NumPy is here performing best. Quick sort is in this case performing slightly better than merge sort, and thus beeing the third best performing sorting algorithm.

5 DISCUSSION

The results after benchmarking sorting algorithms have showed that overall performance of sorting in the implemented sorting functions of Python and NumPy is by far the fastest algorithms in sorting our different data sets. Specifically, if look back on the plot for random sorted data in 8, we see that at a data size of 10^3 bubble sort uses approximately 10 000 times the amount of time as sorting in NumPy, and the difference are increasing as the data size grows. There are several possible reasons for this. One important aspect is that sorting functions in NumPy and Python, both are implemented in C-code, as opposed to the rest of the algorithms we

implemented python. C-code operates on a lower level of abstraction than python code. Python is an interpreted language which translates code into bytecode at run time. The bytecode is then further transformed into machine code, a language the machine is able to understand. The language C on the other hand is a compiled language. During compilation the code is translated directly into machine code. This makes C a faster language compared to Python, and thus using NumPy sort or Python sort will be beneficial for optimal performance [Sikorski and Honig 2012, Chapter 4.]

. Another important aspect is that NumPy uses introsort which is a hybrid combination algorithm as mentioned in the theory part. Implementations able to switch among different sorting algorithms along the sorting process seems beneficial since our benchmarking results has shown that different algorithms performs better or worse depending on the problem at hand. Looking at the results of our own implemented combined algorithm for the three different cases, shown in Figures. 6, 7 and 8, we see that switching from insertion sort to merge sort is beneficial for larger data sizes in two out of three cases. For random sorted data and reverse sorted data the expected run time of insertion sort has an upper boundary of $O(n^2)$ while merge sort runs in $O(nlogn)$, this turns out to be true when carrying out our experiment. However, the algorithm does not recognize if data are sorted beforehand, so for sorted data insertion sort is performing better when the size of the data set exceeds 100.

. Comparing the five algorithms implemented in python we see that quick sort performs best on random data. Random sorted data represents the average case and expected run time for quick sort has then an upper bound of $O(nlogn)$. This is the same upper bound as merge sort, however we observe that quick sort is slightly faster than merge sort for large random sorted date. The theory suggests that the constant factors hidden in quick sort is expected to be quite small compared to other algorithms. Further assuming the constants factors are larger in merge sort compared to quick sort this will result in quick sort achieving a higher performance sorting large data sets of random data. [Cormen et al. 2009, page. 12, 190]

. Continuing comparing the five python implemented sorting algorithms. We observe that when sorting already sorted data in 6 insertion sort performs best as the size of the data increases. In this case insertion sort only needs one iteration over the data set to check that every element is in order. While both quick sort and merge sort will carry on with different divide and conquer strategies even if no sorting is needed. And bubble sort compares elements multiple times as for all cases. The run time of insertion sort has an upper bound of $O(n)$ while for merge sort, quick sort and bubble sort the run time is $O(nlogn)$, $O(n^2)$ and $O(n^2)$ correspondingly. Our benchmark confirms this order. As mentioned earlier we also experience that quick sort fails with a recursion limit error, this is handled internally in python to avoid infinite recursions.

Sorting reverse sorted data we experienced similar results as with sorted data except for insertion sort. Now insertion sort needs to compare all elements multiple times with nested for loops resulting in a run time of $O(n^2)$

6 ACKNOWLEDGEMENTS

REFERENCES

[n.d.].
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.
"NumPy". "2020". "numpy.sort". "https://numpy.org/doc/stable/reference/generated/numpy.sort.html#numpy.sort"
Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis* (1st ed.). Library of Congress Cataloging-in-Publication Data, NA.