Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2020

Anders Mølmen Høst and Muhammad Fahad ljaz
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 there are already sorted, it will move onto the next set. Otherwise, it'll 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.

2.1.2 Bubble-sort.

2.2 Sub-quadratic algorithms

2.2.1 Mergesort. 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.

Listing 1: 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 2: Merge algorithm from Cormen et al. [2009, Ch. 2.3. p 34].

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 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
```

[Cormen et al. 2009]

3 METHODS

```
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 3: Timer function from [?]

4 RESULTS

We benchmark our algorithms on best-case, worst-case and average-case scenarios. Our test data consisting of 10, 100 and 1000 elements. The combined algorithm shifts from insertion-sort to mergesort when the number of elements reach the chosen threshold of 100 elements and in figure 1 we can see that the graphs of insertion-sort and the combined algorithm corresponds at 10 and then the combined algorithms corresponds closely to the mergesort graph after the elements exceed the threshold at 100 elements. (Since we have only three points, the transition looks continuous in our plot, while if we had more data points the shift would be represented by a sharp jump in the graph) In the best-case scenario the algorithms sorts already sorted data. So there is actually no need for sorting. However insertion-sort performs best in all three cases. Using 0.29 milliseconds of time sorting 1000 elements. In the worst case the algorithms sorts elements presorted in reversed order. In this case the overall performance is best in the combined algorithm. Insertion sort is faster sorting 10 elements but mergesort is faster in sorting 1000 elements. The time taken for mergesort sorting 1000 elements is 6 milliseconds while insertion-sort uses 112 milliseconds so mergesort beats insertion-sort by a factor of almost 20. For only 10 elements the situation is the opposite, insertion-sort uses 0.001 millisecond when mergesort uses four times that amount. When the algorithms are benchmarked on random ordered elements the results looks similar to the worst case. Insertion-sort uses less time sorting 10 elements but more time sorting 100 or 1000 elements. Insertion-sort uses 124 milliseconds sorting 1000 elements and 0.002 milliseconds on 10 elements, comparing to mergesort which uses 8 milliseconds and 0.004 milliseconds sorting 1000 and 10 elements respectively.

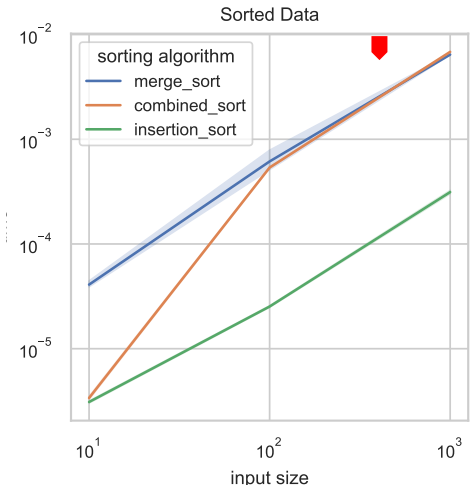

Figure 1: Benchmark results sorted data, n=1000, log-log plot

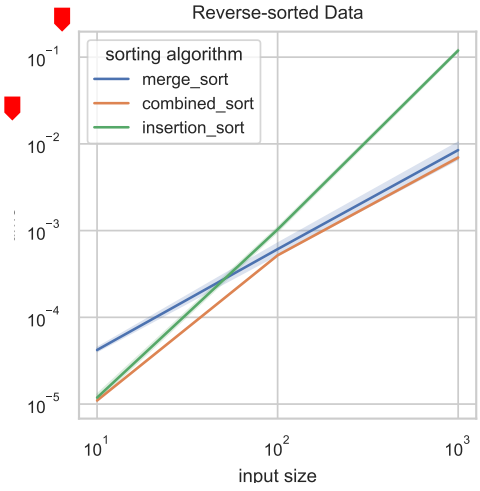

Figure 2: Benchmark results reverse-sorted data, n=1000, log-log plot

5 DISCUSSION

The results after benchmarking sorting algorithms have showed that insertion-sort performs well on small sorted data while merge-sort performs better if the number of elements exceed 100 and the data are either random, representing the average case or reversed-sorted representing the worst case. As the data size grows the performance of the two sorting-algorithms diverge, insertion-sort taking almost 20 times the amount used by mergesort in the worse case for 1000 elements. This is consistent with the theory, stating that in both worst- and average-case, the upper bound of the runtime of insertion-sort is quadratic in the input size. While in mergesort the runtime is, $\theta(nlgn)$, with n representing the size of the input data. The combined algorithm will be a good compromise to achieve more flexibility. However its essential to know what

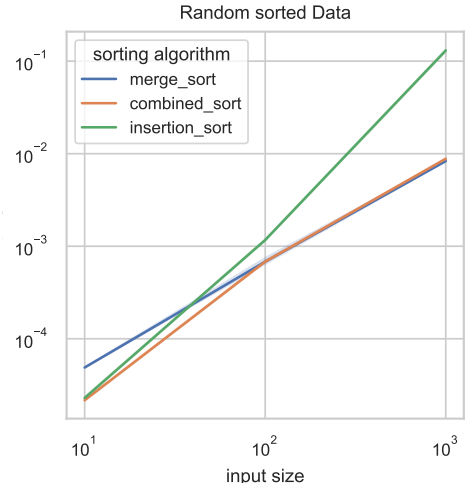

Figure 3: Benchmark results random-sorted data, n=1000, log-log plot

threshold value should be used in advance to optimize performance. Our choice of threshold with data size equals 100 seems to be a bit too high. Overall however, considering the results carried out from this experiment, the performance of the combined algorithm is the best among the three candidates.

6 ACKNOWLEDGEMENTS

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.