

Benchmarking Sorting Algorithms In Python

INF221 Term Paper, NMBU, Autumn 2020

Jon-Mikkel Korsvik
jonkors@nmbu.no

Yva Sandvik
ysandvik@nmbu.no

ABSTRACT

In this paper, we analyse ...

1 INTRODUCTION

Sorting algorithms are used to solve one of the key problems of computer science known as “The sorting problem”. This involves an input sequence of n numbers (a_1, a_2, \dots, a_n) , where the output is a permutation of the input sequence such that the numbers are ordered in an ascending order. The two main aspects to a sorting algorithm is its time complexity (speed) and its space complexity (memory usage). (Kilde: Lecture 6 ipynb, Plessner)

During this investigation we have assessed the performance of the sorting algorithms listed below, under certain assumptions regarding their time complexity. We compared their performance when sorting lists containing different types of elements, as well as their performance as the length of the lists increases.

2 THEORY

In the following subsections we will provide theory, pseudocode as well as details surrounding the methods used when comparing the following sorting algorithms:

- Quadratic algorithms
 - Insertion sort
 - Bubble sort
- Sub-quadratic algorithms
 - Merge sort
 - Quick sort
- Combined algorithm
 - Mergesort switching to insertion sort for small data
- Built-in sorting functions
 - Python ‘sort()’
 - NumPy ‘sort()’

2.1 Algorithm 1 - Insertion sort

Insertion sort (listing 1) is a simple in place and comparison based sorting algorithm. Best case runtime for this algorithm is:

$$T(n) = \Theta(n) . \quad (1)$$

This is achieved when the input array is already sorted. The worst case runtime occurs if the input list is in reversed order. This gives a quadratic runtime of:

$$T(n) = \Theta(n^2) \quad (2)$$

The average runtime is also quadratic, making insertion sort a bad choice for sorting large lists, however it is one of the best and quickest alternatives when it comes to sorting smaller lists.

Listing 1 Insertion sort algorithm from ?, 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
```

2.2 Algorithm 2 - Bubble sort

Bubble sort (listing 2) is known as an inefficient sorting algorithm due to its simplicity. Although bubble sort and insertion sort grow asymptotically at the same rate $\Theta(n^2)$, the difference in bubble sort and insertion sort lies in the number of comparisons. In contrast to insertion sort, bubble sort can make numerous comparisons that do not necessarily result in a swap, making it computationally less efficient.

Listing 2 Bubble sort algorithm from ?, Ch. 2.1.

```
BUBBLE_SORT(A)
1  n = A.length
2  swapped = False
3  rounds = 0
4  while swapped :
5      swapped = False
6      for i = 0 to n-rounds-1
7          if A[i] > A[i + 1] :
8              A[i], A[i + 1] = A[i + 1], A[i]
9              swapped = True
10     rounds + 1
```

2.3 Algorithm 3 - Merge Sort

Merge sort (listing 3) is yet another comparison-based sorting algorithm that uses the divide-and-conquer approach which involves recursively merging together two pre-sorted arrays such that the resulting array also is sorted. (kilde: CLRS s. 30, og nettside, wikipedia). Merge sort is known to be quicker when sorting larger lists. Unlike insertion sort and bubble sort it does not iterate through the entire list several times.

When sorting n objects, merge sort has a consistent average and worst case performance of:

$$T(n) = \Theta(n \lg n) \quad (3)$$

The implementation we have chosen of merge sort, as well as the most common implementations, do not sort in place. Which brings us to one of the drawbacks of merge sort; its memory requirement. The memory size of the input must be allocated for the sorted output to be sorted in, hence it uses more memory space than other in place sorting algorithms. (Wikipedia).

Listing 3 Merge sort algorithm from ?, Ch. 2.1.

```

MERGE_SORT(A)
1  if A.length > 1:
2      mid = A.length/2:
3      L_array = A[: mid]
4      R_array = A[mid :]
5      mergesort(L_array)
6      mergesort(R_array)
7      L_index = 0
8      R_index = 0
9      copy_index = 0
10     while L_index < L_array.length and R_index < R_array.length
11         if L_array[L_index] < R_array[R_index]
12             L_index + 1
13         else :
14             A[copy_index] = R_array[R_index]
15             R_index + 1
16     while L_index < L_array.length
17         A[copy_index] = L_array[L_index]
18         L_index + 1
19         copy_index + 1
20     while R_index < R_array.length
21         A[copy_index] = R_array[R_index]
22         R_index + 1
23         copy_index + 1

```

2.4 Algorithm 4 - Quick sort

Quick sort (listing 4) is an efficient in place divide-and-conquer sorting algorithm. When implemented well it can supposedly be about two or three times faster than its main competitors, such as merge sort.

It shares the same average time complexity as merge sort. However the worst case time complexity of quick sort is its main disadvantage, and is $\Theta(n^2)$ due to its need of a lot of comparisons in the worst case.

2.5 Algorithm 5 - Merge sort combined

Merge sort can be optimized to give "Merge sort combined" (listing 5) by integrating insertion sort and making it a hybrid algorithm. This takes advantage of the fact that insertion sort performs well on smaller lists. It will therefore use fewer comparisons in the worst case than both merge sort and insertion sort, potentially making it a very efficient sorting algorithm.

Listing 4 Quick sort algorithm from ?, Ch. 2.1.

```

SWAP(array, a, b)
1  array[a], array[b] = array[b], array[a]
2  return array

PARTITION(array, start, end)
1  pivotindex = start
2  pivotvalue = array[end]
3  for L_index = start to end
4      if array[L_index] < pivotvalue
5          swap(array, L_index, pivotindex)
6          pivotindex+ = 1
7  swap(array, pivotindex, end)
8  return array, pivotindex

QUICK_SORT(array, start, end)
1  if start < end
2      array, index = partition(array, start, end)
3      array = quick_sort(array, start, index - 1)
4      array = quick_sort(array, index + 1, end)
5  return array

QUICKSORT(array)
1  array = quick_sort(array, 0, array.length - 1)

```

Listing 5 Merge sort combined algorithm from ?, Ch. 2.1.

```

MERGE_SORT_COMBINED(A = list, threshold = 11, comb_algo :
str = "insertion")
1  if A.length > threshold
2      mid = A.length / 2
3      L_array = A[: mid]
4      R_array = A[mid :]
5      merge_sort_combined(L_array, threshold, comb_algo)
6      merge_sort_combined(R_array, threshold, comb_algo)
7      L_index = 0
8      R_index = 0
9      copy_index = 0
10     while L_index < L_array.length and R_index < R_array.length
11         if L_array[L_index] < R_array[R_index]
12             L_index + 1
13         else
14             A[copy_index] = R_array[R_index]
15             R_index + 1
16     while L_index < L_array.length
17         A[copy_index] = L_array[L_index]
18         L_index + 1
19         copy_index + 1
20     while R_index < R_array.length
21         A[copy_index] = R_array[R_index]
22         R_index + 1
23         copy_index + 1
24     else :
25         if comb_algo = "insertion"
26             insertion_sort(A)
27         else : A = np.sort(A)

```

Table 1: Versions of files used for this report; GitLab repository <https://x.y.z>.

File	Git hash
utility.py	8ec07210f
src	396d8a309
plot_creation.ipynb	8ec07210f
benchmark_results.csv	88c28d55c

2.6 Algorithm 6 - Python "sort()"

2.7 Algorithm 7 - Numpy "sort()"

3 METHODS

Short description of what we have done so far and how:

- Our test data is generated using the class function Array-Generator found in our utility file.
- First test data is generated, then our benchmark function times how long each algorithm uses to sort given lists with given lengths.
- The timer function times each test a given number of repetitions and returns all the results (so that they can be saved and used later), as well as showing the mean.
- Mac OS and Windows 10, Python version 3.8.3 and 3.29?
- Git hashes are provided in table 1.

Listing 6 Expert from benchmark code.

```
for algorithm in kwargs['function_list']:
    array_copy = copy(kwargs['array'])
    record[algorithm.__name__] = []
    for _ in range(iters):
        start_time = time.perf_counter()
        algorithm(array_copy) # Runs algo
        end_time = time.perf_counter()
    times.append(bench(func, n))
```

4 RESULTS

The first two algorithms we analysed were the quadratic sorting algorithms Insertion sort and Bubble sort. As mentioned in the theory section, these two sorting algorithms are known as quadratic sorting algorithms because their time complexity is $O(n^2)$.

Plots ...
We then moved onto the sub-quadratic algorithms Merge sort and Quick sort, sharing an average time complexity of $O(n \lg n)$.
Plots ...
Furthermore we compared the subquadratic algorithms with the hybrid Merge Sort Combined.
Plots ...
Finally we compared the built in sorting algorithms Python sort and Numpy sort, and added them to the previous comparison plot with the best performing algorithms.
Plot ...
Until now have found that insertion sort and bubble sort which are quadratic in time complexity combined with other methods

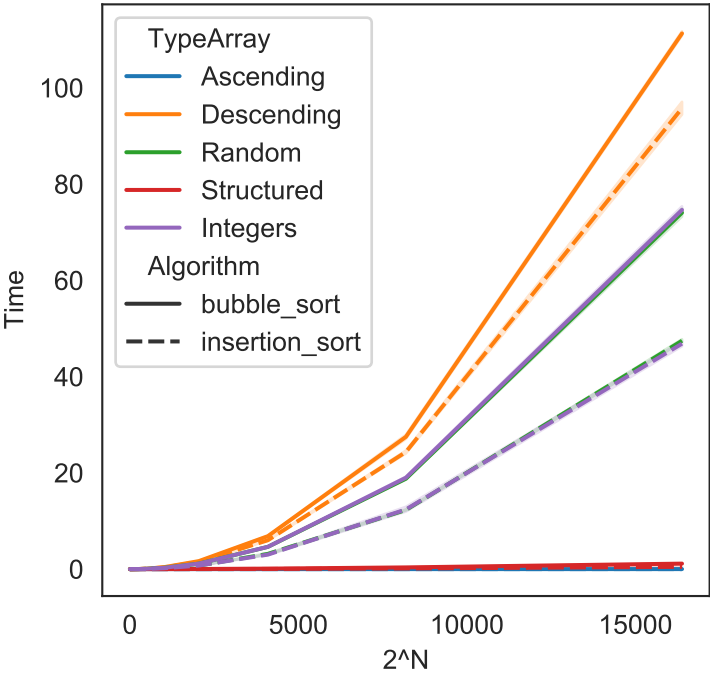


Figure 1: Benchmark results for insertion sort and bubble sort.

like for example merge sort, drastically reduce sorting time by reducing the callstack and memory complexity. Shown in results, by optimizing the difference beneath merge sort.

5 DISCUSSION

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

ACKNOWLEDGMENTS

We are grateful to ...for