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 $a_1, a_2, \ldots, a_n$, where the output is a permutation of the input sequence such that the numbers are ordered in an ascending or descending order. The two main aspects to a sorting algorithm is its time complexity (speed) and its space complexity (memory usage).

During this investigation we have assessed the performance of the sorting algorithms listed below, with certain assumptions regarding their time complexity. We compared their performance using the Python 'time' function, bench-marking their performance when sorting lists containing various types of elements. In addition we assessed the time-development as the length of the lists increased.

2 THEORY

In the following subsections we will provide theory, pseudo-code 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
  - Merge sort 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 run-time for this algorithm is:

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

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

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

The average run-time 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 know as a straightforward and simple sorting algorithm. Both when it comes to implementation and understanding. However its main disadvantage is its inefficiency, especially when sorting large arrays.

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 could in the worst case have to make numerous comparisons that do not necessarily result in a swap, making it computationally slower because of the many comparisons.

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 presorted arrays such that the resulting array also is sorted. 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:

$$T(n) = \Theta(n \lg n) \tag{3}$$

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)
if A.length > 1:
    mid = A.length/2:
    L_array = A[: mid]
    R_array = A[mid :]
    mergesort(L_array)
    mergesort(R_array)
    L_index = 0
    L_index = 0
    copy_index = 0
    while L_index < L_array and R_index < R_array
        if L_array[L_index] < R_array[R_index]
            L_index + 1
        else :
            A[copy_index] = R_array[R_index]
            R_index + 1
    while L_index < L_array.length
        A[copy_index] = L_array[L_index]
        L_index + 1
        copy_index + 1
    while R_index < R_array.length
        A[copy_index] = R_array[R_index]
        R_index + 1
        copy_index + 1
```

2.4 Algorithm 4 - Quick sort

Quick sort listing 4 is an efficient and important 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. Thankfully this can be avoided in almost all cases , making the expected run-time $\Theta(n \lg n)$.

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 therefor 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)
array[a], array[b] = array[b], array[a]
return array

PARTITION(array, start, end)
pivotindex = start
pivotvalue = array[end]
for L_index = start to end
    if array[L_index] < pvotvalue
        swap(array, L_index, pivotindex)
        pivotindex+ = 1
swap(array, pivotindex, end)
return array, pivotindex

QUICK_SORT(array, start, end)
if start < end
    array, index = partition(array, start, end)
    array = quick_sort(array, start, index − 1)
    array = quick_sort(array, index + 1, end)
    return array

QUICKSORT(array)
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")
if A.length > threshold
    mid = A.length / 2
    L_array = A[: mid]
    R_array = A[mid :]
    merge_sort_combined(L_array, threshold, comb_algo)
    merge_sort_combined(R_array, threshold, comb_algo)
    L_index = 0
    L_index = 0
    copy_index = 0
    while L_index < L_array and R_index < R_array
        if L_array[L_index] < R_array[R_index]
            L_index + 1
        else
            A[copy_index] = R_array[R_index]
            R_index + 1
    while L_index < L_array.length
        A[copy_index] = L_array[L_index]
        L_index + 1
        copy_index + 1
    while R_index < R_array.length
        A[copy_index] = R_array[R_index]
        R_index + 1
        copy_index + 1
else :
    if comb_algo = "insertion"
        insertion_sort(A)
    else : A = np.sort(A)
```

2.6 Algorithm 6 - Python "sort()"

Since version 2.3 'Timsort' has been the 'built-in' sorting function in Python. In congruence to merge sort combined, timsort is also a hybrid sorting algorithm derived from merge sort and insertion sort. It is a stable sorting algorithm with the a time complexity of $\Theta(n \lg n)$. It starts by using insertion sort to boosts the smaller naturally sorted elements found in the unsorted array to a sufficient size, after which they are merged using merge sort. By using insertion sort on the smaller lists one reduces the overhead produced by using more complex algorithms on such small lists. The small subsets, also called 'Runs', need to be of a minimum size, often between 32 and 64, however this depends on the length of the array that is to be sorted.

The Python sort() methods do not require any mandatory parameters but they offer two optional parameters. 'key' and 'reverse'. The 'key' parameter serves as a key for the comparison. The reverse parameter defines the order. Default being 'False, which denotes ascending order.

2.7 Algorithm 7 - Numpy "sort()"

Numpy sort() is the 'built-in' numpy function which sorts an array in-place. Numpy sort() can alternate between three sorting algorithms, the default being quick sort, if specified one can also use merge sort or heap sort.

The only mandatory input argument to the numpy sort() function is the input array to be sorted. Otherwise one can choose which axis of the array one wishes to sort, the default it -1. One can choose which algorithm one wants to use (out of the three mentioned above), and lastly one can choose the order one wishes the array to be sorted. Ascending order is the default.

3 METHODS

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

- Our test data is generated using the class function ArrayGenerator 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 Exerpt 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))
```

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`

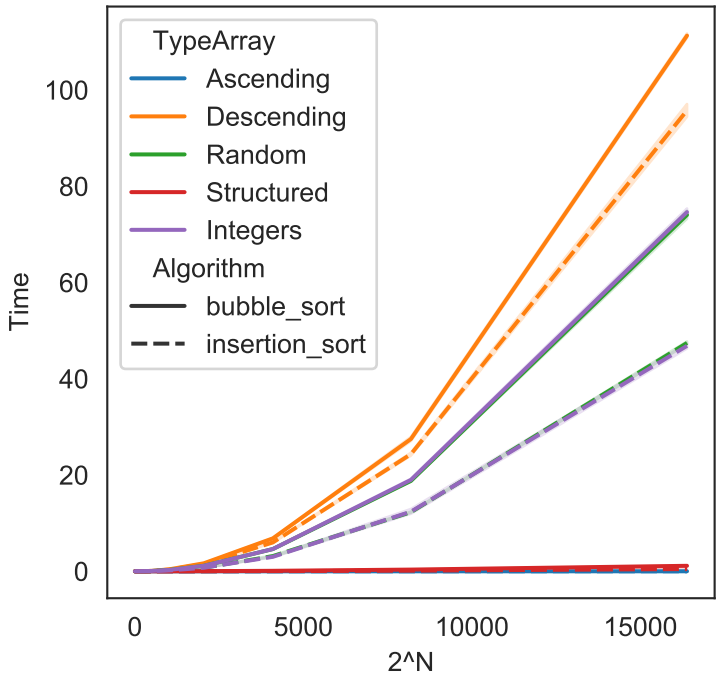

Figure 1: Benchmark results for insertion sort and bubble sort.

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 sub-quadratic 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

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 ….