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, benchmarking their performance when sorting arrays containing various types of elements. In addition we assessed the time-development as the length of the arrays increased.

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
  - 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 **??** 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 **??** 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 **??** 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. Unlike insertion sort and bubble sort it does not iterate through the entire list several times, but instead uses this recursive divide and conquer approach which gives a consistent average and worst case performance of:

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

This is in fact the best possible run-time that can be achieved by a sorting algorithm. In addition, the fact that the input array is split into subsets, opens for the possibility to sort the two subsets simultaneously, also called to parallelize, which naturally increases the efficiency further.

However, on of merge sorts drawbacks stems from the time cost caused by recursion which effects its performance when sorting smaller lists. In addition, the most common implementations of merge sort do not sort in place. Bringing us a second drawback 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 :]
    MERGE_SORT(L_array)
    MERGE_SORT(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 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, and it is also straightforward to parallelize since it also uses the divide and conquer approach. However the worst case time complexity of quick sort is its main disadvantage, and is $\Theta(n^2)$, i.e no better than insertion sort. This becomes an issue when the input is an already sorted array, almost sorted array, or reversed sorted array, as quick sort still needs to do several recursions even though the list is sorted or almost sorted.

The partitioning that occurs when quick sort splits the input array into two subsets is based on a pivot element. The selection of the pivot is critical to the worst case performance of quick sort because it determines the number of necessary recursion levels that will occur. If the algorithm consistently picks the median element as pivot, as opposed to the first, last or a random element, one can guarantee that one will get the best time complexity. Meaning, the worst time complexity can be avoided if the partitioning is balanced, making the expected run-time $\Theta(n \lg n)$, such as merge sort.

In listing **??** you can see how we chose to implement the partitioning in quick sort. We used a median of three optimization to handle the worst case scenarios of input arrays that are in ascending or descending order.

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

```
MEDIAN_OF_THREE(A, low, high)
mid = (low + high) // 2
if A[mid] < A[low]
    exchange A[low] with A[mid]
if A[high] < A[low]
    exchange A[low] with A[high]
if A[mid] < A[high]
    exchange A[mid] with A[high]
PARTITION(A, low, high)
MEDIAN_OF_THREE(A, low, high)
pivot = A[high]
i = low − 1
for j = low to high − 1
    if A[j] ≤ pivot
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
QUICKSORT(A, low, high)
if low < high
    pivot = PARTITION(A, LOW, HIGH)
    QUICKSORT(A, low, pivot − 1)
    QUICKSORT(A, pivot + 1, high)
```

2.5 Algorithm 5 - Merge sort combined

Merge sort can be optimized to give "Merge sort combined" (listing **??**) 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.

2.6 Algorithm 6 - Python "sort()"

Since version 2.3 'Timsort' has been the 'built-in' sorting function in Python. In similarity to merge sort combined, timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. It is a stable sorting algorithm with a the time complexity of $\Theta(n \lg n)$. It starts by using insertion sort to boost the smaller naturally sorted

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

```
MERGE_SORT_COMBINED(A, threshold = 11)
1  if A.length > threshold
2        MERGE_SORT(A)
3  else :
4        INSERTION_SORT(A)
```

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 elements, however this depends on the length of the array that is to be sorted.

The Python sort() method does not require any mandatory parameters but offers 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:

- 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 sorting algorithm in a list a given number of repetitions and returns all the results in a dictionary, as well as printing stats if verbose.
- Timing data is unpacked as a pandas DataFrame with columns 2^N, Time, TypeArray and Algorithm .
- OS : Windows 10, Python version 3.7.9, Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 Mhz, 4 cores, 8 logic threads. 8 GB of installed RAM. **Make and model:** Huawei MateBook X Pro i5 dGPU 8GB 256GB (2018)
- Git hashes are provided in table **??**.

3.1 Data generation - Array Generator

We have decided to use a factor of 2^n for generating arrays. Giving us binary growth of problem sizes with binary increment benchmarking.

Figure 1: Different problem types

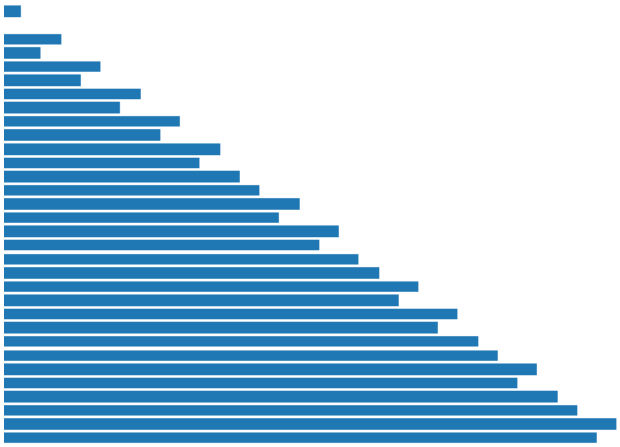

Figure 2: Structured Array

3.1.1 Reasoning behind using binary scale. This will give us a great range from lists of 1 to lists of 1,073,741,824 elements. In addition the binary scale will reduce the size of results by a factor of. $\frac{n}{a\log_2(n)}$ where a is a scalar representing linear step sizes. The benchmark process shaves of time usage by a factor of $\frac{\sum_{k=1}^{n/a} T(k \cdot a)}{\sum_{k=1}^{\log_2 n} T(2^k)}$, where T(x) is the $O(n)$ of an algorithm. This will ease the computation time for generating data as well as plots while retaining information of the trend in runtimes for every algorithm.

3.1.2 Approach. Using an object oriented approach we can set the seed of a random number generator from NumPy and gather the benefits of large data compared to the standard random library in Python. The ArrayGenerator class can create arrays such as shown in figure **??**. We can create random arrays of both float and integer type. Indicated in later plots as correspondingly **Random** and **Integers**.

3.1.3 Creation of the Arrays. Sorted and Reversed are made using NumPy arange method. Random and Integers are created using the random number generator of the class. The nearly sorted array or the structured array is generated as shown in listing **??** and gives an array as shown in figure **??**.

Listing 6 Excerpt from ArrayGenerator in utility.py

```
ArrayGenerator.structured_array(self, n):
    """
    Creates array with ascending structure.
    """
    A = np.arange(0, int(2**n))

    # Shuffle range
    s_r = int(((2**n))//((n**2)/2))

    for i in range(0, int(2**n), s_r):

        #In place shuffle within indices
        self.rng.shuffle(A[i: i + s_r])

    return A.astype('int32')
```

The Structured Array will be shuffled for any list with number of elements > 0 since s_r(shuffle range) is at it's minimum at $n = 3 \Rightarrow$ ShuffleRange = $int(1.7778) = 2$

3.2 Runtime Benchmarking

The main bunch of work is done by the *repeating_timer* decorator shown in listing **??**, which can be used with single or multiple algorithms (as a callable) several times on the same array. It handles in place algorithms by assigning a new copy before every iteration, and thereafter saves the results from every iteration in a dictionary with algorithm name as the key and the list of timings as the value.

Listing 7 Excerpt from repeating_timer decorator in utility.py

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

To generate results we used `notebook/generate_data_script.py`. Over time we generated multiple results, and merged them together with `src/merge_csv.py` which gave `data/merged_results.csv` as an output. To store results we have created an elegant way to store all data points with equality. This can be manipulated easily for creating new variables and plotting. See table **??**.

To keep us updated on how the benchmark was doing while running, we could look at the terminal outputs shown below:

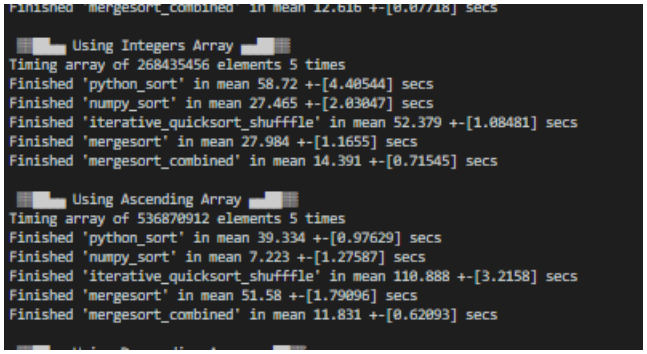

Figure 3: Terminal output

Table 1: 5 Random samples from results

Results DataFrame			
Algorithm	2^N	TypeArray	Time
python_sort	15	Integers	0.003036
numpy_sort	21	Descending	0.028119
numpy_sort	1	Structured	0.000002
numpy_sort	15	Random	0.002030
mergesort	15	Structured	0.002981

Table 2: Versions of files used for this report; GitLab repository :https://gitlab.com/nmbu.no/emner/inf221/h2020/student-term-papers/team_19/inf221-term-paper-team19.

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

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

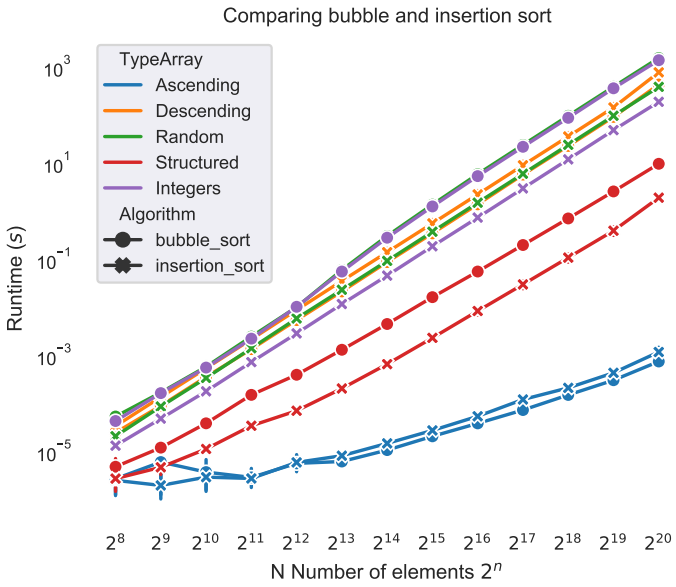

Figure 4: Benchmark results for insertion sort and bubble sort.

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. **??**.

ACKNOWLEDGMENTS

We are grateful to …for ….