Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2019

Bishnu Poudel
bishnu.poudel@nmbu.no
MS in Data Science, NMBU

Mohamed Radwan
mohamed.radwan@nmbu.no
MS in Data Science, NMBU

ABSTRACT

In this paper, we generate the runtimes for three pure sorting algorithms that follow the divide-and-conquer approach. In addition, we benchmark the inbuilt numpy sort function- np.sort() and also the python sort function - sorted(). Then we compare these real-life runtimes with the theoritical runtimes that we have learnt as a part of the Algorithms and data structure course at NMBU. We also analyse and plot the distributions of runtimes we foundData is collected for list sizes from 10 to 10 million in order to study the asymtotic behavior of these algorithms. The work has been done using python and its libraries.

1 INTRODUCTION

Sorting and searching are basic and routine operations for computers of any type. Much research has already been done in the subject, and computers today use the most robust algorithms possible that meet their specific requirement/application. Many applications use a blend of 2 or more algorithms depending on the nature input data or the application. However, it would be interesting to see how the sorting algorithms behave stand-alone in real life.

In the following section, we are benchmarking sorting algorithms in python based on their runtimes. This paper is also the final term-paper for the 'Computer Science for Data Scientists' course at NMBU. Here, we get to compare the theoritical runtimes of sorting algorithms to their real-life runtimes.

The main questions we are trying to address are: Do the algorithms show their theoritical average case behavior in real situations? Which of the stand alone sort is more efficient? Do these standalone sorts have a chance against the built-in sorts of numpy and python? How are the runtimes of each run distributed, is there any statistical significance in the distribution?

In the Theory section, we describe the psuedo-code of the algorithms together with their theoritical runtimes. We discuss the best, average and the worst cases. In the Methods section, we describe the python implementation of the algorithms, together with the python functions we wrote to extract the runtime information. We also discuss the type and amount of data we collected. Results section has facts and figures from our analysis. In the Discussion section, we summarize our findings and compare them to the theoritical expectations. We also compare the standalone algorithms with the pre-implemented algorithms in numpy and python. Acknowledgements and References conclude the paper.

2 THEORY

2.1 Heap Sort

Listing 1 Heap sort algorithm from **?**, Ch. 6.4.

```
HEAP-SORT(A)
    BUILD-MAX_HEAP(A)
    for i = A.length downto 2
        swap A[1] and A[i]
        A.heap_size = A.heap_size - 1
        MAX-HEAPIFY(A, 1)
```

Heap sort uses the max-heap (or min heap property) to sort elements in an array. Due to this, heap sort is not stable. Therefore, the keys having exact value might be interchanged. Theoritically, all 3 cases of heap sort (worst, best and average) are of the order $n * logn$. We wrote a python implementation of psuedo code above to sort the array in place. It will be interesting to see which of the three sets of data (random, ordered, reversely ordered) data performs the best on heap sort.

2.2 Merge Sort

Listing 2 Merge sort algorithm from **?**, Ch. 2.3.1.

```
MERGE-SORT(A,p,q)
    if p < r
    q = (p + r)/2
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Here the MERGE function simply merges two lists, both of which are already sorted. Merge sort probably got its name from this sub-process of the algorithm.Merge sort is the most intuitive divide and conquer approach to sorting, as it is dividing the list recursively in two equal halves (if the list has odd number of elements, one half has 1 more element than the other).

Merge sort has the runtime of the order of $n * logn$ in all of its best, average and worst cases. It will be interesting to see if the already sorted data is the best case for a merge sort.

2.3 Quick Sort

In case of quicksort, most of the work is done by the PARTITION function. It does the inplace swapping of the list

Listing 3 Quick sort algorithm from **?**, Ch. 2.3.1.

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)


PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] <= x
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[r]
    return i + 1
```

elements. It also returns the position of the pivot element to the QUICKSORT function.

Quicksort has a average and best case runtime of the order of $n * logn$, (the best case has a smaller constant of course). However, in the case where each subsequent partition has only one less element than the last partition, it runs into its worst case and has a theoritical runtime of n^2. It will be interesting to see if we indeed run into this scenario.

2.4 Numpy's numpy.sort()

The numpy sort function uses quicksort and a combination of few other sorts depending on the input data. Yet to put reference article or link here...

2.5 Python's default sorted()

Python uses a now popularly used sort algorithm in many programming languages, called timsort which is named after its inventor. Yet to put reference article or link here...

3 METHODS

We wrote the python implementations for heap sort, merge sort and quick sort following the psuedo code from CLRS. We wrote the code in a way that the only input we need to provide is the input array. The functions will calculate the length of the array or subarrayif their algorithm needs the length. Merge sort and quick sort call themselves recursively with 3 parameters, while heap sort does it with a single parameter. All the python codes, data generated and figures can be found at https://github.com/vsnupoudel/termpaper01. It was interesting to see that we could implement quicksort in a few lines. Below we can see the partition function used for quicksort.

Secondly, to benchmark the 3 algorithms we implemented along with numpy.sort() and sorted(), we use the timeit library. The skeleton of the function was provided by professor H.E. Plesser, which is built on. Since the process can slow down due to other processes in the computer, w e repeated

Listing 4 Quick sort PARTITION FUNCTION

```
def quick_comparison_and_swap (Inputlist, start, end):
    pivot_last = Inputlist[end]
    index_less_than = start -1

    for comparison_index in range(start,end):
        if Inputlist[comparison_index]<= pivot_last:
            index_less_than += 1
            Inputlist[index_less_than], Inputlist[comparison_in
            = Inputlist[comparison_index], Inputlist[index_less

    Inputlist[end] , Inputlist[index_less_than+1] \
    =   Inputlist[index_less_than+1], Inputlist[end]

    return index_less_than+1
```

each of the timeit experiments 7 times and took the fastest time among the 7. We export the runtimes of these algorithms for list of size ranging from 10 to 10 million. We use the same seed all the time, so that the data we use is exactly same at all times.

Listing 5 Time it function used with parameters

```
import numpy as np
import timeit
import copy

def timing_function(number_of_data_points, sort_type,randomizat
    np.random.seed (seed_number)
    test_data = np.random.random(number_of_data_points ,)
    test_data = list(test_data)
    if randomization_type=='reverse':
        test_data= sorted(test_data, reverse =True)
    elif randomization_type=='sorted':
        test_data= sorted(test_data)

    clock = timeit.Timer ( stmt ='sort_func ( copy ( data) ) ',
                           globals ={ 'sort_func': sort_type ,
                            'data': test_data ,
                            'copy': copy.copy })
    n_ar , t_ar = clock.autorange ()
    t = clock.repeat ( repeat =7, number = n_ar )
    return t
```

After that, we compare the runtimes of 5 algorithms for all data sizes in a number of line graphs. Similary, we plot a scatter plot of the 7 runtimes we got for each of the algorithms and see if they show any statistical significance. We also compare the runtime for reverse sorted and already sorted data and see if it drastically different than the random data for any of the algorithm.