Analysis of Sorting Algorithms

Bahadır Özkan
Istanbul,Turkey
bahadir.ozkan@bahcesehir.edu.tr

***Abstract*—There is not a panacea for sorting algorithms rather there are more efficient algorithms in some cases. In this paper, 6 of these algorithms are analyzed in terms of speed where the array was consisted of random, sorted and reverse sorted integers.**

Keywords—sorting algorithms, analysis of algorithms, insertion sort, quick sort, heap sort.

I. Introduction

The goal to analyze the algorithms is to predict the performances of them. This knowledge then can be used in the design decisions. The performance of an algorithm depends on time and space resources required for it to execute. Today, in the analysis of the algorithms, the focus is on the time rather than the space constraint. It is true that the data structures take time however, there are a variety of good options at one's disposal. Hence, in this paper, space criteria will not be discussed. Six sorting algorithms were analyzed in terms of execution time against an array of 100,000 integers, in random, sorted and reverse sorted order.

II. Comparison-based sorting algorithms

Algorithms that sort a list or array based on only the input are called comparison-based sorting algorithms. These techniques do not need any statistical or descriptive data related to the input. This quality makes comparison-based sorting algorithms simple and powerful. Some methods are straight-forward to understand and implement whereas others are built on these simple ones to achieve efficient sorting. These latter ones are harder to grasp works almost always better than the simple ones. The algorithms analyzed and their computation complexities will be discussed next:

A. *Insertion Sort*

Insertion Sort is the first of the simple methods that has a time complexity of Ω(n) in the best case and n^2 for the worst and average cases. If the array is already sorted it will take n times to reach the end of the array and no other computations will be made. This will be seen in the experimental results section.

B. *Selection Sort*

Best, worst and average complexity of the selection sort is n^2. Here, sorted array does not help the performance because all items will be compared to each-other until the end of the list.

C. *Bubble Sort*

Bubble sort and its optimized has the worst-case time complexity of O(n^2). Sorting plays a crucial factor on the performance. Best case scenario when the array is sorted the time complexity is Ω(n). On average the complexity is θ(n^2).

D. *Merge Sort*

Merge Sort's time complexity is nlog(n) for all cases. Merge sort uses divide and conquer method and recursively splits the data, sorts it and then merges it back. The number of splits does not change with the sorting condition.

E. *Quick Sort*

Quick sort also has an average and best time complexity of nlog(n), however, its worst-case complexity is O(n^2) though, this is a rare situation. The pivot selection of the Quick Sort affects the performance. Worst-case condition is achieved when the leftmost element is selected as the pivot and the array is already sorted. This can be remedied by choosing a random pivot or the mean element as the pivot. In the experiment, middle element is chosen as the pivot. This approach can go to a O(n^2) in some cases but working with sorted and reverse sorted arrays, it is a better option to choose than the first or last elements. Quick Sort does in-place sorting unlike Merge Sort where an extra O(n) data needs to be stored.

F. *Heap Sort*

It is an improved version of the selection sort using an efficient heap data structure. It is an iterative method where like Merge Sort all time complexities are nlog(n). If the data is close to a sorted state, Heap Sort will perform better than Merge and Quick Sorts.

III. Experimental setup

A laptop that has the following specifications had been used for the analysis: 1,8 GHz Intel Core i5, 4 GB 1600 MHz DDR3. As a programming language, first Python was used both in local and in Google Colab with GPU enabled. However, a single iteration of the Insertion Sort algorithm took more than 20 minutes, therefore the analysis was carried to C++11 environment. The results as it will be discussed in the next section are significantly faster for the C++ when working with an abundance of data. All six algorithms were iteratively executed 10 times for all cases and the records below has been collected. Source code can be accessed from https://github.com/bahadirozkan/sortingAlgorithms

IV. Experimental results

TABLE I. Performance of Insertion sort

Execution time in seconds	
Insertion Sort with random order	***Seconds***
Average	28.3533
Maximum	29.485
Minimum	27.128
Insertion Sort with sorted order	***Seconds***
Average	0.0029
Maximum	0.005
Minimum	0.002
Insertion Sort with reverse sorted order	Seconds
Average	62.5433
Maximum	67.674
Minimum	56.005

TABLE II. PERFORMANCE OF SELECTION SORT

Execution time in seconds	
Selection Sort with random order	***Seconds***
Average	50.8898
Maximum	56
Minimum	48.374
Selection Sort with sorted order	***Seconds***
Average	48.8448
Maximum	53.473
Minimum	46.839
Selection Sort with reverse sorted order	***Seconds***
Average	51.3585
Maximum	55.635
Minimum	49.16

TABLE III. PERFORMANCE OF BUBBLE SORT

Execution time in seconds	
Bubble Sort with random order	***Seconds***
Average	96.0891
Maximum	98.039
Minimum	93.995
Bubble Sort with sorted order	***Seconds***
Average	51.012
Maximum	52.4
Minimum	49.618
Bubble Sort with reverse sorted order	***Seconds***
Average	107.809
Maximum	127.789
Minimum	101.981

TABLE IV. PERFORMANCE OF MERGE SORT

Execution time in seconds	
Merge Sort with random order	***Seconds***
Average	0.6167
Maximum	0.664
Minimum	0.596
Merge Sort with sorted order	***Seconds***
Average	0.6077
Maximum	0.641
Minimum	0.59
Merge Sort with reverse sorted order	***Seconds***
Average	0.6003
Maximum	0.628
Minimum	0.585

TABLE V. PERFORMANCE OF QUICK SORT

Execution time in seconds	
Quick Sort with random order	***Seconds***
Average	0.0521
Maximum	0.093
Minimum	0.038
Quick Sort with sorted order	***Seconds***
Average	0.025
Maximum	0.081
Minimum	0.014
Quick Sort with reverse sorted order	***Seconds***
Average	0.0323
Maximum	0.092
Minimum	0.014

TABLE VI. PERFORMANCE OF HEAP SORT

Execution time in seconds	
Quick Sort with random order	***Seconds***
Average	0.1104
Maximum	0.119
Minimum	0.101
Quick Sort with sorted order	***Seconds***
Average	0.1154
Maximum	0.211
Minimum	0.092
Quick Sort with reverse sorted order	***Seconds***
Average	0.0925
Maximum	0.097
Minimum	0.082

V. CONCLUSION

Some findings from the experimental results are as follows:

- Insertion Sort gave the best results for the sorted data due to the time complexity of $\Omega(n)$.
- High-level algorithms which are Merge, Quick and Heap Sort performed better over the rest. Quick Sort among them needed less time to execute. (See figure 2)
- Bubble Sort had the worst performance. (See figure 1)
- Merge Sort's, Selection Sort's and Heap Sort's performance was not affected much by the order. Whereas Insertion Sort was affected the most.
- Choosing the middle item as pivot in Quick Sort worked as expected and prevented the worst-case scenario.

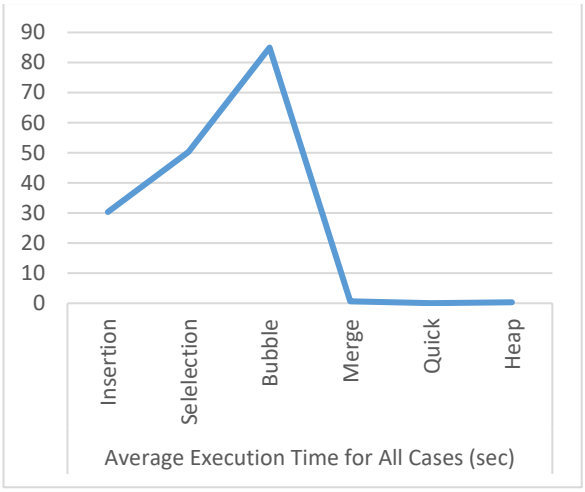

Fig. 1. Average Execution Time For All Cases in seconds

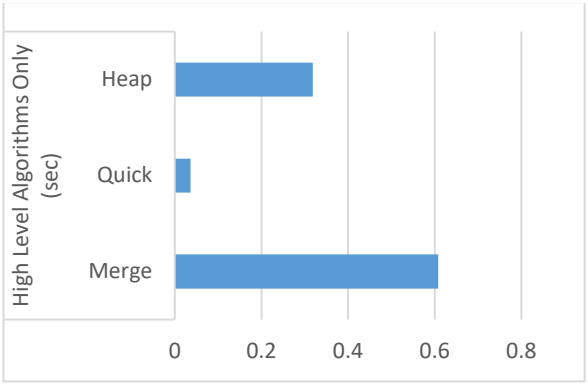

Fig. 2. Performance of Heap, Quick and Merge Sort Algorithms