# The Sorting Showdown: The Comparison Between Merge, Quick, Insertion and Heap Sort

December 15, 2022

Adilet Baimyrza uulu

## 1 Introduction

In this paper we will compare 4 well-known comparison-based algorithms. Each algorithm is going to sort an array of size *n*, where the integer is bounded to the range (-*n*, *n*). The items in the array must be sorted in an ascending order.

### 1.1 Merge sort

Merge sort is a divide-and-conquer algorithm that works by dividing the input list into smaller sub-arrays, sorting these sub-arrays, and then merging them back together to create the final sorted array. It uses a recursive approach to divide the array into smaller and smaller sub-arrays until each sub-array consists of only a single element, which is, by default, sorted. These single-element sub-arrays are then merged in pairs, using a comparison-based sorting technique, to create larger and larger sorted sub-arrays. The process continues until the entire initial array is sorted.

The algorithm is stable and belongs to the group of not-in-place sorting algorithms with a time complexity of *O(n log n)* in average and worst cases.

### 1.2 Quick sort

Quicksort is another divide-and-conquer algorithm that works by selecting a pivot element from the input array and partitioning the list into two sub-arrays based on the pivot. The pivot is chosen in such a way that all the elements in the left sub-array are less than or equal to the pivot, and all the elements in the right sub-array are greater than the pivot. The sub-arrays are then recursively sorted using the same process, until the entire input list is sorted.

Quicksort algorithm is highly efficient in the average case having time complexity of *O(n log n)*, but it has a time complexity of *O(n^2)* in the worst case. It is also not a stable algorithm.

### 1.3 Insertion sort

Insertion sort is a simple sorting algorithm that works by iterating over the elements of the input array and inserting each element into its correct position in the final sorted array. It starts by assuming that the first element in the list is already sorted, and then it iterates over the remaining elements, comparing each element to the elements that have already been sorted. When an element is encountered, it is inserted into the correct position in the sorted array, shifting all the other elements to the right by swapping to make room. This process continues until all the elements have been inserted into the sorted array.

It has time complexity of *O(n^2)* in the average case as well as in the worst case, but it is a stable algorithm and in-place.

### 1.4 Heap sort

Heapsort is a sorting algorithm that uses a data structure known as a heap to organize and sort the input data. A heap is a tree-like data structure that satisfies the heap property, which states that the value of each node in the tree is greater than or equal to the values of its children (if any). Heapsort works by first building a heap from the input array, and then repeatedly removing the largest element from the heap and adding it to the end of the array, hence making a sorted array in-place. This process continues until the heap is empty, at which point the final sorted array contains all the elements of the original input array in sorted order.

Its time complexity is *O(n log n)* in average and worst cases. And it is also stable and in-place.

## 2 Methodology

The above algorithms were implemented in C++ language and were tested on my laptop in Visual Studio. The results were obtained for the arrays of bigger size such as **100, 200, 300, . . . , 5000** and for smaller arrays starting from **1, 2, 3, 4, …, 100**. Elements of the array for both types of arrays were randomly shuffled integers of values in range (*-n, n*), where *n* indicates the size of the array. For each algorithm, there was given the same input data. Each array size was tested **100** times. The result for each size is the average time it took for each algorithm to sort the input array.

## 3 Results

Graphs of results for the average case for all four algorithms tested on bigger arrays are presented in *Figure 2*. On that graph, we can see that Insertion sort took the longest time to sort the array, compared to all other algorithms. While we can't see much difference between other algorithms, in *Figure 1*, where

the results of the Insertion sort were excluded, we can see that Quick sort is more than 2 times faster than Heap sort and roughly 3 times faster than Merge sort.

Graphs of results for the average case for arrays of smaller size are presented in *Figure 3*. There, if we do not consider what happened, while measuring the time between array size 17 and 33. We can see that, on early stages, Insertion sort is being faster than Merge sort and Heapsort, however still slower than Quicksort. But as the array size gets bigger the time needed to sort for the Insertion sort increases quadratically.

# 4 Conclusion

In conclusion, our comparison of four well-known sorting algorithms has shown that the Quick sort is highly efficient in the average case. Heap and Merge sort algorithms, being slower than Quick sort, are still efficient, with a time complexity of *O(n log n)* in the average case. Both algorithms are also stable. However, if we take into consideration Quick sort's worst case, which has a time complexity of *O(n^2)*, its graph will not much differ from the Insertion sort's, making it less reliable. Although Quick sort is the fastest in the average case, the Heap sort is more reliable and still very fast in average case and in its worst case. So, the Heap sort is the best. Finally, the insertion sort algorithm is the least efficient, with a time complexity of *O(n^2)* in the average case.

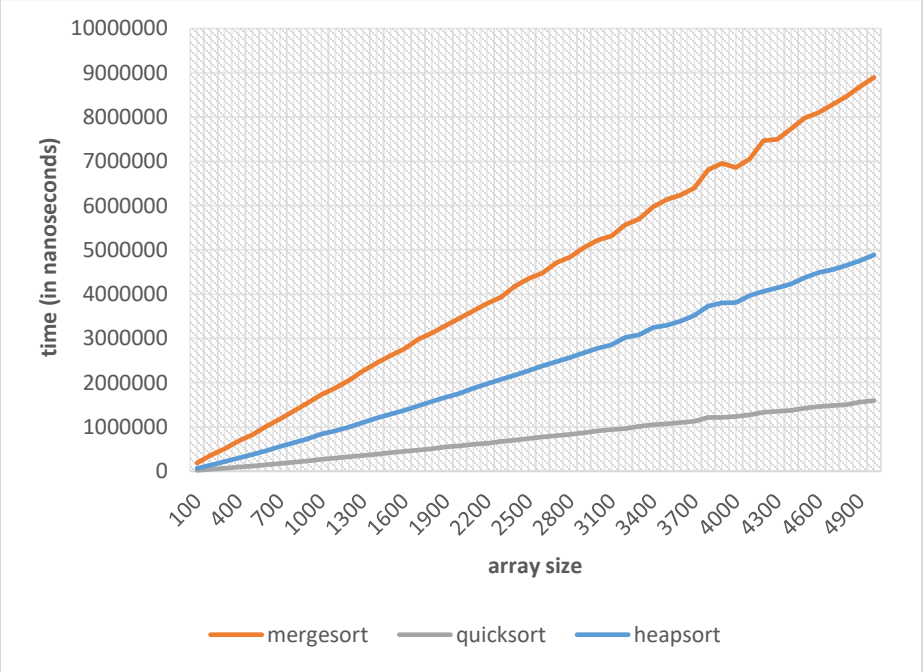


Figure 1: Average time each algorithm took to sort randomly shuffled arrays in range (-n,n) for bigger size arrays, excluding Insertion sort

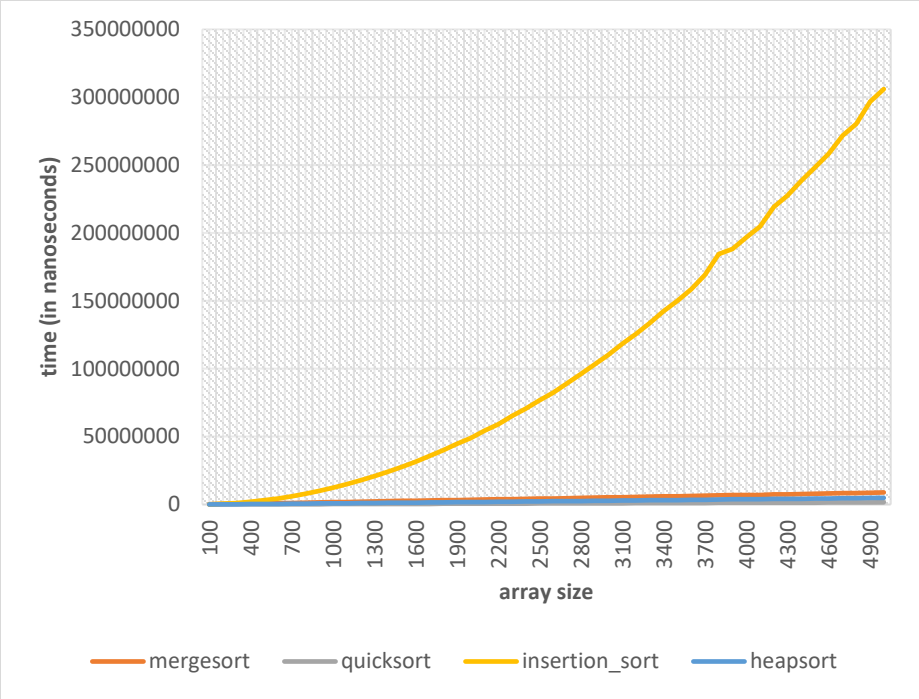


Figure 2: Average time each algorithm took to sort randomly shuffled arrays in range (-n,n) for bigger size arrays

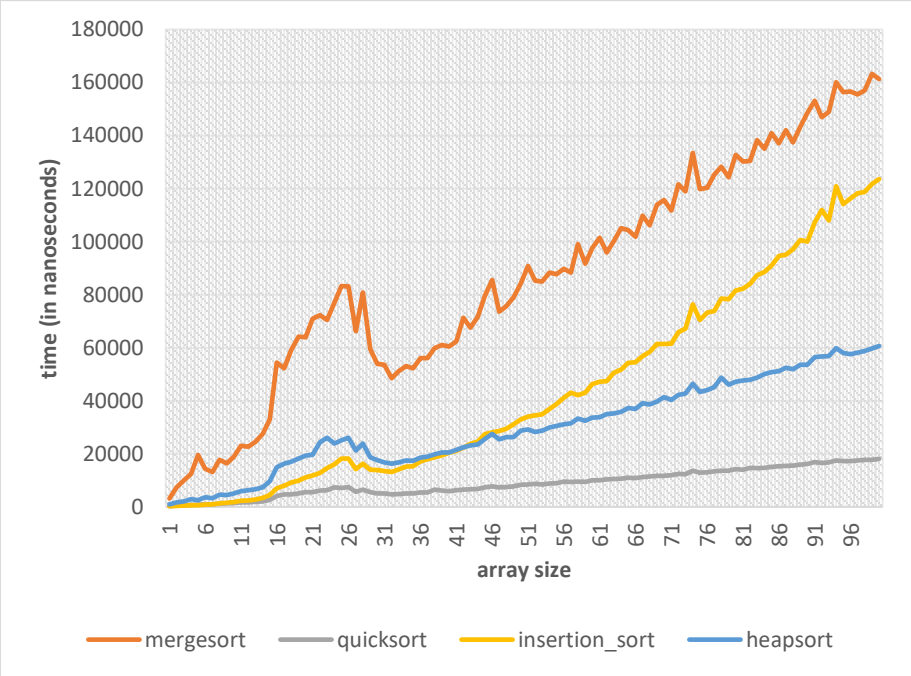


Figure 3: Average time each algorithm took to sort randomly shuffled arrays in range (-n,n) for smaller size arrays