Performing the best for little

Ayna Sultanova

December 21, 2022

1 Introduction

In this paper, we investigate the efficiency of specific sorting algorithms, their performance, and their comparisons: heap sort, insertion sort, quick sort, and merge sort. The comparison will be based on the time the algorithms need to sort an array of randomly generated numbers with different sizes. According to the results, a corresponding hybrid sorting algorithm will be created.

1.1 Heap sort

The heap sort is an algorithm whose main task is to **heapify** a certain array. The idea behind the heap sort is to find the highest value and place it at the end, repeating the process until everything is sorted. This algorithm works with a special data structure called the **heap**.

A heap is a structure with two main conditions: first element with starting index of 0 called top has the largest value (max heap) and the value of the child (min heap) the element cannot be bigger than the value of the parent.

Heapifying is the process where we make a heap from the given array array. After, we swap the top of the heap with its last element and continue doing the same operations on the rest of the array until there is only one element in the heap, which means the array is sorted.

The complexity of the algorithm is the same for all cases – $O(n \log n)$.

The algorithm is in-place but unstable.

1.2 Insertion sort

Main principle of insertion sort is to take one value, iterate through the array and find its correct position in the array. In each iteration, the algorithm compares the current value with the value before it. If the current value is greater than the value before it, the current value remains the same. Otherwise, if the current element is smaller than its predecessor, compare it to the values before. Move the greater values one position up to make space for the swapped value.

Insertion sort is used when the number of elements is small. It can also be useful when the input array is almost sorted, and only a few elements are misplaced in a complete big array.

The average complexity of insertion sort is $O(n^2)$. The insertion sort has a best case: when values in an array are sorted in ascending order. In this case, the time complexity of the insertion sort is $O(n)$. The worst case of this algorithm is when values are sorted in descending order. In such case time, complexity is $O(n^2)$ same as the average case.

The algorithm is stable and in-place.

1.3 Quick sort

Quick sort is a recursive partitioning. The main part of this algorithm consists of partitioning. First, we choose a value called the pivot point from the array. In our case, our pivot point is the first value. The array is reordered in a way that all values smaller than the pivot value are moved before it and all values larger than the pivot value are moved after it, with values equaling the pivot going either way. For this, we create left and right indexes. We assign the left index to the position of the second element, the right index is the last value of the array. We start comparing right index values with pivot. If it is bigger, we decrement right and go to the next element of the array. If it is less than we increment the left index, then we swap left and right index values. And repeat these operations until the right index reaches the pivot. After, we swap the pivot value with the left index element incremented by 1.

The above step is repeated for the left array of smaller values as well as done separately for the right array with greater values. At this point, elements are already sorted. Finally, elements are combined in the sorted array.

In the best case, quick sort is $O(n \log n)$ and the average case is the same.

The quick sort has a worst case, which is if after each recursive call the pivot ends up in the first or last position in the array. In these cases, quick sort performs in $O(n^2)$ time.

Quick sort is a stable and in-place algorithm.

1.4 Merge sort

Merge sort is a sorting algorithm, which is commonly used in computer science. The idea behind merge sort is theoretically simple: divide and conquer. It makes two sorted arrays from an unsorted one, then merges them into a sorted array. Merge sort in most cases contains two parts: dividing and merging.

In the first part, we divide an array into two parts and recursively call the same division function. We pass a pointer to the array, the index of the first and the last element as parameters of the function (or the second pointer might be a certain value). In the second part, we divide the given array into left and right arrays and we fill them in with the values from the main array. Then the main array gets filled with the values of these arrays in a sorted way.

Merge sort keeps on dividing the array into equal parts until it can no more be divided. By definition, if it is only one element in the array, it is sorted. Then, merge sort combines the smaller sorted arrays keeping the new array sorted too.

Merge sort is a stable sort which means that the same element in an array keep their original positions concerning each other. The overall time complexity of merge sort is $O(n \log n)$ and there are no worst or best cases for this algorithm. Additionally, merge sort is stable, but this algorithm is not in-place, which means it requires additional memory.

2 Hybrid sorting algorithm

In this paper, the hybrid sorting algorithm will be based on the quick sort and insertion sort.

Insertion sort in general performs worse than other algorithms, but it shows some good performance on the small arrays. Quick sort in general performs better than the other algorithms, but it performs worse than the insertion sort in small arrays with certain sizes.

In Figure 2, you can see that, when quick sort and insertion sort approach values 55-65, they perform the same, but after 65 the best performance goes to the side of quick sort.

Based on this information, we combined the code of insertion and quick sort. Arrays with sizes up to 65 will be sorted by insertion sort. Arrays with sizes 65-10000 will be sorted by quick sort.

3 Methodology

The algorithms were implemented in C++. The algorithms were tested on arrays of sizes from 100, 200, 300, . . . , 10000.

The results are presented in nanoseconds and represent the average time each algorithm takes to sort the array.

4 Results

Graphs of results for the average case of all sorting algorithms are presented in Figures 1, 2,3, and 4. Insertion sort takes several times more in general, but in small arrays, with sizes up to 60-65, it performs better than others (Figure 3). Quick sort takes the least time compared to others, but merge sort and heap sort performances are slightly the same on large arrays. In small arrays heap sort and merge sort take longer than other algorithms. Heap sort and merge sort perform almost the same, but with a minor difference.

While quick sort performs slower than insertion sort in arrays with sizes up to 60-65 (Figure 3), insertion sort later takes quadratic time to sort arrays with larger sizes, exceeding merge sort and heap sort (Figure 1). Using these results, a hybrid sorting algorithm was created based on insertion sort and quick sort.

In general, hybrid sort performs better than the other algorithms, showing the best time performance (Figure 2). The Hybrid sorting algorithm also shows its best on small arrays, sorting the arrays faster than insertion sort, quick sort, merge sort, and heap sort (Figure 4).

5 Conclusions

Insertion sort in the general case was the worst sorting algorithm. Merge sort, heap sort and quick sort generally performed the same.

The hybrid sorting algorithm shows the best results in all cases.

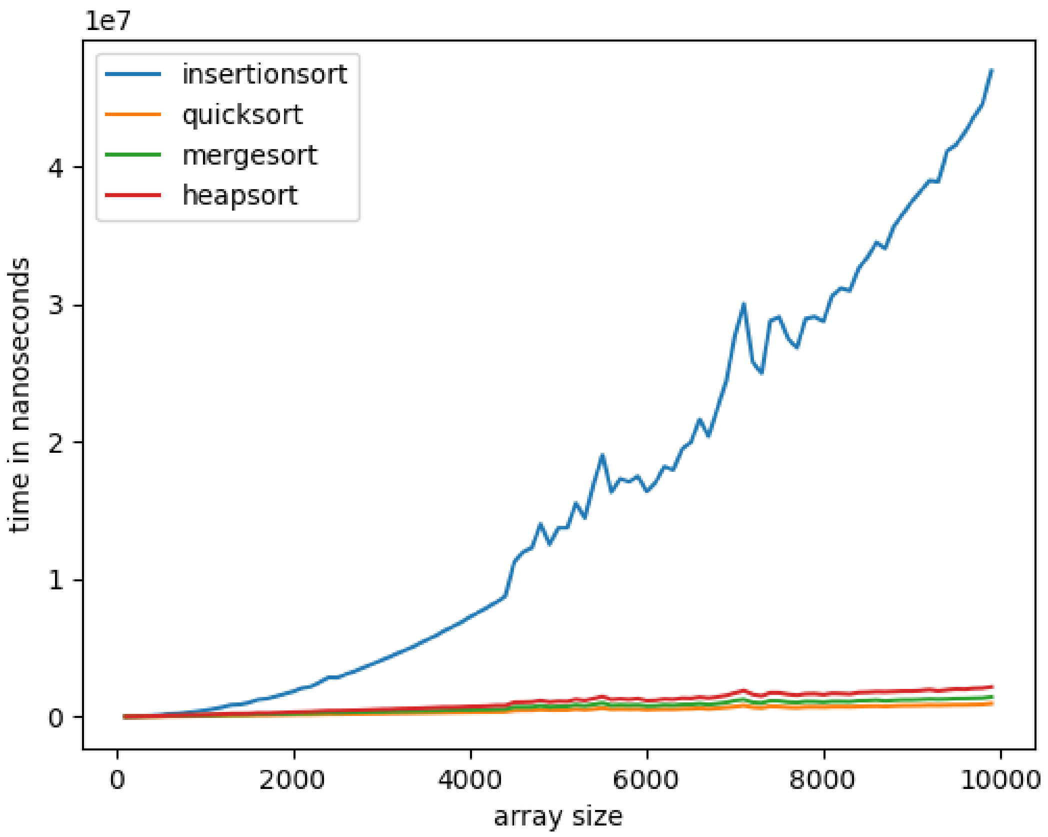

Figure 1: Average time each algorithm took to sort arrays with sizes up to 10000 elements

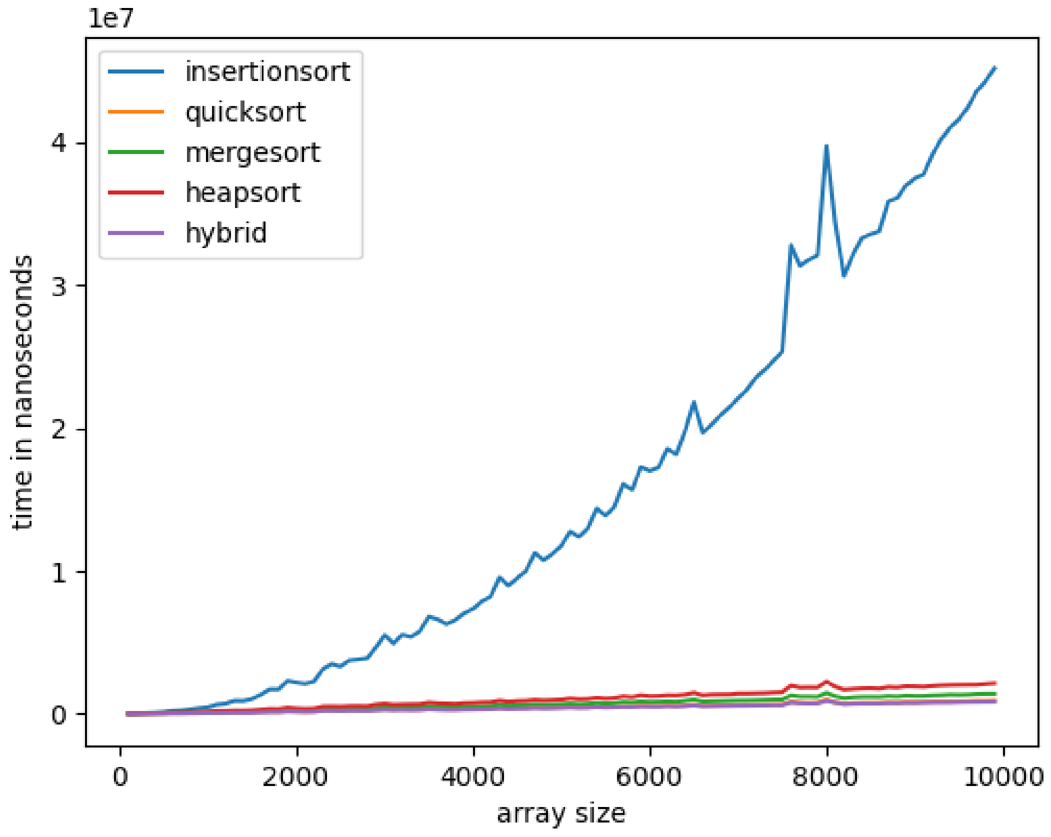

Figure 2: Average time each algorithm (including hybrid sort) took to sort arrays with sizes up to 10000 elements

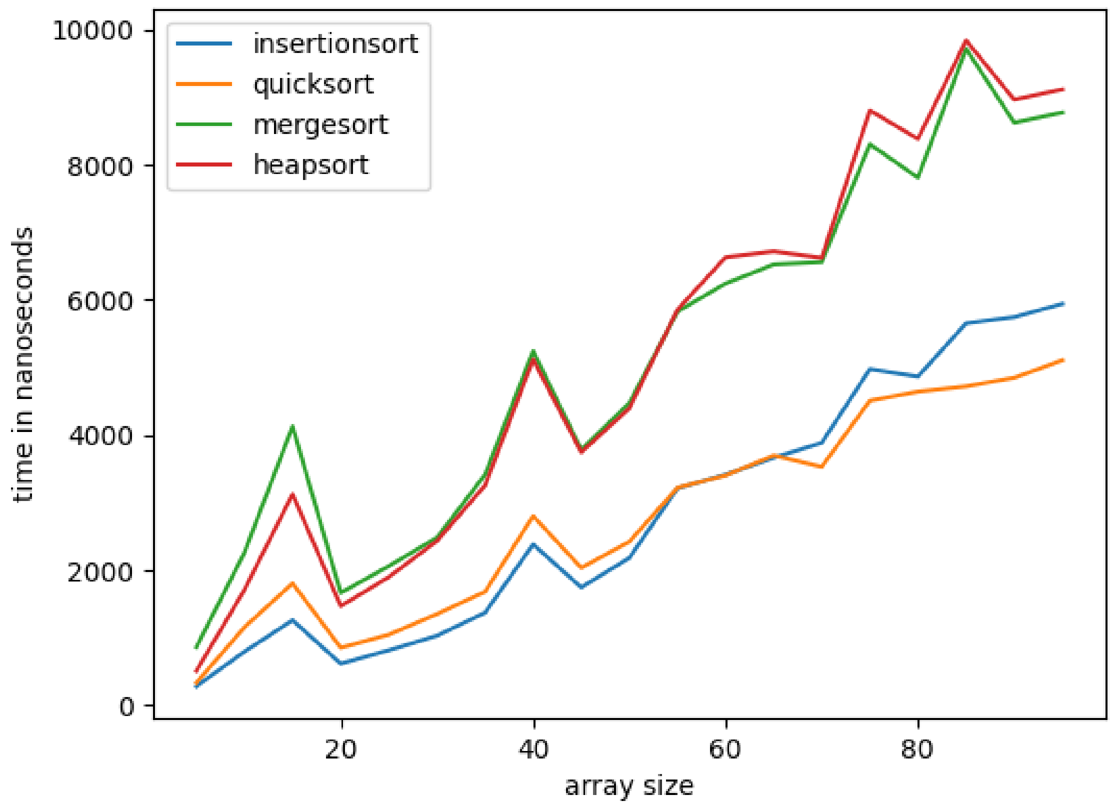

Figure 3: Average time each algorithm took to sort arrays with sizes up to 100 elements

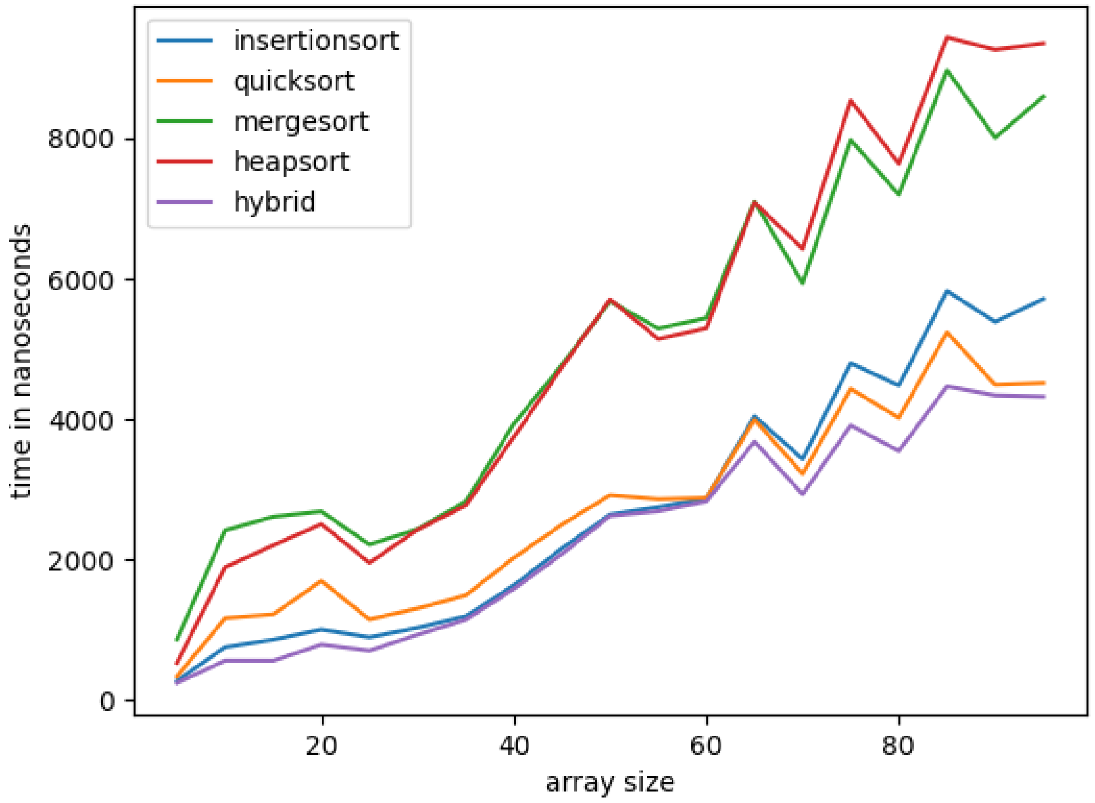

Figure 4: Average time each algorithm (including hybrid sort) took to sort arrays with sizes up to 100 elements