**Sorting quickly: comparison between insertion, merge, heap, and quick sort**

Siarhei Miachkou

December 04, 2022

1. **Introduction**

In this paper we approach the problem of sorting: that is arranging an array of n integers in the ascending order. The final goal of this research is to find the fastest sorting algorithm either by choosing from the existing ones or combining a multiple of them.

We will investigate four of the well-known sorting algorithms: insertion sort, merge sort, heap sort and quick sort.

**1.1 Insertion sort**

Insertion sort algorithm takes the complex problem of sorting element by element. We iterate through the array finding the right place for each element individually assuming that everything before it has been already put in the correct (here ascending) order. When we approach the given integer, we “remember” it and go back to the beginning of the array comparing our integer to all of it predecessors. There are only two possible cases: if our integer is smaller, we copy the compared element to the next position moving it up the list and go on searching, if our integer is bigger or equal, we place it to the right of the compared element and stop the iteration. Once we fall out of the list, we know that the number we have chosen is the smallest one, so we can put it at the very first position since all other elements moved to the right by exactly one spot. Once we apply this procedure to all numbers from our array, we will sort them in the ascending order. Average complexity: O(n^2), Worst case: O(n^2), Best case: O(n). The algorithm is in-place and stable.

**1.2 Merge sort**

The core of the merge sort algorithm (as can be seen from the name) is merging. By merging we mean taking two sorted arrays and “glue” them together keeping the integers sorted. This is a much simpler task than sorting, here’s how it’s done: we go independently through our arrays picking out the smallest elements. We do not compare the integers by their position i.e., 1st with the 1st, 2nd with 2nd and so on. Our iteration goes farther through the array only when we take a number from it. Once we reach the end of one of the arrays, we simply copy all the remaining elements from the second one to our combined array, achieving the sorted collection of all numbers from both arrays. What merge sort does is essentially this but many times. We break the array we were given in two until we end up with the sorted parts (single elements) and then we go back up by merging all of them with each other, building our initial array back while also sorting it in the process. Average complexity: O(n \* logn). No worst or best case. The algorithm is not in-place (O(n) space complexity) but stable.

* 1. **Heap sort**

Heap sort algorithm sorts the array treating it like a heap structure. Heap which is used for sorting has some of the key features required: each element has at most two children; each parent element is bigger than or equal to all its children. First, we need to arrange our array in this way. For this we assume that it is indeed a heap and then check it from bottom to top fixing all the mistakes. We start from the first element that has any children and check if it is bigger than or equal to them as the heap structure requires it to be, if there’s a child bigger than its parent, we swap their values and go down the affected root recursively to check if the heap didn’t get ruined by doing similar checks and swapping more elements if needed. Once we get to the top element, we will have the heap in which all parent elements are bigger than their children. Now we can use this heap we have just built to sort our array. The element at the top of the heap is the biggest element we had in our list, so we swap it with the last element, sending the largest integer to the end. Most likely this operation ruins our heap, so we need to fix it again, but now going from the top. We can ignore the last element since it’s already at the right position, so we decrease the number of elements that take part in our heap by one. Once our heap is empty, we end up with a sorted array. Average complexity: O(n \* logn), Worst case: O(n \* logn), Best Case: O(n). The algorithm is in-place but not stable.

* 1. **Quick sort**

Quick sort algorithm sorts array one can say gradually arranging elements to left and to the right sides of the pivot. The pivot is just one of the array’s elements and there are several ways to choose it, but to see the difference that comes from the option chosen we need to understand the process of sorting first. For now, let’s always make the first element of the array our pivot and describe the Hoare method of quick sorting. We make two pointers, the first one (left) we put next to our pivot, the second one (right) we put at the end of the array. We want our left pointer to find the element which is larger than the pivot and the right pointer to find the element which is smaller than the pivot and the swap their values. This will be a step in a right direction since the bigger number must appear closer to the right and vice versa. If our pointers cross their ways, we can no longer do any swaps since left pointer is now to the right of the right which makes zero sense and swaps like this will only do us harm. So, we got everything we could from our pivot and now we swap its value with one of our right pointers because we are sure that everything to the right of it is now bigger than the pivot and everything to the left is smaller. Thus, our array is “more sorted” than it was and what we must do next is recursively run the same procedure on the parts of the array now separated by our pivot value. These parts will get smaller and smaller and once they are just single elements our array is sorted. This is extremely effective since it divides our array into much smaller parts, or does it? The dilemma of choosing the pivot is that it is important where you end up putting it after running the Hoare procedure. For example, if every time our pivot ends up right next to where it was, we will have the array of almost the same size to sort once again which is a terrible blow to the performance of the algorithm (O(n^2) complexity). Taking the first element as the pivot is especially risky since it might be often that we run our algorithm on almost sorted arrays making the worst-case scenario very plausible. There are two ways to avoid this: take middle element as your pivot (good for sorted arrays) or take the random element as your pivot (the best practical solution). And with this issue out of the way quick sort algorithm becomes an extremely powerful tool for sorting arrays. Average complexity: O(n \* logn), Worst case: O(n^2), Best case: O(n \* logn). The algorithm is in-place but not stable.

1. **Methodology**

All algorithms were implemented in C++. They were tested on arrays of sizes 100, 200, 300, …, 10000 and of sizes 1, 2, 3, …, 200. The arrays were filled with the random numbers form the range of (-array size / 2, array size / 2). Each array size was tested 100 times. The result for each size is the average time it took each algorithm to sort the given array.

1. **Results**

As it was expected for arrays of the large sizes insertion sort performed severely worse than three other algorithms (Figure 1). Between three algorithms all having O(n \* logn) complexity merge sort stands out in a bad way taking noticeably longer to do the sorting, and it is very clear that quick sort is the best performing algorithm (Figure 2). However, it is also important to investigate what happens with the smaller arrays. Here insertion sort as a simpler algorithm beats quick sort approximately up until to the point when array size becomes more than 75 (Figure 3). This means that to get the fastest sorting algorithm we must combine the simplicity of the insertion sort which allows it to sort small arrays faster with the effectiveness of the quick sort so much needed for larger arrays. For this a hybrid sort algorithm was created. The idea is simple: if array has less than 75 elements use insertion sort otherwise use quick sort. Our hybrid sort algorithm outperformed the previous leader quick sort by a noticeable margin (Figure 4) and no longer has an issue of being slower than insertion sort when dealing with small arrays (Figures 5). Average complexity: O(n\*logn), Best case: O(n), Worst case: O(n^2). This algorithm is in-place but not stable.

1. **Conclusions**

By running the tests, we found what sorting algorithms are the best for the specific array sizes and by wisely combine them in a way that highlights their strong features and eliminates their weaknesses we came up with a sorting algorithm that is superior to all the others. We can conclude that there is no sorting algorithm which is always the best, so the way to achieve the fastest sorting for all the array sizes is to combine several of the sorting algorithms into one hybrid sort algorithm that carefully chooses which one is to use for each case to have the best possible performance.

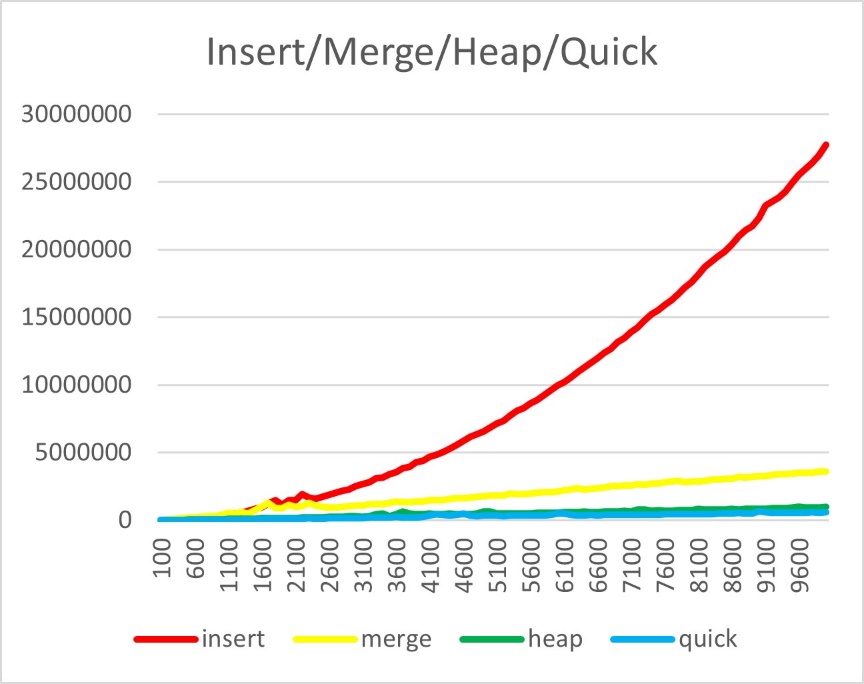


Figure 1: All 4 algorithms for large array sizes

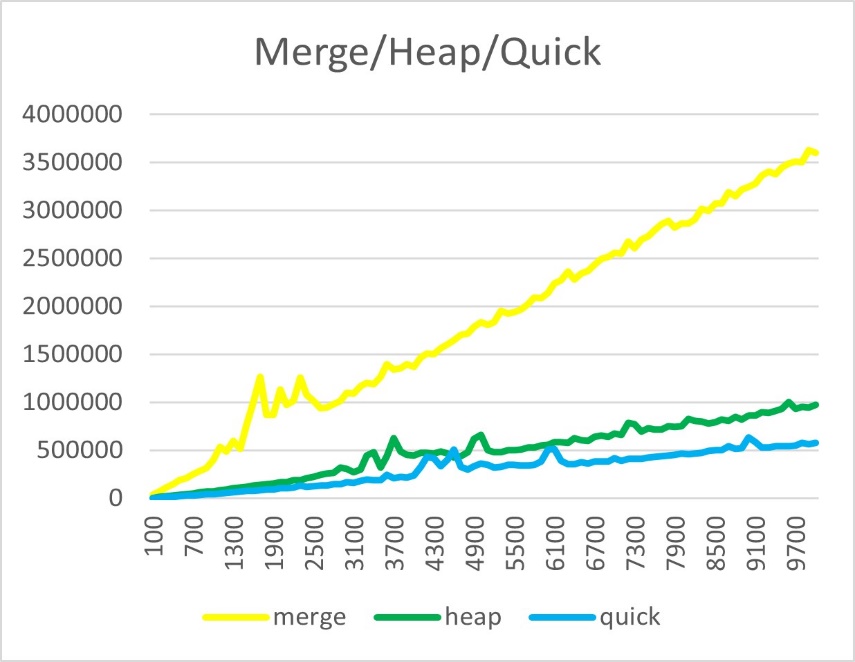


Figure 2: Merge, heap and quick sort compared

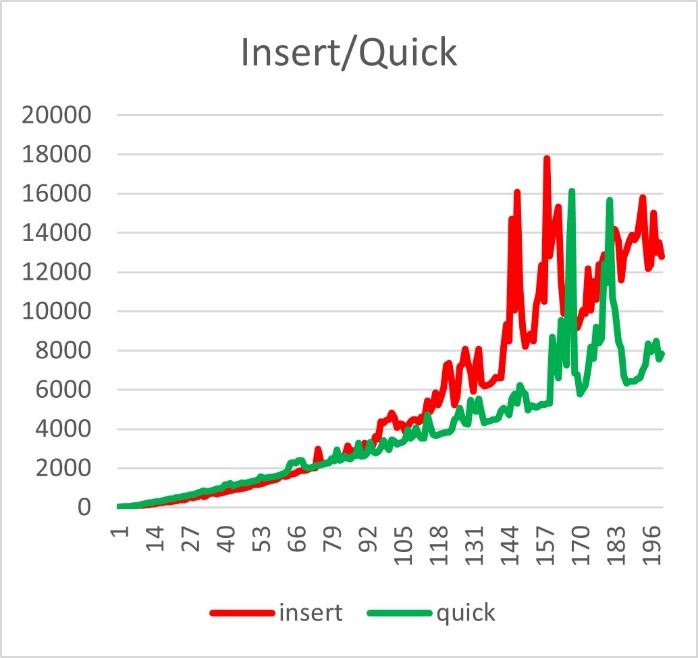


Figure 3: Insert and quick sort performance for small arrays

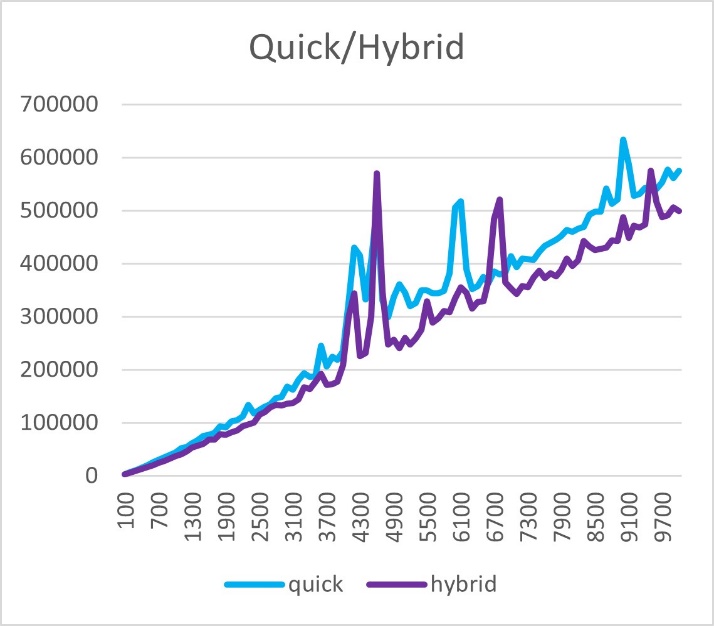


Figure 4: Quick sort and hybrid sort comparison for large arrays

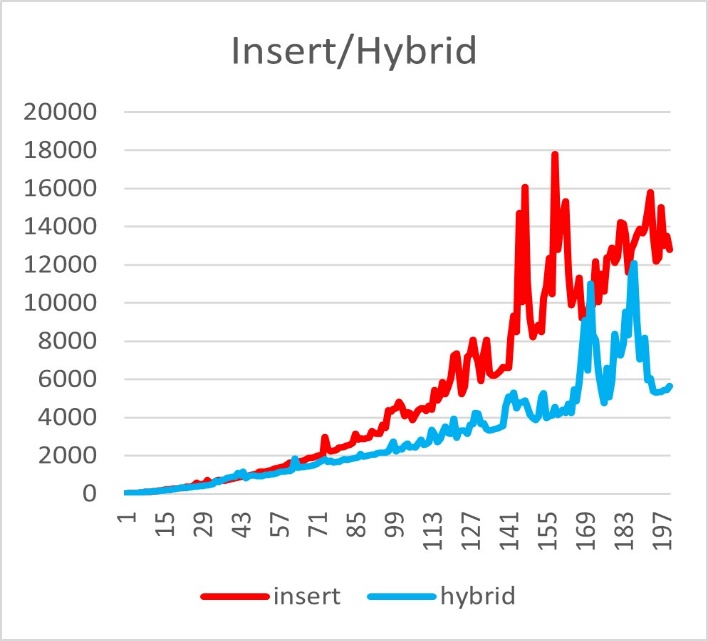


Figure 5: Hybrid and insertion sort for small arrays