Comparing three algorithms:

merge sort, quick sort, heap sort

Khashimov Akmalkhon

January 19, 2021

1. Goal

Our goal: find out the most efficient sorting algorithm for various input sizes among merge sort, quick sort, heap sort. CRITERIA: Time complexity, space complexity

2. Description

The algorithms below sort data in non-decreasing order.

Merge sort: recursively divide the array into two parts until we reach the array size of 1, and continuously merge the arrays into sorted ones. We only swap elements if they are in the wrong place. Attributes: *Not-In-Place, Stable, Non-Adaptive*.

This algorithm must always recursively reach each element of the array to sort it, for this reason the *best case, worst case, and average case* of this algorithm is always *O(n*log(n))*.

Heap sort: use the heap data structure to have the maximum element in the array and continuously put that maximum element into the end of the array and decrease the size, which obviously finally gives us sorted array. Attributes: *In-Place, Non-Stable, Non-Adaptive*.

*Best, worst, average case: O(n*log(n)).*

Quick sort: find elements in the input array which are greater and smaller than a chosen element from the array and swap them. Recursively repeat the action for each half-part of the array, divided by the previously chosen element. Partition schema used: Hoare's partitioning. Attributes: *In-Place, Non-Stable, Non-Adaptive*.

Since quick sort must make sure that each element is in its sorted place, this makes the algorithm's *best case* the same as its *average case* O(n*log(n)) with minor actual difference in time. Quick sort's worst case is when the input array is sorted in inverse direction to what the algorithm is intended to. This makes for each element pass through the whole array. This adds another loop under the loop we already have in quick sort through the array, which makes the *worst-case* O(n^2), which is terrible. Before using quick sort, better make sure the array is not sorted inversely to what is intended.

P.S. I could have used many variations of the algorithms above with various minor improvements, but I will still have the same core of sorting, which means that it shall not make major changes in the results we get. For this reason, I just picked the classic Hoare's partitioning using the first element as the pivot (while we can use multi-pivot).

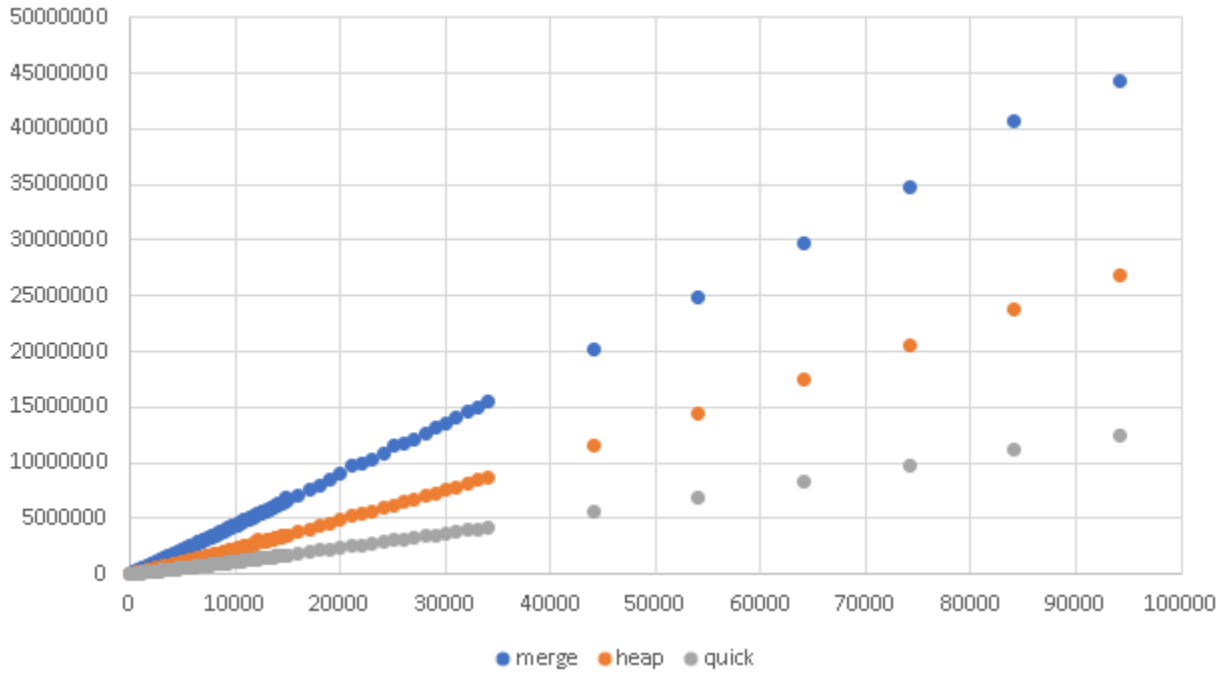
3. Methodology

Sum up mean of total time taken for 1000 times implemented for generated data for each algorithm for sizes: 50-10000. The same data was used for all algorithms for each size. Language used for algorithms and measurement: C++. I used Hoare's partitioning schema. When it comes to picking the pivot in quick sort, even though I could pick a random element or median of first, last, and middle element, which would statistically reduce probability that I would get a worst case, however I just pick the first element of the array, since I am sure that I am working with truly (or at least close to truly) random numbers. But in real life cases, where data may be and mostly is – truly messy, it is a good idea to implement median of three (or even 5 – 0%,25%,50%,75%,100%), of course it depends on the data you work on.

4. Results

Throughout the project, horizontal axis is used for the array size and vertical axis is used for nanoseconds the algorithm took for that specific array size to sort, unless otherwise stated.

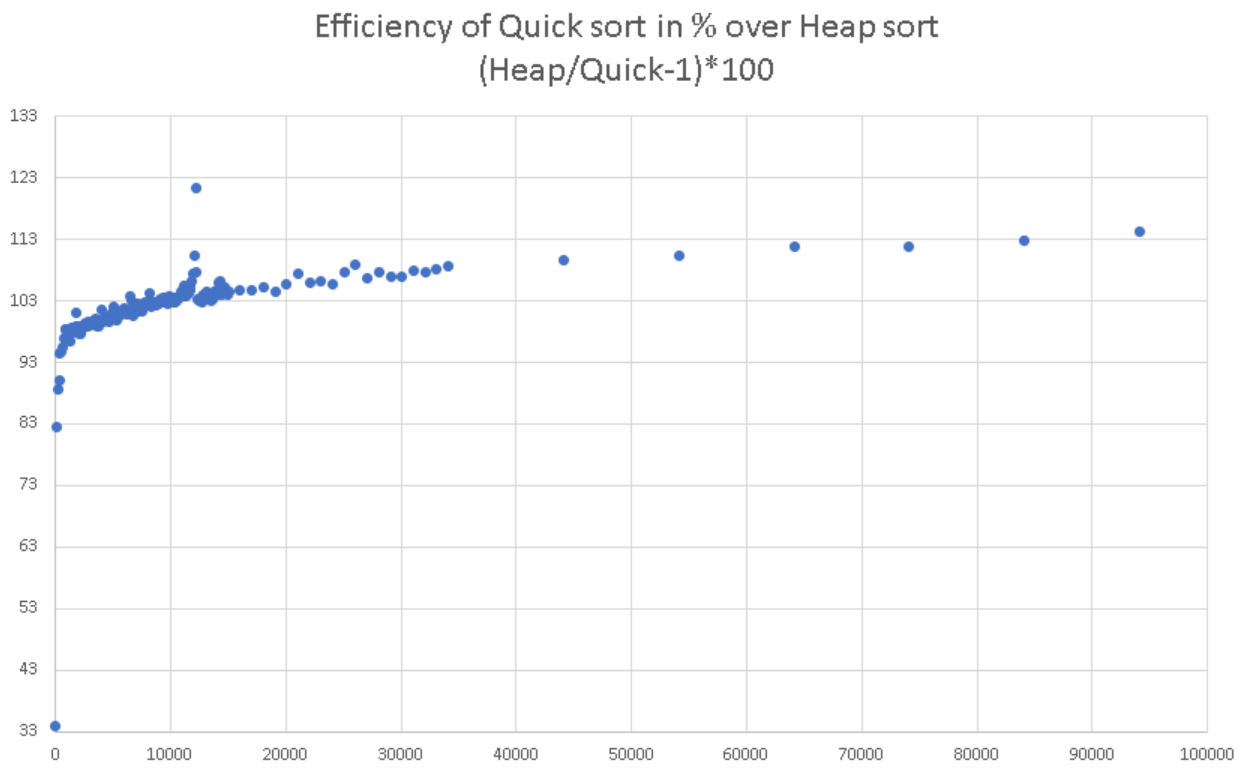
I got the time data generated for each algorithm – let us look at its visualization below.



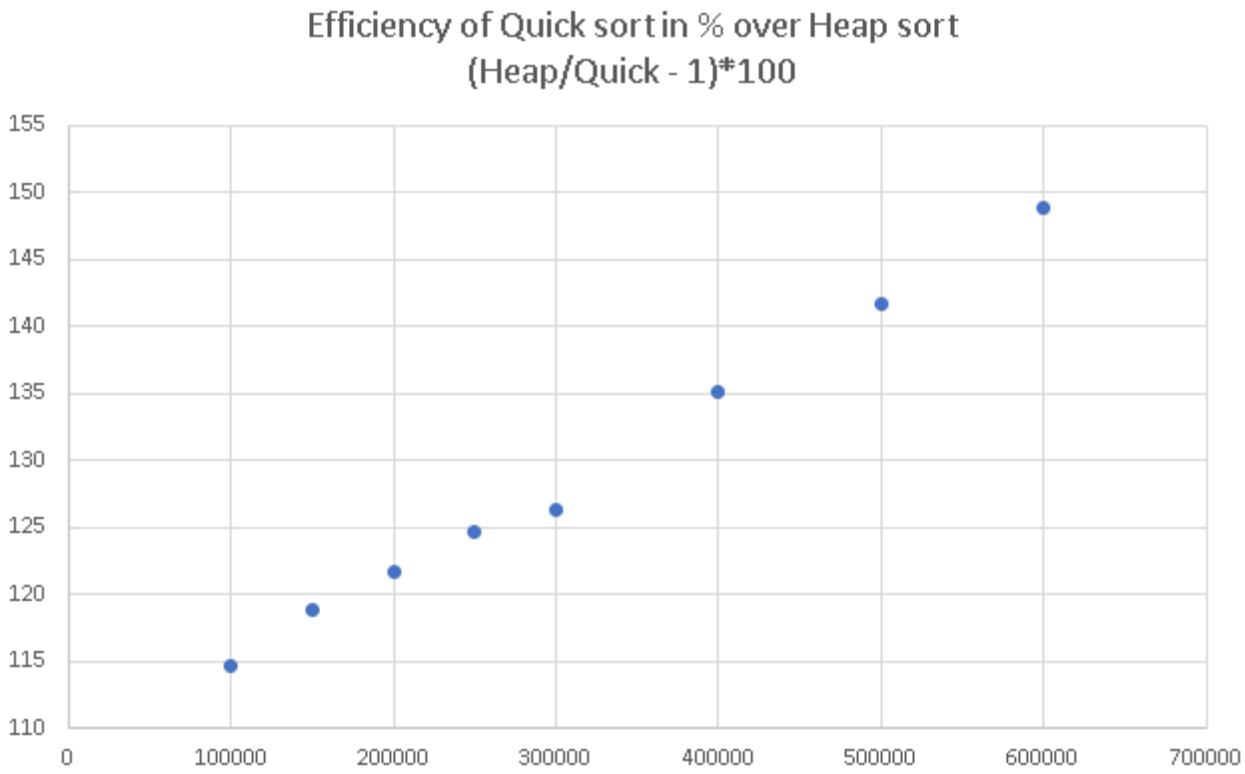
The graph clearly shows the trends of each algorithm. We clearly see that quick sort is the most efficient and the merge sort is the least efficient, heap sort standing in between.

However, in modern world nobody believes into words, numbers are the subject of interest now.

Let us compare quick sort to heap sort and heap sort to merge sort. As heap sort is present in both comparisons, we can compare quick sort to merge sort later, if it is necessary.



There seems to be outliers in the array size less than 5000, however with closer look at grander scales, we can see that…



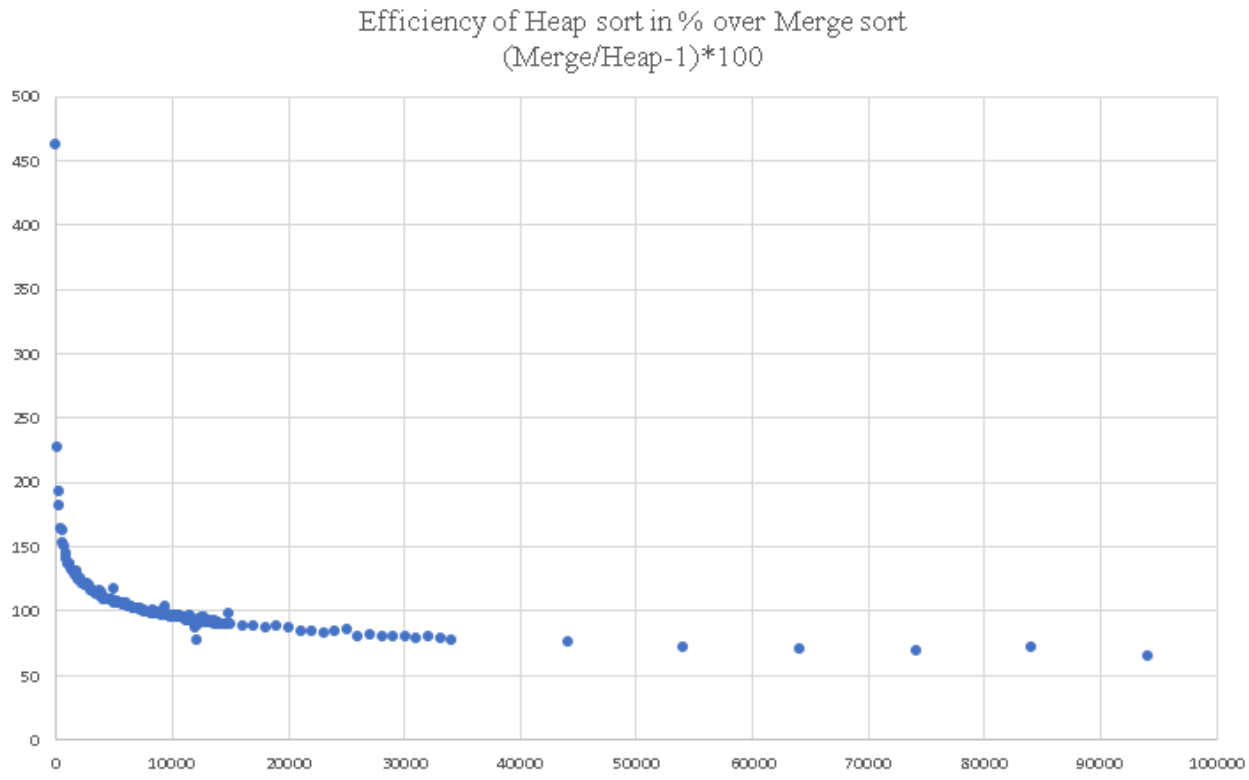
… the input array size increases, quick sort becomes continuously more efficient than heap sort. And those data points are not outliers but rather clear indicators of that.

From my data I can say that on average for each increase of 100000 in input array size, quick sort becomes 5.4% more efficient than heap sort.

Conclusion for Quick sort compared to Heap sort:

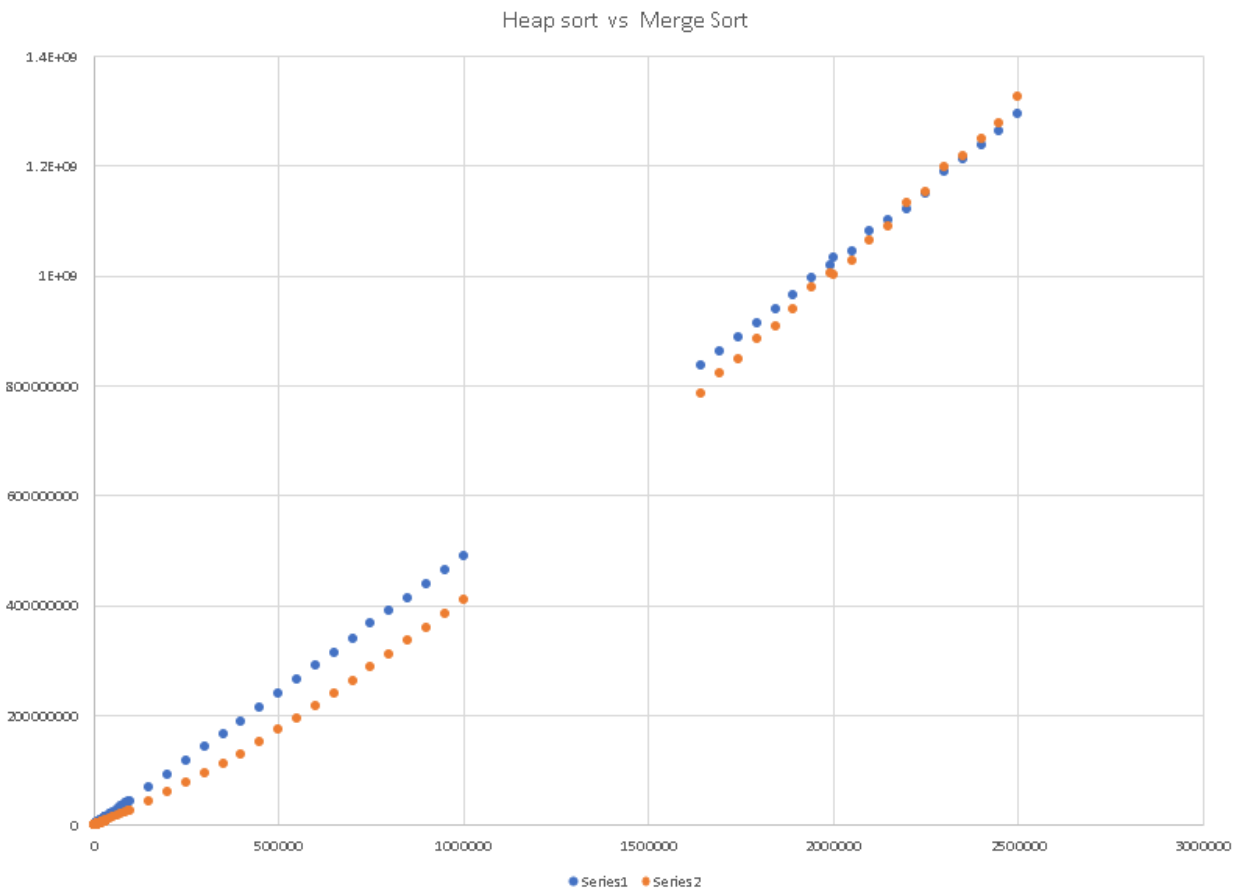
Quick sort is at least 33% more efficient than heap sort and about 90%+-15% more efficient for small (10-1000) and medium scales (1000-10000), but in grander scales with increase in the sorting size, quick sort's efficiency over heap sort grows linearly, at approximate rate of5.4% per 100000.

Next: heap sort vs merge sort.



Now that is interesting. Does heapsort and merge sort become equally efficient at some point? We need to take a bigger picture and test even grander scales.

One night of calculations, and here is the result.

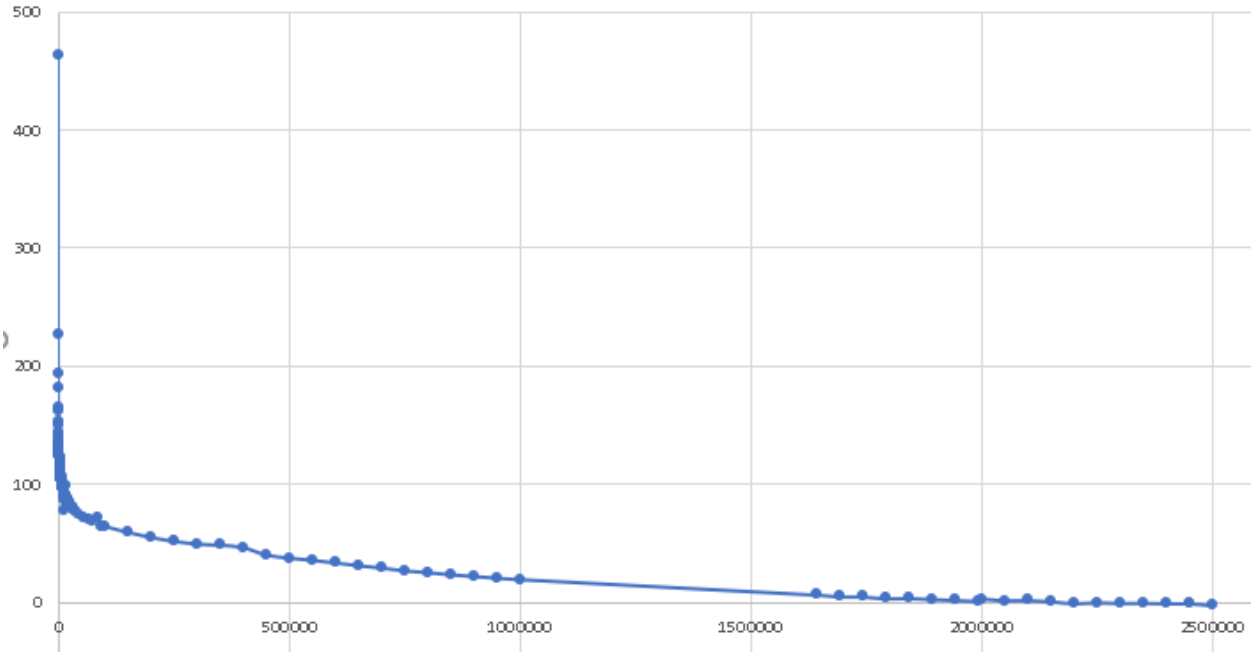


Indeed, the trend is clear. Blue represents MERGE SORT and orange represents HEAP SORT.

Looking at data, merge sort becomes more efficient starting input array size of approximately 2250000.

Conclusions for heapsort vs merge sort:

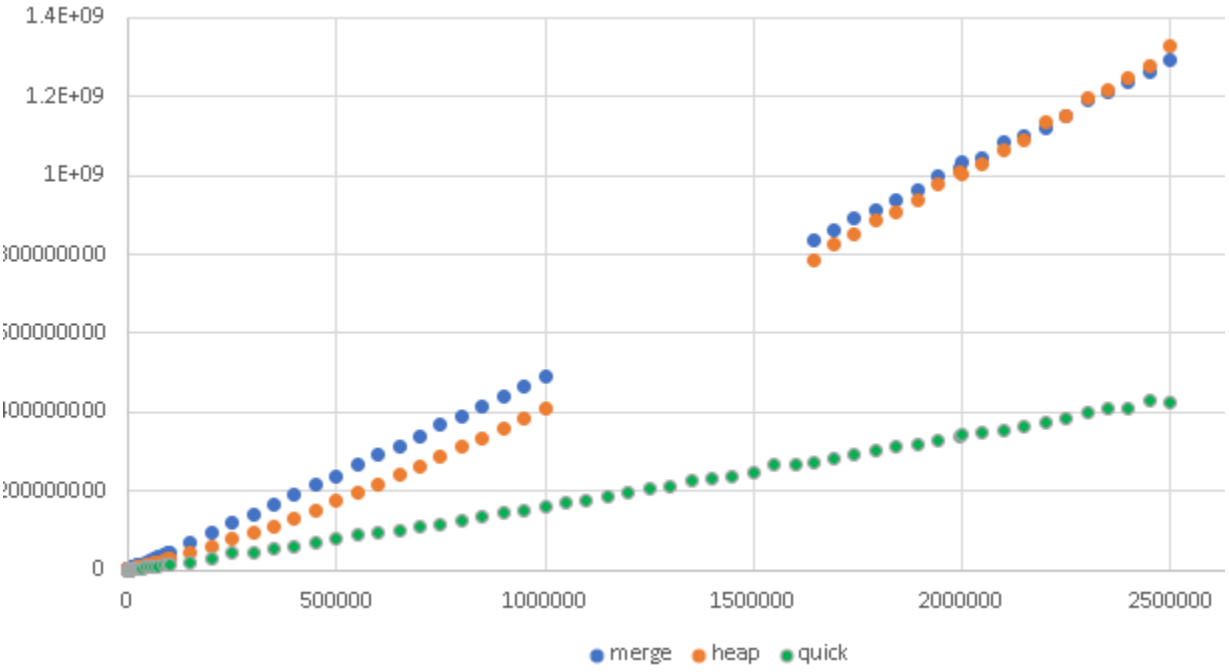
Using heapsort is more time efficient than merge sort for array size less than 2250000, however the further we increase the size of the array, the change in time efficiency might not be significant enough to pick heap sort and you might want to prefer stability of merge sort. For this case, here is the graph of the tradeoff for you below.



Vertical: Heap sort efficiency over merge sort, in %; Mathematically: 100*(Merge/Heap-1)).
Horizontal: Size of array. How much efficiency are you ready to tradeoff for stability with additional space complexity?

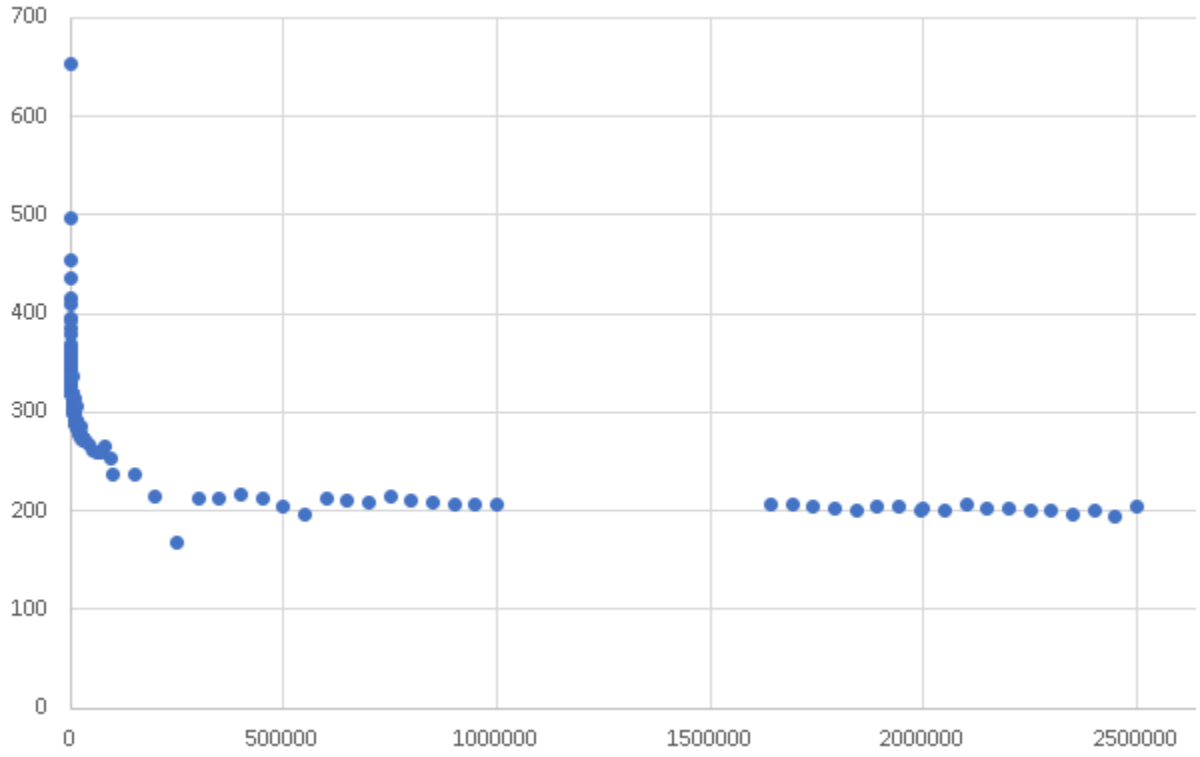
5. Conclusions

Finally let us get the big picture of all the algorithms.



Even considering that heap sort is more efficient than merge sort in range 0-2250000 and it is in-place compared to merge sort with its O(n) space complexity, we have a much better alternative of quick sort with the same attributes. Quick sort is significantly more efficient than both heap sort and merge sort both in short run and long run. However, we might need stability while sorting, in this case merge sort would obviously be the choice. Finally, looking at the graph, it is the choice between merge sort and quick sort, depending on requirements.

Here is the graph of quick sort vs merge sort.



Vertical axis: quick sort's efficiency compared to merge sort, in %; Mathematically: 100*(Merge/Quick – 1). Horizontal: input array size.

Obviously for small scale sorting quick sort shows huge efficiency over merge sort, however in larger scales quick sort is on average by 200 % more efficient than merge sort, which actually means three times more efficient. Let us put it this way: ***quick sort is at least and on average three times more efficient than merge sort. But we might want to trade this huge efficiency for stability that comes with additional space complexity of O(n) in some cases, depending on what we want.***

P.S. (There is one downside of quick sort: the worst case. Before sorting using quick sort, make sure that the array is not already in the inverse direction. It is a good idea to use a random element as a pivot, or median of three elements (usually first, central, and last).)