Comparison of sorting algorithms

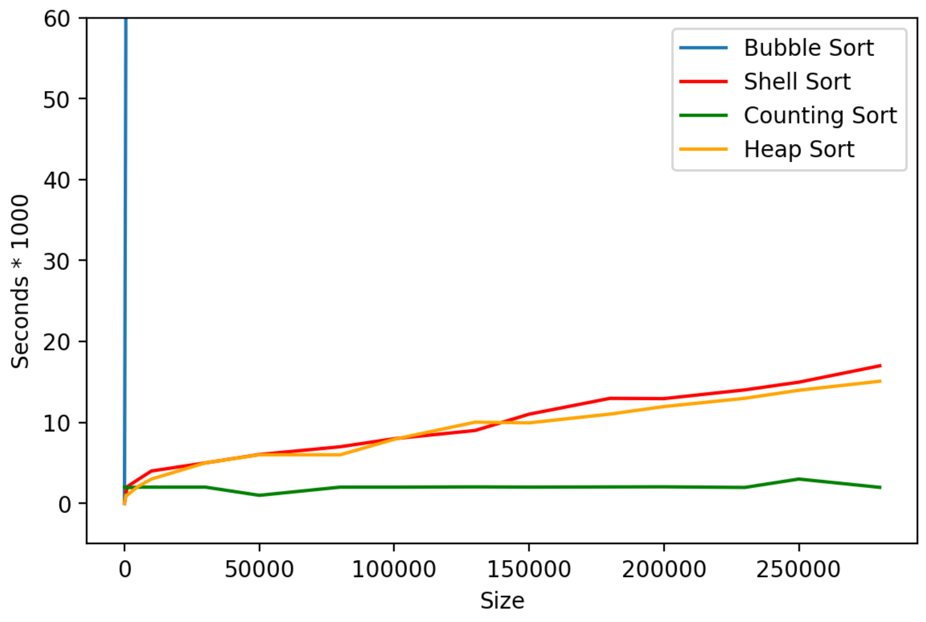
Sebastian Chwilczyński
Karol Cyganik

1. Bubble, Shell, Counting and Heap Sort comparison for arrays filled with random integers.

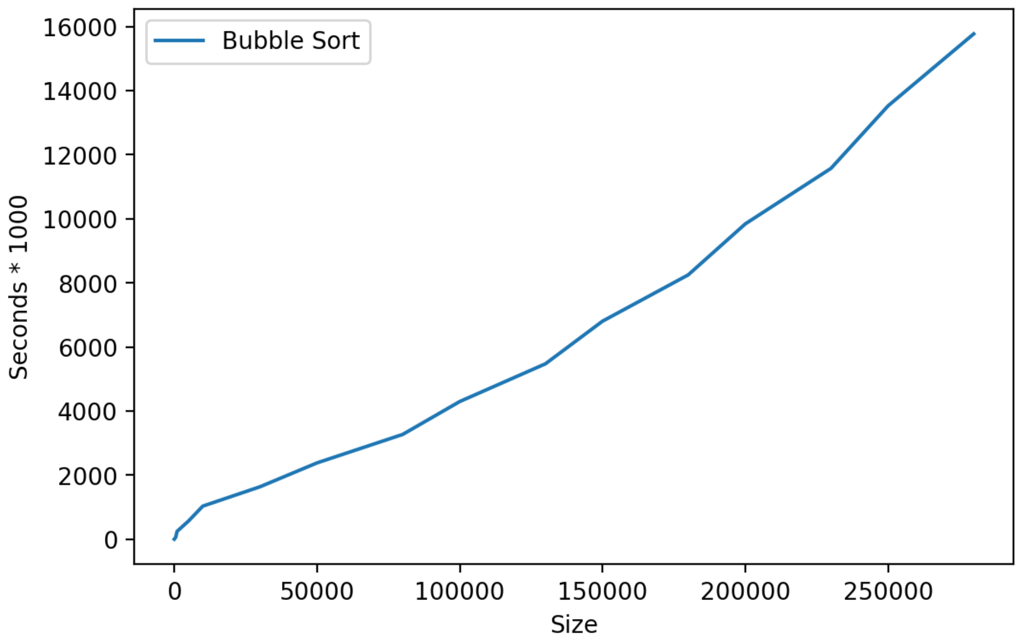
	Bubble Sort	Shell Sort	Counting Sort	Heap Sort
Best case	O(n^2)	O(nlog(n))	O(n+k)	O(nlog(n))
Average case	O(n^2)	O(nlog(n))	O(n+k)	O(nlog(n))
Worst case	O(n^2)	O(n^2)	O(n+k)	O(nlog(n))

Table above shows us the complexity of four algorithms which took part in our experiment. At the first look of the eye we can see, that definitely, Bubble Sort has the worst complexity, while Shell and Heap sort have nearly the same complexities and Counting is non-arguably the fastest one. However, this costs – Counting sort needs a lot more extra memory to run, its extra memory complexity is equal to O(MAX), where MAX is the biggest element in the array.

We selected 20 measuring point using such function $f(x) = x * 5000 + 100$



Too see anything we had to zoom closer and limit y axis values to 60, but having visible differences in other sorting algorithms we cannot see the full shape of Bubble sort so plot below stays only for BS to show its shape.



First thing which can be seen at this plot is a huge difference between Bubble Sort and other algorithms. Its complexity O(n^2), caused by iterating n*n times over the array and swapping elements, is confirmed above on the plot, where the shape of a line is a nice parabola. It's definitely a worst time complexity algorithm and it's not worth to be used because of time it needs to make all computation (at most – comparisons and swaps). But it may be considered if time complexity doesn't matter for someone or he wants a short, working code.

Then noticeable is small difference in Shell and Heap sort. The thing here is that we used uniformly distributed random numbers to fill an array, so we didn't obtain the Shell Sort's worst case. Furthermore we got similar results, because the average case complexity of these algorithms is the same – O(nlog(n)).
The complexities of these algorithms are similar, but the way they work is very different. Shell Sort chooses the 'gap' at the beginning, then compares the numbers from i and $i + gap$ position and swaps them if first was bigger. After one iteration in reduces the gap twice and so on, going to 1. Then it becomes working as Insertion Sort. Heap Sort in comparison uses a data structure called heap, to sort the array. It creates the Max-Heap binary tree from a list and then it: swaps elements, remove them and do heapify – propagate larger nodes up, in a sub-tree it swaps parent with one bigger child.

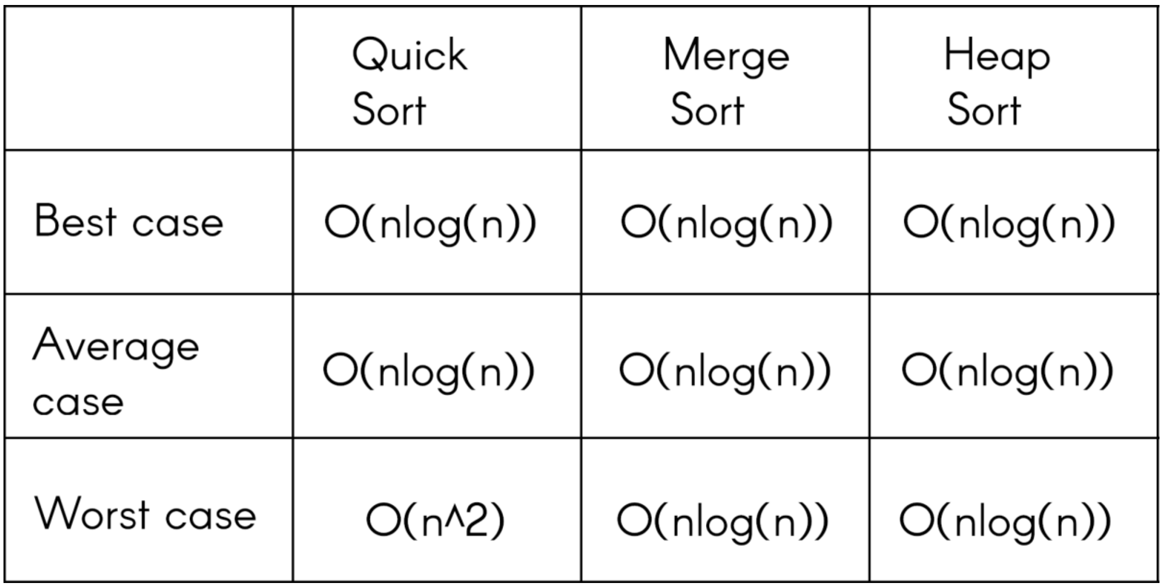
The only algorithm left is Counting Sort. It has the best time complexity and as we can see on the plot, it can't be even compared when we speak about time needed to run the algorithm. Moreover, the biggest n, the most significant is the difference

between CS and other algorithms.
So – why we don’t use Counting Sort all the time? As it’s written above, it needs a lot of extra memory to run. It’s because we need to create as many buckets as the range from 0 to the biggest number in the array. We count every number occurrence in bucket and then print them iterating from 0 to the maximum element. Its memory complexity is a big limitation of this algorithm and that’s why it’s not used very often. So summing up, it’s used when we have small range of integers and we really need linear time complexity.

As CS’s memory complexity is failing, BS’s time complexity is bad and they can be used only in several specific cases which doesn’t occur often, which one should we use – Shell Sort or Heap Sort? They have same extra memory complexities O(1), same time complexities in best and average cases, same time need to run (as it’s visible on the plot), in theory they differ just by worst case tine complexity. First answer which comes to our mind is definitely Heap Sort – better time complexity in worst case, when other comparable parameters are really similar. It’s partially true, Heap Sort is very fast algorithm, which doesn’t need extra space to perform computations, but it has some cons. As data structure is not very stable and it’s not adaptive, what means when it gets partially sorted array, it still has to make computations in some time. So there are not many practical uses of Heap Sort. On the other hand we have Shell Sort which is very similar to HS, but has worse worst case complexity but same O(1) extra memory consumption. Time complexity here really depends on the gap we choose.

2. MS, QS and HS comparison for different data types of the sequence

At first lets recall time complexities of these algorithms, to know what we should expect at all.

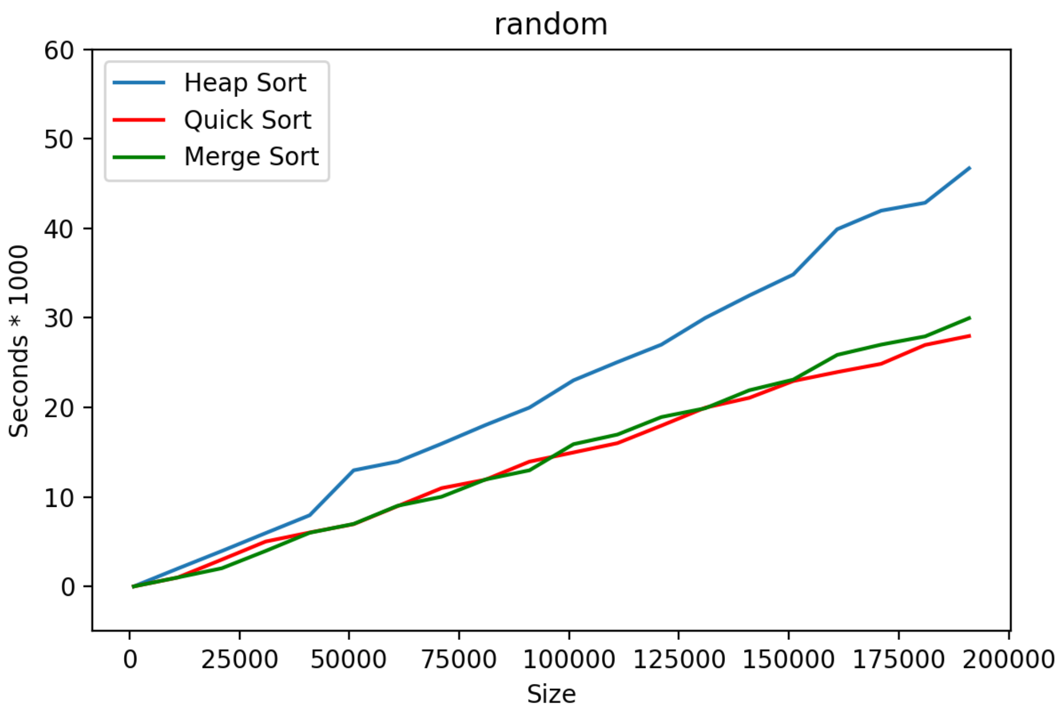


	Quick Sort	Merge Sort	Heap Sort
Best case	O(nlog(n))	O(nlog(n))	O(nlog(n))
Average case	O(nlog(n))	O(nlog(n))	O(nlog(n))
Worst case	O(n^2)	O(nlog(n))	O(nlog(n))

In theory, the only difference is that Worst case in Quick sort differs and is equal to O(n^2). What’s more QS and HP are in-place algorithms so they don’t need any additional memory to perform sorting, whereas Merge Sort is not in-place algorithm and it needs additional array of size n to perform its duty. So on paper it may seem that heap sort is the best out of these three. So let's check it!

We selected 20 measuring point using such function $f(x) = x * 10000 + 1000$

Case 1 – random distribution of the number

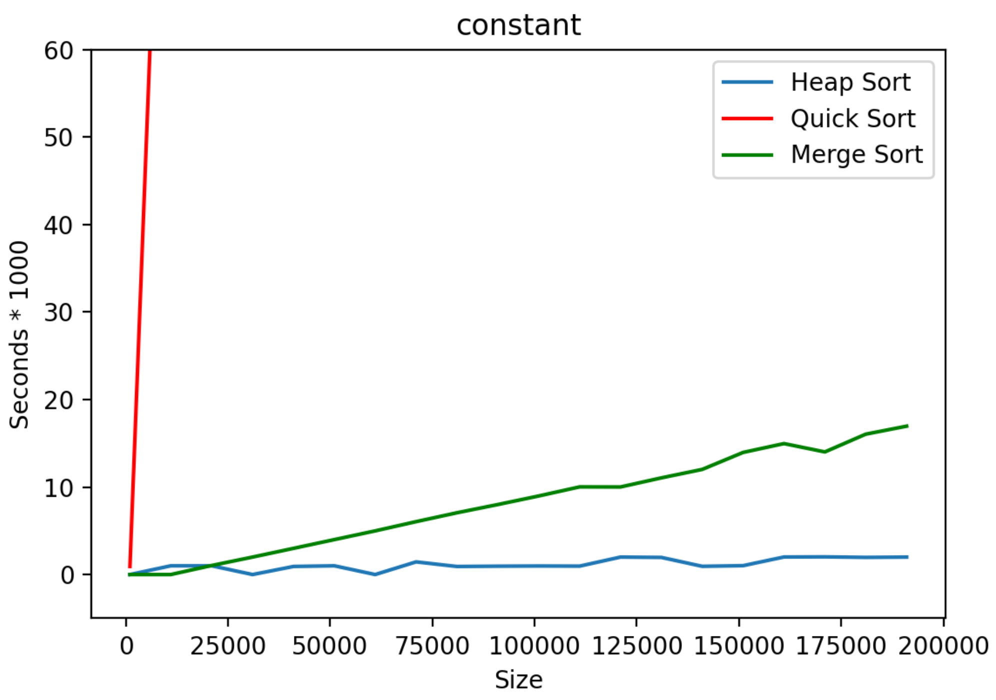


As we can see Merge and Quick sort performance is almost the same, but surprisingly Heap Sort efficiency is much worse and it gets worse in comparison to QS and MS as we increase numbers of elements. But why? Basically while sorting we can do 2 operations, namely we can either compare elements or swap them. We know that swapping is computationally more expensive than comparing and probably that is the reason of the difference. We tried to count how many swaps MS and QS does while sorting array of size n = 111000, we did it by simply increasing counter every time we invoke the swap function. The results are as follows:

```
Size of the array 111000
Heap swaps = 1764137   Quick swaps = 1172011
```

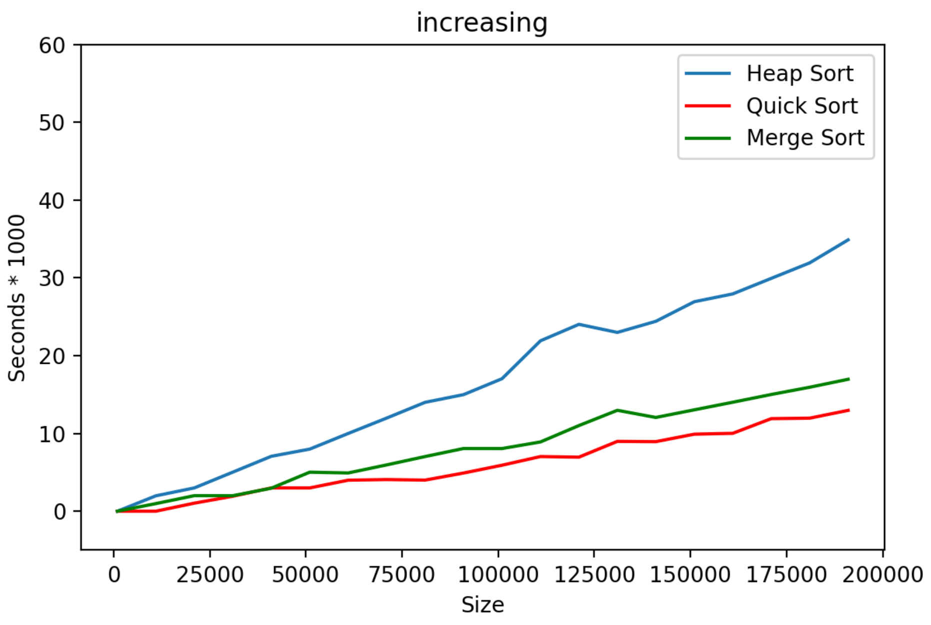
Probably in different data distribution these counter will differ, but it seems that number of swaps is indeed the issue of heap sort. Unfortunately merge sort didn't swap elements at all it just inserts them so it is hard to compare it in this realm. Since merge sort always behaves in the same manner we think that time of its execution shouldn't differ in all cases.

Case 2 – constant value



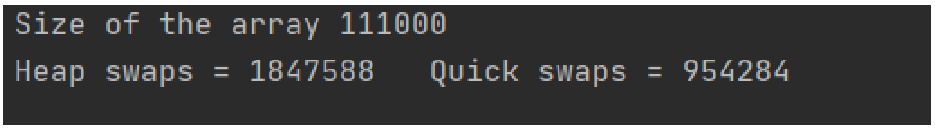
At first glance outcomes looks not so good. Do we hit the worst case of Quick Sort? It turns out so. It is because in every partition there is no element greater or less than our pivot element so our left flag always ends up at the end of the array so our next partitions have size n-1, n-2, n-3 and so on and we know that Sum of all natural numbers from 1 up to n is $\frac{n*(n-1)}{2}$ so the complexity is O(n^2). But what happened in case of Heap sort why its time is like constant or O(n)? Actually this is a special case as we have max heap already given. Values are never pushed down the heap in this situation. So our heapify function takes O(1).

Case 3 – Increasing



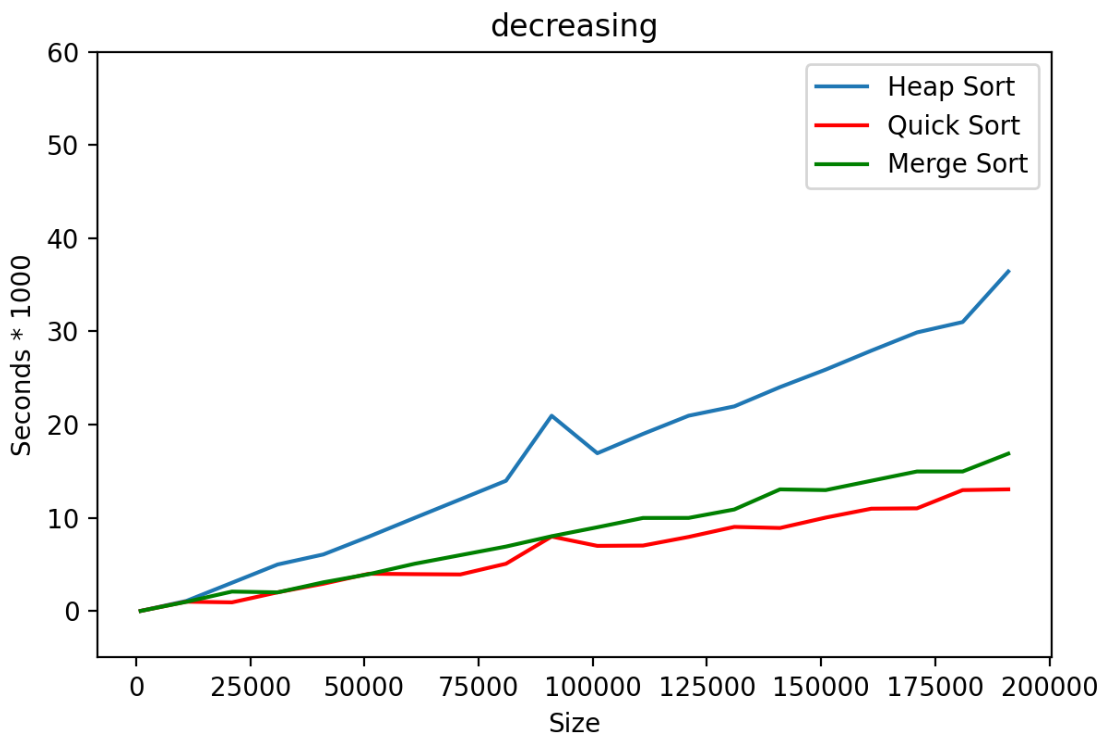
This case is kind of funny because we sort already sorted array. Trend looks similarly as in uniformly distributed random values but generally it takes less time to sort for all algorithms. What we can observe is that QS is quicker than MS. This is because in scenario of increasing numbers when we pick middle element as a pivot we always pick median and this is perfect case for QS. When we would select last element as a pivot then we will obtain the worst case, as pivot will be always smallest or largest element. So this is crucial part for QS to select appropriate pivot.

Let’s see amount of swaps in HS and QS



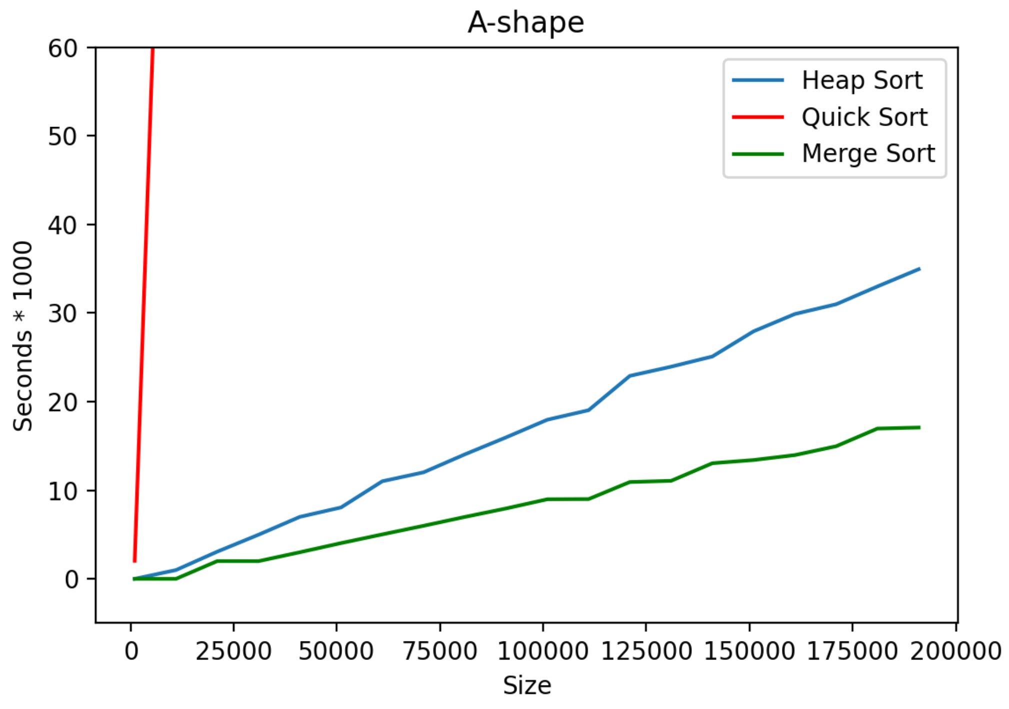
```
Size of the array 111000
Heap swaps = 1847588   Quick swaps = 954284
```

Case 4 – Decreasing

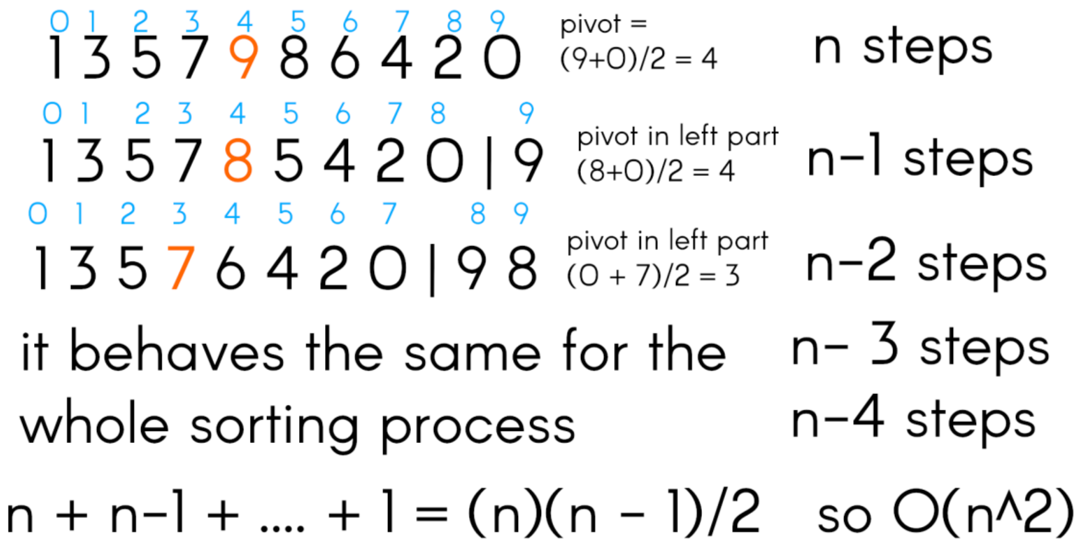


Now we are sorting array that is also sorted but in reverse order. Decreasing and Increasing plots looks almost the same. We have weird peak at around 100000 elements in Heap Sort but probably it is just problem with measuring time.

Case 5 – A shape



Whereas MS and HS looks casually like on previous graphs, it seems that we again hit the QS worst case. Here is a quick proof why it happened. Consider 10-elements A shape array:

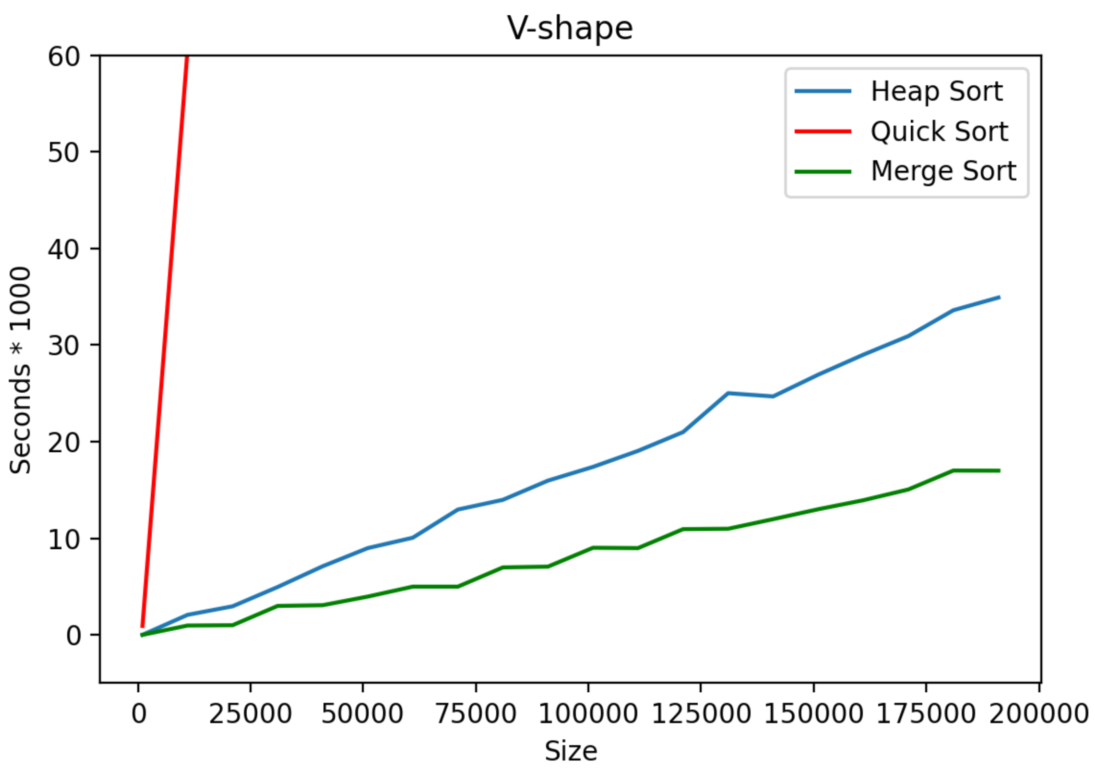


```
0 1 2 3 4 5 6 7 8 9
1 3 5 7 9 8 6 4 2 0     pivot =              n steps
                        (9+0)/2 = 4
0 1 2 3 4 5 6 7 8   9
1 3 5 7 8 5 4 2 0 | 9   pivot in left part   n-1 steps
                        (8+0)/2 = 4
0 1 2 3 4 5 6 7   8 9
1 3 5 7 6 4 2 0 | 9 8   pivot in left part   n-2 steps
                        (0 + 7)/2 = 3
```

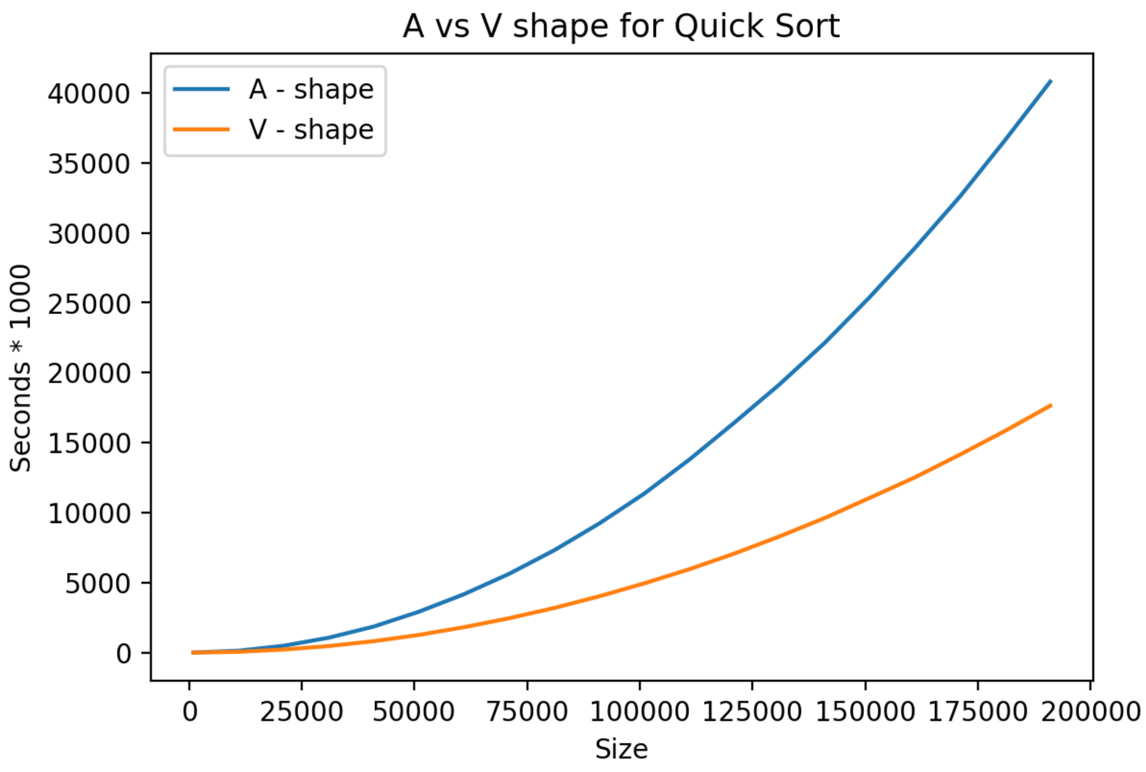
it behaves the same for the whole sorting process — n- 3 steps, n-4 steps

n + n-1 + ... + 1 = (n)(n - 1)/2   so O(n^2)

Case 6 – V shape



Here trend is the same as in A shape case (MS and HS plots looks almost the same) but QS parabola seems to grow slowly, let’s compare them on separate plot:



And the difference is quite huge! We can see that both functions are parabolas but definitely the blue one has greater slope, so A – shape seems to be harder case to sort for QS.

Conclusions

Taking into account only complexities one may say that merge sort is the best, but we one have to take into account additional memory consumption that in case of big inputs may be essential. So maybe heap sort is the best? Yes it always produces similar size vs. time functions it don't need any additional space but Is much slower than QS and MS. So is Quick Sort the best? In most cases it is the fastest one and don't need additional memory, but those 3 worst cases destroy everything. So taking into consideration only this experiment and only these 3 implementations of those sorting algorithms, the conclusion about which is the best is it depends on input data. So should we, before sorting linearly scan array and after that decide which one to use? Definitely it is not the best idea. Why Quick Sort is implemented inside sort method inside programming languages? In our experiment we take into account only one implementation of QS namely the one with pivot as the middle element, but there are few possible improvements to be done so let's look at them.

Median effect and possible improvements for QS

In Quick sort, worst case happens when somehow every next partition is only slightly smaller than previous one, so every time we have to do check in almost linear time. It occurs When the pivotal element is one of the least or greatest element in our partition. Then complexity is O(n^2). One can say that this is rather impossible when we have random distribution of our data and indeed it is but sometimes our data is structured in a way that may cause worst case. So now consider situation when every time our pivotal element is a median of a set, then in each partition we divide our set into two equal halves and we have best case of the best cases then. Searching for median every time is without sense, but we can increase our chances for selecting at least something that is not outsider by choosing pivot a the median of 1^{st}, middle and last element or choose 3 random elements and then pick median from them. This action is not so computationally expensive but decreases chances of worst case 3 times. So better pivot selection is one of the way to improve Quick Sort performance. Another way is to use different partition scheme that is called "Hoare's partitioning scheme" in that way we can decrease amount of swaps. We can also try to minimize the recursion depth and actually this step is crucial, as in worst case without optimizing it our program may crash. Further, we can do is to use not recursive method to sort elements when the total number of elements in partition is below some threshold. One more improvement we can do is to use variation of QS called Intro sort, which uses Heap sort to sort the current partition if the Quicksort recursion goes too deep, indicating that a worst case has occurred – this is how std::sort method in C++ is implemented.

Quick sort vs. 99999, 100000 and 100001 elements and random distribution, A-shape and V-shape

	99999	100000	100001
random	0.015973	0.046793	0.014373
A-shape	30.195445	30.096247	11.876895
V-shape	13.708019	10.957038	10.309157

On the intersection we have time in second. What is interesting that in case of A-shape and 100001 elements sorting time is much faster than for 1 and 2 elements less.