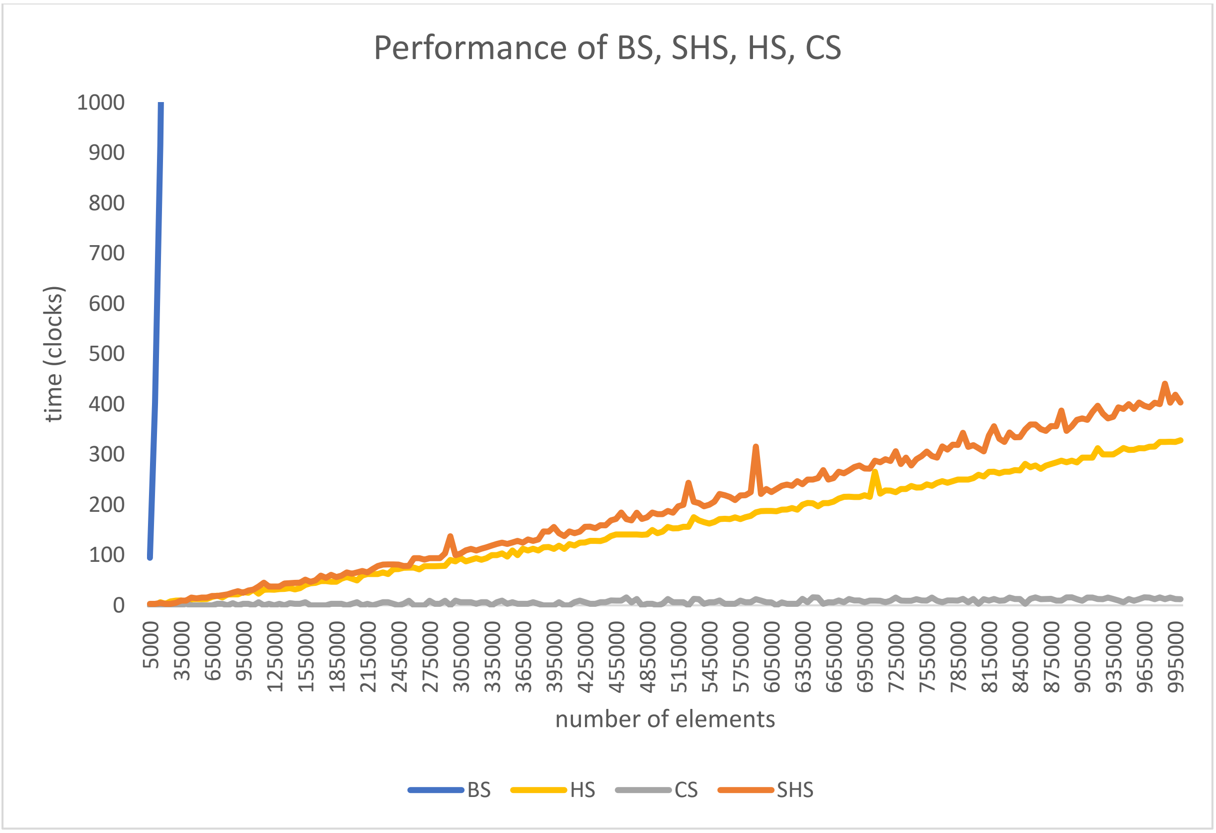
I



I 1 Chart

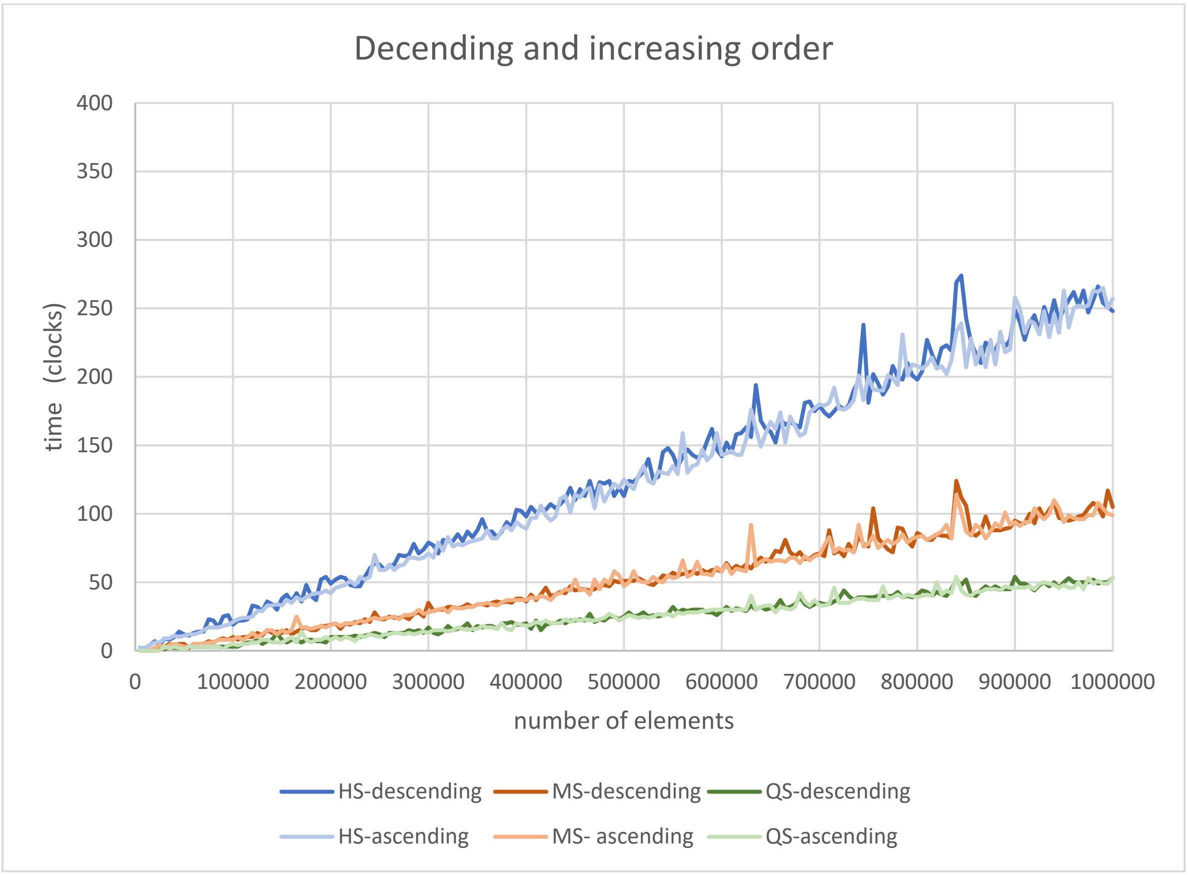
A clear winner as for time complexity turned out to be the Counting Sort algorithm (CS) , depicted with a gray line on the above chart, it considerably outperforms the other sorting algorithms due to its linear time complexity O(n+k), furthermore the usage of a small range(the array of length k being of similar length to the main array (of length n)) also helps. However lower time complexity comes with a major drawback- for large amounts of data (especially when the range if sizeable) - in such cases CS is characterized by excessive memory consumption.

Secondly comes the Heap Sort (HS), algorithm, with O(nlogn) complexity (the chart might be misleading on the first glance because of the proportions of the axis). In contrast to CS, HS does not consume additional memory space for algorithm-specific structures, therefore it seems to be a good pick for a time-efficient sorting algorithm, the more so since for all cases its time complexity is the same(which can be written symbolically as Ω= θ=O).

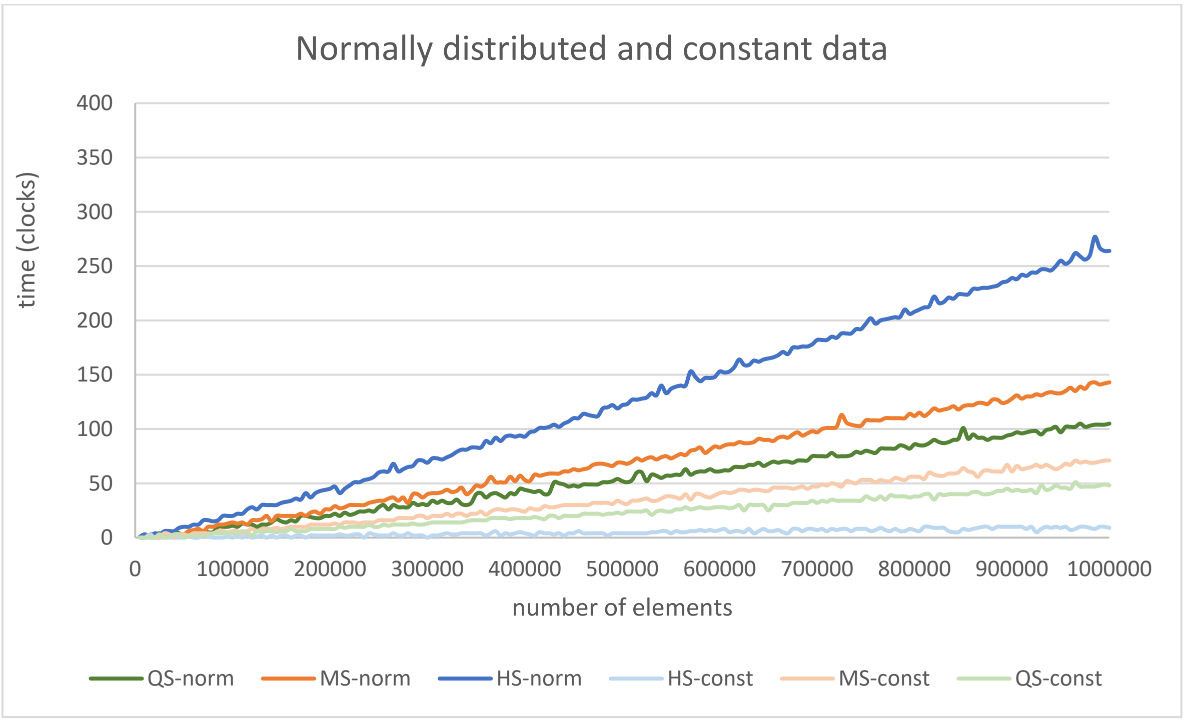
The orange line represents the Shell Sort algorithm (SHS), it's notably more ragged than the 3 other lines – owing to the fact that SHS is an unstable algorithm – in the 'classic'(Shell's) worst case scenario it behaves like a tweaked version of insertion sort. Because of the instability, it's quite difficult to state what it's time complexity is, but our best approximation would be θ (n*log(n)^2) – for the average case, with O(n^(3/2) being the worst case scenario and Ω(nlog(n)) being the best. As for the memory consumption it's sorting in place without any additional structure needed.

The algorithm delivering the worst performance(as for computational complexity) is Bubble Sort(BS), although it doesn't consume additional memory, it's O(n^2) computational complexity renders it completely impractical for most applications. As can be observed looking at the chart, BS's blue line skyrockets and is therefore is outrun by all the other sorting algorithms presented above by orders of magnitude.

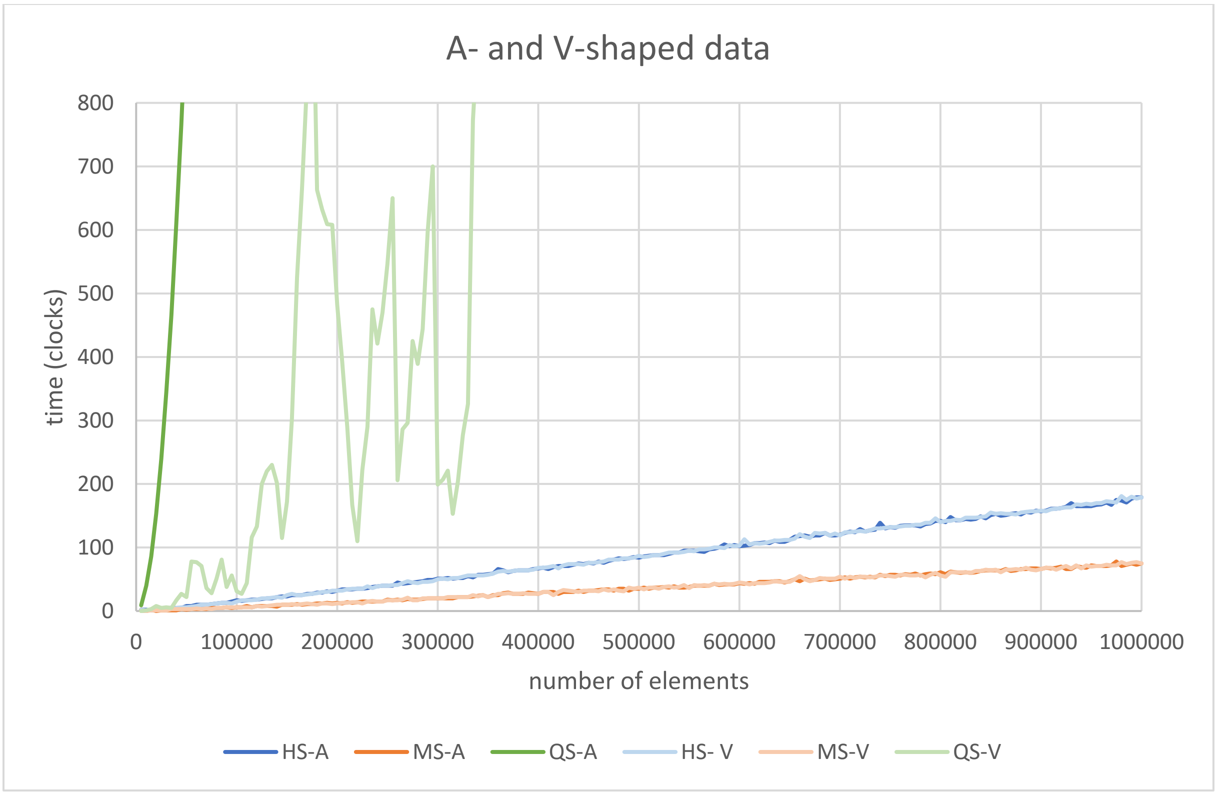
II Overview of presented algorithms:



II 1 Chart



II 2 Chart



II 3 Chart

For ascending and descending data, the plots of all 3 sorting algorithms look strikingly similar, which seems to be rather counterintuitive. However - for instance in the HS algorithm - because of the fact that the "heap indices" are mapped onto an array, the procedure do not differ in any significant way. An unrelated, but analogous phenomenon occurs in QS as a result of constantly choosing the right pivot – from the point of view of the algorithm it doesn't matter if the order of elements is right – the crucial part is to choose such a pivot which divides the array into 2 subproblems (subarrays) of roughly the same size – which here happens every time since the subarrays are pre-sorted.

For Normally distributed and constant-value arrays one can spot a clear trend – comparing these two- the time complexity of the algorithms working on the latter is lower, with heap sort scoring astonishingly well – this can be explained away with the knowledge of the fact that this is exactly the best case of the HS algorithm (no need for the heapify procedure to displace items) so it's just O(n). In both cases (normal distribution and constant value QS outperforms MS – that might be due to a plethora of factors – certainly MS's space complexity being one of them. A similar observation can be made for HS and merge sort working on the normally distributed data – MS outperforms HS – one possible explanation can be that heap sort has to swap values(2 reads and 2 writes) while MS just moves them. From this follows that moving and swapping is quite computationally costly and since quick sort performs less of those (in MS 100% of the data is moved, since an additional array is used and in HS an entire heap is build by swapping elements and placing the maximum element at the end) it's less costly. Considering this, one has to put emphasis on the fact that all of the upper 5 lines are all O(nlogn) – only the constants differ.

For the last chart conclusions for HS and MS drawn from the previous example still hold (merge sort performs even better because of the pre-sorted character of the data). As For the QS algorithm, it is clear that the worst possible case is depicted – this is especially true for the A-shape data: Each time an extremum or a value close to it is selected to be the pivot, it results in performing the quick sort procedure recursively on an array of similar size (as opposed to similar sized sub-arrays or even subarrays of a ratio 1/9 – this too would result in a time complexity of O(nlog(n)). A similar phenomenon occurs for the V-shaped arrays, although after some time QS seizes to make very bad choices due to the nature of the problem(or at least our implementation of it) (this is depicted in the Remark section). A solution to this would be to select the median as the next pivot(making the best possible choice), or at least to pick the median of a subset of the array as the next pivot(ensuring not picking the worst possible pivot).

II b)

1	3	5	7	9	11	13	15	17	$\boxed{19}$	20	18	16	14	12	10	8	6	4	2

N comparisons 2 swaps

1	3	5	7	9	11	13	15	$\boxed{17}$	2	4	18	16	14	12	10	8	6	$\boxed{20}$	19

(N-2) comparisons 2 swaps + 1 comparison 1 swap

1	3	5	7	9	11	13	$\boxed{15}$	6	2	4	8	16	14	12	10	$\boxed{18}$	17	19	20

(N-4) comparisons 2 swaps + 1 comparisons 1 swap

1	3	5	7	9	11	$\boxed{13}$	10	6	2	4	8	12	14	$\boxed{16}$	15	17	18	20	19

(N-6) comparisons 2 swaps + 1 comparisons 1 swap

1	3	5	7	9	$\boxed{11}$	12	10	6	2	4	8	$\boxed{13}$	14	15	16	17	18	20	19

(N-8) comparisons 2 swaps + 1 comparisons 1 swap

1	3	5	7	$\boxed{9}$	8	4	10	6	2	$\boxed{12}$	11	13	14	15	16	17	18	20	19

(N-10) comparisons 2 swaps + 1 comparisons 1 swap

1	3	5	$\boxed{7}$	2	8	4	6	$\boxed{10}$	9	11	12	13	14	15	16	17	18	20	19

(N-12) comparisons 2 swaps + 1 comparisons 1 swap

1	3	$\boxed{5}$	6	2	4	$\boxed{8}$	7	9	10	11	12	13	14	15	16	17	18	20	19

(N-14) comparisons 2 swaps + 1 comparisons 1 swap

1	$\boxed{3}$	4	2	$\boxed{6}$	5	7	8	9	10	11	12	13	14	15	16	17	18	20	19

(N-16) comparisons 1 swap + 1 comparisons 1 swap

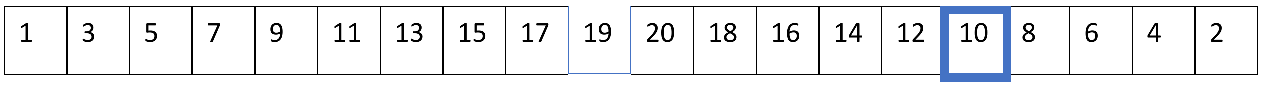
$\boxed{1}$	2	$\boxed{4}$	3	5	6	7	8	9	10	11	12	13	14	15	16	17	18	20	19

(N-18) comparisons 0 swaps + 1 comparisons 1 swap

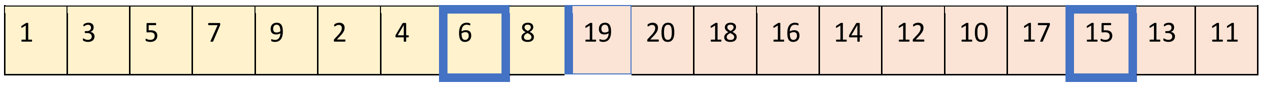
$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$	$\boxed{8}$	$\boxed{9}$	$\boxed{10}$	$\boxed{11}$	$\boxed{12}$	$\boxed{13}$	$\boxed{14}$	$\boxed{15}$	$\boxed{16}$	$\boxed{17}$	$\boxed{18}$	$\boxed{20}$	$\boxed{19}$

Approximate total count ~ ((N^2)/4+ N/2) comparisons + N swaps - at least for our implementation this holds true

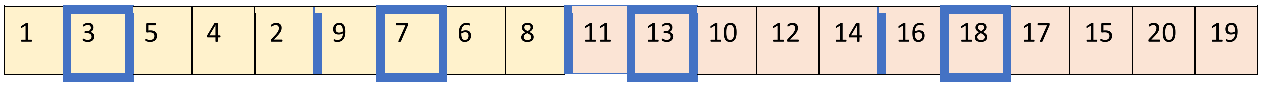
**** here the A-shape was used for the V-shaped data as seen in the chart one can not draw such consistent conclusions (at a certain point QS seizes to make very bad "pivot decisions").*



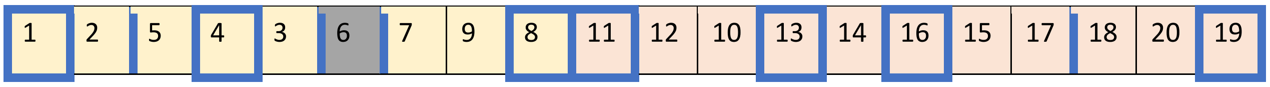
N comparisons + 4 swaps



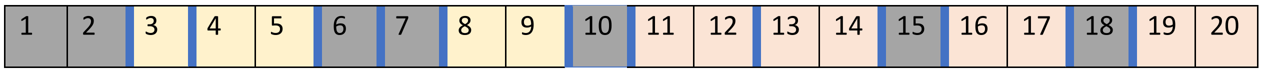
N comparisons + 2 swaps + 4 swaps



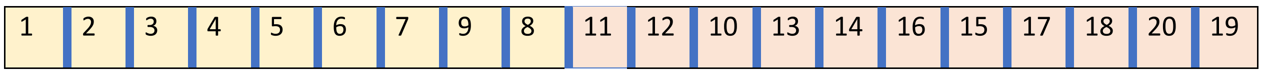
N comparisons + 1 swap + 1 swap + 1 swap + 1 swap



(N – 1) comparisons + 1 swap + 1 swap + 2 swaps +1 swap + 1 swap



(N – 7) +1 swap

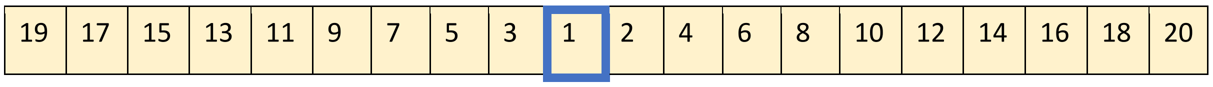


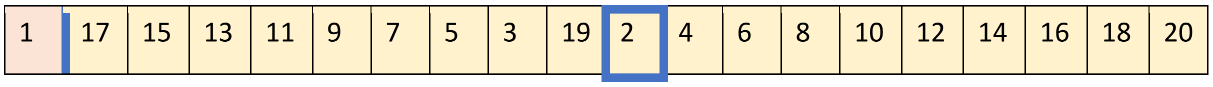
*Number of comparisons ~ O(N*ceil(log(n)))*

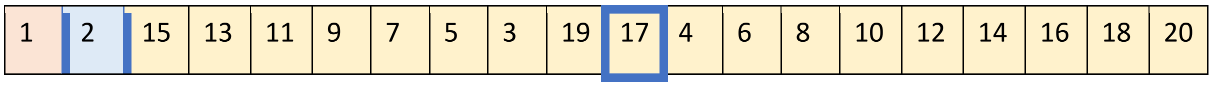
On the previous page the procedure executed by the QS (middle pivot- worst case scenario) algorithm is outlined in a step-by step manner in a simplified way. The complexity of which (overall) being O(n^2) - this is significantly worse than the performance delivered by choosing the median(or a value close to it) as the pivot. As stated before even dividing the subarrays in a seemingly very uneven ratio results in far better time complexity – thus it seems quite redundant to find the exact median of the data – using the "median of medians" algorithm is satisfying though(or any other algorithm not being too computationally heavy). One can use this approach to avoid O(n^2) complexity in the classical "worst case".

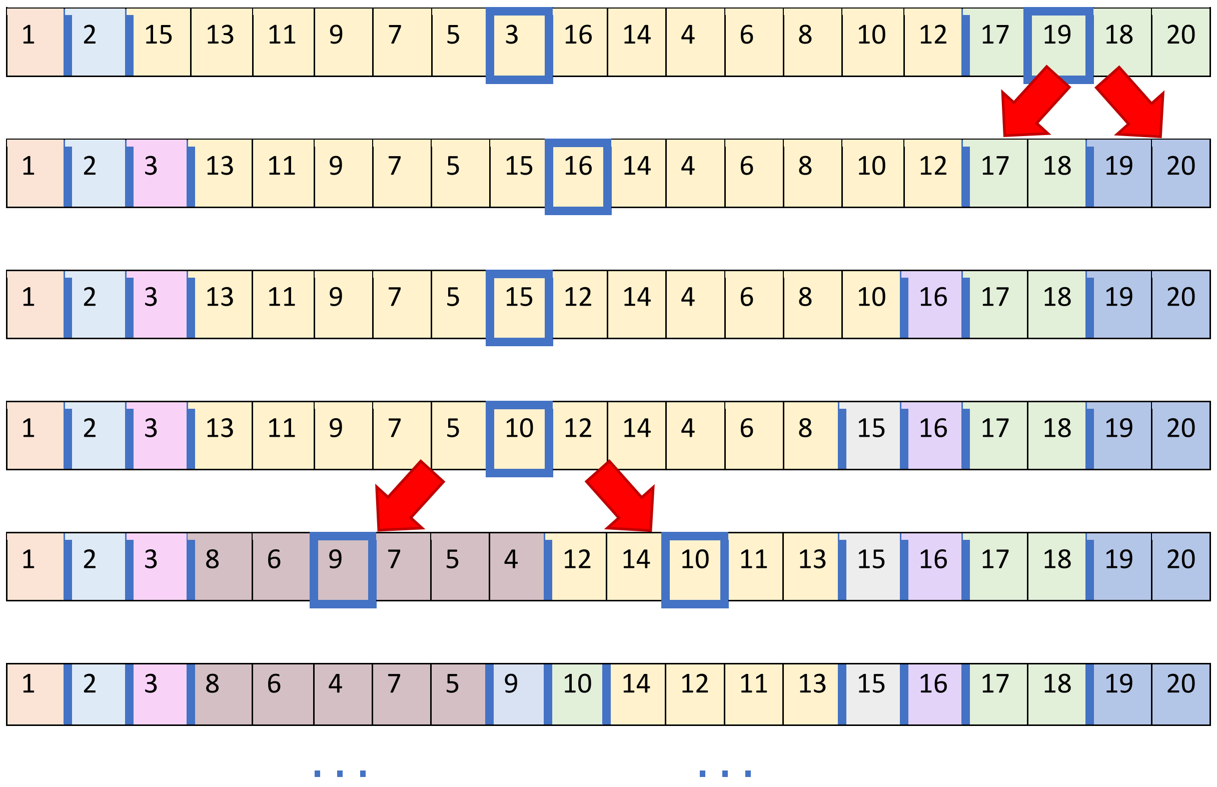
REMARK on the workings of the QS on V- shaped data.

In contrast to the A-shaped data the V-shaped (at least in our implementation) data does not consistently result in O(n^2) time complexity when fed to the QS (crucial parts marked):









Such “good” pivot choices might seem rare and one can argue that they are outweighed by the “good pivot” choices – nevertheless even one single “good” at the beginning can largely improve the performance of the algorithm, and most of the times – more of such “good” pivot choices happen.