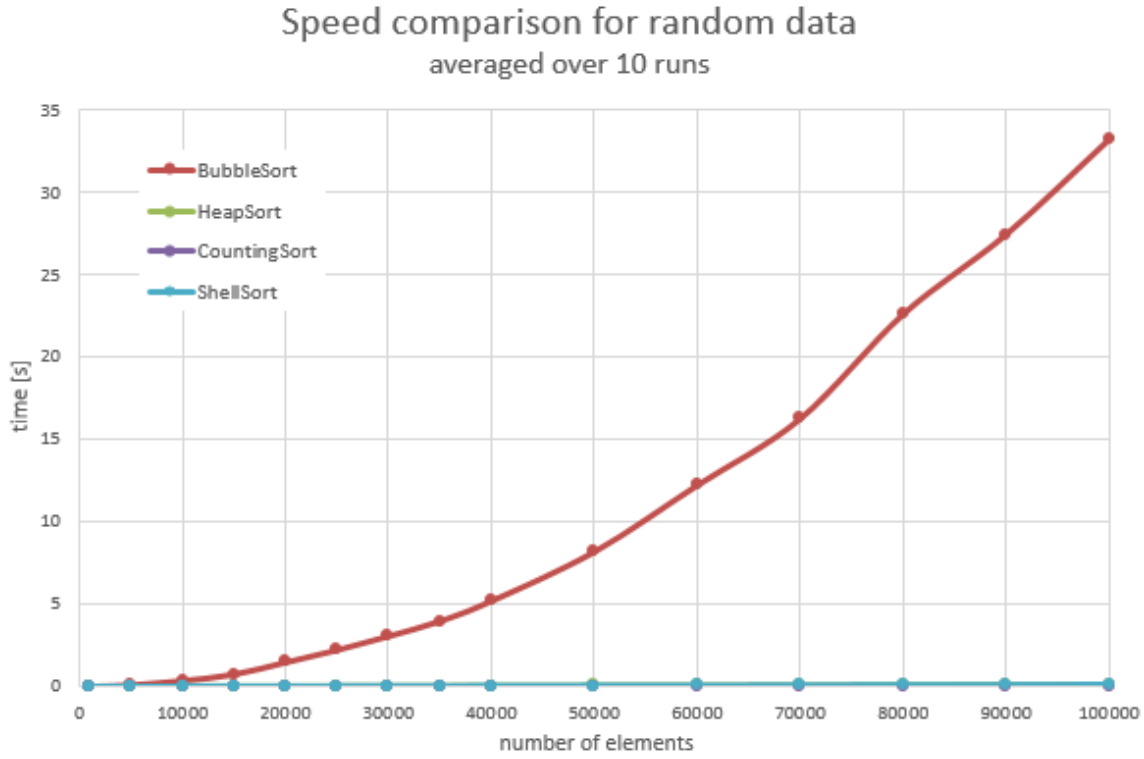
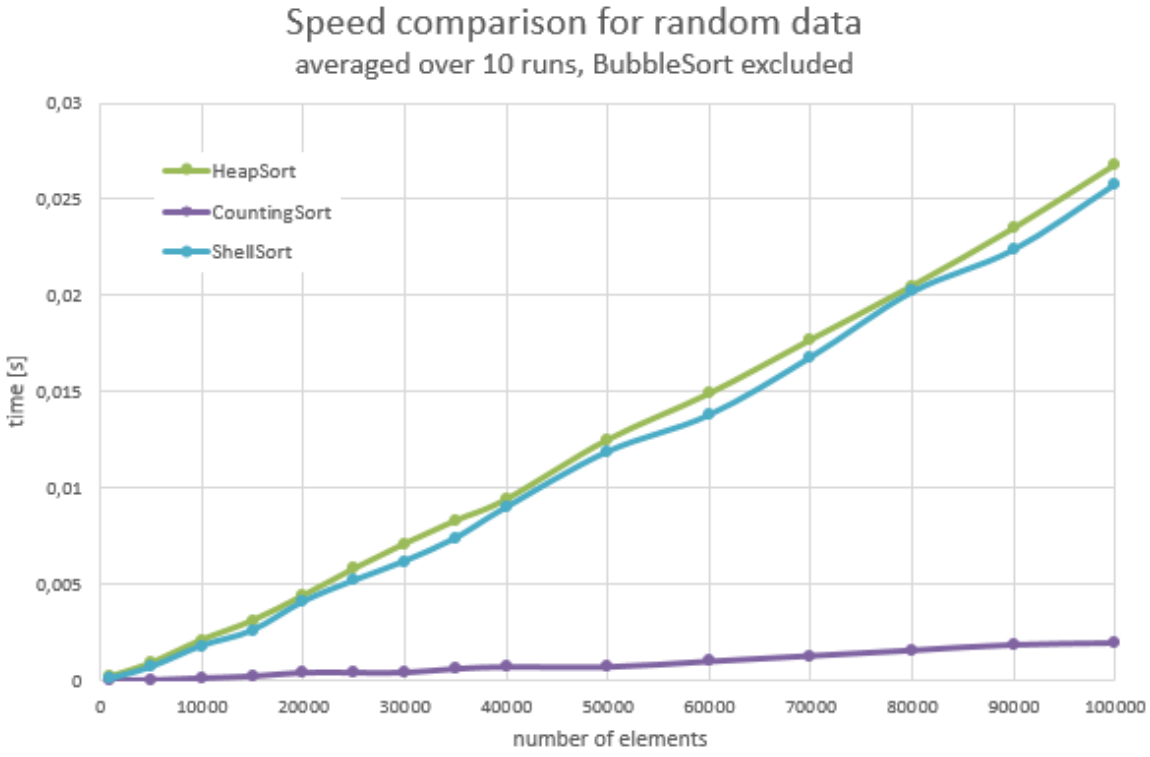
Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

I.1. *Compare the speed of 4 sorting methods: BS, HS, CS, ShS for the array of integers randomly generated according to the uniform probability distribution. Submit a chart t=f(n) where: t - sorting time; n - number of elements of the sequence. The number of elements must be selected in such a way that measurements can be taken properly. The results are presented in one chart (at least 15 measuring points).*





I.2. *Formulate conclusions on the computational complexity of the methods studied and their relationship with the efficiency of sorting and the memory activities of each method.*

Algorithm	**Complexity**	**Memory**
Bubble sort	n^2	1
Heap sort	n*log n	1
Shell sort	n*log n	1
Counting sort	n+r	n+r

*r = range of the numbers to be sorted

Performing experiment for random data confirmed that Bubble sort has the worst time complexity, which is $O(n^2)$. In comparison, Heap sort, Shell sort and Counting sort performed significantly better with complexity which seems to be O(0). Only after excluding BS from the presented chart, can one notice deviations. Both HS and ShS produced similar results – time complexity O(n*log n). CS was the fastest algorithm, with complexity O(n+r).

With computation time shorter than 0.005 s for 100 000 elements, Counting sort may seem to be the best choice. However, it requires using additional array to store the counts of numbers, so its memory consumption is n+r (r - range of sorted numbers). The rest of discussed algorithms is *in situ*, so they do not require additional storage. Sorting is performed within the initial array. Bubble sort is clearly the worst one, although it has one advantage – it is simple to understand and intuitive. When it comes to the last two, Shell sort was slightly faster than Heap sort.

II.1. *For different input types compare the effectiveness of 3 sorting algorithms.*
a) QS with middle selected pivot,
b) HS
c) MS.
Examine the performance for the following data types of the sequence:
- random (uniform distribution)
- constant value (e.g. equal to 0)
- increasing order (step equal to 1)
- descending order (step equal to 1)
- ascending-descending order (A shape – increase odd numbers - decrease even)
- descending-ascending order (V shape – decrease odd numbers - increase even)

Prepare charts t=f(n) where: t - sorting time; n - number of array elements for different data types (2 consecutive types per chart - 6). The number of elements must be selected in such a way that measurements can be taken properly.

The results are presented in charts - one for two data types (at least 15 measuring points).
Note: the data used to create the charts is an average over 10 runs

Chart no. 1

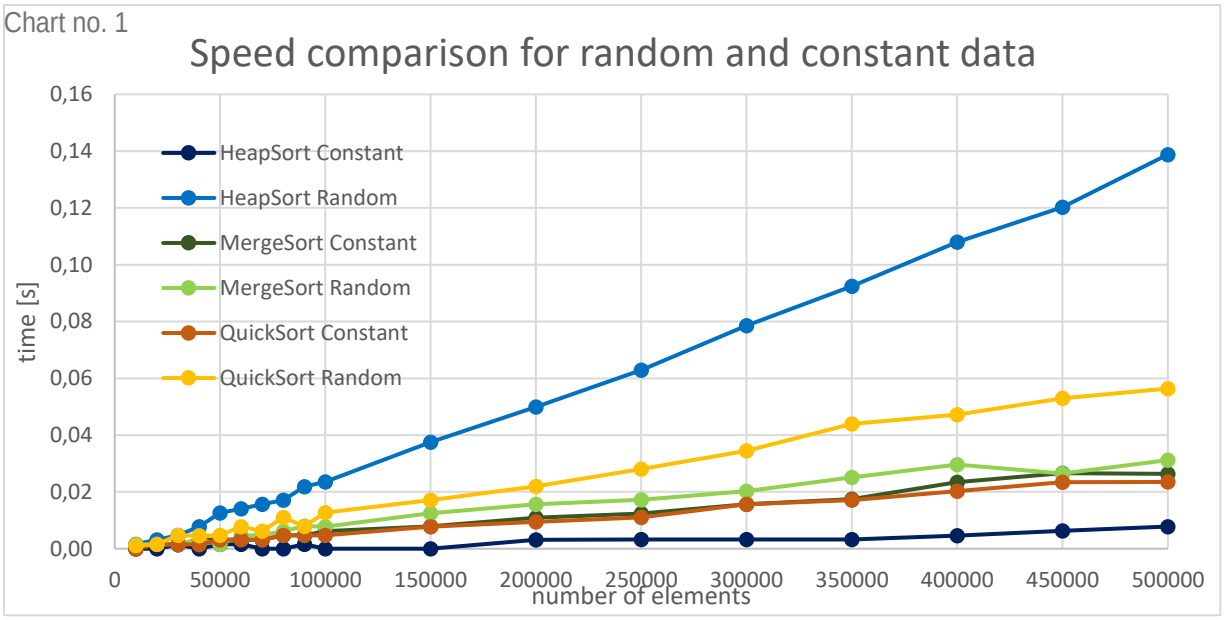


Chart no. 2

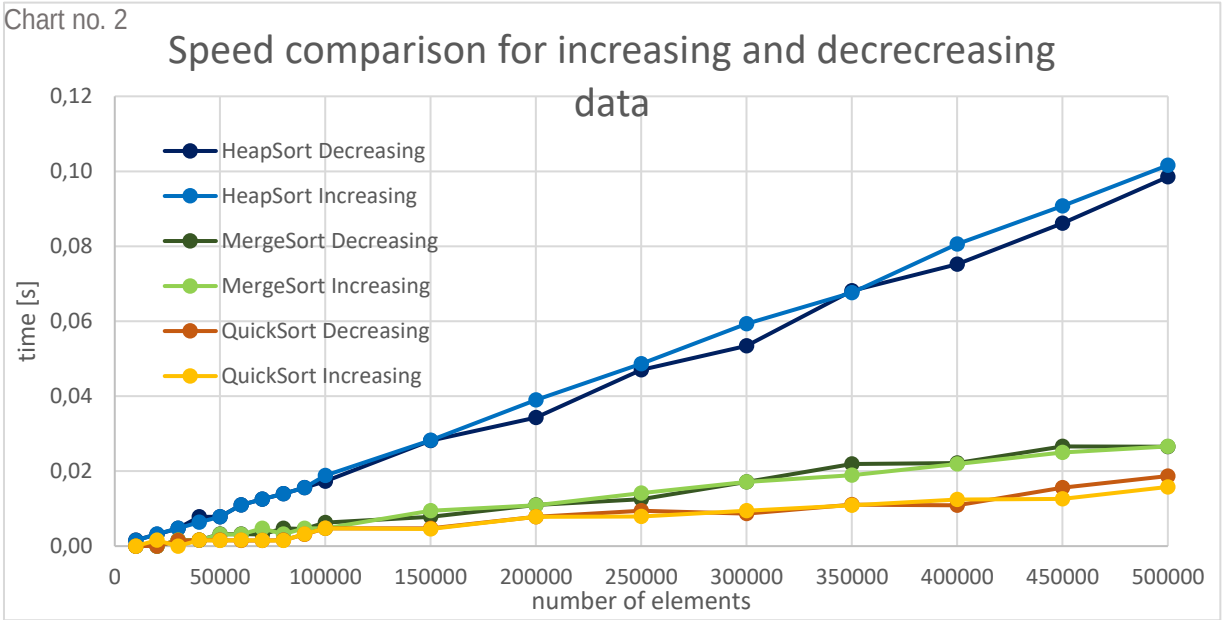
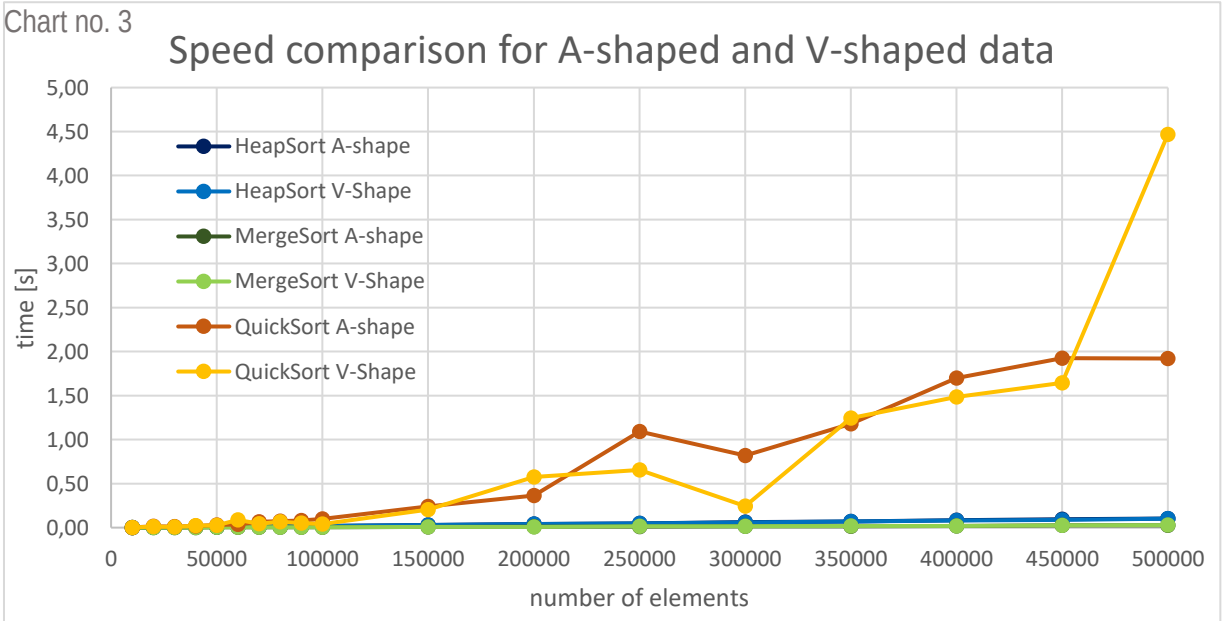
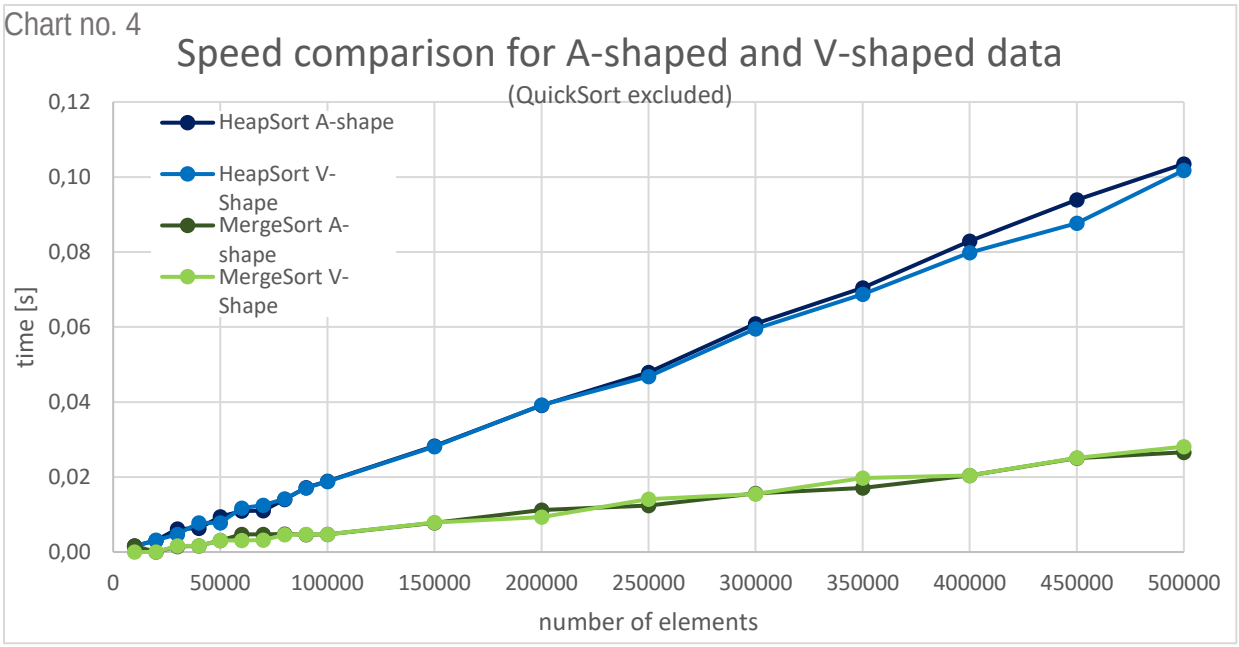


Chart no. 3

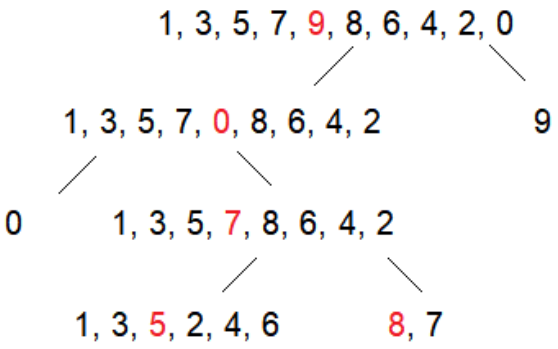




II.2. *Draw conclusions on the computational complexity and computational efficiency of QS execution and the behavior of the algorithm in the worst case scenario and for the different types of data.*

From the experiment performed one may think the efficiency of QuickSort is determined by the way the numbers in an array are arranged in the beginning. In fact, the time spent on computations depends on whether successive arrays' divisions made while sorting are balanced or not. (which element is chosen as the pivot point)

Considering the worst case when a pivot always divides a n-element array into two most imbalanced arrays (i.e. of sizes 1 and n-1) it can be seen that the total time needed to sort the whole array (complexity) $T(n) = T(n-1) + O(n) = T(n-2) + O(n-1) + O(n) = \ldots O(n^2)$ (O(n) - time needed to partition n elements). Such a situation occurs when the picked pivot is an extreme value. In the experiment this condition is met when a data type is A-shaped or V-shaped (chart no.3) – extreme values are placed in the middle of an array as well as the pivot which results in poor performance of the algorithm. The part that might be surprising at first is the appearance of the lines of QuickSort (in the case of A- and V-shape) as the number of elements increases. The result might come from the fact that the algorithm implemented does not always make the worst possible choice (as can be observed below; the example considers an A-shaped array of n=10, pivots are marked in red). This may contribute to the deviations visible on the chart.



…

In the best case scenario, if the number selected as a pivot always divides an array as equally as possible (their sizes either are equal or differ by 1), the complexity of the QuickSort will be O(n*logn) since within each partition (which number equals to logn) there are n elements in total to be split. This case can be seen on the chart no. 2 where QuickSort performed the best compared to MergeSort and HeapSort.

Analyzing the average case, QuickSort will for sure perform worse than in the case mentioned above, although there are common proofs (e.g. using percentiles) showing that the complexity is also O(n*logn). It can be observed on the 1st chart (random data type) where the line representing QuickSort is placed between MergeSort and HeapSort (taking into consideration the fact that both of these algorithms' complexity is O(n*logn)).

The behavior of the QuickSort with constant data type provided is very similar to MergeSort. In fact, this particular case can be treated as a one of the best case scenarios (an array will be divided as equally as possible).

What is the median effect on QS sorting time? What is it used for?

The median affects QuickSort sorting time in a significant way. By selecting the median value of an array the algorithm will be allowed to split the list into approximately equal parts. Therefore, assigning the median value as a pivot can be used to reduce the runtime of a program. However, the problem of using the median is that one has to know all the values in an array to be able to choose the proper value as a pivot. Therefore, despite the fact that the median is the optimal value in theory, in practice it might not be such easy.

An interesting solution to the problem of choosing pivot may be "median-of-three". It states taking the median of the first, middle and last element of the partition for the pivot. This rule counters the case of sorted (or reverse sorted) array and gives better approximation of the optimal pivot (median of the whole set).

Examination of the properties of other sorting algorithms

HeapSort: It is the slowest of the sorting algorithms of the complexity O(n*logn) but it does not require neither massive recursion nor multiple arrays to work.

MergeSort: The algorithm is slightly faster than HeapSort for larger n; however, it requires additional memory to create the second array.

Having looked at the charts, one could have noticed that the type of data does not influence MergeSort and HeapSort at all (to have a clearer view on A-shape and V-shape data type in MergeSort and HeapSort, QuickSort was removed from the chart no. 4). The only difference is in the case of HeapSort when constant data type is involved. It is quite intuitive since according to the algorithm there will be no swaps in the heapifying part.

The comparison of complexity and memory used to perform QuickSort, MergeSort, and HeapSort:

Algorithm	Complexity*	Memory
QuickSort	n^2 / n*logn / n*logn	n*logn
MergeSort	n*logn/ n*logn/ n*logn	n
HeapSort	n*logn/ n*logn/ n*logn	n*logn

*worst, average and best case respectively

Sources:

- Wprowadzenie do algorytmów T. Cormen, C. Leiserson, R. Rivest
- https://en.wikipedia.org/wiki/Sorting_algorithm
- http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych
- https://www.geeksforgeeks.org/sorting-algorithms/