Project report		Academic year:
Subject: **Algorithms and data structures**		**2020/21**
Project name: **Analysis of different sorting algorithms**		Subject takes place at: **N/A**
Faculty, field of study, semester: **FC, AI, II**	Full name: **Jan Gruszczyński**	Grade:

Exercise 1

Speed comparison of four distinct sorting methods (Bubble Sort, Heap Sort, Counting Sort, Shell Sort). Sorting arrays of integers generated according to the uniform probability distribution.

At each measuring point, sorting time was measured 30 times on consequently new generated arrays, then an average was taken as representative of obtained values. This was done in order to get more accurate data, less corrupted by inherent system instability.

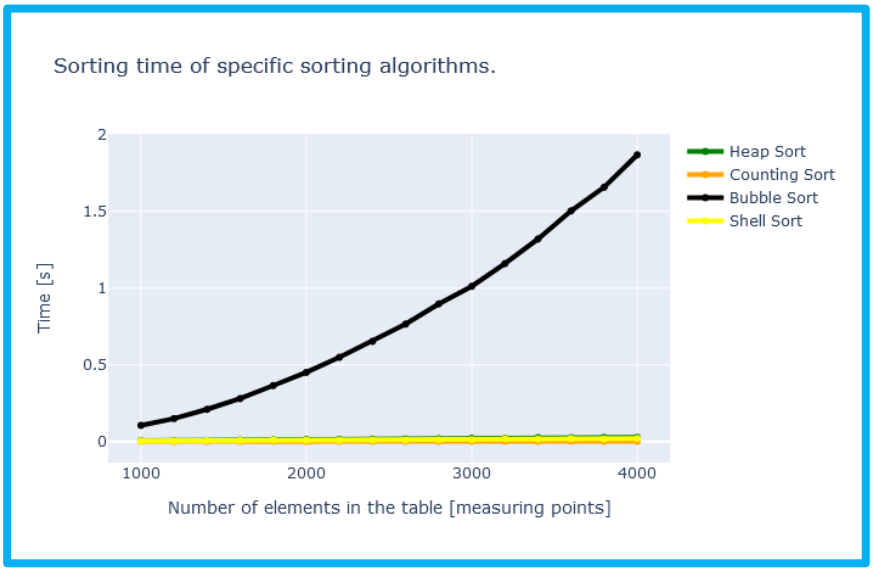

Figure 1: Speed comparison of HS, CS, BS and ShS. Range of numbers in sorted array same as array length.

Clearly Bubble Sort is the worst one of compared sorting algorithms

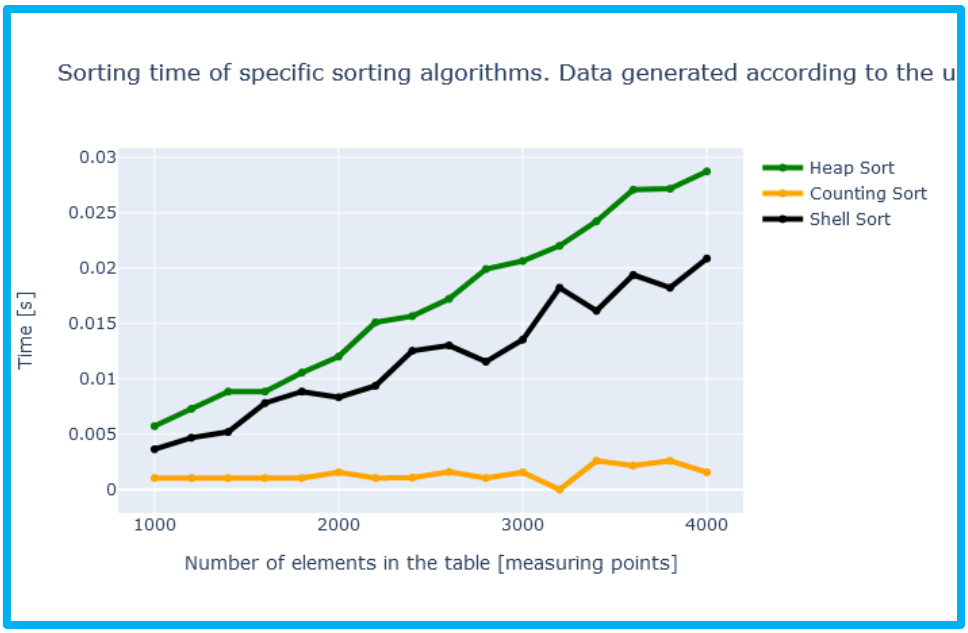

Figure 2: Speed comparison of HS, CS, and ShS. Bubble Sort omitted. Range of numbers in sorted array same as array length.

Above charts state that counting sort is the fastest sorting algorithm.

Theoretical time and memory activities of the given algorithms

Sorting algorithm	**Feature**	**Average Time complexity**	**Memory complexity**
Heap Sort		O(n*log(n))	O(n+k)
Shell Sort		$O(n^{1,25})$	O(n)
Counting Sort		O(n+k)	O(n)
Bubble Sort		O(n^2)	O(n)

Although Heap Sort has better time complexity than Shell Sort. On my Chart Shell Sort appears to be better, because $n^{1,25}$ is smaller than n*log(n) for small numbers (*Figure 3*). Additionally, it's worth noticing that if the range of sorted numbers is equal to 1*n the fastest algorithm is Counting sort.

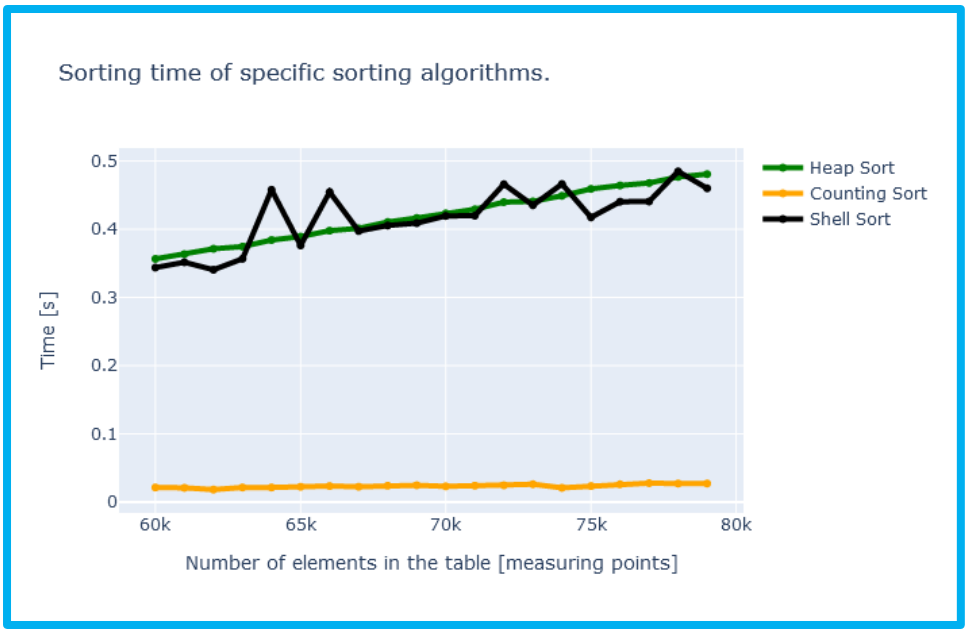

Figure 3: The bigger the n the time gap between Shell Sort and Heap Sort becomes smaller. Eventually Heap Sort should become more efficient.

However if the range of numbers in sorted arrays becomes larger, Counting Sort quickly becomes the least efficient algorithm (*Figure 4*).

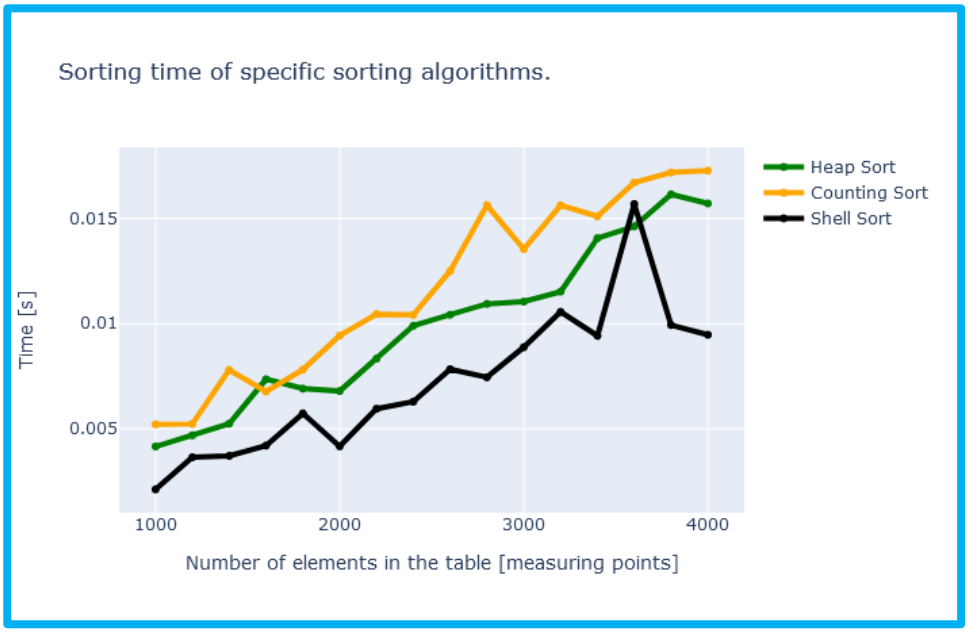

*Figure 4: Speed comparison of HS, CS, and ShS. Bubble Sort omitted. Range of numbers in sorted array is equal to 25 * n where n is the length of sorted array.*

In conclusion:

- Bubble Sort should not be used.
- If the range of numbers in sorted array is near the length of sorted array, counting sort is the most efficient method.

- If the range of numbers in sorted array is bigger (more than 10*n=length of array), Shell Sort should be used for smaller arrays, and Heap Sort for bigger arrays. (At least 100 thousand elements).

Exercise 2

Comparison of efficiency of Quick sort with middle selected pivot, Heap Sort and Merge Sort. Six types of data: random, constant, increasing, decreasing, V shaped, A shaped.

Once again at each measuring point, sorting time was measured 30 times on consequently new generated arrays, then the an average was taken as representative of obtained values. This was done in order to get more accurate data.

Range of values in arrays, appears not to make a difference in comparison.

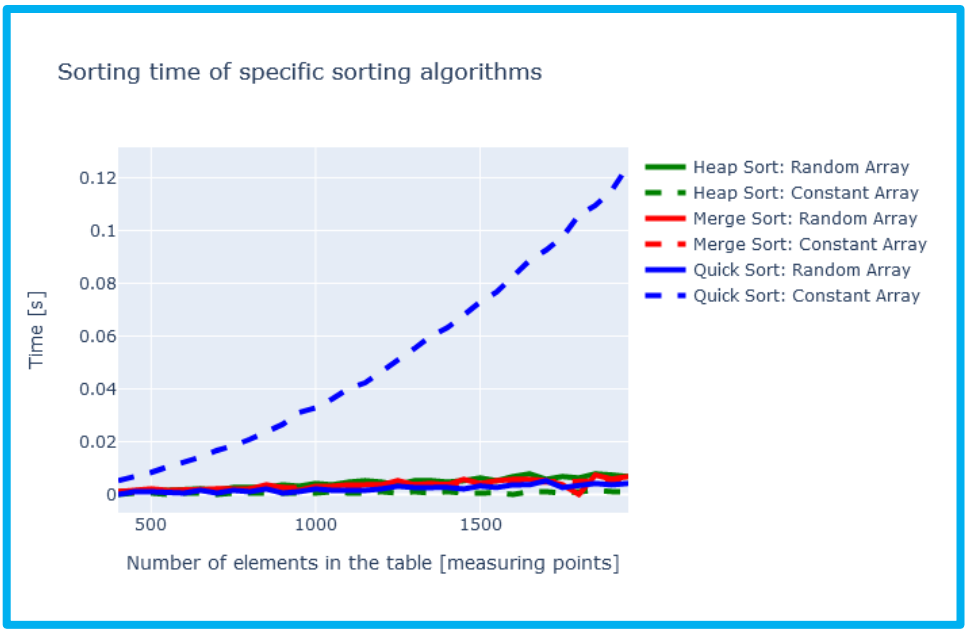

Figure 5: Speed comparison of HS, MS, and QS. Two data type: constant and random. Constant equal to one.

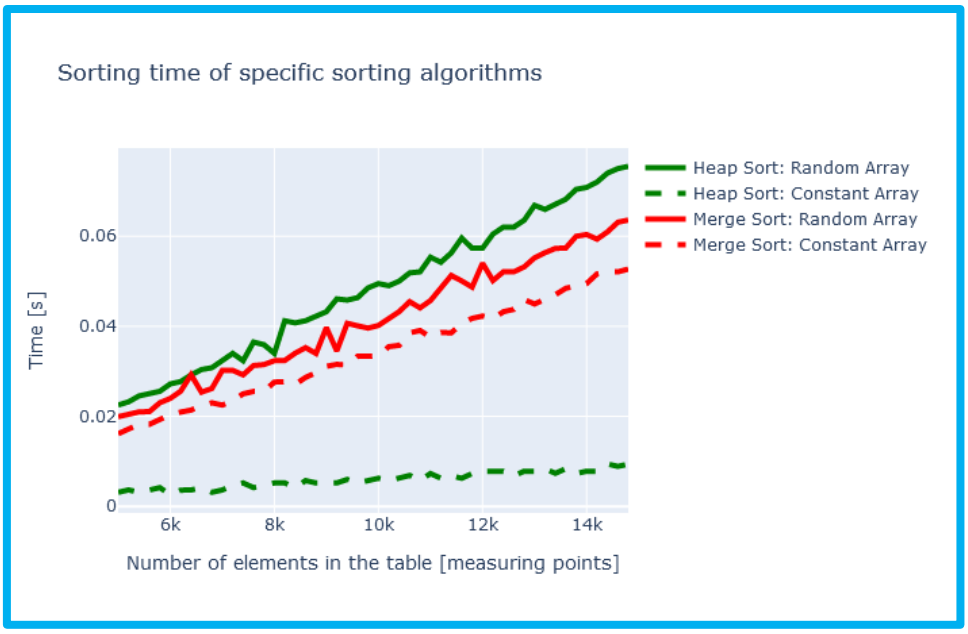

Figure 6: Speed comparison of HS, MS. Quick Sort omitted. (Recursion depth error) Two data type: constant and random. Constant equal to one.

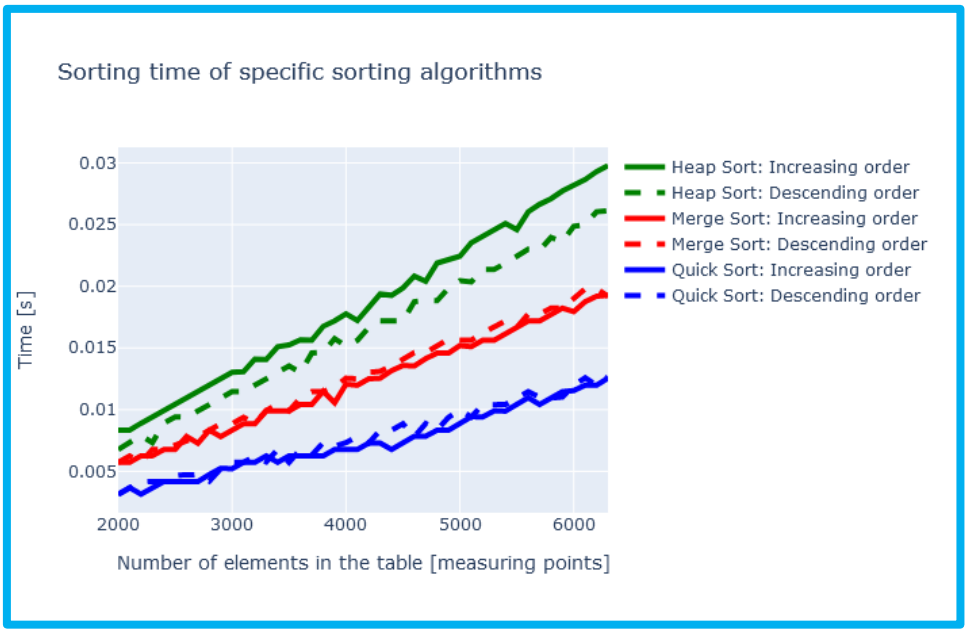

Figure 7: Speed comparison of HS, MS, and QS. Two data types: Array of increasing elements, and array of decreasing elements.

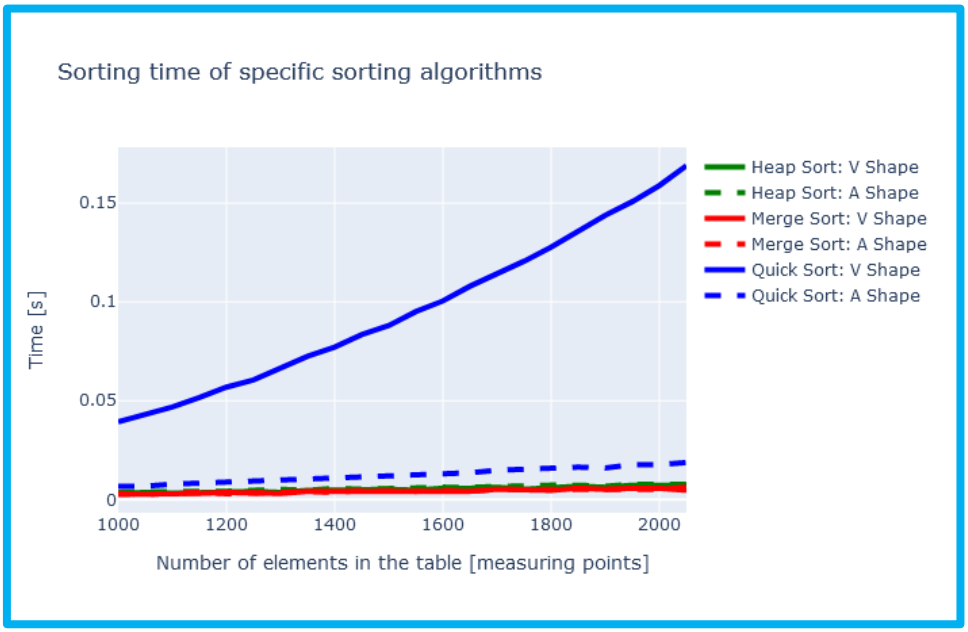

Figure 8: Speed comparison of HS, MS, and QS. Two data types: V shaped Array, and A shaped array.

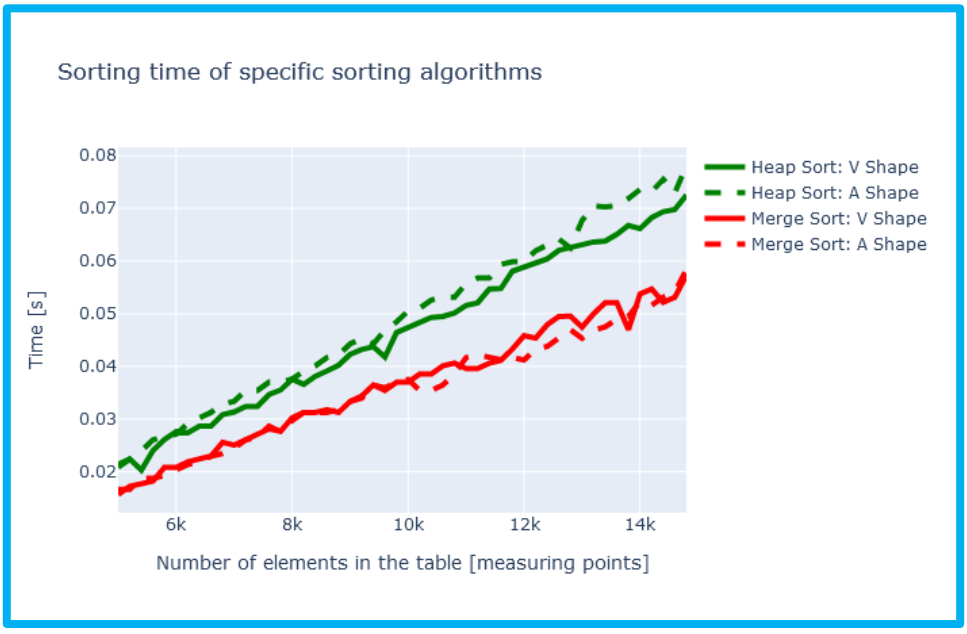

Figure 9: Speed comparison of HS and MS. Two data types: V shaped Array, and A shaped array.

Sorting algorithm	Feature	Time complexity	Memory complexity	Worst Time Complexity
Heap Sort		O(n*log(n))	O(n+k)	O(n*log(n))
Merge Sort		O(n*log(n))	O(n)	O(n*log(n))
Quick Sort		O(n*log(n))	O(n)	$O(n^2)$

Conclusion:

- Quick Sort seems to be the best sorting method of array that is generated randomly (uniform distribution). (*Figure 5)*
- Array of constant values is one of the worst case scenarios for Quick Sort. (Every element of array is moved to one side of the pivot, which requires n-1 splits). Similar situation occurs with V-shaped data. Such data requires maximal amount of swaps from Quick Sort algorithm. (*Figure 5 and 8*)
- Thanks to selecting middle pivot, Quick Sort is really efficient when sorting already sorted arrays. Pivot is the median value, thus array is divided into two equal halves, which requires the least amount of swaps. Which is the best case scenario for Quick Sort (Concerns only Quick Sort with middle selected pivot, choosing different pivot would cause a performance drop.). (*Figure 7*)
- Merge Sort appears to be overall a more efficient algorithm than Heap Sort. Excluding best case scenario for Heap Sort, which is array of constants. (*Figure 6*)