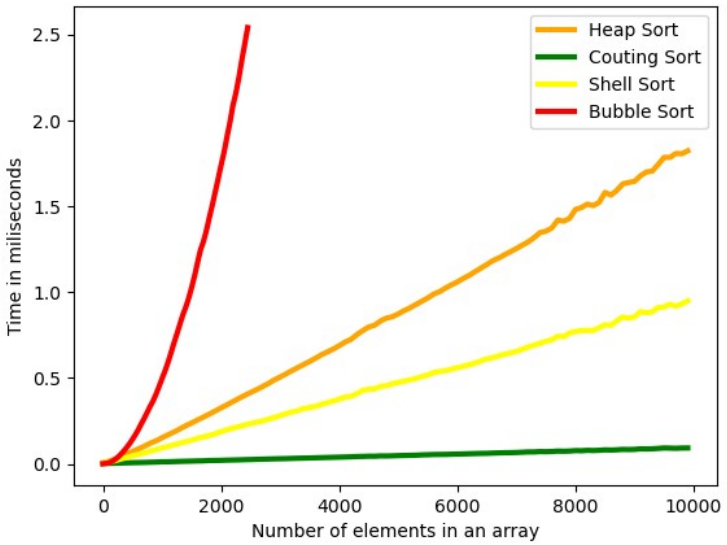
# RAPORT ON SORTING ALGORITHMS

Author: *Adam Korba*
Index: *151 962*
Course: *Algorithms and Data Structures*
Teacher: *Grzegorz Pawlak*

Raport consists of results from two experiments on most popular sorting algorithms, with focus on their time complexity. All algorithms are implemented in C++.

## EXPERIMENT I:

In this experiment I compared these algorithms:
-Bubble Sort
-Counting Sort
-Heap Sort
-Shell Sort




On the figure on the right you can see time taken to sort array with n elements. Arrays were generated randomly and had values from 0 to 100. Now I will try to summarize my findings and thoughts on each algorithm. From worst to best.

**Bubble sort**

Simply the worst of them. I implemented optimized version of this algorithm (Bubble Sort Turbo) however it couldn't improve perfomance enough to even consider this algorithm a viable option for sorting arrays with higher amount of elements. It has time complexity of $O(n^2)$. At least it doesn't use unnecesary space. It is sorting in place.

**Heap sort**

Much better than its succesor. In this one time taken to sort big arrays increases much slower.
It has time complexity of O(nlogn). Great thing about this algorithm is that it is also sorting in place, so it only occupies in memory space needed to store array being sorted.

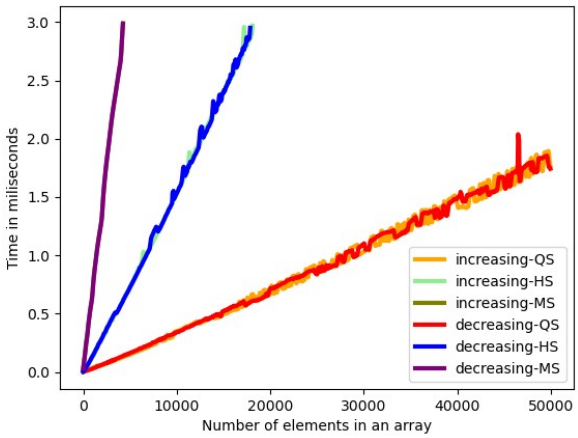
**Shell sort**

This is a great algorithm, but time needed to complete task depends greatly on gap sequence used.
I decided to use Marcin Ciura's sequence (Ciura was a Polish computer scientist), which he derived from experimentation. It isn't greatest for big arrays but for arrays around 10000 elements or lower it is suprisingly efficient (better than heap sort). Ciura didn't manage to calculate complexity of shell sort using this sequence. But complexity was measured well for other gap types. And some of them achive worst case $O(n^{4/3})$, so it would be worse than heap sort for bigger arrays. Shell sort is also sorting in place so it doesn't take much space in memory.

### Counting sort

This sort performed exceptionally well in the experiment. It came from the fact that there wasn't big range of values in arrays (max value – k = 100). Counting sort's time complexity is linear – O(n + k). It has some serious drawbacks, first of all it uses quite a lot of additional space. We need another array that stores occurences of each value in array, so it is k elements long. Counting sort also wouldn't work for floating point values. So it has amazing performance in some specific cases where dataset has many repeating values within relatively small range.

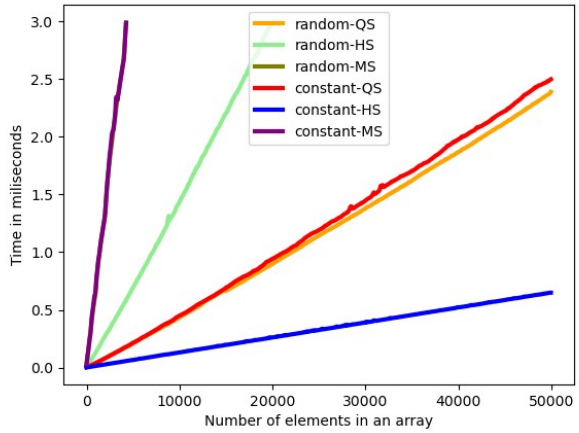
## EXPERIMENT II:




In this experiment I compared performances following algorithms:
-Quick Sort
-Heap Sort
-Merge sort
Sorting times were measured for different types of data to find other cases.

### Merge sort

It turned out to be worst of them in all types. It could be result of me not implementing it well enough, with some improvements this algorithm should have complexity O(nlogn), as it relies on divide and conquer method. Poor performance could also come from allocating memory for auxiliary arrays.

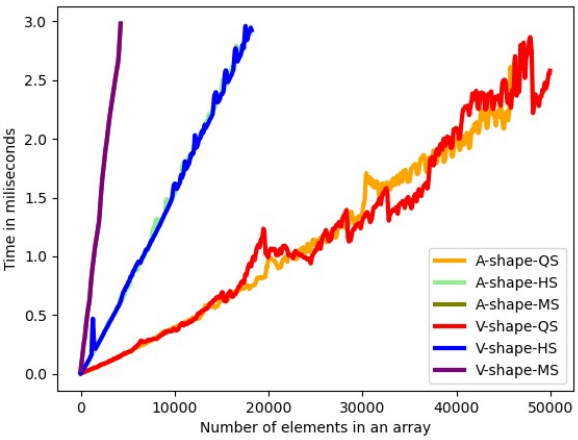



Remark: *On the graph Merge sort results hide behind each other*

### Heap sort

Not much to add. It works great for arrays with all values equal to some constant, because heap is already created (all elements are equal so tree is monotonic)

### Quick sort




It is the fastest of all algorithms tested in this document. I achieved performance of O(nlogn).
We can see that it works really good for increasing and decreasing data. And is slightly worse for random or constant arrays. The worst results were observed in A shaped and V shaped. Problem in these data types is that we hit maximum/minimum which makes quick sort inefficient. In order to addres this issue median method for choosing pivot element is often used. It ensures that algorithm never chooses a max/min element from array as pivot. It decreases chance of Worst Case in QS.