POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Report

SORTING ALGORITHMS

Adam Korba, 151962

Teacher
PhD. Grzegorz Pawlak

POZNAŃ 2022

Contents

Chapter 1

Introduction

Goal of this report is to implement and compare following sorting algorithms and show their strengths and weaknesses:

- Insertion sort - IS
- Selection sort – SS
- Bubble sort - BS
- Heap sort - HS
- Quick sort - QS
- Merge sort MS
- Shell sort -ShS
- Counting sort – CS

All algorithms were implemented in C++ programming language and tested on different randomly generated arrays/

Chapter 2

Exercises

2.1 Exercise 1

2.1.1 Implementation details

All algorithms were implemented in C++ using ordinary arrays to ensure the highest efficiency possible. Function: `int * randomArray(int n, int min = 0, int max = 100, char type)` returns pointer to randomly generated array using rand() function. You can specify different things

- n - number of elements in array
- min/max - the bottom and upper bound of values
- type - the order of elements (ex. i - increasing sequence, A - a shaped sequence, r - random sequence)

2.1.2 Experiment

In the first experiment, 4 sorting algorithms were chosen: BS, HS, CS, and ShS. An experiment was performed for arrays in sizes ranging from 0 to 500000 elements. Time was measured with function `clock()` from C header `<time.h>`. This function returns processor time at a given moment so we can measure the number of clock ticks elapsed since the start of sorting, and then divide it by `CLOCKS_PER_SEC` to get approximate time in seconds. The drawback of this method is that time in seconds may vary from the actual time elapsed, however, I decided to use this function as it gives more consistent results even despite dynamic CPU throttling.

Table below presents expected time and space complexities of tested sorting algorithms:

Table 2.1: Sorting algorithms time and space complexities complexity

Algorithm	Worst Case	Average Case	Best Case	Additional Space
Bubble sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n)$	in place
Heap sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	in place
Shell sort	$\Omega(n^{\frac{4}{3}})$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	in place
Counting sort	$\Omega(n+k)$	$\Theta(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(k)$

REMARKS: in Shell sort, it is difficult to prove an average case and it very much depends on the gap sequence chosen. 'in place" means that the algorithm doesn't require any additional space in memory to perform sorting.

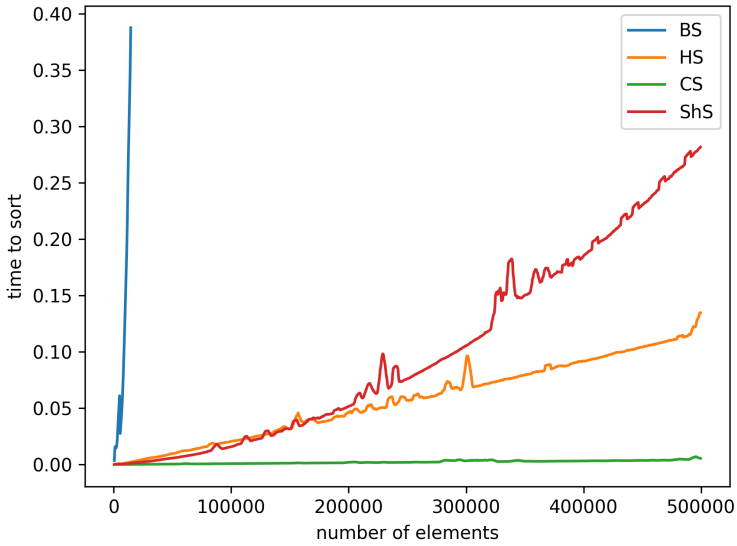


FIGURE 2.1: Results of experiment 1

2.1.3 Bubble sort

- Simply the worst of them. I implemented optimized version of this algorithm (Bubble Sort Turbo) however, it couldn't improve performance enough to even consider this algorithm a viable option for sorting arrays with a higher amount of elements. It has square time complexity. The only positive aspect is that it sorts an array in place. (it means that it doesn't need additional space)

2.1.4 Heap sort

Much better than its successor. In this one time taken to sort big arrays increases very slowly. It has logarithmic time complexity. The great thing about this algorithm is that it is also sorting in place. This algorithm utilizes the "divide and conquer" strategy.

2.1.5 Shell sort

Very interesting results. The algorithm performs better than heap sort for arrays to a size of about 100000. The reason may lie in the gap sequence used. In my implementation there is Marcin Ciura's sequence used (Ciura was a Polish computer scientist), he derived this sequence from experimentation. It isn't the greatest for big arrays but works surprisingly well for smaller ones. Ciura didn't manage to calculate the complexity of shell sort using this sequence. But complexity was measured well for other gap types. Some of them achieve worst-case $\Omega(n^{\frac{4}{3}})$.

2.1.6 Counting sort

This sort performed exceptionally well in the experiment. It came from the fact that there wasn't a big range of values in arrays (max value – k = 100). Counting sort's time complexity is linear. It has some serious drawbacks, first of all it uses quite a lot of additional space. We need another array that stores occurrences 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 the dataset has many repeating values within a relatively small range.

2.2 Exercise 2

2.2.1 Experiment

In the second experiment, the following algorithms were chosen: QS with middle selected pivot, HS and MS.

This table presents their expected time and space complexities:

Table 2.2: Sorting algorithms time and space complexities complexity

Algorithm	Worst Case	Average Case	Best Case	Additional Space
Quick sort	$\Omega(n^2)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	in place
Heap sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	in place
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

2.2.2 Constant and random

In this part duality of the heap, sort can be observed. It performed worst of the three algorithms for random data, however, it had very good performance for constant arrays (because such an array already forms a heap).

Merge sort performed worse than quicksort. And in both of them, it was easier to constant arrays as much of the work is omitted.

So far results are as expected, because all algorithms solved the task in a reasonable time, without any problems. (logarithmic time complexity)

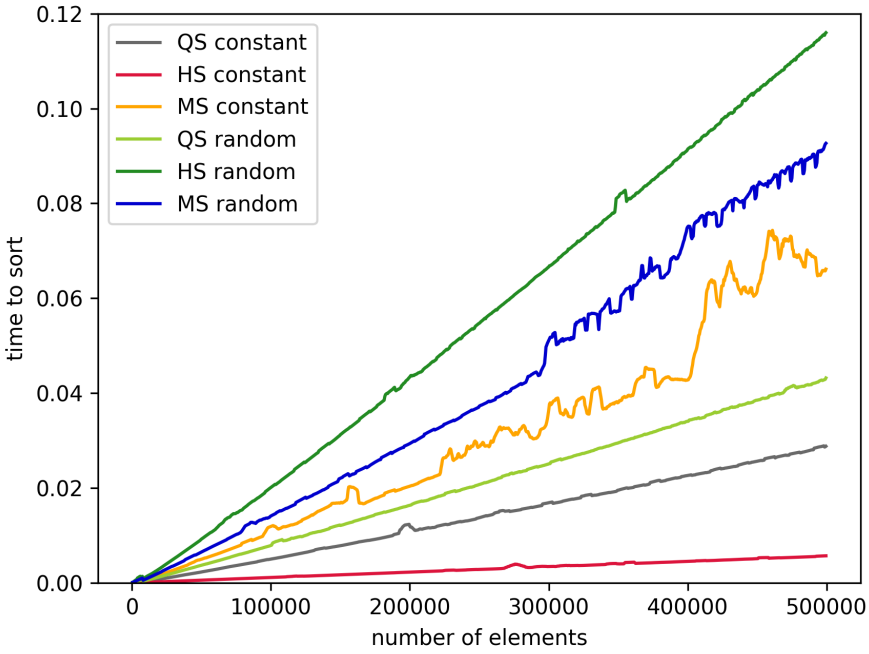
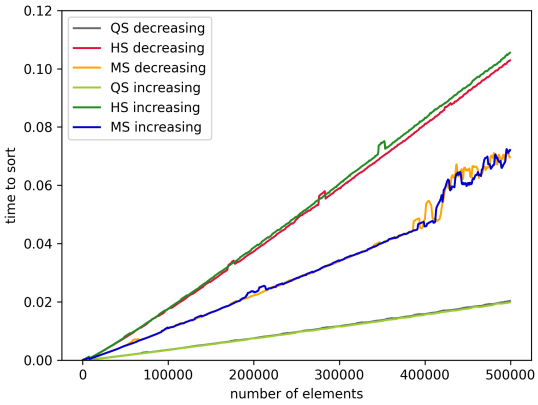
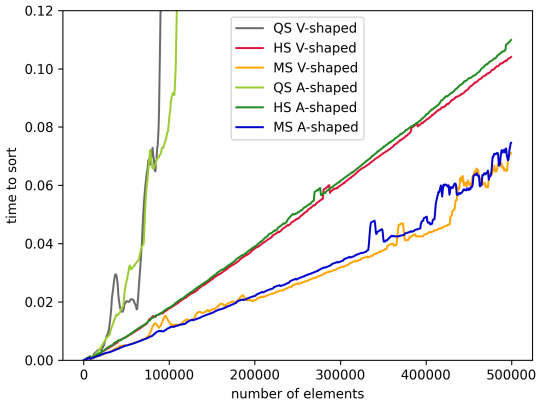


Figure 2.2: Results of experiment 2 part 1



figureA figure



figureAnother figure

2.2.3 Increasing and decreasing

In this part performances of all algorithms overlap because despite the sequence already being sorted we need to make a lot of work to ensure that. Quicksort performed the best, then merge sort and heap sort turned out to be the worst here.

2.2.4 A-shaped and V-shaped

As for merge sort, we don't see much difference from the previous experiment that is since this algorithm performs all steps even if an array is sorted. We also don't observe the difference in heap sort complexity.

Quicksort turned out to be very inefficient here. The reason for that is in A-shaped and V-shaped data we get a worst case as we always hit maximum/minimum when choosing pivot.

One way to fix this issue would be to implement the median pivot choosing technique. In this method, we would take the median from the leftmost, middle, and rightmost elements and would choose it as a pivot. In this way, we would avoid this situation for A-shaped and V-shaped.

Chapter 3

Conclusions

This experiment has shown the strengths and weaknesses of the most popular sorting algorithms. Quicksort is almost always the fastest one, however, we have to be conscious of the worst-case scenario in A-shaped and V-shaped data. We could implement quicksort with median selected pivot, but the safe option would be to use merge sort or heap sort. Shell sort showed potential for smaller input sizes. Bubble sort can be treated as educational sorting algorithm, but it proved to be very inefficient. There is also counting sort which has shown to be the best of them in the scenario when the range of values in the array isn't great and is an integer.

© 2022 Adam Korba

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.