Sorting an array

Ayda Ghalkhanbaz

Fall 2023

Introduction

This assignment involves exploring three different sorting methods and analyzing the time it takes for each of these algorithms to sort data as the size of arrays grows. Later in this report, the time complexity of these algorithms will be discussed based on some benchmark results.

Selection sort

The main purpose of this task was to implement the selection sort and derive an expression representing the behaviour of this algorithm.

Method

The implementation uses a nested for-loop and a few simple if-statements. The logic is to set the first element of the array as the smallest and then searching through the entire array for a smaller value than the candidate. If a smaller value is found, these two swap their positions. This process will be iterated till the end of the array is reached.

Result

The benchmarking results are shown in table 1 (see page 4). As we can observe, the time growth behaves like a quadratic function. Using Excel, the expression $t(n) = 0.0003n^2 + 0.04n$ was derived where 'n' is the size of the array and '$t(n)$' represents the execution time for running benchmarks on arrays of size n.

Discussion

According to the results presented in table 1 and the derived expression discussed in the result section, we can verify that the time complexity of selection sort is indeed $\mathcal{O}(n^2)$. The implementation of this method involves

a nested for-loop which leads to a quadratic time complexity in relation to the size of the array.

Insertion sort

In this task, the insertion sort algorithm is implemented and similar to the previous task, a time-size expression is derived. Later in the discussion section, a comparative analysis of both selection and insertion sort will be provided.

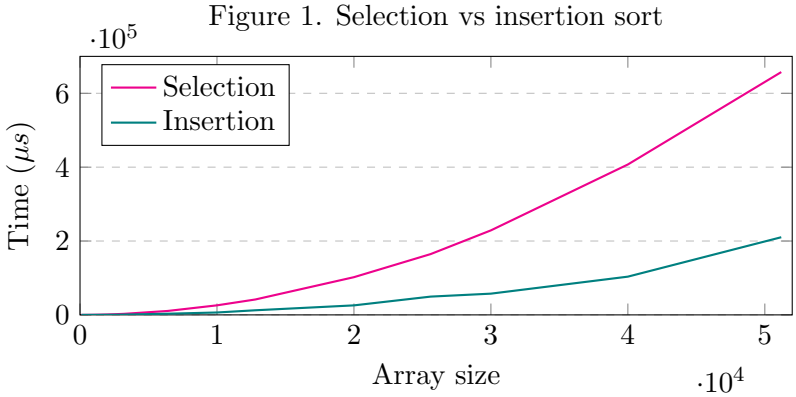
Method

Like the previous task, implementing this algorithm was quite straightforward since a part of the code was already provided in the assignment. This method was completed after adding some conditions and a few line codes. The underlying logic of this algorithm is that every element in the array compares with the element to its left, and if it is smaller then it moves to the left.

Result

The Benchmarking of insertion sort gave the results which are displayed in table 1. Using the data, the expression $t(n) = 0.00008n^2 - 0.15n$ was derived. To provide a clearer comparison between selection and insertion sort, the data is also plotted, as shown in figure 1.

Figure 1. Selection vs insertion sort



Discussion

According to the obtained results, it is evident that selection sort is approximately 3.5 times slower than the insertion sort, although both of their graphs show the same behaviour, which is a quadratic function. In fact, both of

these algorithms has same time complexity in their average and worst case, i.e. $\mathcal{O}(n^2)$. However, insertion sort has a linear time complexity $\mathcal{O}(n)$ in its best case,that is when the data is nearly sorted, while the selection sort maintains its $\mathcal{O}(n^2)$ time complexity in the worst case scenario.

Even though the selection sort has a better time complexity in numbers of swaps, insertion sort minimizes the number of unnecessary swaps and comparisons, which makes this method faster than the selection sort in practice.

In conclusion, both of these algorithms share the same time complexity in their average case scenario, but as mentioned earlier insertion sort can be faster in practice, even though it is not the case in theory. Nevertheless, there are more efficient and faster algorithms which are preferred for sorting larger datasets.

Merge sort

In this task a faster sorting algorithm was implemented. Merge sort implementation involved a recursive method. Similar to the two previous tasks, a time-size expression was derived. The time complexity of merge sort and a comparative analysis of all these three algorithms have been covered in the discussion section.

Method

Implementing this task was followed by some challenges. Since a recursive method is used in the implementation, it took a while to understand the underlying logic of this algorithm. However, it was easy to write the code because of the clear instructions in the assignment. Following these instructions resulted in a completed and working algorithm. In summary, the merge algorithm divides an array into two parts, sorts each of parts by utilizing recursion (calling a method in itself), and finally merges the two sorted parts together.

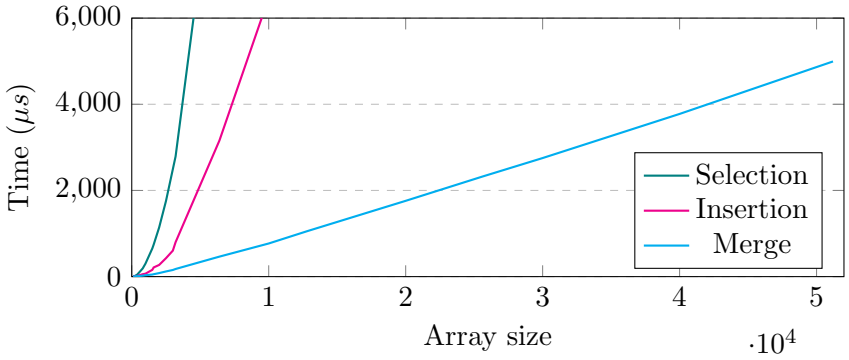
Result

The results of benchmarking for merge sort are shown in table 1 as well. The expression $t(n) = 0.098n - 87.7$ represents the behaviour of this sorting method based on the data from table 1. Figure 2 provides a clearer comparative visualisation between all three algorithms.

Array size	Selection	Insertion	$\frac{Selection}{Insertion}$	Merge	$\frac{Insertion}{Merge}$
100	16	8	2.01	6	1.33
200	11	3	3.26	6	0.63
400	46	15	3.17	10	1.39
800	190	56	3.43	22	2.51
1600	743	213	3.49	56	3.82
3200	2789	799	3.49	177	4.55
6400	10631	3158	3.37	464	6.78
12800	41695	12467	3.34	1054	11.88
25600	164625	49490	3.33	2322	21.40
51200	657509	210277	3.13	4994	42.04

Table 1: Benchmark results of selection, insertion and merge sort in μs

Figure 2. Selection, insertion vs merge sort



Discussion

From the data shown in table 1, we can observe that merge sort is faster in comparison with insertion and selection sort. Now if we compare insertion and merge sort, as the size of array doubles, the ratio of the execution time grows doubles as well. Thus, merge sort is preferred in sorting of large datasets.

Due to the divide–conquer algorithm that merge sort has, sorting the larger datasets would not take a long time. This algorithm e divides a very large array into manageable parts. Through recursive methods, the array divides into two parts repeatedly until the smallest arrays have a size of one. After the sorting of all these arrays, the parts are merged and resulting in a completely sorted array.

The merge sort algorithm has a time complexity of $\mathcal{O}(n \cdot \log(n)$ which explains the execution time in comparison with insertion and selection sort.