Sorting an Array

Abdinasir Abdi

September 2024

1 Introduction

This report concludes the fourth assignment in course ID1021, taken in the fall of 2024. The assignment explores three different sorting algorithms and how they differ. Each algorithm works on different-sized arrays and evaluates how the performance time changes.

2 Selection Sort

The first search algorithm explored is called selection sort. This algorithm follows simple logic and starts by comparing the first element in the sequence with the rest of the array and takes the smallest value, which is then swapped with the first element. This process is repeated until the end of the array is reached and all the elements in the array are sorted. The method's functionality can be implemented by the following pseudo code:

```
n = length of array

for i from 0 to n - 2
    minIndex = i

    for j from i + 1 to n - 1
        if array[j] < array[minIndex]
            minIndex = j

    if minIndex != i
        swap(array[i], array[minIndex])
```

One iteration is not enough to give an accurate measurement. The sorting algorithm iterated multiple times sorting different-sized arrays and eventually returned the the average time for all iterations. The following results were obtained through a benchmark on selection sort:

Iterations	**Size (n)**	**Average Runtime**
1000	100	6.7
1000	200	13.3
1000	400	34.7
1000	800	159
1000	1600	575
1000	3200	1968
1000	6400	10900

Table 1: Selection sort for different sized arrays, measured in μs

From the results, the selection sort algorithm follows the quadratic time complexity of $O(n^2)$. The logic for the search algorithm is simple to implement and very capable of handling smaller data sets but inefficient for larger data sets.

3 Insertion Sort

The next sorting algorithm, insertion sort, builds a sorted array one element at a time. It sequentially takes one element and compares it to previous elements in the array. All previous elements that are larger than the current one shift to the right and the current selected element is temporarily inserted where it belongs. This process is repeated until all elements are checked and sequentially sorted. The following pseudo-code was used to construct the insertion sort algorithm:

```
insertionSort(array)
for i from 1 to length(array) - 1
currentValue = array[i]
j = i - 1

while j >= 0 and array[j] > currentValue
     array[j + 1] = array[j]
    j = j - 1

array[j + 1] = currentValue
```

The following results were obtained through a benchmark on insertion sort:

Iterations	**Size (n)**	**Average Runtime**
1000	100	4.2
1000	200	16
1000	400	39
1000	800	159
1000	1600	658
1000	3200	2236
1000	6400	8350

Table 2: Insertion sort for different sized arrays, measured in μs

The results confirm the time complexity to be quadratic $O(n^2)$. The performance is slightly better than the selection sort algorithm but still follows the same pattern of change and is still inefficient for bigger-sized arrays.

4 Merge Sort

Merge sort is a more advanced sorting algorithm that follows a divide-and-conquer approach. The algorithm starts by dividing the array in half, recursively sorting each half, and then merges it into a newly sorted array. Merge sort works as follows:

- Divide: Sequentially dividing the array into two smaller subarrays until each subarray only has one element, which naturally makes it sorted.
- Conquer: All subarrays are recursively sorted.
- Merge: The now sorted subarrays are then merged and sorted to form a single sorted array.

The following were obtained through a benchmark on merge sort:

Iterations	**Size (n)**	**Average Runtime**
1000	100	8.8
1000	200	13.4
1000	400	21.7
1000	800	49.5
1000	1600	110
1000	3200	191
1000	6400	325

Table 3: Merge sort for different sized arrays, measured in μs

From a simple observation, it would seem that merge sort follows a linear time complexity. Looking at it further and measuring for larger data sets confirms the time complexity to be $O(n * log(n))$. Merge sort is an incredible sorting algorithm, highly efficient, and the time complexity is the same for best, worst, and average cases, making it desired for larger data sets.

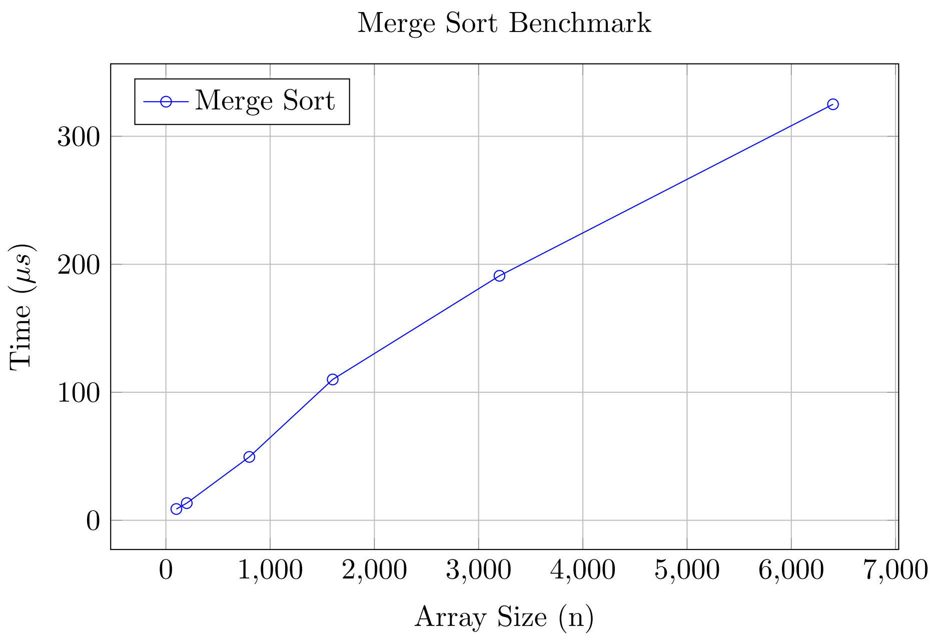

Figure 1: Runtime as array gets bigger

5 Conclusion

Selection -and insertion sort have the same time complexity of $O(n^2)$ and perform better than merge sort for smaller data sets. Both sorting algorithms follow a similar logic, using comparison and swapping procedures to sort the array correctly. It becomes tedious and inefficient as the array gets bigger because the algorithm iterates over every element in the array. Merge sort on the other hand divides the array in half and recursive sorts both parts. The final step recursively sorts the subarrays and puts all elements into a final sorted array. By following this operation logic the runtime for sorting an array becomes very efficient for larger arrays.

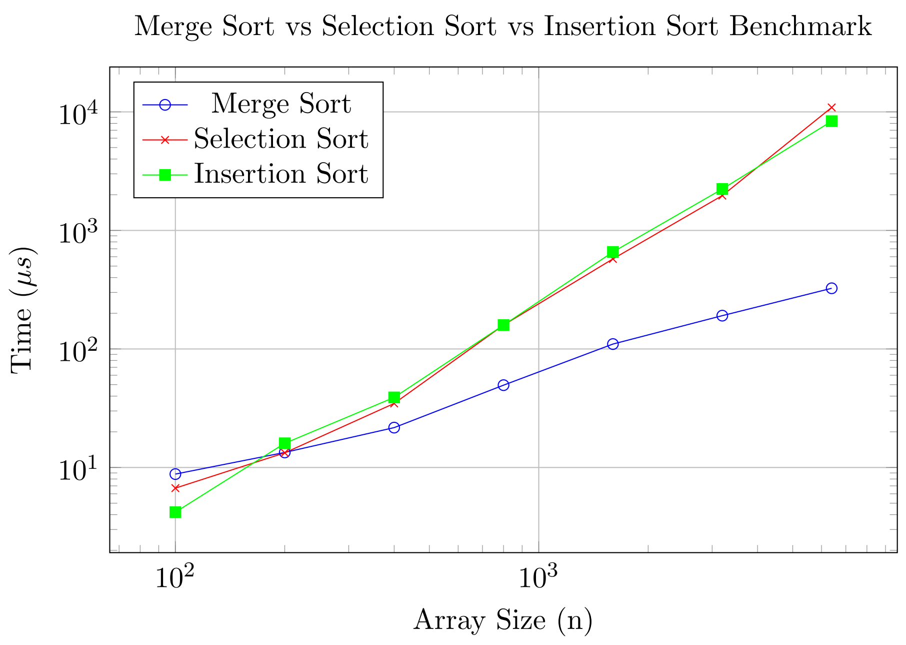

Figure 2: Logarithmic runtime comparison between Merge Sort, Selection Sort, and Insertion Sort