Performance of Sorting Algorithms on *n sized* Array

Muhammad Shahwaiz Abbasi

VT 2026

Introduction

This report presents the fourth assignment for the Data Structures and Algorithms (ID1021) course. We will be analyzing the performance of different sorting algorithms on n sized array. We will be covering three types of sorting algorithms:

- **Selection Sort**
- **Insertion Sort**
- **Merge Sort**

We will also compare these three algorithms at the end of the report.

I measured time in nanoseconds using a function from `Assignment 1` and then converted it to milliseconds to make the graph clearer and more readable.

The objective of this assignment is to understand how sorting works and how it can be done in different ways, as well as in an efficient way.

Selection Sort

We will start with a simple algorithm that is not very efficient but simple to implement.

```
void selection_sort(int array[], int size){
    for (int i = 0; i < size - 1; i++){
        int candidate = i;
        for (int j = i; j < size; j++){
            if (array[j] < array[candidate]){
                candidate = j;
            }
        }
```

```
            if (candidate != i){
                int e = array[i];
                array[i] = array [candidate];
                array[candidate] = e;
            }
        }

}
```

The code above starts by initializing an index to the first position in the given array. It then finds the smallest element from that index to the end of the array. If there is a smallest element, the smallest value is swapped with the element at the current position and then move forward to the next position in the array. The process repeats until the end of the array

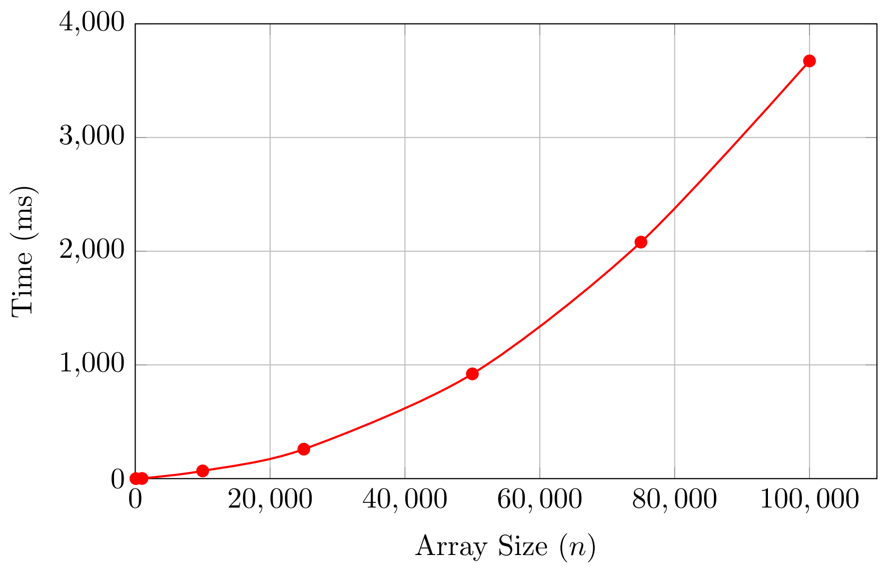

Figure 1: Selection Sort algorithm's time performance

Looking at `Figure 1`, when the array size doubles, the runtime increases by almost four times. Therefore the time complexity for the Selection Sort algorithm can be written as $\mathcal{O}(n^2)$. The main reason behind this is that selection sort must go through every element in the array and it does not exit even if the array is already sorted. This makes selection sort inefficient.

Insertion Sort

The second sort we will be implementing is `insertion sort`. It is generally more efficient than selection sort.

```
void insertion_sort(int array[], int size) {
    for (int i = 1; i < size; i++) {
        for (int j = i; j > 0 && array[j] < array[j - 1]; j--) {
            swap(array, j, j - 1);
        }
    }
}
```

The code above initializes with the first index in the array. It checks if the item at the index is smaller than the item before, if true it then swaps them both and continues checking. After swapping it moves the index forward one step and repeats until the entire array is sorted. Insertion Sort works best when the array is already sorted, giving us a time complexity of $\mathcal{O}(n)$.

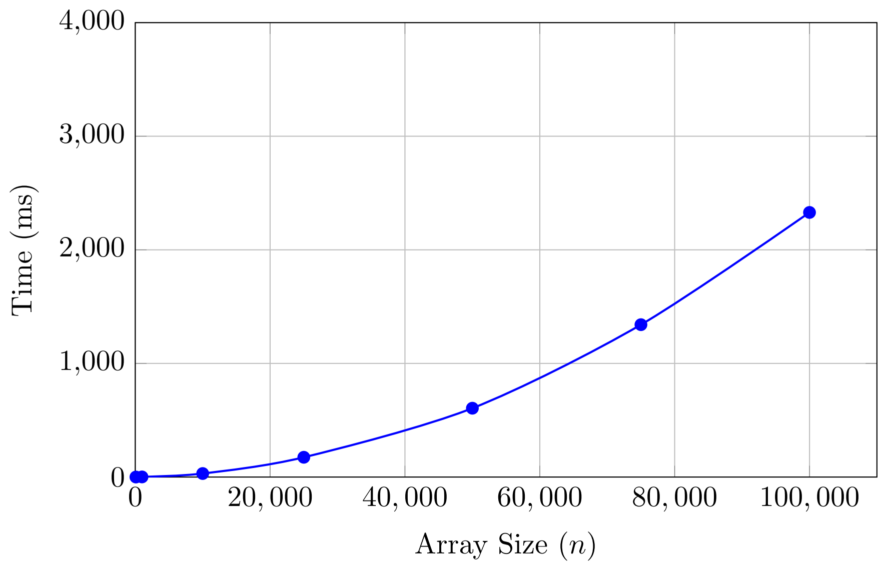

Figure 2: Insertion Sort algorithm's time performance

`Figure 2` might look similar to `Figure 1` but looking closely it tells us that insertion sort is more time efficient than selection sort. Although both algorithms share the same *worst case* and *average case* time complexity of $\mathcal{O}(n^2)$.

Merge Sort

Last but not least, we will be implementing merge sort. This is the most time efficient of all the sorting algorithms that we have implemented so far.

```
void mergesort(int *org, int n) {
    if (n == 0)
    return;
    int *aux = (int*)malloc(n * sizeof(int));
    sort(org, aux, 0, n-1);
    free(aux);
}
void sort(int *org, int *aux, int lo, int hi) {
    if (lo != hi) {
        int mid = (lo + hi)/2;
        sort(org, aux, lo, mid);
        sort(org, aux, mid+1, hi);
        merge(org, aux, lo, mid, hi);
    }
}
void merge(int *org, int *aux, int lo, int mid, int hi) {
    for ( int i = lo; i <= hi; i++ ) {
        aux[i] = org[i];
    }
    int i = lo;
    int j = mid+1;
    for ( int k = lo; k <= hi; k++) {
        if (i > mid) {
            org[k] = aux[j++];
        } else if (j > hi) {
            org[k] = aux[i++]   ;
        } else if (aux[i] <= aux[j]) {
            org[k] = aux[i++];
        } else {
            org[k] = aux[j++];
        }
    }
}
```

Above are the three functions that are combined to create a merge sort. The function `mergesort` is basically the setup function that we used when we want to sort in `main`. This function takes two arguments, the original array and the size of the array. We start by checking whether there is something to sort or not. Then we allocate space in memory for our auxiliary array. At last, the function calls the recursive sort function to sort the entire array.

The `Sort` function is a recursive sort function that takes four arguments. Two arrays and a starting index and an ending index of the array. It starts by finding the middle index to split the array. The function calls itself to first sort the first half of the array then calls again to sort the second half of the array with the help of the middle index. This is called *recursively sorting*. Finally, it calls a merge function where it merges the two halves of the array into one sorted array.

The third function we made is the `merge` function. It merges two already sorted sub arrays. It takes 5 arguments. Two of them are the arrays and the rest are integers that represents the first, middle and last indices of the array. This function starts by copying all the elements from the original array into the auxiliary array. It then checks if either left sub array or right sub array is empty. If one side is empty it takes the remaining elements from the other side. After that it compares the current elements from both subarrays and insert the smaller value in the right position in original array.

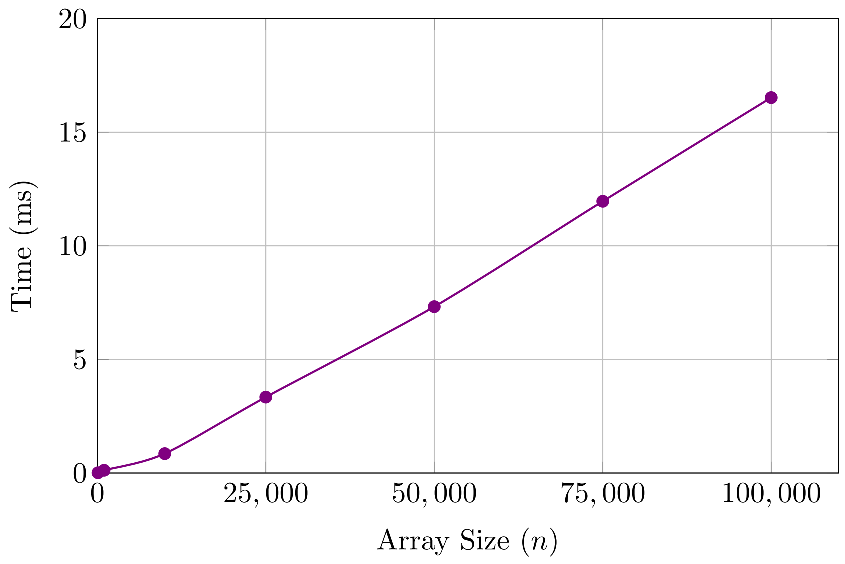

Figure 3: Merge Sort algorithm's time performance

As seen in `Figure 3` the graph looks like a linear graph with the complexity of $\mathcal{O}(n)$ but the merge sort actually has a time complexity of $\mathcal{O}(nlog(n))$. The graph is almost identical for smaller numbers.

A Comparison of Selection, Insertion and Merge Sort

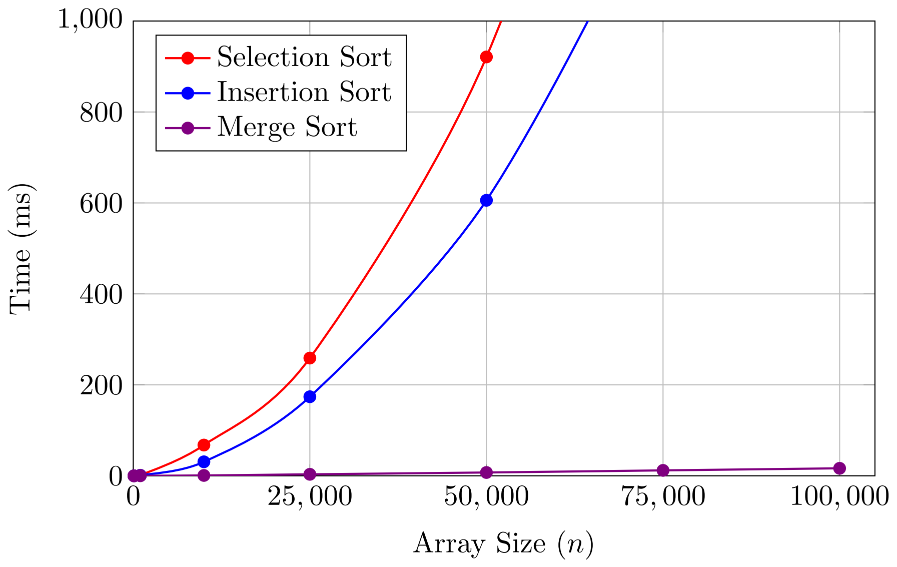

Figure 4: Performance analysis of sorting algorithms on an n-sized array.

Looking at `Figure 4`, we see that merge sort is the most efficient sorting algorithm among selection sort and insertion sort.