Program 4 Report

Disha Mahajan

Contents

[Introduction 3](#_Toc119955917)

[BubbleSort Versus Insertion Sort 4](#_Toc119955918)

[ShellSort Compared to Other Sorts 5](#_Toc119955919)

[MergeSort Versus IterativeMergeSort 6](#_Toc119955920)

[QuickSort 7](#_Toc119955921)

[Conclusion 8](#_Toc119955922)

# Introduction

Through this program I learned a variety of skills related to time complexity analysis and found out which sorting methods were better than others. My discovery allowed me to find that the slowest sorting algorithm was BubbleSort while the fastest sorting algorithm was QuickSort. I made some interesting discoveries through my analysis which will be insightful to depict through a multitude of graphs. I will compare each sorting method in this report. In the end, I will show the final graph which contains all the sorting method, although some sorting methods take such little time that it will not be noticeable in the graph.

# BubbleSort Versus Insertion Sort

BubbleSort and InsertionSort are similar in the aspect that the worst-case time complexity and average case time complexity are **both** **O(n2).** BubbleSort compares elements that are next to each other and swaps if they are in the wrong order. InsertionSort on the other hand constructs a sorted list by moving elements one at a time. Thus, the primary thing to note between both sorting methods is that BubbleSort, because it swaps adjacent elements only, takes a larger number of swaps than InsertionSort, which sorts by inserting the selected element in the proper place of the sorted portion it is building. Due to requiring a higher number of swaps to occur, BubbleSort takes more time to complete than InsertionSort. This is reflected in ***Figure 1***. To better show InsertionSort and its curve, please take a look at **Figure 2**.

***Figure 1***

**Figure 2**

Thus, the following can be concluded:

* BubbleSort has more swaps than InsertionSorts
* InsertionSort is faster than BubbleSort
* BubbleSort swaps adjacent elements while InsertionSort inserts elements in a sorted side of the vector or array

# ShellSort Compared to Other Sorts

ShellSort has an average case time complexity of **O(n3/2)** and a worst-case time complexity of **O(n2).**

**Figure 3**

When we compare ShellSort to other sorting methods such as MergeSort **(O(n\*log(n))** or IterativeMergeSort **(O(n\*log(n)))** we can notice that it is slower than these algorithms; however, when we compare ShellSort to InsertionSort **(O(n2))** we can see that it is significantly faster. The strength of ShellSort lies in how it can swap elements that are far apart from each other. In fact, it is popularly known to be an optimization of InsertionSort. InsertionSort will still make many movements to move elements far apart, but ShellSort is better for this, thus it is faster. Such comparisons can be seen visually in ***Figure 3*** which shows this is a graphical format.

# MergeSort Versus IterativeMergeSort

MergeSort and IterativeMergeSort both have an average and worst-case time complexity of **O(n\*log(n))**. What is noticeable from the graph and through theory is that IterativeMergeSort is slightly faster than recursive, or normal, MergeSort. Essentially, IterativeMergeSort and MergeSort both apply the same algorithm. They keep halving an unsorted array or vector until only one element remains. Then they sort these halves and finally merge them together to get a sorted list. The reason IterativeMergeSort is faster than recursive MergeSort is because when executing the algorithm, MergeSort creates and deletes multiple local arrays/vectors every call, while IterativeMergeSort does not use extra space and has a temporary vector/array already allocated beforehand. It does not need to create several versions. Thus, by not using extra memory, IterativeMergeSort is slightly faster than normal, or recursive, MergeSort. This is shown in ***Figure 4***.

**Figure 4**

# QuickSort

QuickSort is the fastest sorting algorithm. It has an average case time complexity of **O(n\*log(n))** and a worst case time complexity of **O(n2).** The reason it is the fastest algorithm is because it partitions the input in the list without taking extra space. To elaborate, QuickSort is better because it does not need to allocate extra memory in order to sort. For example, when compared to MergeSort, QuickSort is faster. This is because MergeSort allocates memory every time it partitions which makes it less efficient. Take a look at **Figure 5** to see the drastic time difference of execution between QuickSort and MergeSort.

**Figure 5**

# Conclusion

Through my analysis it can be concluded that BubbleSort is the slowest sorting algorithm with an average time complexity of **O(n2).** Next, InsertionSort is the second slowest sorting algorithm with a average case time complexity of **O(n2).** Thirdly, ShellSort is the third slowest sorting algorithm with a time complexity of **O(n3/2).** After this, MergeSort is the fourth slowest sorting algorithm and has an average case time complexity of **O(n\*log(n)).** IterativeMergeSort has the same average case time complexity as MergeSort but is a slightly faster algorithm which puts it as the second to last slowest sorting algorithm. Lastly, the quickest sortest algorithm is QuickSort with an average case time complexity of **O(n\*log(n)).** **Table 1** and **Figure 6** further show this analysis in visual format. Note that in Figure 5 only the QuickSort and InsertionSort are visible since the rest of the algorithms are much smaller.

**Table 1**

|  |  |  |
| --- | --- | --- |
| Time Complexities | | |
| Algorithms | Average Case | Worst Case |
| QuickSort | O(n\*log(n)) | O(n2) |
| IterativeMergeSort | O(n\*log(n)) | O(n\*log(n)) |
| MergeSort | O(n\*log(n)) | O(n\*log(n)) |
| ShellSort | O(n3/2) | O(n2) |
| InsertionSort | O(n2) | O(n2) |
| BubbleSort | O(n2) | O(n2) |

**Figure 6**