**Sorting Algorithms**

**By: Trinity McCann**

**Co-Researchers: Same Burns & Emily Henderson**

**CS404 with Wajeb Gharibi**

**4/29/2025**

**Table of Contents**

**Introduction** ................................................................................................................... 2  
**Objectives** ...................................................................................................................... 2 **Methodology** ................................................................................................................. 2

**1. Algorithms** ................................................................................................................. 2  
 1.1 Overview .............................................................................................................. 2  
 1.2 Time Complexity .................................................................................................. 3  
 Figure 1.1: Time Complexities Chart ..................................................................... 3

**2. Quick Sort** .................................................................................................................. 3  
 2.1 Code Implementation ........................................................................................... 4  
 Figure 2.1: Code for Quick Sort ............................................................................. 4  
 2.2 Performance Tests ........................................................................................…….4  
 Figure 2.2: Test with 200 Input Size ....................................................................... 4  
 Figure 2.3: Test with 1,000 Input Size .................................................................... 4  
 Figure 2.4: Test with 10,000 Input Size .................................................................. 5

**3. Bubble Sort** ............................................................................................................... 5

**4. Merge Sort** ................................................................................................................. 5

**Conclusion** .................................................................................................................... 5

**References** .................................................................................................................... 5

# Introduction

The following documentation outlines the combined research efforts completed by Trinity McCann, Sam Burns, and Emily Henderson. The purpose of this project is to gather, analyze, test, and report on the effects, efficiencies, and findings regarding a variety of sorting algorithms in attempts to further understand definitions of sorting algorithms, their time complexities, real-world applications, and how they compare to one another.

By the end of this research, the three authors and all readers of the documentation shall come out with a deeper level of understanding and greater knowledge of sorting algorithms with the ability to both discuss them in depth and apply them in realistic scenarios.

# Objectives

The research for this documentation will be divided up into three different scopes, each done by a prospective co-researcher. Each of the scopes will undertake a specific sorting algorithm. The algorithms, alongside their respective researchers are as follows: Quick Sort by McCann, Bubble Sort by Henderson, and Merge Sort by Burns. This specific selection of sorting algorithms gives us greater insight into how time complexity affects efficiency where Quick Sort is generally one of the fastest algorithms, Bubble Sort is one of the slowest, and Merge Sort standing as a control group.

# 

# Methodology

Requirements for the completion of the research includes that each of the three sorting algorithms be coded and tested by their respective researcher. In addition, visuals shall be provided for each sorting algorithm, showing the nature of the search and how it works. Relevant code and all visuals will appear in the documentation and presentation for the comprehensive understanding of the readers. After coding and testing is complete, the researchers will compare their findings and provide an agreed upon conclusion of the findings.

The goal of this document is to outline overviews, findings, and the Quick Sort algorithm. It will not go into full detail of the other two algorithms as there are separate documents addressing those algorithms in their entirety completed by the two co-researchers.

# Algorithms

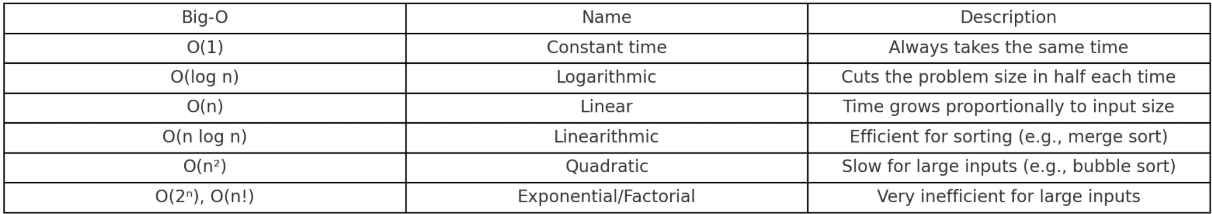
An algorithm is a procedure that is systematically designed to solve computation problems step-by-step. For something to count as an algorithm, it must provide some sort of solution to some problem that can be repeated by following its outlined steps.

Some important definitions for algorithms include well-designed, correct, time efficient, and space efficient. A well-designed algorithm is one that is definite, easy to understand/execute, and can be completed in a finite number of steps. A correct algorithm is one that consistently produces the desired output for all valid inputs within a given problem’s constraints. Time efficiency is defined as an algorithm’s ability to accomplish its goal with minimal wasted effort and resources. Space efficiency is defined as the most effective use of working space (usually memory) during execution.

# Time Complexity

Time complexity discusses the tendencies of an algorithm as the input size reaches infinity. While time complexity does not give us an exact time estimate of a specific input, it is useful for determining and comparing the efficiencies of algorithms, especially when large datasets are considered.

Time complexity can be written in three forms: Big O which represents “Worst Case” or Upper Bound, Big Theta (Θ) which represents “Average Case”, and Big Omega (Ω) which represents “Best Case” or the Lower Bound.

Due to the difficulty of accurately calculating the average case, time complexity is usually determined by its Big O notation. The following chart is useful for understanding some of the most common growths.

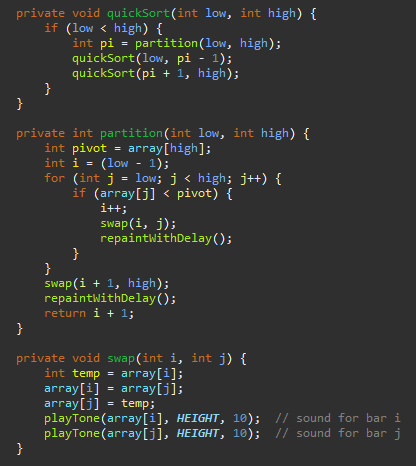
# Quick Sort

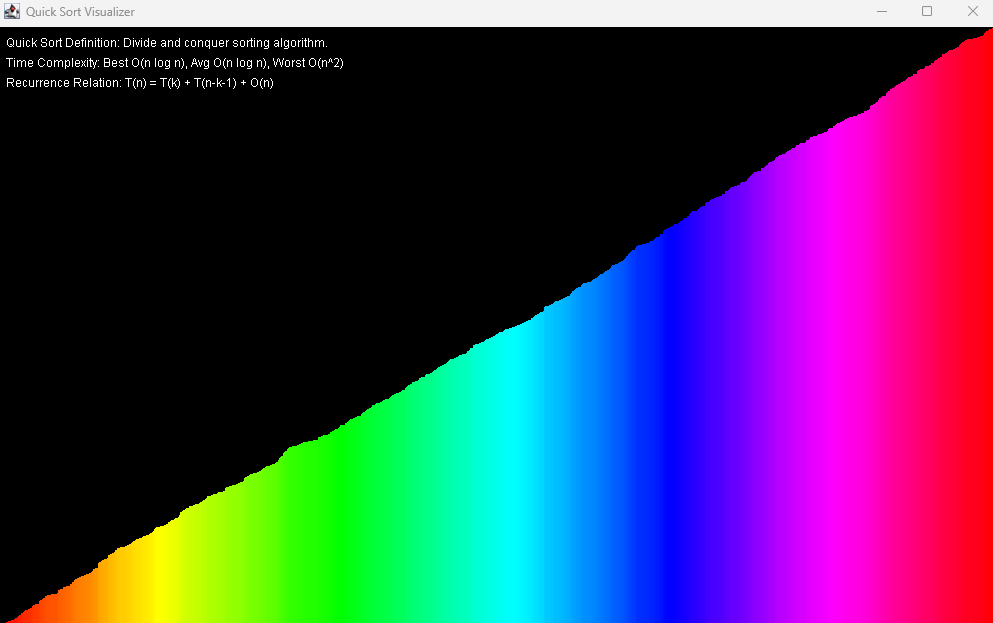
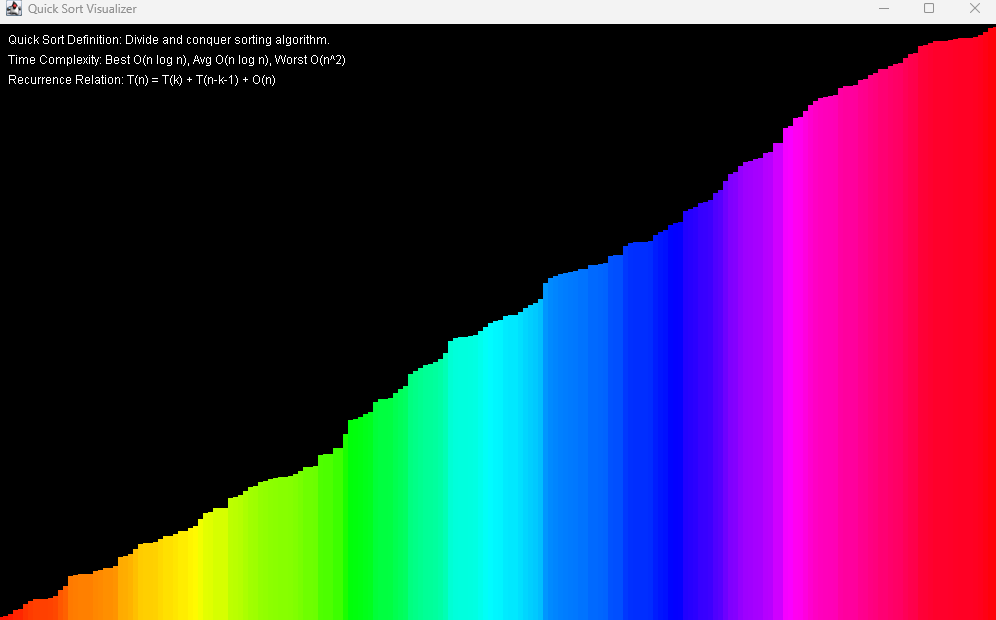
Quick Sort is a sorting algorithm that gets its name from its impeccable ability to swiftly sort even large datasets. Though Quick Sort’s worst case is defined as O(n2), which puts it on par with most other sorting algorithms, its greatness lies in its average case of Ө(nlogn). This average case makes it one of the fastest and most efficient sorting algorithms used today.

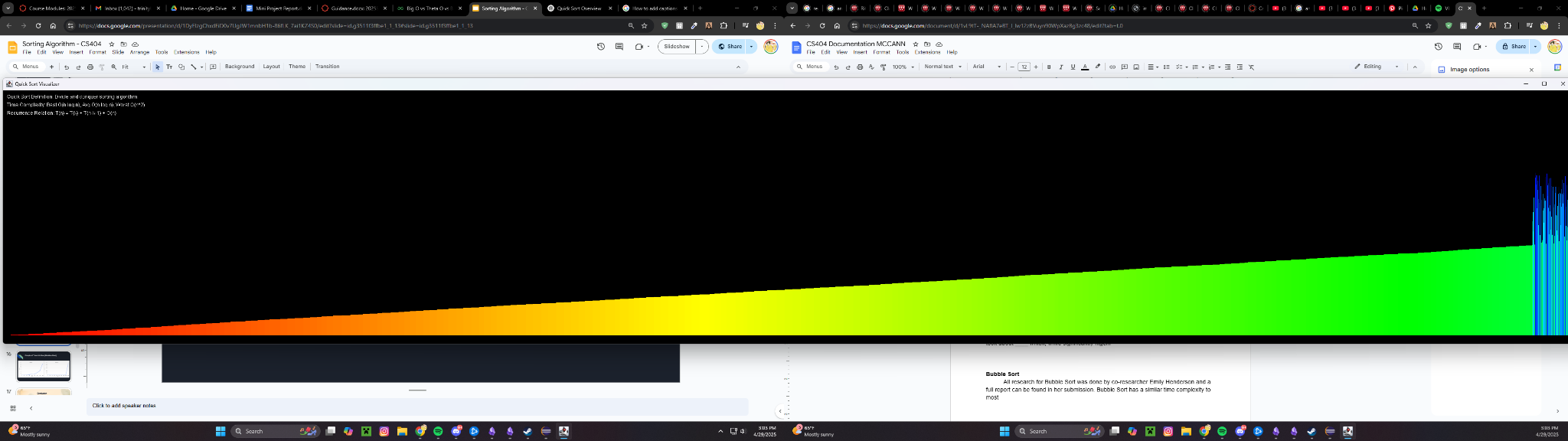
There are three simple steps to complete the Quick Sort algorithm onto a set of data:

1. Choose a pivot point in the array (This is often the first or last element of the array)
2. Partition the array by arranging the elements so that everything lower goes to the left and everything higher goes to the right
3. Recursively apply these steps to the sub-arrays

Quick Sort has a number of key details about it that can help us find out why it is so efficient. Firstly, Quick Sort utilizes the Divide & Conquer Approach, where the data is broken up into smaller sub-arrays and those sub-arrays are handled independently of each other before being arranged back together. Quick Sort is also known as an In-Place algorithm, which refers to its ability to not require extra memory proportional to the input size (This is dissimilar to other popular sorting algorithms, such as Merge Sort). Quick Sort is an unstable sorting algorithm, which means that it does not preserve the relative order of equal elements. So, in a dataset where the chronological ordering of added values is necessary to remain the same, Quick Sort will not be a viable option. Quick Sort is considered random, meaning that the efficiency of any given data set will change depending on the pivot point selected within the array. Lastly, Quick Sort is considered recursive by nature, as it will continuously apply itself to the subarrays it creates.

The following implementation of Quick Sort has three primary methods that achieve the three steps of the sorting algorithm.The first method, labelled “quickSort” serves as a sort of main method that runs and calls the other functions with their variables. Its key detail is that it establishes the variable known as “pi” which presents the pivot point that Quick Sort is so well known for. Next up is the “partition” method which completes the divide and conquer approach of Quick Sort.”partition” sets the pivot as the first value in the array and begins to sort through them. It determines sorting by determining if a number is lower than it, if it is it will run the “swap” method. The swap method will put lower values before the pivot and keep higher values after itself.

All the code completed for Quick Sort was completed with assistance of OpenAI’s ChatGPT 4o model and compiled using Eclipse. It creates a visual that utilizes color and sound to effectively show sorting. The sorting done inside of the presentation only includes an input size of 200 which took about 20-25 seconds of compilation time on average. I did some testing with much larger input sizes which scales greatly in time. It should be taken into consideration that the program is not only doing the compilations but providing a visual and sounds as well which take up compilation time. The input size of 1,000 to about 2 minutes and 50 second and a final run of a data input size of 10,000 took about 29 minutes and 40 seconds which, while significantly higher, does reflect the input size and time complexity of Θ(nlogn) very well. 

My computer screen could not show the sorted 10,000 input in its entirety. I like these visualizations because you can tell how large the input size is by the width of the bars and softer fade of the gradient.

# Bubble Sort

All research for Bubble Sort was completed by co-researcher Emily Henderson and a full report can be found in her submission.

# Merge Sort

All research for Merge Sort was completed by co-researcher Sam Burns and a full report can be found in her submission.

# Conclusion

Firstly, our research and findings were considered to be a resounding success. Sorting algorithms have been long studied and tested so it is very good news that our findings match that which is well known. The algorithms we hypothesized to do the best and worst accurately reflected such and our theories regarding their best use-cases proved to be accurate as well.

Bubble Sort’s implementation quickly skyrocketed into the hours whereas Merge Sort and Quick Sort swiftly outshone it. The differences in Merge Sort and Quick Sort were also easily visible through minor amounts of testing. In conclusion, Quick Sort stays one of the best, most efficient sorting algorithms. Each algorithm has their use-cases however. Merge Sort, unlike Quick Sort, is stable. This means it retains the original ordering of exact values from the input data which is extremely useful in many real world scenarios that might make one choose it in places where Quick Sort would ruin the sorting of data.

# References

Wajeb Gharibi, Ph.D. Associate Teaching Professor, Division of Computing

ChatGPT (For assistance in generation of code)