Project: Algorithm Analysis

Kavan Singh

May 10, 2023

**TABLE OF CONTENTS**

**Part 1: Comparison of Sorting Algorithms**

Introduction……………………………………………………………………………. 3-7

Methodology………………………………………………………………………….. 8-10

Results……………………………………………………………………………….. 11-21

Analysis……………………………………………………………………………... 22-30

**Part 2: Problem-Solving and Analysis**

Brute-Force Approach………………………………………………………….………. 31

Efficient Approach……………………………………………………………….……... 32

References………………………………………………………………………………………. 33

***Part 1: Comparison of Sorting Algorithms***

**INTRODUCTION**

During the semester, I concentrated on analyzing and determining the time complexity and rate of growth for various algorithms using varying input sizes. The objective of the first part of this project is to observe how the theoretical analysis of an assortment of sorting algorithms compares with their actual performance. The sorting algorithms I will be focusing on are the following:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Merge Sort
5. Quicksort
6. Heapsort
7. Counting Sort
8. Radix Sort

**Insertion Sort**

Insertion sort works by inserting elements one by one into a sorted portion of the array until all elements are sorted. In the best-case scenario, the input array is already sorted, and the algorithm requires only one comparison per element, resulting in a time complexity of O(n). In the worst-case scenario, the input array is sorted in reverse order, and every element must be moved to its correct position in the sorted array. This results in a time complexity of O(n2), as each element must be compared to all the elements that come before it. In the average case scenario, the time complexity of insertion sort is also O(n2), as the average number of comparisons and swaps required to sort an array of size n is proportional to n2.

**Selection Sort**

The way Selection Sort works is by dividing an input list into two separate parts. The parts consist of a sorted and unsorted list. After, the smallest element is taken from the unsorted list and transferred to the sorted list. This is done until the full list is sorted. The time complexity of the selection sort algorithm is O(n2) where n is the number of elements in the list. Although it is not very efficient when compared with other sorting algorithms, it is easy to implement and understand. If used, it should be when small lists are present.

**Bubble Sort**

Bubble Sort is a sorting algorithm that swaps adjacent elements in ascending order and descending order depending on the goal of what kind of comparisons you require. For example, as a comparison-based sorting algorithm, if a smaller element comes after a larger element, it may be swapped if the goal is to have larger elements at the beginning of the list. The time complexity of bubble sort is O(n2). Since it is inefficient, it is limited to when it is best used. When a list is partially sorted, it may be used as it would require fewer swaps as the comparisons would identify it as not necessary.

**Merge Sort**

Merge-Sort is a sorting algorithm that uses the divide and conquers technique to sort an array or list of elements. The way it works is by dividing the input array in half recursively, sorting the halves, and then combining them to get the sorted solution. This sorting algorithm utilizes the merge operation which does the comparisons between two arrays or lists and combines them. For the merge operation, the time complexity is O(n). However, for the full Merge-Sort algorithm, the time complexity is O(nlgn). This makes it one of the most desirable sorting algorithms to implement for large arrays or lists. A disadvantage of Merge-Sort is that for the merge operation, temporary arrays are put to use, so more space is needed. Something merge-sort is widely known for is its stability. This means it preserves the relative order of equal elements of a dataset.

**Quicksort**

Quicksort, similar to Merge-sort, is a recursive sorting algorithm that follows the divide-and-conquer technique. It works by selecting a pivot element from an input array and then partitioning it into sub-arrays. Two sub-arrays will be formed where one contains elements less than the partition element and the other contains elements that are larger than the partition element. The pivot or partition element is placed accordingly to its correct position and the procedure is repeated on the sub-arrays that were made from the input array until the entire original array is sorted. For this sorting algorithm, the time complexity is heavily dependent on if the sub-arrays created are balanced or not. If balanced, the average time complexity is O(nlgn), and when not balanced, the average time complexity is O(n2). Overall quicksort requires less memory usage, is usually good for larger arrays, and is known for cache efficiency. It should be probably avoided when arrays are smaller and have the same elements as they will not be made use of properly and the partition step becomes purposeless.

**Heapsort**

Heapsort uses a binary heap data structure that takes the smallest element and inserts it into the beginning of the array. The best-case time complexity of Heapsort is O(nlgn), which occurs when the input array is already sorted. In this case, the heapify operation will take O(n) time and the time complexity of the Heapsort is O(nlgn). The worst-case time complexity of Heapsort is also O(nlgn). This occurs when the input array is in reverse order, and the heapify operation takes O(lgn) time for each element. In this case, the time complexity of Heapsort is O(nlgn), and finally, the average-case time complexity is also O(nlgn) because the heapify operation takes O(lgn) time for each element. Heapsort is considered one of the most efficient comparison-based sorting algorithms, but it is also unstable.

**Counting Sort**

Counting sort is a non-comparison-based sorting algorithm that sorts an array by counting the number of occurrences of each unique element in the array. The best-case time complexity of Counting Sort is O(n + k), where n is the size of the input array and k is the range of the input data. This occurs when all the elements in the input array are distinct. In this case, the counting and summing operations take O(n) time, and the time complexity of the algorithm is O(n + k). The worst-case time complexity of Counting Sort is also O(n + k). This occurs when all the elements in the input array are the same, or when the range of the input data is significantly larger than the number of elements. In this case, the counting and summing operations take O(k) time. The average-case time complexity of counting sort is also O(n + k). Although the algorithm's running time may depend on the range of the input data, Counting Sort is generally considered one of the most efficient sorting algorithms when the range of the input data is relatively small compared to the size of the input array.

**Radix Sort**

Radix sort is a non-comparison-based sorting algorithm that sorts an array by sorting the elements by their digits, from the least significant digit to the most significant digit. The time complexity of Radix Sort is O(d(n+k)), where d is the number of digits in the maximum element in the array, n is the size of the input array, and k is the range of the input data. The best-case time complexity of radix sort is O(d(n+k)), which occurs when all the elements in the input array have the same number of digits. The worst-case time complexity of Radix Sort is also O(d(n+k)), which occurs when all the elements in the input array have different numbers of digits. The average-case time complexity of Radix Sort is also O(d(n+k)). Radix sort is considered to be efficient when the number of digits is relatively small compared to the size of the input array, and the range of the input data is not too large.

**METHODOLOGY**

**CPU | RAM | IDE**

*AMD Ryzen 7 3700X @* 4.4 GHz | 32 GB | Windows Subsystem for Linux on Ubuntu through the Z Shell kernel

**Timing Mechanism:** Milliseconds, calculated via high\_resolution\_clock::now() and duration\_cast<milliseconds>(end-start) functions in the *chrono* library of C++.

When I first thought of this project, I was reminded of a time I was tasked to test different hash functions hundreds of times against multiple files. I did not want to have to put myself, through having to run each file multiple times manually, so I decided to automate the process. my methodology was implemented in f my parts: create test files, create the scripts to iterate across the test files, test the files, and put the results in a readable format that provided the data I needed to complete the assignment. I benchmarked my implementation of all eight sorting algorithms on my machine. I tested all iterations (Approximately five hundred runs) through the Windows Subsystem for Linux – a virtual machine – on Ubuntu through the Z Shell kernel.

First, I wanted to generate a large field of files for us to be able to test. I wrote a program that first generated files filled with random numbers to sort. The following were the sizes I tested: 100, 1000, 10000, 50000, 100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000, and 1 million elements for a total of f my teen different test files. Initially, I filled those files with random data. This generated the average case for all sorting algorithms as the data *could* be inadvertently sorted or could be truly random considering I used the C standard library’s **srand()** function for the random number generator. I did try to mitigate this by seeding the random number generator with **srand(time(NULL));** however, this still creates pseudo-random numbers that could or could not affect my results. Therefore, I used modulus to better scatter the random numbers across 0 - 9999. Once this program was done, titled *generator.cc*, I ran a script titled *file\_generator.sh* to generate those files. At the group's request, I modified his program to include files to test the best case (everything is already sorted) and the worst case (everything is in reverse order) across all algorithms. These files were slightly modified and can be found in the ***s my ce\_files*** directory in my [GitHub](https://github.com/kavansingh582/CSCI115-Term-Project) repository.

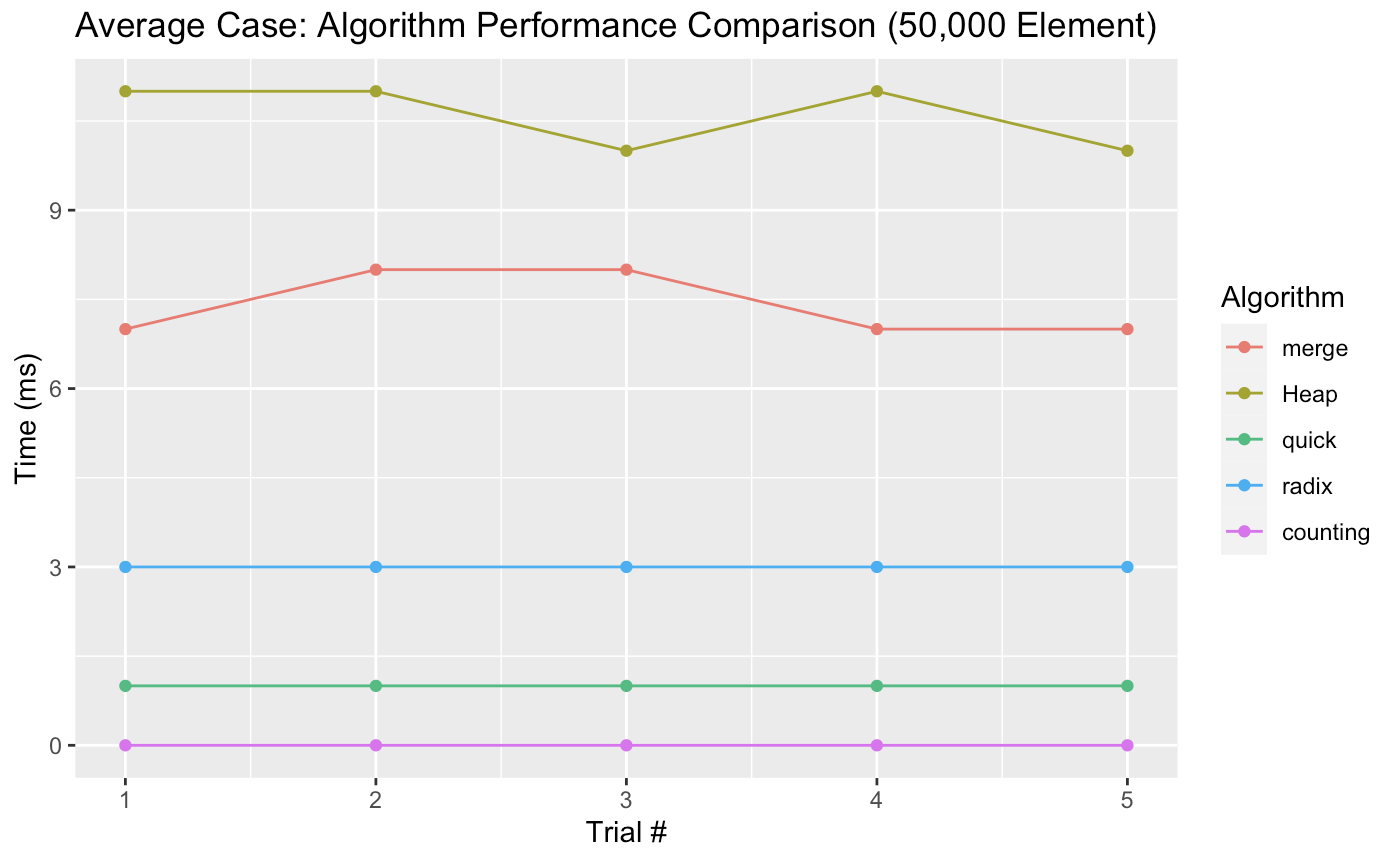
After I generated the files, I created a driver program named *main.cpp*. Inside this file, I developed two separate ways for us to test my program: Through Command Line Arguments (which is how the scripts are used) and manually through user input. Regardless of testing, the overall flow is the same. The user decides which sorting algorithm they want to use and which file they want to read from. This creates an array of the size of the test file and passes that array, the size, and the sorting algorithm the user chose into a function that runs the chosen sorting algorithm. Once that function executes, it returns the sorted array as well as prints the time it took to sort the number of elements inside the array. The *README* text file goes further into how to compile and execute the program on y my machine.

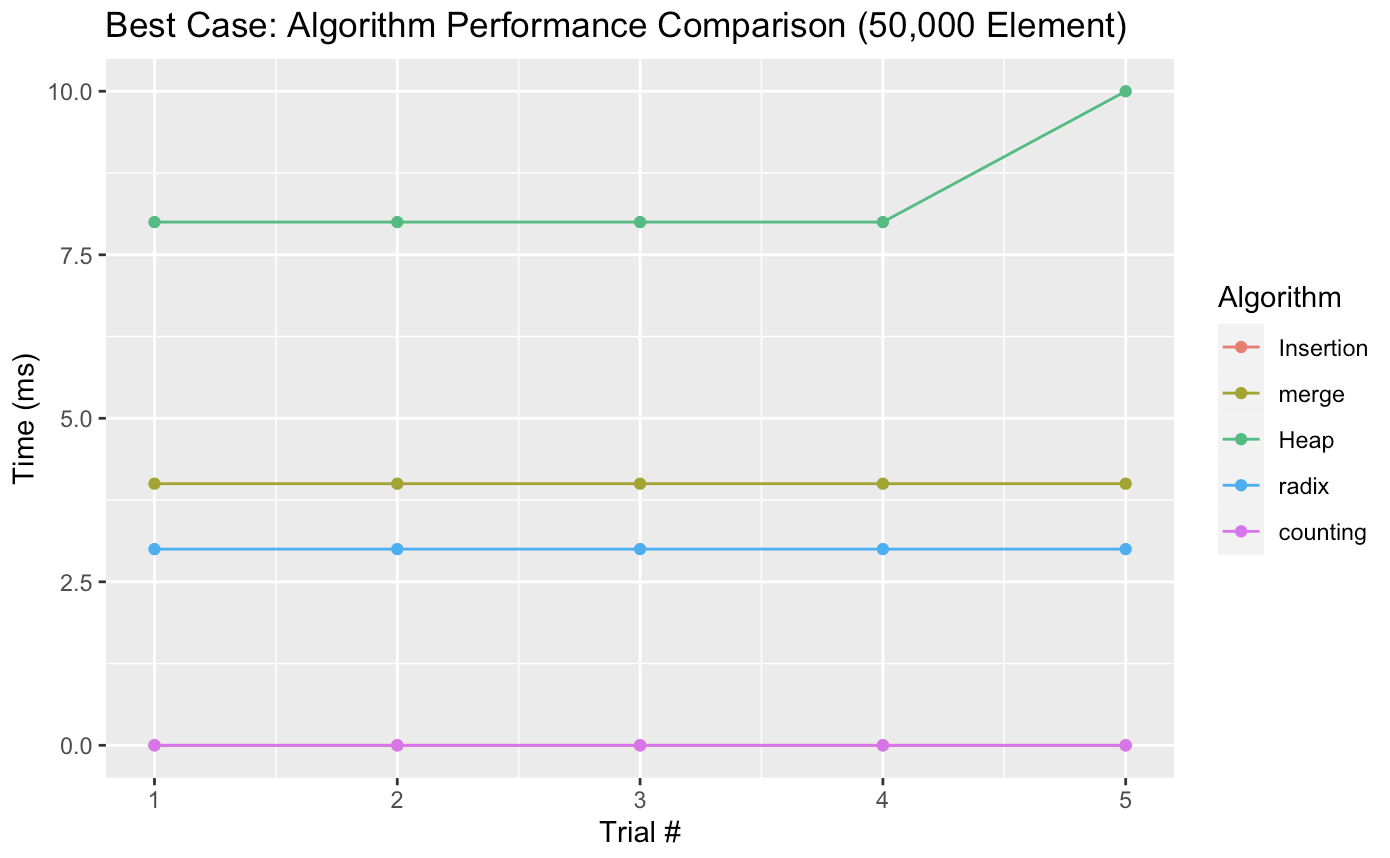
Once the driver program finished, I developed a script that would automatically run through every file and every algorithm and put the output into a text file. This script would automatically compile the program at the start of the script and run through each file (f my teen total), across each algorithm (eight algorithms), and run those files on those algorithms five times each. This results in five-hundred sixty (560) total runs. Not only that but I ran this three different times as well: Once for the best case (sorted already), once for the average case (could have some elements pre-sorted), and once for the worst case (non-sorted or reverse order). This results in one-thousand six-hundred eighty runs to get my data. This is why I did not want to run each iteration by hand.

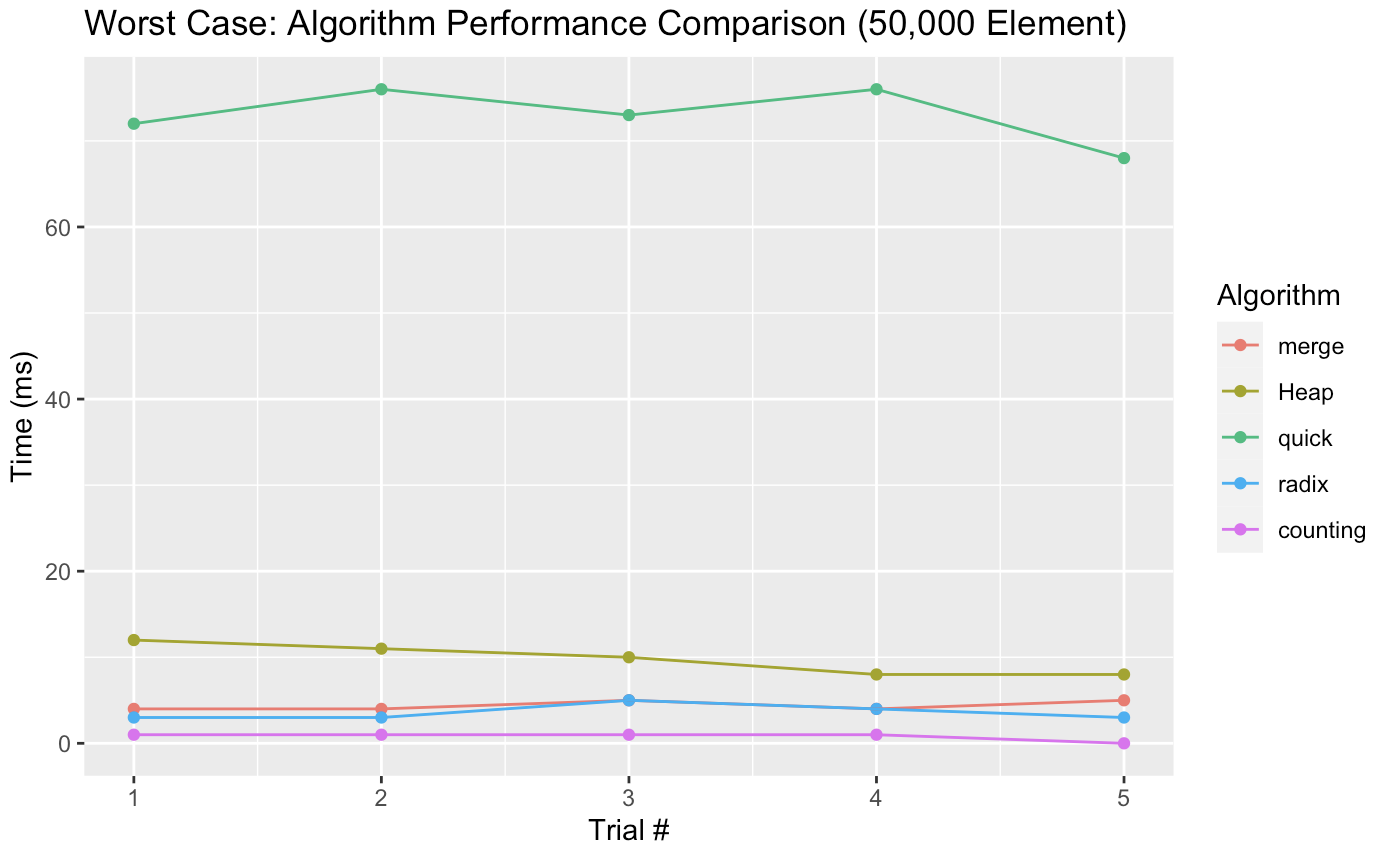
Once I received all the data required, I put it in a readable format that allowed the group to go through to check all the data so I could graph my results. The files I used to graph my results were *worst\_case.txt*, *best\_case.txt*, and *average\_case.txt*. These also can be found in my GitHub repository under ***output\_files***.

**RESULTS**

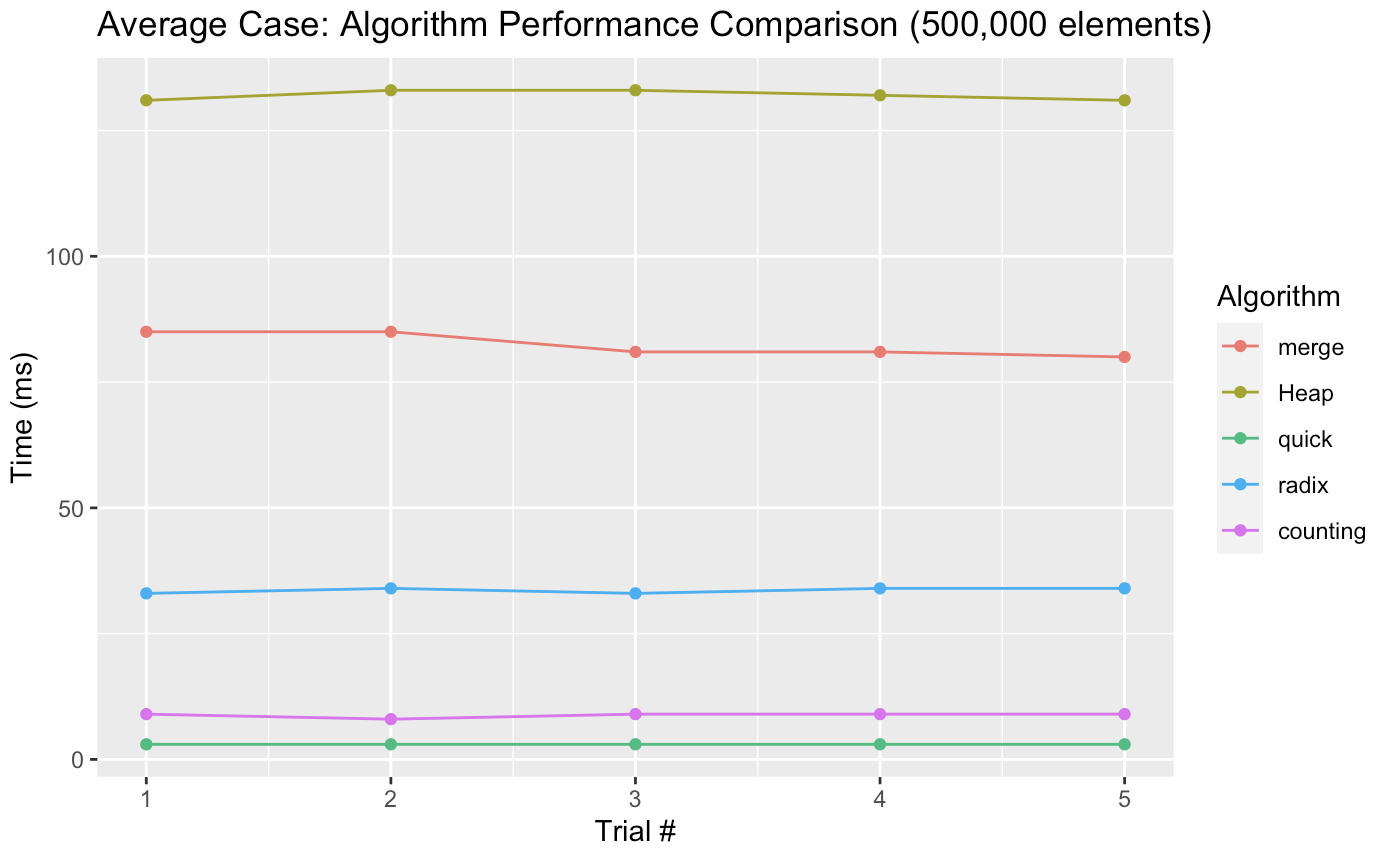
The graphs of data sets for 50k numbers on each algorithm over 5 trials are shown below. This was graphed using a point-line graph over the 5 trials, and the data was programmed and analyzed in R-Studio. This is for the average-case, best-case, and worst-case data where the numbers are in random, sorted, and reverse order respectively. It is important to note that the time required for radix, merge, and counting sort should not vary significantly between average, worst, and best cases. This is because the algorithms' time complexity remains the same regardless of the presented cases since the manipulation of the algorithms isn't influenced by the speed or slowness of comparisons. I cut out significantly time-consuming algorithms such as insertion, bubble, and selection sort because the time taken for these comparisons was significantly higher than all of the others, and they skewed data. However, in the case of the best-case data, I kept insertion in and removed quicksort, because insertion was incredibly fast in the best-case scenario, while quicksort was one of the three slowest algorithms for the best-case data. All of these times are taken in milliseconds.

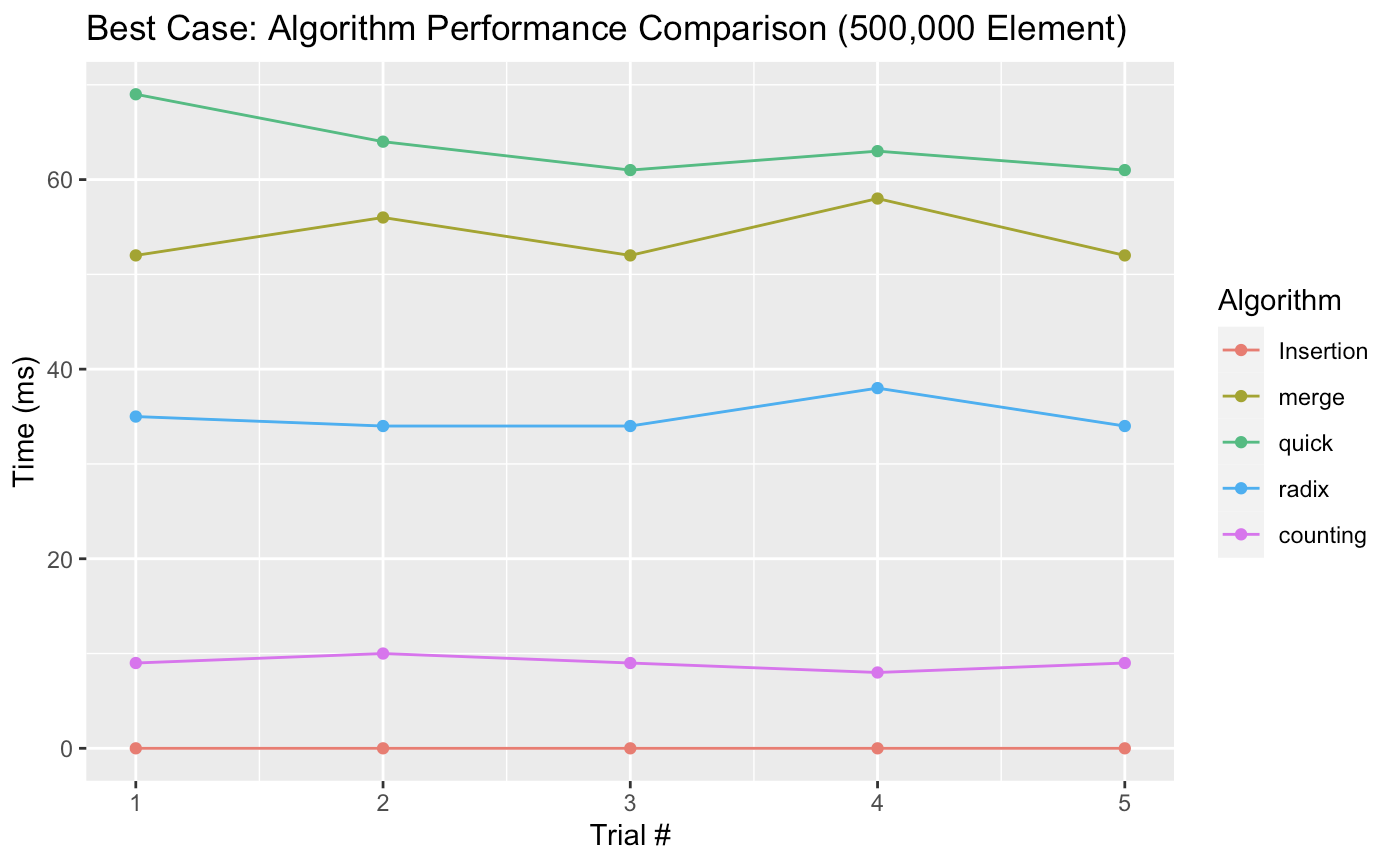
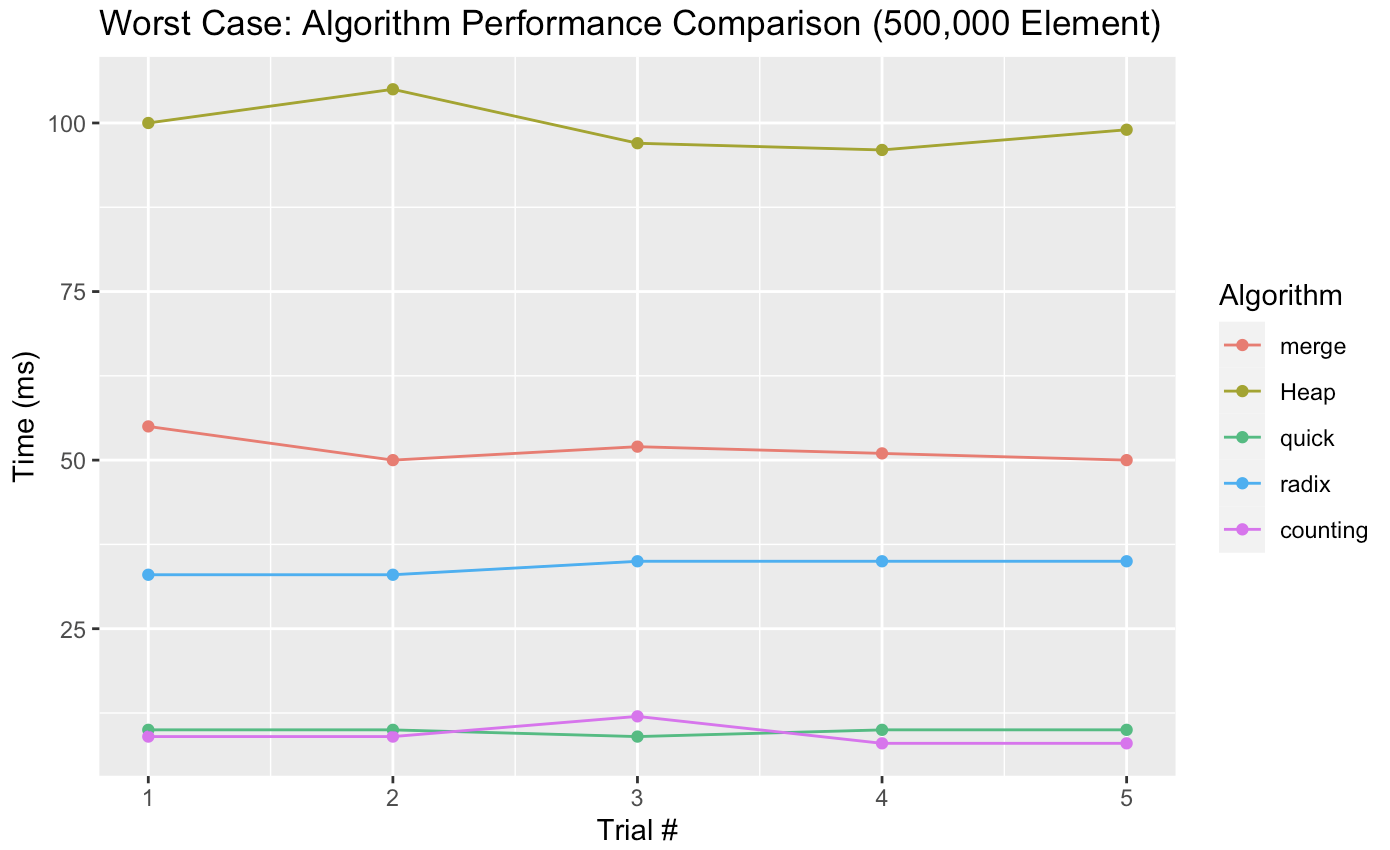




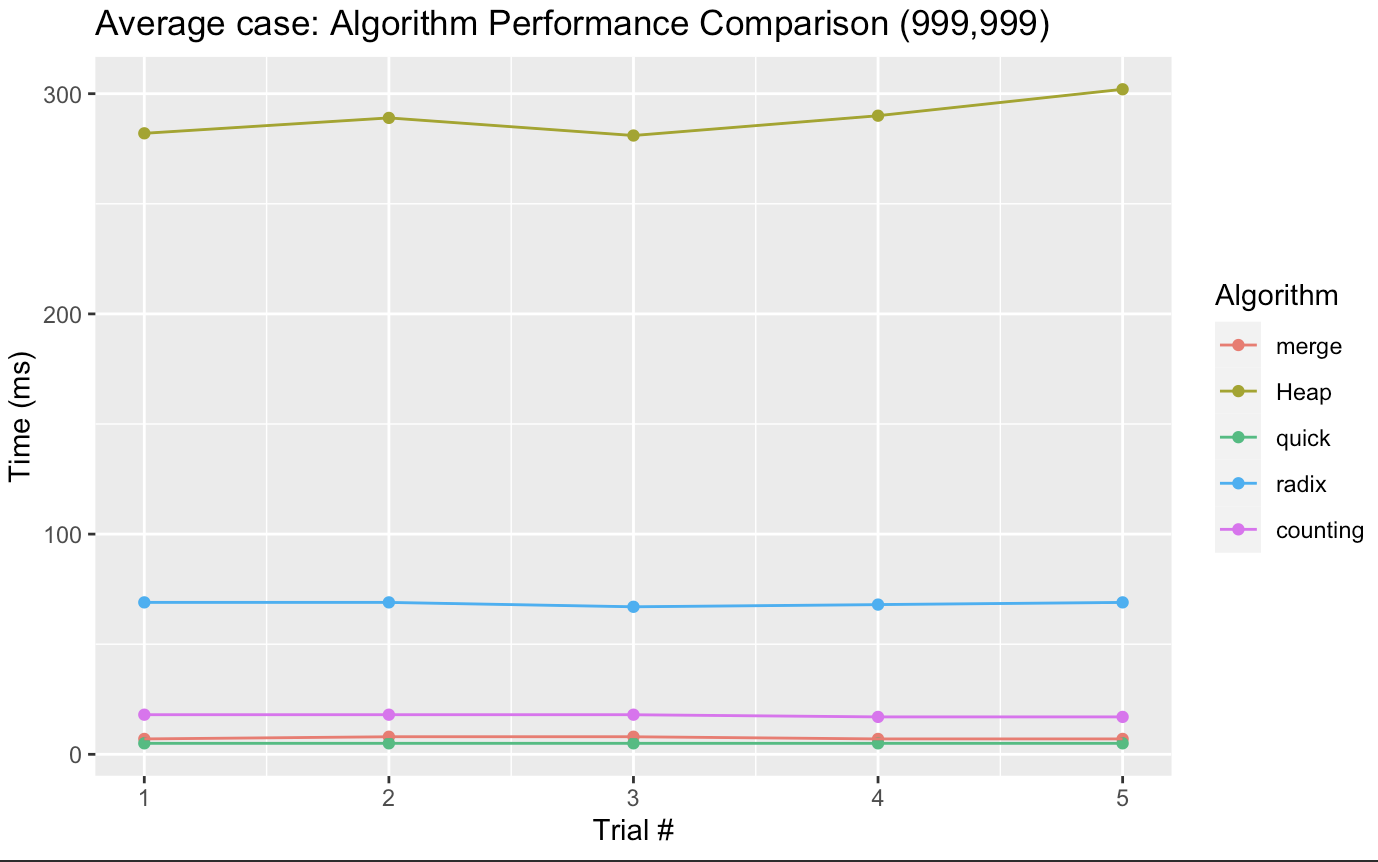


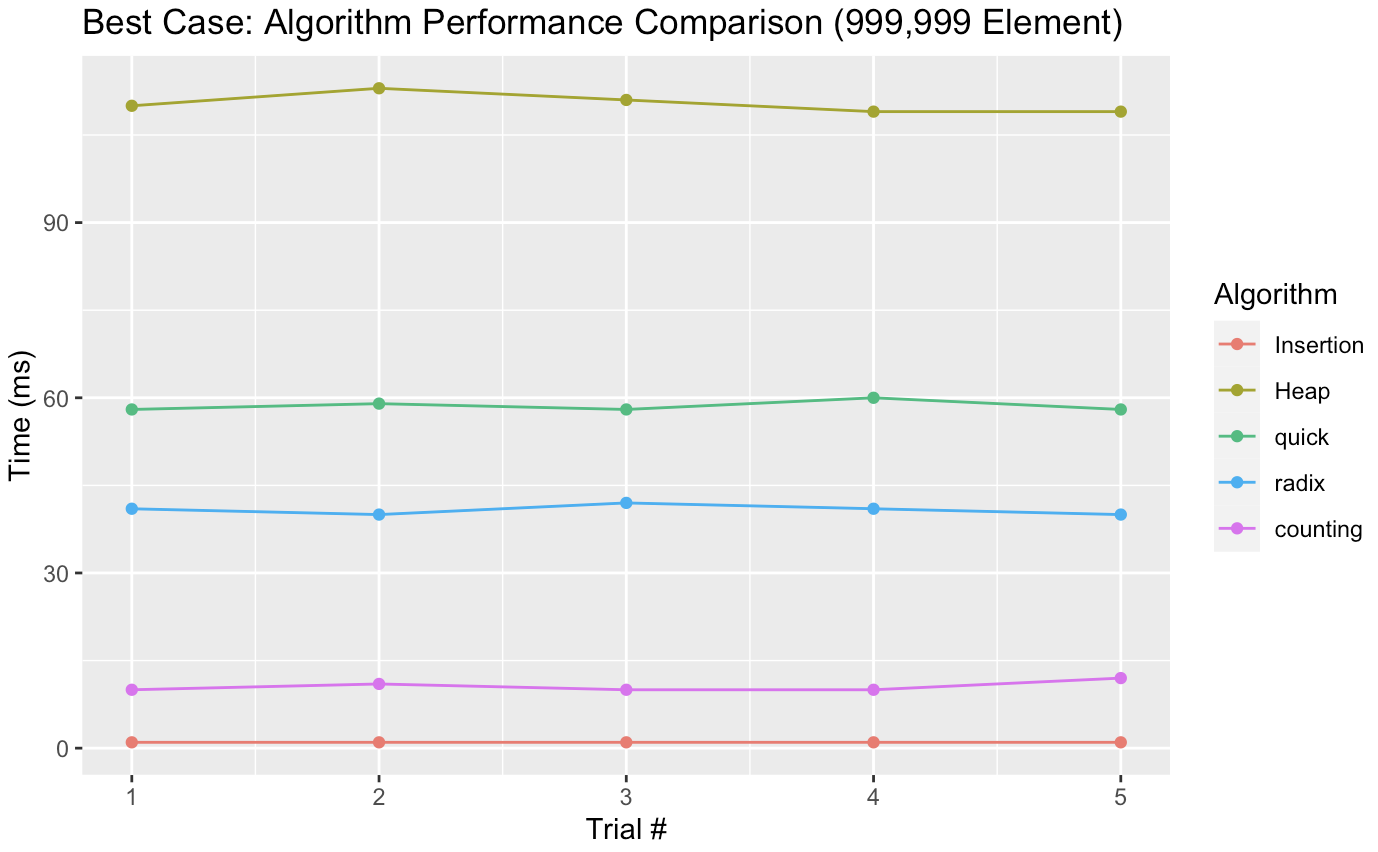
500k: The graphs of data sets for 500k numbers on each algorithm over 5 trials are shown below. This was graphed using a point-line graph over the 5 trials, and the data was programmed and analyzed in R-Studio. This is for the average-case data, best-case data, and worst-case data where the numbers are in random order, sorted order, and reverse order respectively. It is important to note that the time for radix, merge, and counting sort should not significantly change between average, worst, and best case, due to the algorithms’ complexity not changing when different cases are presented because of the manipulation of the algorithms not being faster or slower because of comparisons. I cut out significantly time-consuming algorithms of insertion, bubble, and selection sort because the time taken for these comparisons was significantly higher than all of the others, and skewed data. However, in the case of the best-case data, I kept insertion in and removed heapsort, because insertion was incredibly fast in the best-case scenario, while heap was one of the three slowest algorithms for the best-case data. All of these times are taken in milliseconds.

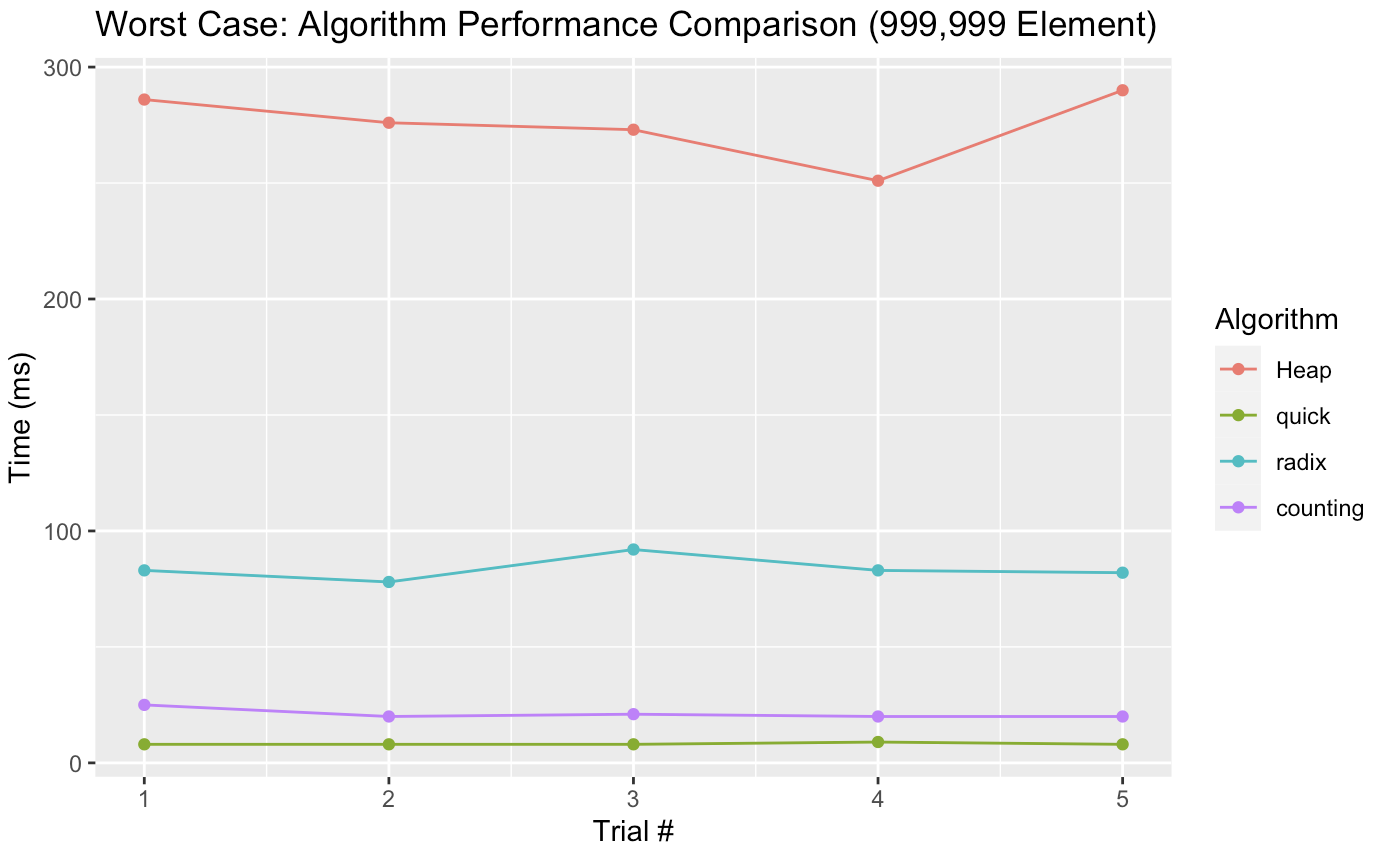




The graphs of data sets for 999k numbers on each algorithm over 5 trials are shown below. This was graphed using a point-line graph over the 5 trials, and the data was programmed and analyzed in R-Studio. This is for the average-case data, best-case data, and worst-case data where the numbers are in random order, sorted order, and reverse order respectively. It is important to note that the time for radix, merge, and counting sort should not significantly change between average, worst, and best case, due to the algorithms’ complexity not changing when different cases are presented because of the manipulation of the algorithms not being faster or slower because of comparisons. I cut out significantly time-consuming algorithms of insertion, bubble, and selection sort because the time taken for these comparisons was significantly higher than all of the others, and skewed data. However, in the case of the best-case data, I kept insertion in and removed heapsort, because insertion was incredibly fast in the best-case scenario, while heap was one of the three slowest algorithms for the best-case data. As I began to analyze the larger data sets of 999k, merge sort encountered a segmentation fault. This is due to the recursion of merge sort recursively splitting 999k elements into one element piece. Merge sort is therefore not present in the graph of 999k elements in any of the cases. All of these times are taken in milliseconds.







| Algorithm | 50k elements  Time (ms) | 500k elements  Time (ms) | 999k elements  Time (ms) |
| --- | --- | --- | --- |
| Insertion | 2519, 2492, 2556, 2527, 2514 | 247350, 249371, 248040, 248584, 247670 | 1022999, 1013824, 1022435, 1012717, 1052448 |
| Selection | 2596, 2642, 2571, 2617, 2571 | 257645, 256258, 257880, 255090, 256887 | 1071455, 1092951, 1086339, 1087624, 1081141 |
| Bubble | 5913, 5922, 6009, 5926, 6026 | 597788, 602089, 599388, 603636, 598508 | 2550689, 2485765, 2465673, 2434416, 2430646 |
| Merge | 7, 8, 8, 7, 7 | 85, 85, 81, 81, 80 | Segmentation Fault |
| Heap | 11, 11, 10, 11, 10 | 131, 133, 133, 132, 131 | 282, 289, 281, 290, 302 |
| Quick | 1, 1, 1, 1, 1 | 3, 3, 3, 3, 3 | 5, 5, 5, 5, 5 |
| Radix | 3, 3, 3, 3, 3 | 33, 34, 33, 34, 34 | 69, 69, 67, 68, 69 |
| Counting | 0, 0, 0, 0, 0 | 9, 8, 9, 9, 9 | 18, 18, 18, 17, 17 |

Table 1: Average Case

| Algorithm | 50k elements  Time (ms) | 500k elements  Time (ms) | 999k elements  Time (ms) |
| --- | --- | --- | --- |
| Insertion | 0, 0, 0, 0, 0 | 0, 0, 0, 0, 0 | 1, 1, 1, 1, 1 |
| Selection | 2618, 2578, 2627, 2593, 2589 | 255929, 263074, 264082, 271636, 274226 | 386131, 395339, 389042, 396824, 386659 |
| Bubble | 2041, 2066, 2094, 2034, 2036 | 209858, 206600, 210786, 206127, 210409 | 294561, 287274, 294836, 288063, 292297 |
| Merge | 4, 4, 4, 4, 4 | 52, 56, 52, 58, 52 | Segmentation Fault |
| Heap | 8, 8, 8, 8, 10 | 96, 98, 101, 95, 94 | 110, 113, 111, 109, 109 |
| Quick | 61, 56, 57, 57, 58 | 69, 64, 61, 63, 61 | 58, 59, 58, 60, 58 |
| Radix | 3, 3, 3, 3, 3 | 35, 34, 34, 38, 34 | 41, 40, 42, 41, 40 |
| Counting | 0, 0, 0, 0, 0 | 9, 10, 9, 8, 9 | 10, 11, 10, 10, 12 |

Table 2: Best Case

| Algorithm | 50k elements  Time (ms) | 500k elements  Time (ms) | 999k elements  Time (ms) |
| --- | --- | --- | --- |
| Insertion | 5518, 5408, 5404, 5439, 5221 | 516443, 521338, 523357, 523556, 517298 | 2026715, 2025065, 2021892, 2029297, 2023097 |
| Selection | 3709, 3540, 3514, 3537, 3620 | 275699, 277699, 277900, 276408, 274553 | 1061027, 1067244, 1062933, 1058746, 1063299 |
| Bubble | 5091, 5058, 5030, 5208, 5079 | 483833, 484596, 485941, 485076, 48531 | 1916732, 1923062, 1990701, 2034879, 2095570 |
| Merge | 4, 4, 5, 4, 5 | 55, 50, 52, 51, 50 | Segmentation Fault |
| Heap | 12, 11, 10, 8, 8 | 100, 105, 97, 96, 99 | 286, 276, 273, 251, 290 |
| Quick | 72, 76, 73, 76, 68 | 10, 10, 9, 10, 10 | 8, 8, 8, 9, 8 |
| Radix | 3, 3, 5, 4,3 | 33, 33, 35, 35, 35 | 83, 78, 92, 83, 82 |
| Counting | 1, 1, 1, 1, 0 | 9, 9, 12, 8, 8 | 25, 20, 21, 20, 20 |

Table 3: Worst Case

**ANALYSIS**

**Best Case:**

In the best case, Insertion sort performed as the best sorting algorithm in comparison to the others. This is expected, and the reasoning is that insertion sort in the best case, which is when the array is already sorted, has a time complexity of O(n). This time complexity results because of the methodology of insertion sort. When the input array is already sorted, there will be no need to perform any swaps since every element is already in the correct position. The algorithm simply iterates over each element of the array once to ensure that it is greater than or equal to the element before it. Therefore, it will not traverse through the array a second time to find the correct place, because it is already less than the number before it and greater than the number after it. This O(n) complexity is comparable to that of counting sort, which also is a close competitor for the fastest algorithm in the best case. However, the reason that insertion sort performs better in theory and practice, is because insertion sort relies only on comparisons while counting sort will initialize three arrays: one, which is the argument that is passed in; another that is the cumulative sum, which is the size of the range of numbers, and a third array that is the same size as the argument that is passed in. Especially on larger sets of numbers, such as the 999k element list, this means that counting sort will create another array of 999k elements and traverse it, which takes more time than the comparisons. This means that counting sort technically has O(N+R) complexity. Overall, the results were surprising because I discounted insertion sort in the best case, but after analysis, the data lines up with the theoretical time complexity of both of these algorithms.

**Worst Case:**

In the worst case, the following sorting algorithms were the best in the 50k elements, 500k, and 999k elements respectively: Counting, Counting, and QuickSort.

At 50k elements and 500k elements, counting sort was the fastest. This was expected because the time complexity of counting sort is O(n) which is asymptotically faster than the other sorting algorithm, especially comparison algorithms. Technically the time complexity of counting sort is O(n+r) where r is the range of the inputs. However, the time complexity will just be O(n) if the range is smaller than the input size. At the beginning of the program, I set the range of the input from 0 to 10,000 which is smaller than the input size. During the 500k elements, counting sort was slightly faster than quicksort. However, there was a trial where quicksort was faster than counting sort marginally. This may be due to luck and the computer may just have processed the algorithm faster. However statistically, counting sort was faster for f my other trials, thus counting sort was faster at 500k elements as well.

At 999k elements, quicksort was faster than counting sort. This is because, at higher numbers, counting sort has a significantly higher space complexity. The methodology of counting sort is that it takes in one array as an argument but outputs three arrays of high memory as the output and transverses through them. This takes time as well as memory. Thus, quicksort was faster because the reverse is not significant since the quicksort algorithm depends on the partition. This is how quicksort was faster than counting sort at a high input size of 999k.

**Average Case:**

In the average case, the following sorting algorithms were the best for sizes of 50k, 500k, and 999k elements respectively: Counting sort, Counting sort, and Quicksort. At 50k elements and 500k elements, counting sort was the fastest. This was expected because the time complexity of counting sort is O(n) which is asymptotically faster than the other sorting algorithm, especially comparison algorithms. Being in the worst case or average case does not matter for counting sort, because it is not a comparison sort, and therefore does not change. Technically the time complexity of counting sort is O(n+r) where r is the range of the inputs. However, the time complexity will just be O(n) if the range is smaller than the input size. At the beginning of the program, I set the range of the input from 0 to 10,000 which is smaller than the input size. During the 500k elements, counting sort was slightly faster than quicksort. However, there was a trial where quicksort was faster than counting sort marginally. This may be due to luck and the computer may just have processed the algorithm faster. However statistically, counting sort was faster for f my other trials, thus counting sort was faster at 500k elements as well.

At 999k elements, quicksort was faster than counting sort. This is because, at higher numbers, counting sort has a significantly higher space complexity. The methodology of counting sort is that it takes in one array as an argument but outputs three arrays of high memory as the output and transverses through them. This takes time as well as memory. Thus, quicksort was faster because the reverse is not significant since the quicksort algorithm depends on the partition. This is how quicksort was faster than counting sort at a high input size of 999k.

**Experimental VS Theoretical:**

**Insertion Sort**

Insertion Sort, the best case time complexity is O(n) and the average and worst case time complexity is O(n2). In terms of my 999k elements trial, the experimental results seem to correspond and agree with the theoretical expectations. In the best case, insertion sort was able to complete transverse through all the elements in 1 ms for all five trials. However, in the average case, insertion sort sorted the elements in 1022999 ms, 1013824 ms, 1022435 ms, 1012717 ms, and 1052448 ms. In the worst case, insertion sort sorted the elements in 2026715 ms, 2025065 ms, 2021892, 2029297, and 2023097 ms. The average and worst cases are significantly higher than the best case, which agrees with the theoretical expectations of trials.

**Selection Sort**

Selection Sort, the best case, average case, and worst case time complexity is O(n2). In terms of my 999k elements trial, the experimental results seem to correspond and agree with the theoretical expectations. In the best case, selection sort sorted the elements in 386131 ms, 395339 ms, 389042 ms, 396824 ms, and 386659 ms. However, in the average case, selection sort sorted the elements in 1071455 ms, 1092951 ms, 1086339 ms, 1087624 ms, and 1081141 ms. In the worst case, selection sort sorted the elements in 1061027 ms, 1067244 ms, 1062933 ms, 1058746 ms, and 1063299 ms, The best, average and the worst case are significantly high with an input of 999k elements. Although the best case seems to be 3 times as longer than the average and worst case, they all correspond to the same time complexity since the constants are dropped the leading term since it still has roughly the same rate of growth.

**Bubble Sort**

For Bubble Sort, the best case time complexity is O(n) and the average and worst case time complexity is O(n2). In terms of my 999k elements trial, the experimental results correspond and agree with the theoretical expectations for the average and worst case, but disagree with the best case. In the best case, bubble sort sorted all the elements in 294561 ms, 287274 ms, 294836 ms, 288063 ms, and 292297 ms. In the average case, bubble sort sorted the elements in 2550689, 2485765 ms, 2465673 ms, 2434416 ms, and 2430646 ms. In the worst case, bubble sort sorted the elements in 1916732 ms, 1923062 ms, 1990701 ms, 2034879 ms, and 2095570 ms. For the best, average, and worst case, they seem to be all similar where the best case should be significantly faster than the average and worst case. This is because even in the best case where the array is already sorted. Bubble sort still needs to iterate over all the elements in the array to confirm that it is already sorted. Thus, the array will always require the maximum number of iterations to confirm it, regardless of the order of the elements.

**Merge Sort**

Merge Sort, the best case, average case, and worst case time complexity is O(nlgn). For this sort, the 50k elements trial was used. The experimental results correspond and agree with the theoretical expectations This is because at higher input size a segmentation fault occurred. In the best case, merge sort sorted the elements in 4 ms for all of the trials. In the average case, merge sort sorted the elements in 7 ms, 8 ms, 8 ms, 7 ms, 7 ms. In the worst case, merge sort sorted the elements in 4 ms, 4 ms, 5 ms, 4 ms, 5 ms. The best, average, and the worst case are all similar. This is because regardless of the order of the array, merge sort will split the array and then merge them back together so they should all have similar run times. Thus, the experimental results correspond and agree with the theoretical expectation.

**Heap Sort**

Heap Sort, the best case, average case, and worst case time complexity is O(nlgn). In terms of my 999k elements trial, the experimental results seem to correspond and agree with the theoretical expectations. In the best case, heap sort sorted the elements in 110 ms, 113 ms, 111, 109 ms, and 109 ms. However, in the average case, heap sort sorted the elements in 282 ms, 289 ms, 281 ms, 290 ms, and 302 ms. In the worst case, heap sort sorted the elements in 286 ms, 276 ms, 273 ms, 251 ms, and 290 ms, The best, average, and the worst case are similar. Although the average and worst case seems to be 2 times as long as the best case, they all correspond to the same time complexity since the constants are dropped, the leading term since it still has roughly the same rate of growth. This is because regardless of the order the heaps have to be heapify and the heap has to be rebuilt. Due to this all of the runtime should be similar. This corresponds and agrees with the theoretical expectations.

**Quick Sort:**

For Quick Sort, the time complexity of the quicksort algorithm is O(nlgn) in the average case, where n is the number of elements to be sorted. However, in the worst-case scenario, when the pivot chosen is the smallest or largest element in the array, the time complexity becomes O(n2). The average-case time complexity of quicksort is achieved when the pivot element is chosen randomly and is usually the fastest among the sorting algorithms. The algorithm works by partitioning the array into two subarrays based on a pivot element. All the elements less than or equal to the pivot element are placed in one subarray and all the elements greater than the pivot element are placed in another subarray. This process is repeated recursively for each subarray until the entire array is sorted. Therefore, it should not significantly change between best and worst case arrays, but rather partitions. For these trials, the partitions became relatively worse in my best and worst cases, which means that the running times in those cases should be larger. The orientation of numbers (sorted vs. unsorted) should not matter. In terms of my 999k element trial, the experimental results correspond and agree with the theoretical expectations for the best and worst case and agree that the average case should be the worst because of the partition choice. In the best case, quicksort sorted all the elements in 5 ms, 5 ms, 5 ms, 5 ms, 5 ms. In the average case, quicksort sorted the elements in 69 ms, 69 ms, 67 ms, 68 ms, and 69 ms. In the worst case, quicksort sorted the elements in 8 ms, 8 ms, 8 ms, 9 ms, and 8 ms. These results are significantly different and correspond with the idea that the order of numbers in an array does not increase the time of quicksort.

**Radix Sort:**

For Radix Sort, O(d \* (n + k)) is the time complexity, where "n" is the number of elements to be sorted, "k" is the maximum value of any element, and "d" is the number of digits in the maximum value. and the average and worst case time complexity is the same. Radix sort does not take comparisons, depending on which algorithm you choose to radix. The algorithm works by performing a stable sort on each digit, and the most common implementation of this is with counting sort, because of its O(N+R) complexity. Therefore, it should not significantly change between the best and worst case. In terms of my 999k element trial, the experimental results correspond and agree with the theoretical expectations for the average and worst case and agree that the best case should be the same. In the best case, radix sort sorted all the elements in 41 ms, 40 ms, 42 ms, 41 ms, and 40 ms. In the average case, radix sort sorted the elements in 69 ms, 69 ms, 67 ms, 68 ms, and 69 ms. In the worst case, radix sorted the elements in 83ms, 78ms, 92 ms, 83 ms, and 82 ms. These results are relatively similar and correspond with the idea that the order of numbers in an array does not increase the time of radix sort.

**Counting Sort:**

For Counting Sort, the best case time complexity is O(N+R), where N is the input size, and R is the range of numbers. and the average and worst case time complexity is the same. Counting sort does not make comparisons, which is why it does not change in the worst vs. best case. In terms of my 999k element trial, the experimental results correspond and agree with the theoretical expectations for the average and worst case and agree that the best case should be the same. In the best case, counting sort sorted all the elements in 10 ms, 11 ms, 10 ms, 10 ms, and 12 ms. In the average case, counting sort sorted the elements in 18 ms, 18 ms, 18 ms, 17 ms, and 17 ms. In the worst case, radix sort sorted the elements in 9 ms, 9 ms, 12 ms, 8 ms, and 8 ms. These results are relatively similar and correspond with the idea that the order of numbers in an array does not increase the time of counting sort. For the complexity of O(N+R), it is important to note that if R is less than N, or the range is less than the input size, then the complexity becomes O(n), which is what happened in this case, since all of my numbers are between 1 and 9999.

**Comparisons and their relations to execution time:**

The number of comparisons is a good predictor of the execution time for comparison sorts, and comparisons are a reasonable choice of basic operation for analyzing these algorithms. In comparison sorts, elements are sorted by comparing pairs of elements and swapping them if they are out of order. Therefore, the number of comparisons made during the sort is a good indicator of the amount of work the algorithm performs. The execution time of a comparison sort is directly proportional to the number of comparisons, making it a useful metric for evaluating the efficiency of the algorithm. However, the number of comparisons is not the only factor that affects the execution time of comparison sorts. The actual implementation of the algorithm, the size of the data set, and the distribution of the data all play a role in contributing to the running time. Furthermore, comparisons may not be the best choice of basic operation for analyzing some specific sorting algorithms. For example, radix sort, which is a non-comparison sort, sorts elements by grouping them by digit values rather than comparing them. In such cases, a different metric may be more appropriate for analyzing the efficiency of the algorithm. While the number of comparisons is a good predictor of the execution time for comparison sorts, it is not the only factor to consider, and it may not be the best choice of basic operation for analyzing every sorting algorithm.

***Part 2: Problem-Solving and Analysis***

**BRUTE-FORCE APPROACH**

The problem is to design and implement an efficient algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. The algorithm shown is a brute force method that compares each element in pairs and goes through the entire list to see if the pair adds up to the desired value.

brute\_sum\_pair(S, n, x) **Times**

1. for i**←**1 to n-1 n

2. do for j←i+1 to n

3. do if S[i] + S[j] = x

4. then return TRUE 1

5. return FALSE 1

From the diagram above, if I were to solve the summations and combine the times of each line, the running time will be n(n-1)/2 or O(n2). This is because as I access an index, I search the whole list before moving on to the next one. This method is highly inefficient as there are possible calculations using integer values greater than x (impossible for the sum to be x) and the entire list is accessed multiple times. However, if memory space is a concern, then the only space required is the size of the list.

**EFFICIENT APPROACH**

Table\_sum\_pair(S, n, m, x)  **Times**

1. T**←** an empty hash table of size m; //m is less than or equal to n 1

2. for i**←**1 to n n+1

3. do if CHAINED-HASH-SEARCH(T, x - S[i]) != NIL n

4. then return TRUE 1

5. CHAINED-HASH-INSERT(T, S[i]) n

6. return FALSE 1

The given efficient approach utilizes a hash table with chaining to solve the problem. It initializes an empty hash table of size m taking constant time. In my program, I used the unordered\_set library to initialize the empty hash table. The for loop iterates i=1 to n, meaning n iterations are taken. The loop involves a search operation taking constant time. Overall, the total time complexity of this approach is 1+n+1+1+3n+1=3n+4. When simplified, the time complexity becomes O(n), which is an efficient approach to be used when finding table sum pairs. Other operations such as table search, creation, or insertion take constant time, O(1), since they are not dependent on input size.

**REFERENCES**

“Bubble sort algorithm,” *GeeksforGeeks*, 21-Apr-2023. [Online]. Available: <https://www.geeksforgeeks.org/bubble-sort/>. [Accessed: 21-Apr-2023].

“Counting sort,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/counting-sort/>. [Accessed: 22-Apr-2023].

“Heap sort,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/heap-sort/>. [Accessed: 22-Apr-2023].

“Insertion sort - data structure and algorithm tutorials,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/insertion-sort/>. [Accessed: 22-Apr-2023].

“Merge sort algorithm,” *GeeksforGeeks*, 31-Mar-2023. [Online]. Available: <https://www.geeksforgeeks.org/merge-sort/>. [Accessed: 21-Apr-2023].

“Quicksort,” *GeeksforGeeks*, 27-Apr-2023. [Online]. Available: <https://www.geeksforgeeks.org/quick-sort/>. [Accessed: 21-Apr-2023].

“Radix sort,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/radix-sort/>. [Accessed: 22-Apr-2023].

“Selection sort algorithm – data structure and algorithm tutorials,” *GeeksforGeeks*, 19-Apr-2023. [Online]. Available: <https://www.geeksforgeeks.org/selection-sort/>. [Accessed: 21-Apr-2023].