Thomas Rafeld, M12808058

Ryan Winterhalter, M12727389

Lab 11 Report

**Objectives:**

The objectives for this module were to understand the different sorting algorithms, be able to describe the strengths and weaknesses of the different sorting algorithms, and to build and compare different sorting techniques. The first objective is important to this class because, to be able to implement these algorithms, we must understand them. For this class we are trying to learn these sorting methods and then use them in a lab. To be able to do that, we must first understand how they operate. This objective is important to a career in engineering because at some point you might either need to use the sorting algorithm or explain them to another person. In either case, you would have to understand the sorting algorithms to be able to complete them. The second objective is important to this class as it will help us increase our understanding of the algorithms. By being able to describe their strengths and weaknesses, we will better understand the sorting algorithms and when to utilize them. This same reasoning would apply to a career in engineering; by being able to describe the strengths and weaknesses, when working on a project that requires sorting, we would be able to choose the best sorting implementation for that project. The final objective was to build and compare the different sorting techniques. This is important for this class because we need to be able to implement the different sorting algorithms to show that we have learned them. We are trying to understand data structures, and being able to sort these data structures is an important part of that. For a career in Engineering, this would be important if we had to use some sort of sorting for a project. Being able to build the different sorting algorithms would be extremely important for that project. Comparing them would be important to help us select the best sorting algorithm to use for what data is being sorted.

**Task 2:**

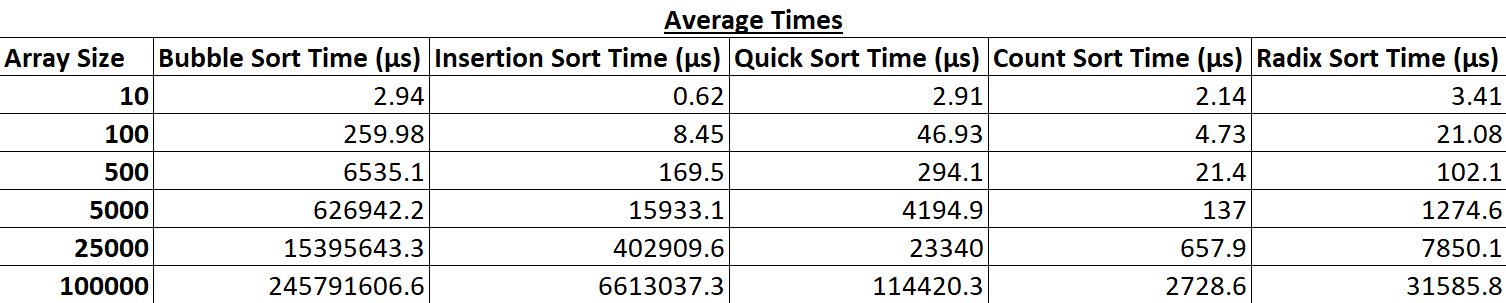
****

Table 1: Average performance times for the different sorting algorithms on arrays of different sizes.

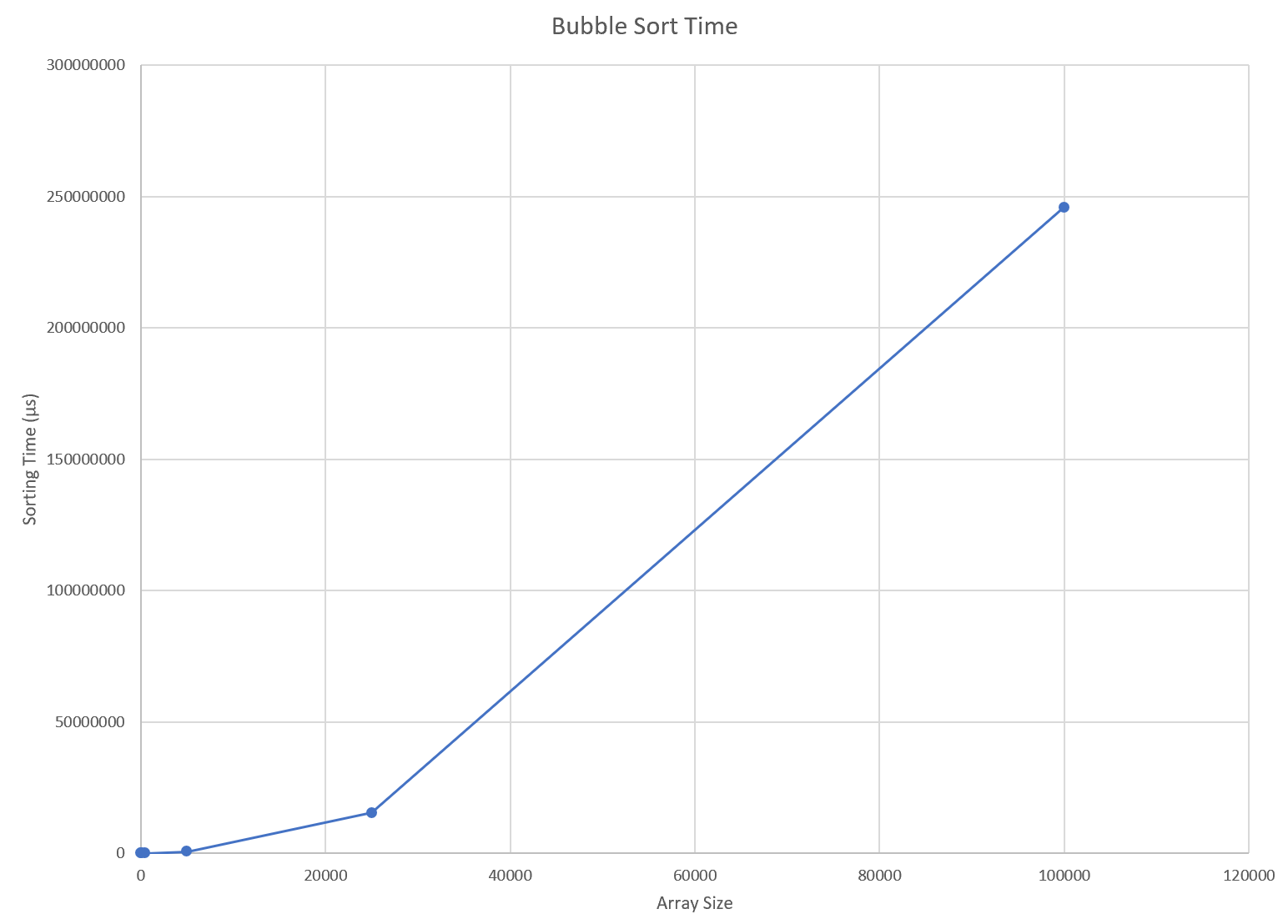


Figure 1: Graph of the bubble sort times for arrays of various sizes.

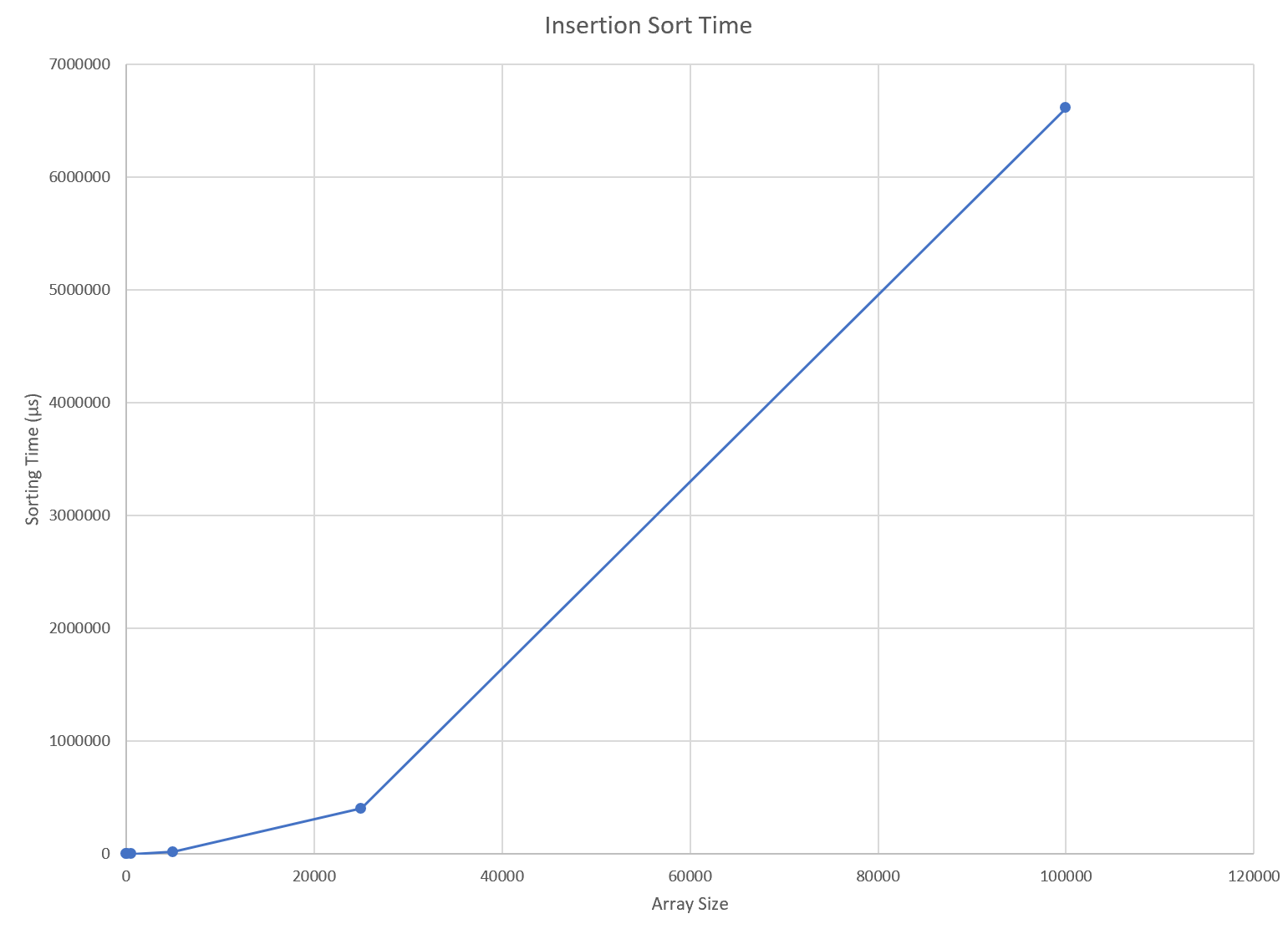


Figure 2: Graph of the insertion sort times for arrays of various sizes.

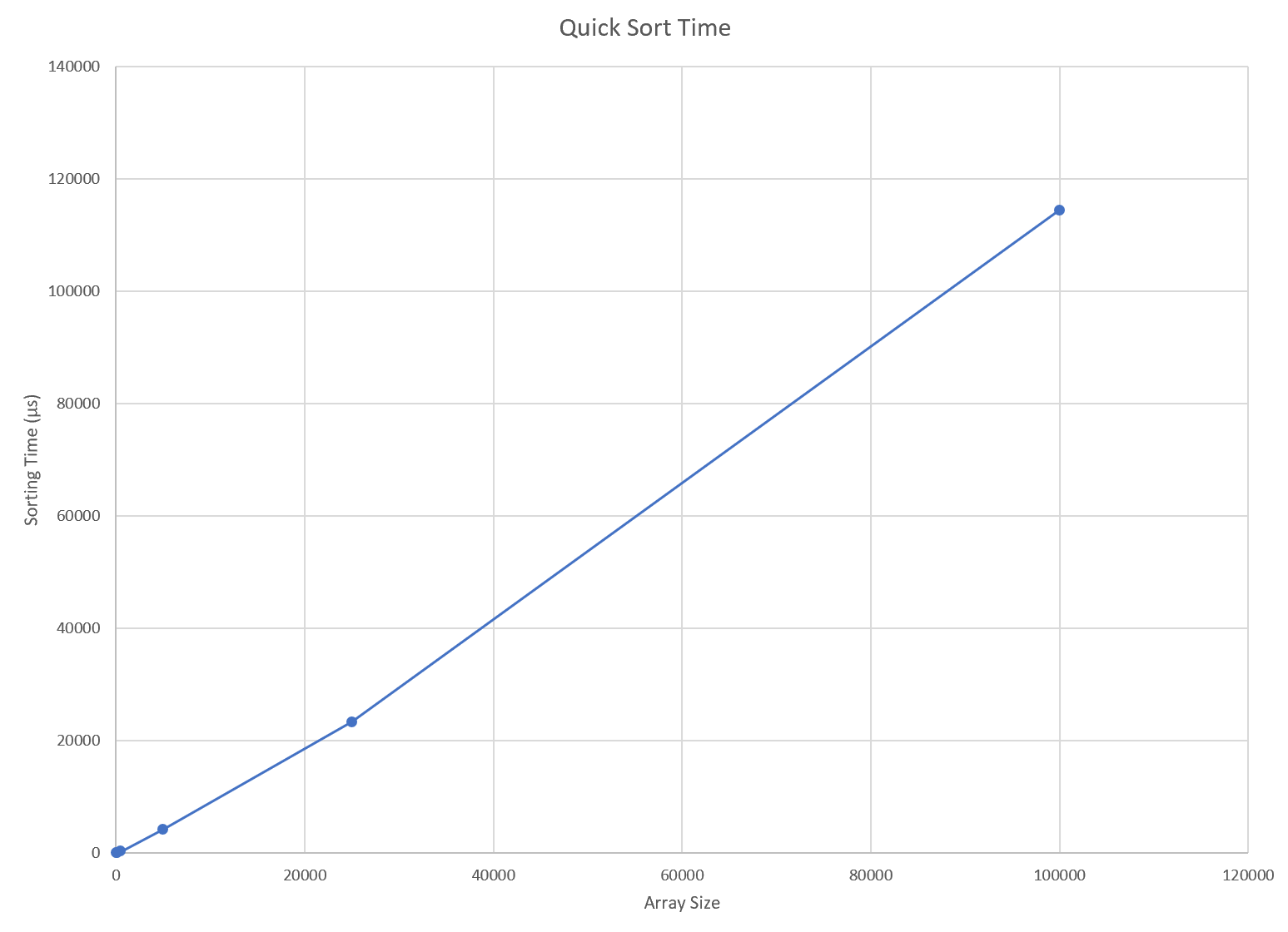


Figure 3: Graph of the quick sort times for arrays of various sizes.

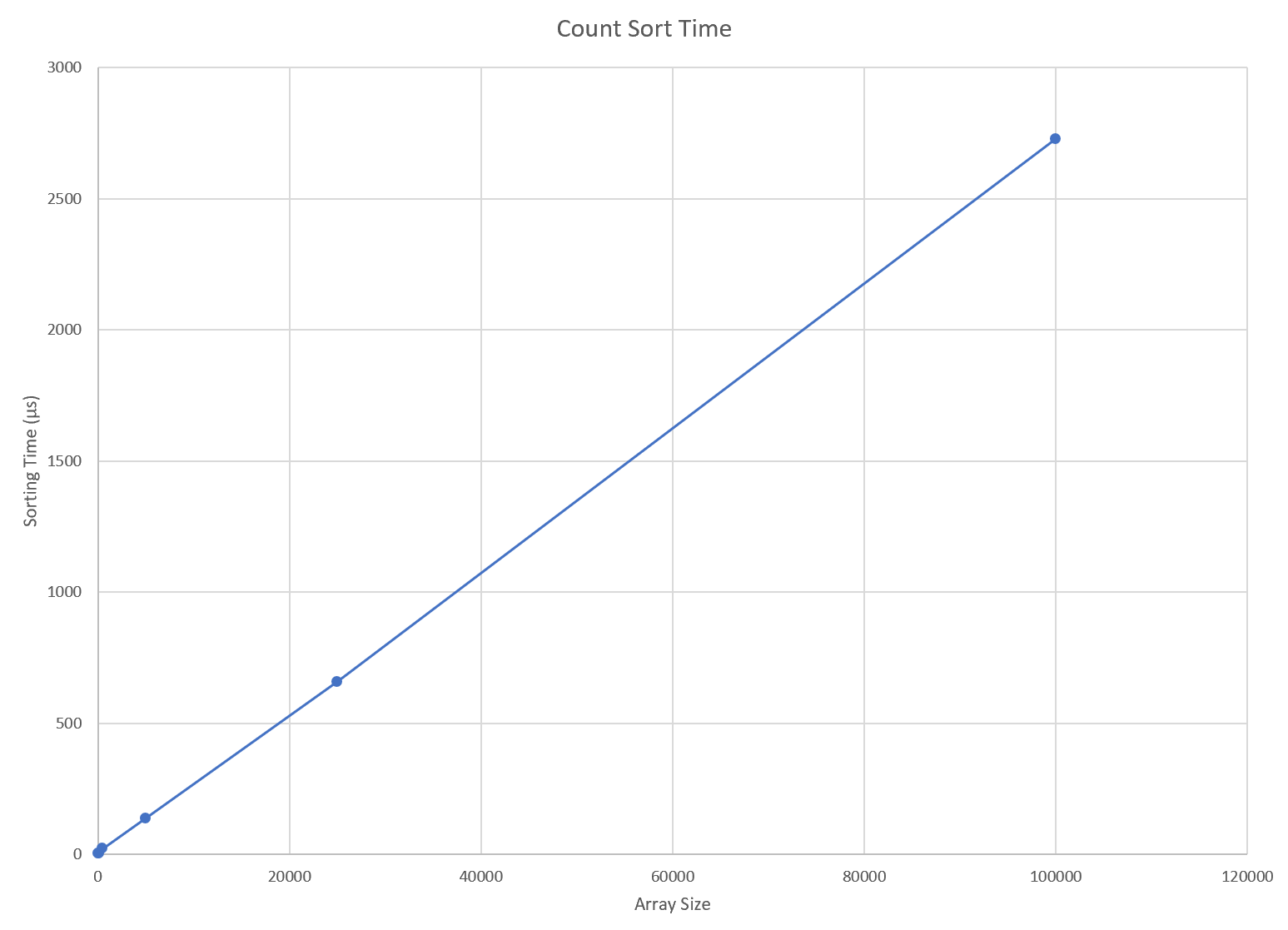


Figure 4: Graph of the count sort times for arrays of various sizes.

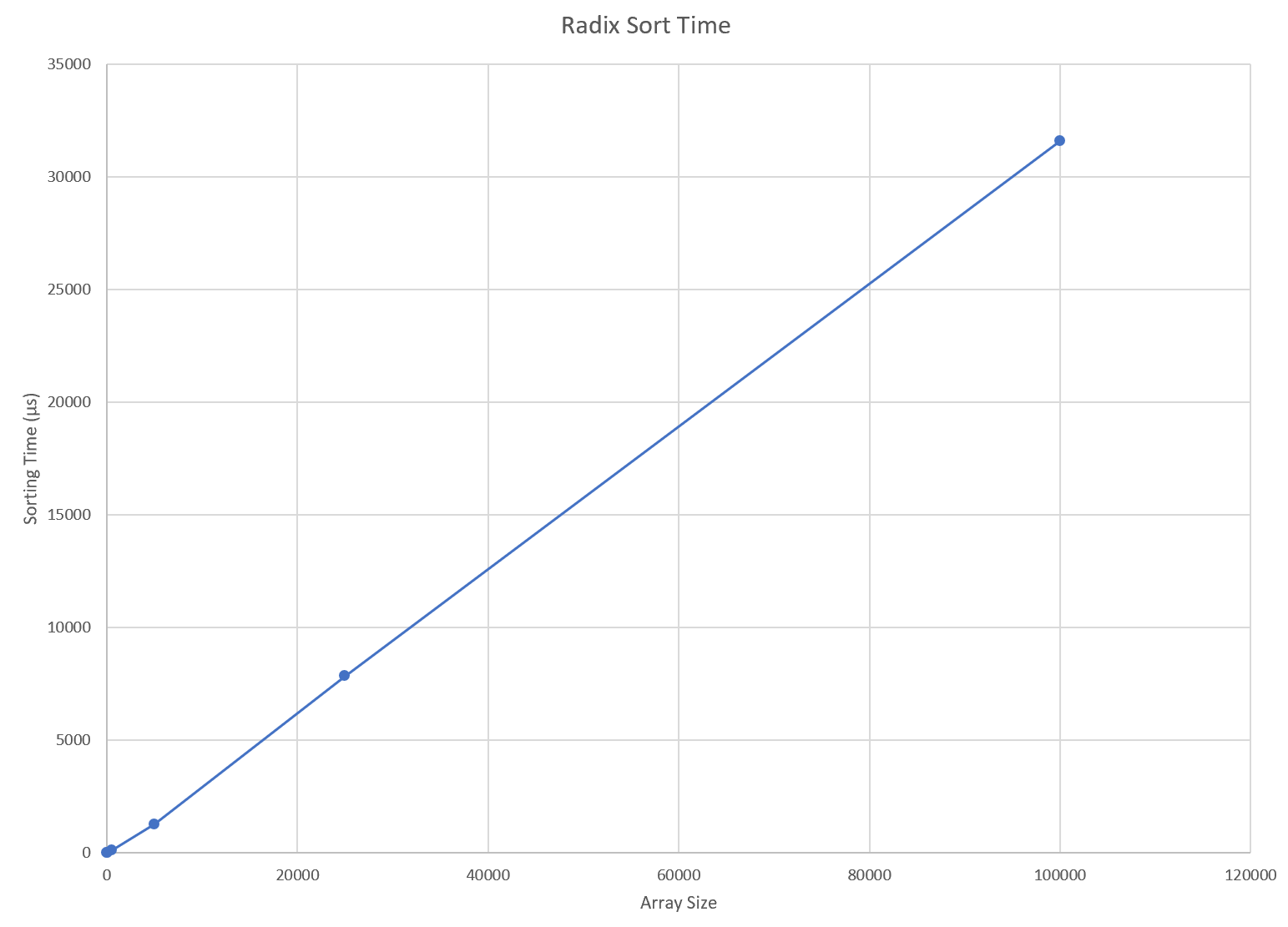


Figure 5: Graph of the radix sort times for arrays of various sizes.

Our performance data for all of the different sorting algorithms matches with what we expected. As bubble sort and insertion sort both have average performances of O(n2), we expected that their performance times would increase exponentially as the sizes of our arrays increased. For bubble sort, we expected our sorting times from 10 to 100 to be about 100 times as large, from 100 to 500 to be about 25 times as large, from 500 to 5,000 to be about 100 times as large, from 5,000 to 25,000 to be about 25 times as large, and from 25,000 to 100,000 to be about 16 times as large. These expectations were based on the relative increase in size from one array to the next and the O(n2) average performance for bubble sort. For insertion sort, we had similar expectations. We expected our sorting times from 10 to 100 to be about 100 times as large, from 100 to 500 to be about 25 times as large, from 500 to 5,000 to be about 100 times as large, from 5,000 to 25,000 to be about 25 times as large, and from 25,000 to 100,000 to be about 16 times as large. Like bubble sort, these expectations were based on the relative increase in size from one array to the next and the O(n2) average performance for insertion sort. Our expectations match with our data, as both graphs for bubble sort and insertion sort show non-linear growth as the size of the array increases. As quick sort has an average performance of O(nlogn), we expected that its performance time would increase non-linearly as the sizes of our arrays increased, but would not increase at a rate as high as that for bubble and insertion sort. We expected our sorting times from 10 to 100 to be about 20 times as large, from 100 to 500 to be about 8.5 times as large, from 500 to 5,000 to be about 20 times as large, from 5,000 to 25,000 to be about 8.5 times as large, and from 25,000 to 100,000 to be about 8 times as large. These expectations were based on the relative increase in size from one array to the next and the O(nlogn) average performance for quick sort. Looking at our graph for our quick sort times, we can see that there is non-linear growth in the sorting time as the size of our array increases, but the growth in the sorting time is not as large as that for either bubble sort or quick sort. As count sort and radix sort both have average performances of O(n), we expected that their performance times would increase linearly as the sizes of our arrays increased. For count sort, we expected our sorting times from 10 to 100 to be about 10 times as large, from 100 to 500 to be about 5 times as large, from 500 to 5,000 to be about 10 times as large, from 5,000 to 25,000 to be about 5 times as large, and from 25,000 to 100,000 to be about 4 times as large. These expectations were based on the relative increase in size from one array to the next and the O(n) average performance for count sort. For radix sort, we had similar expectations. We expected our sorting times from 10 to 100 to be about 10 times as large, from 100 to 500 to be about 5 times as large, from 500 to 5,000 to be about 10 times as large, from 5,000 to 25,000 to be about 5 times as large, and from 25,000 to 100,000 to be about 4 times as large. Like count sort, these expectations were based on the relative increase in size from one array to the next and the O(n) average performance for radix sort. Our expectations match with our data, as both graphs for count sort and radix sort show approximately linear growth as the sizes of the arrays increase.

**Group Contributions:**

The lab was worked on together by both Ryan and Thomas while on a call together in Microsoft Teams. For tasks 1 and 2, we worked together to write all of the sorting algorithms in our header file, using the GeeksForGeeks sorting implementations provided in the module for help, and develop the main ‘.cpp’ file that would populate the arrays of different sizes with random values and record the time for how long it took to sort each array for each different sorting implementation. We decided to use Ryan’s header file (task 1) and Thomas’s main file (task 2) for these tasks.

For task 3, we modified our code from our earlier linked lists lab assignment (Lab 4) to allow our linked list (populated by the “students” data type) to be sorted using three different sorting algorithms. As each sorting algorithm needed to sort the data (in both an ascending and descending order) according to one of the characteristics of the “student” data type, we decided to sort the students’ first names using bubble sort, the students’ M-numbers using insertion sort, and the students’ last names using merge sort. Thomas wrote the code for the bubble sort and insertion sort algorithms and Ryan wrote the code for the merge sort algorithm, using help from video tutorials on YouTube (links are in the task 3 header comments). For the final grade, each member of the group should receive 100 percent of the grade as we feel that we both evenly contributed to the lab and worked together for almost the whole time it was being worked on.