# California State University, Fresno

# DEPARTMENT OF COMPUTER SCIENCE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class: | **Algorithms & Data Structures** | | | Semester: | **Fall 2021** |
|  | | | | | |
| Points |  | Document author: | **Marlon Branham** | | |
|  | Author’s email: | **zekebranham@mail.fresnostate.edu** | | |
| Laboratory number: | **Laboratory 2** | | |
|  | | | | | |

**1. Statement of Objectives**

In this lab, we were to explore two different types of sorting algorithms. One being Selection Sort, a non-recursive function that sorts elements of an array using iterative methods. The second being Merge Sort, a recursive function that sorts elements by breaking the problem down into smaller segments recursively, then merging them together at the end. With these two algorithms, we were to determine which of the two were more efficient, that being faster in this case, and report how we know. The significance of this lab is to gain knowledge as to which algorithm is faster when dealing with large numbers of data. The report will be discussing the experimental procedures of the lab, my approach to the lab, how I measured the efficiency, and any problems I encountered.

**2. Experimental Procedure**

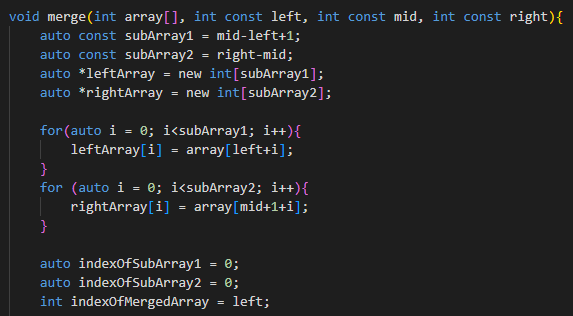
The Experimental Procedure portion of this report will be broken down into 3 sections, the first being the Selection Sort Algorithm, the second being the Merge Sort Algorithm, and the third being the calculation of the results.

A computer screen shot of a program code

Description automatically generatedThe first part of the procedure was to create a Selection Sort algorithm to sort a previously fully sorted array, half-sorted array, and a reverse-order array. Selection sort is relatively straightforward in its function, so I created a 7 line algorithm to complete this task by using a function I created called selectionSort() that takes in an array and an integer representing the size of the array.

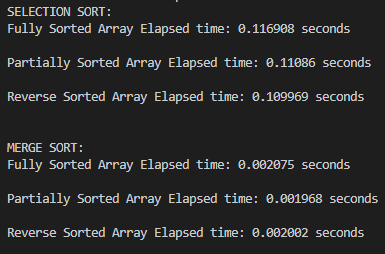
For the second part of the procedure, it was the same as goals as the first part but instead of iterative methods, use a recursive Merge Sort algorithm. This step was a bit more difficult in the fact that it was my first time creating a full Merge Sort algorithm. Like the first, I used functions, I created two functions for my Merge Sort algorithm, one being *merge*, that takes in the array, the leftmost variable of the array, the center of the array, and the rightmost variable of the array. The *merge* function merged the subarrays created by the latter part of the Merge Sort algorithm and does the actual sorting of the subarrays. The second part of the Merge Sort algorithm is a function called *mergeSort* that recursively calls itself to complete the algorithm by calling the *merge* function to create one A computer screen shot of a program

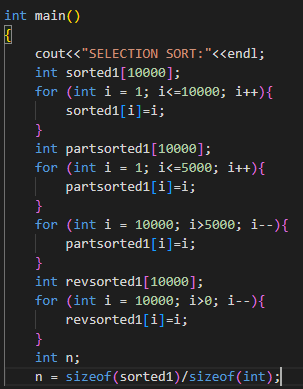
Description automatically generatedfully sorted array.



For the final part of the procedure, we were to time the speeds of the 3 arrays (sorted, half-sorted, and, reverse-order) being sorted using the two algorithms and comparing the two. Firstly, I created 3 arrays of 10,000 values each in *main* and timed the sorting algorithms using the system clock. More pictures of this will be shown in the Results section.

**3. Analysis**

A computer screen shot of a program

Description automatically generatedThe results for this experiment were personally surprising to me, contrary to my initial thinking, Merge Sort was the clear winner. To report the results, as discussed previously in the Experimental Procedures portion of this report, I created 3 arrays of 10,000 elements each for both sorting algorithms. One being fully sorted already, the second being half-sorted, and the third being reverse-ordered, essentially trying to illustrate best-case, average-case, and worst-case scenarios. I timed each algorithm using the system clock. My results show that the Merge Sort algorithm was roughly 65x faster on average than that of Selection Sort when looking at the worst-case times. With 10,000 elements that difference is under a quarter of a second, but when in terms of 1,000,000 items, the Insertion Sort would be immensely slower than that of Merge Sort.

**4. Encountered Problems**

Some issues I faced while doing this lab did not necessarily have anything to do with the algorithms at all. My problems came to light when I was trying to create arrays for the two algorithms to sort. This was difficult because finding an array size that would be large enough to truly compare speeds of each algorithm, and small enough for not only my computer to handle and for Selection Sort to run in a reasonable amount of time. Initially I tried 100 elements, but this was too small to time, then I tried 1,000 elements, and this again was too small to get a great calculation. I jumped to 100,000 elements hoping it would be the perfect number to get data out of, but it was too large for Selection Sort to complete in a decent time and was a bit large for my computer to handle. I eventually settled on 10,000 elements to get the best data possible without overstressing my computer.

**5. Conclusions**

In conclusion, the results show Merge Sort to be the more effective algorithm when handling large amounts of data. This is due to Merge Sort’s divide-and-conquer functioning. Initially, I thought Merge Sort would take longer due to how many arrays it would be handling, the recursiveness, and the overall length of the algorithm. We look at the 7 lines it took for Selection Sort, and the 50+ lines of code it took for Merge Sort, it would seem that Selection Sort is faster since the compiler is doing seemingly “less.” But the reality is that the extent of the Merge Sort algorithm’s code is needed for better functionality and handling.

**6. References**

<https://www.geeksforgeeks.org/cpp-program-for-merge-sort/>

<https://www.programiz.com/dsa/merge-sort>