**Project 3 – Scaling Behaviors of Sorting Algorithms**

1. *Requirements*:

This project looked at different sorting four different sorting algorithms and their efficiencies when the size of the array was increased. The sorting algorithms analyzed were the insertion sort, heap sort, merge sort and quick sort. The program allowed an input which allowed the user to select which sort method they would like to analyze during execution. The output of the program would be the results of the average of five iterations of different array sizes of the time it took to sort the array, the number of comparisons the sorting algorithm used as well as the number of moves the sorting algorithm utilized to sort the array into ascending order.

1. *Method*:

The project utilized a Driver class which handled the setup and execution of generating the randomly filled arrays and iterating to obtain an average for the desired sorting parameters. The driver class used a switch case to determine which sorting algorithm the user wanted to utilize and created a sorting object from the desired sorting class. A SortTimer

1. *Implementation*:

The program utilized a Driver class which instantiated the variables and methods needed to execute the sorting algorithm desired from the user. Each sorting algorithm was written in their separate classes (InsertionSort, MergeSort, HeapSort, QuickSort) which had an interface with SortAlgorithm. The interface required a sort method with a double array as well as a SortTimer object as input with a void output. The SortTimer class was used to quantify the performance of the sorting algorithms by keeping track of elapsed time as well as a count of comparisons and moves. The various sorting algorithm classes utilized the algorithms discussed during class.

1. *Testing*:

In order to ensure that the sorting algorithms operated correctly, I made a sample array from the examples mentioned in class that went through step by step of each sorting algorithm. I monitored the steps as the sorting algorithm executed to ensure the validity of the sort method as well as verified the resulting sorted array was sorted in ascending order. A verify method was also implemented in the Driver class so that when run with the sorted array as an input, it would check the array to determine if the sorted array is really in ascending order. If it wasn’t in ascending order, it would output *false* which could be used to display that information to the user.

1. *Findings*:

From this project, I learned different ways to measure the efficiency of an algorithm as well as what can affect the efficiency. After executing the program with all four sorting algorithms, Figures 1, 2 and 3 were generated to illustrate the change in efficiency as the array size increased. After obtaining these results, using the equation resulted in the constant factor for each of the parameters, with the results being tabulated in Table 1. It was interesting to see how the differences in sorting time across the different array sizes. As the array size increased, quick sort was generally more efficient in sorting the randomized array into ascending order.

Figure 1 – Time to Sort vs. Increasing Array Size

Figure 2 – Count of Comparisons vs. Increasing Array Size

Figure 3 – Count of Moves vs. Increasing Array Size

Table 1 – Constant factor of various sorting algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| **Constant Factor** | **Time (microseconds)** | **Comparisons** | **Moves** |
| **Insertion** | 0.020367438 | 0.204869401 | 0.223387734 |
| **Heap** | 0.327907268 | 3.863672853 | 3.642415806 |
| **Merge** | 12.74311153 | 2.728304168 | 2.33191664 |
| **Quick** | 1.397477469 | 2.431977947 | 1.019340918 |