Mohamed Kharma

Professor: Ahmet C. Yuksel

Class: CSC 22000

Sorting Algorithms Report

For this assignment, I decided to write all my code into a single cpp file. The program will take user input of whatever size the user like the array to have, then it will store random numbers from range 0 to 1000 inside the elements of the array that the user created. Then the algorithms are going to be called one at time to sort that array. Once the array starts to get sorted by any algorithm, a timer will begin counting the time that the algorithm took to sort the array in nanoseconds. To keep track of time, I added the time library <chrono>. Then added about six timers (one for each algorithm) and got the results of each algorithm in nanoseconds. Lastly, the arrays will get deleted from the heap to free up the space they used.

Using nanoseconds(ns) as a time unite, the performance of each algorithm is shown below:

Size n	InsertionSort	MergeSort	HeapSort	QuickSort	Randomized QuickSort	RadixSort
10	500	4600	1500	700	19300	1800
100	10300	43100	11600	7200	12900	6300
1,000	739600	521300	209500	124500	93100	80000
10,000	59314300	3157700	1751100	1213000	1086000	751900
100,000	5626502400	40107200	23308000	21543100	21176600	8667300
1,000,000	869129975000	394819618	639389496	1635043742	1613261156	105624738

Looking at the code result, we can clearly see that some algorithms are faster than others depending on the size that the user entered. As we learned in our previous lectures, this is mainly because some algorithms may require less space, and some can vary their speed depending on inputs while some are always consistent with their speeds. For example, using InsertionSort for huge arrays would not be efficient as it takes n^2 time on average to sort any array. However, using stable algorithms such as MergeSort will be more efficient for these types of arrays because it guarantee a constant-case of nlogn time for sorting an array. Therefore, we can conclude that all sorting algorithms have their own best-case and worst-case performance depending on the user input and the size of the elements being sorted and how much is already sorted in the existing data. Those circumstances and the chosen algorithm can give a huge difference in performance.

While working on this assignment, I had to overcome multiple obstacles. First, keeping track of the time each algorithm takes to sort the elements of the array, which I had to learn how to use the time library to calculate the amount of time a code would take to fully run. Second, not losing track of the original array that was created with random values and to make sure that all the algorithms sorted that array one at a time. Therefore, having one array wasn't going to be enough and I decided to have two of them. One to store all original elements of the array that was randomly created based on the user array size called "randArray". The second one called "temprandArray" was used for the algorithms. Each time the "temprandArray" passed through an algorithm and get sorted, it will reset and copy the elements of "randArray" which had the original array values. Another obstacle was the memory usage for large arrays such as 1 million elements which was an easy fix since I took advantage of the heap to store the array.