Colton Morley

CS 302.1001

October 7, 2020

PA2 Report

For this assignment I have created multiple cpp files and header files that work together to test and compare the speed of MergeSort and BubbleSort algorithms. Using RandGen.cpp, I created three different files, containing 1,000, 10,000, and 100,000 integers respectively, all ranging from 0 to 106. Due to my limited knowledge of the rand() and srand() functions, the second file generates the same first 1,000 numbers as the first file, followed by 9,000 random numbers, and the third the same 10,000 as the second, followed by 90,000 more random numbers. I do not believe that this issue has a large impact in comparing the two sorting algorithms and the times should still be accurate. Below is the output of the collected data from the different sorting methods in tabular form, and screenshots of the actual output of my program can be found in the Log.docx file located in the same folder as this file.

Data of Time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BubbleSort (unsorted) | MergeSort (unsorted) | BubbleSort (sorted) | MergeSort (sorted) |
| 1,000 integers | .00530s | .00022s | .0000039s | .0000804s |
| 10,000 integers | .54927s | .00204s | .0000512s | .0013996s |
| 100,000 integers | 54.911s | .02250s | .0003984s | .0130377s |

Data of Comparisons

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BubbleSort (unsorted) | MergeSort (unsorted) | BubbleSort (sorted) | MergeSort (sorted) |
| 1,000 integers | 499,329 | 8722 | 999 | 5,044 |
| 10,000 integers | 49,989,114 | 120,548 | 9,999 | 69,008 |
| 100,000 integers | 4,999,935,294 | 1,536,486 | 99,999 | 853,905 |

Data of Swaps

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BubbleSort (unsorted) | MergeSort (unsorted) | BubbleSort (sorted) | MergeSort (sorted) |
| 1,000 integers | 255,437 | 19,952 | 0 | 19,952 |
| 10,000 integers | 24,802,236 | 267,232 | 0 | 267,232 |
| 100,000 integers | 2,490,399,265 | 3,337,856 | 0 | 3,337,856 |

The worst case for BubbleSort is O(n2), which shows in the tables above. As the number of integers in the unsorted list increase, the time, comparisons, and swaps are all very high, and the highest among all test cases. However, when using a sorted list, BubbleSort shifts from the worst option to the best option, as its best case is 0(n). When this list is already sorted, the BubbleSort only takes one iteration through the array to determine the list is sorted.

MergeSort is much more efficient when dealing with large and unsorted lists. The worst case for MergeSort is n\*log(n) which results in much more efficient sorting for worst case scenarios when compared to BubbleSort. However, the number of swaps did not change between sorted and unsorted lists and the number of comparisons was still relatively large when compared to others.

Using these numbers, it can be determined that the worst case for BubbleSort is much worse than the worst case for MergeSort, but the best case for BubbleSort is much better than the best case for MergeSort.

After seeing firsthand how much longer it can take when not using more efficient sorting algorithms, it is clear that using efficient algorithms can save people a lot of time in the workplace. This will make these people’s lives easier and also will make things more efficient and therefore cheaper. The less time they take to run means less man hours and electricity paid for. Efficient computing and an efficient economy go hand in hand because the more efficient that these computer algorithms can make the workplace, the more efficient businesses and things related to the economy will be.

\*I am not sure if the graders would like just a simple CMakeLists.txt file or if you would like all the resulting files from cmake so that the only command needed to compile is ‘make’. For this assignment I have done it in the latter form, but please let me know if you would prefer another way.