Jeffrey Awuku-Boateng & Faisal Hossain

CSCI 2720 - Assignment 5 April 16, 2021

# Table 1 - Experimental Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Input Type** | **# Comparisons** | **Comments about time complexity and # comparisons** |
| Selection Sort | Ordered File | 499949985 | The time complexity of an Ordered selection sort is O(n2).  The size of our constant input is 10,000 so expect an output of 8-9 digits which aligns with the Big-O time complexity of this algorithm. |
| Selection Sort | Random File | 49995004 | The time complexity of Random selection sort is O(n2). The size of our constant input is 10,000 so expect 100,000 and output of 8-9 digits which aligns with the Big-O time complexity of this algorithm. |
| Selection Sort | Reversed File | 49995002 | The time complexity of Reverse selection sort is O(n2). The size of our constant input is 10,000 so expect 100,000 and output of 8-9 digits which aligns with the Big-O time complexity of this algorithm. |
| Merge Sort | Ordered File | 69008 | The complexity for ordered data is achieving maximum efficiency with O(n) space. The time complexity of an ordered Merge Sort is NlogN. |
| Merge Sort | Random File | 120464 | The Time complexity of Random Merge Sort is NlogN. The size of our constant input is 10,000 so expect 4,000 with an  output 5-6 digits. This aligns with the Big-O Time Complexity of this algorithm. |
| Merge Sort | Reversed File | 67028 | The Time complexity of Reverse Merge Sort is NlogN. The size of our constant input is 10,000 so expect 4,000 with an  output 5-6 digits. This aligns with the Big-O Time Complexity of this algorithm. |
| Heap Sort | Ordered File | 256526 | The complexity for Heap Sort is O(nlogn) for all input type cases. |
| Heap Sort | Random File | 235480 | The complexity for Heap Sort is O(nlogn) for all input type cases. |
| Heap Sort | Reversed File | 236740 | The complexity for Heap Sort is O(nlogn) for all input type cases. |
| Quick Sort - FP | Ordered File | 50004999 | The complexity for Quick Sort will be O(n2) if we give the sorted data. |
| Quick Sort - FP | Random File | 162640 | This input type is the best-case scenario for Quick Sort complexity, when using a first pivot. |
| Quick Sort - FP | Reversed File | 49995000 | The complexity here will also be O(n2) if we give the reversed data. |
| Quick Sort - RP | Ordered File | 41252992 | The complexity here has improved when using a random pivot, in comparison to using a first pivot, but not by much. |
| Quick Sort - RP | Random File | 168047 | This input type is also the best-case scenario for Quick Sort complexity, when using a random pivot. |
| Quick Sort - RP | Reversed File | 26970696 | The time complexity of a Reverse Quick Sort Random Pivot is O(n2). The size of our constant input is 10,000 hence we expect 100,000, expect 8-9 digits. This does not align with the Big-O Time Complexity of this algorithm. This implementation outputs a smaller number of comparisons when compared to the out of the First Pivot Quicksort, but not by much. |

**Table 1 Analysis**

1. We used an extra vector to implement the Merge Sort algorithm. This vector stored split sub-arrays temporarily which we then sort regrouped back together. Hence our Merge Sort had a space complexity of O(n).

1. From the results in Table 1, we can conclude that Merge Sort is the most efficient algorithm in all cases (Best case, Average case, worst-case). Merge Sort performs at maximum efficiency when the data is either sorted or in reverse order. When the data is in random order, Merge sort has a time complexity of NlogN. Merge Sort compared to the other algorithms has the least number of comparisons. Merge Sort is best used on Linkedlists and for merging two arrays that have already been sorted. If you however do not have a lot of space to work with then QuickSort is the best algorithm to use but it has some cons like number of comparisons and takes a longer amount of time to perform.

# Table 2 - Theoretical Results

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Sample Size** | **10** | **100** | **500** | **1000** | **10000** | **20000** | **50000** | **100000** |
| Selection Sort | 48 | 4950 | 124750 | 499500 | 49995000 | 199990000 | 1249975000 | 4999950000 |
| Worst Case O(N2) | 100 | 1000 | 250000 | 1000000 | 100000000 | 400,000,000 | 2500000000 | 10000000000 |
| Merge | 23 | 543 | 3865 | 8701 | 120429 | 260826 | 718308 | 1536586 |
| Heap | 45 | 1050 | 8032 | 16922 | 246969 | 510816 | 1409630 | 3019766 |
| QuickSort - FP | 26 | 675 | 4600 | 11350 | 159100 | 325039 | 986273 | 2096294 |
| QuickSort - RP | 21 | 650 | 4500 | 10900 | 159100 | 370000 | 948750 | 2060000 |
| Avg Case O(N\*logN) | 33 | 664 | 4482 | 9965 | 132877 | 285754 | 780482 | 1660964 |

**Discussions:**

## i. Selection Sort

*Figure A: Selection Sort Graph* (x - sample size, y - # of comparisons)

In Figure A above, the blue line represents the worst-case time complexity for Selection Sort, which is O(n2). The theoretical results and the experimental results align for the Selection Sort algorithm. The experimental results are close to the Big-O worst case complexity’s Big-O which is O(n2). Since the Big-O is the worst case, we expect the complexity for Selection Sort to be equal to or less than this value. In all input cases, the number of comparisons was the same and a **O(n2)** complexity was met. **ii. Merge Sort**

*Figure B: Merge Sort Graph* (x - sample size, y - # of comparisons)

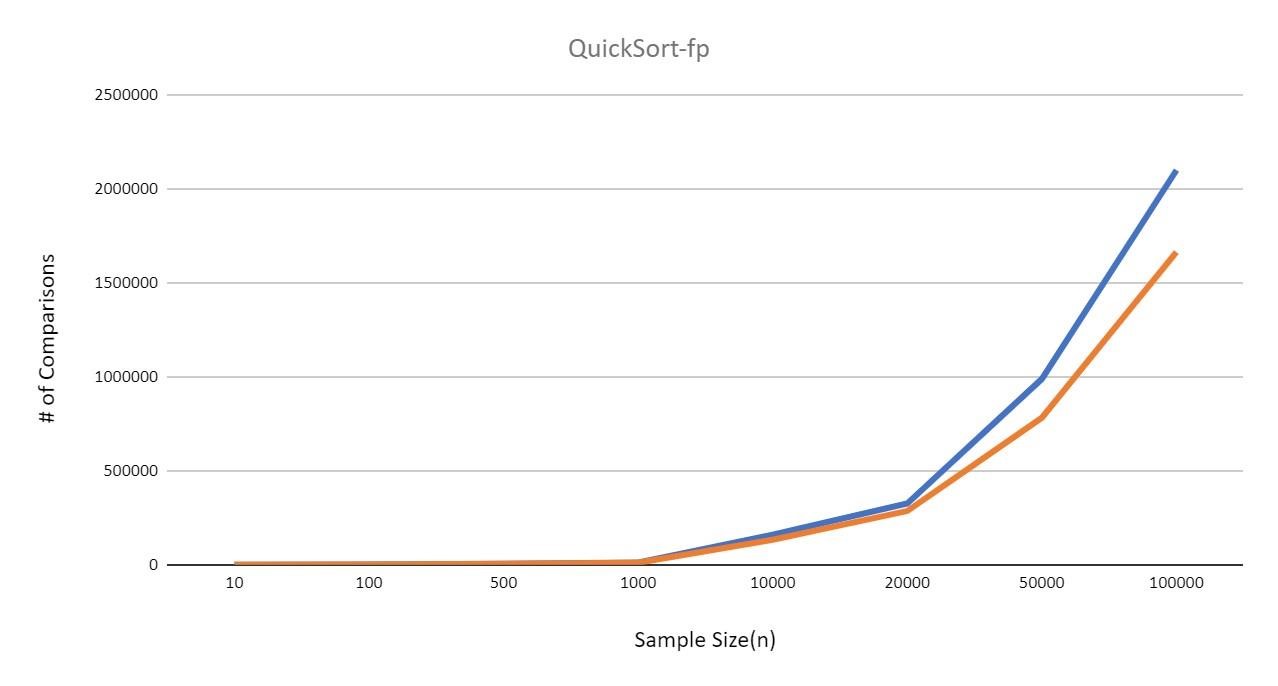
In Figure B above, the red line represents the worst-case time complexity for Merge Sort, which is O(nlogn). We can see that the blue line, our Merge Sort, runs close to this nlogn bound, following roughly the same curve. The theoretical results and the experimental results align for the Merge Sort algorithm. Merge Sort algorithm achieves maximum efficiency with both ordered data and reversed data. When given random data, this algorithm is less efficient in comparison to the other two input types. Overall, the experimental results coincide with the Big-O complexity **O(nlogn).** Being that the Big-O is the worst-case scenario, we expect the complexity for Merge Sort to be equal to or less than this value. Based on Table 1, we can see a **O(nlogn)** complexity was met.

## iii. Heap Sort

*Figure C: Heap Sort Graph* (x - sample size, y - # of comparisons)

In Figure C above, the orange line represents the average time complexity for Heap Sort, which is O(nlogn). The blue line, our Heap Sort, runs close to this nlogn bound, following roughly the same curve. We can see here that our heap sort implementation runs a bit above average to our predicted O(nlogn**)** time. This slight inconsistency may have been due to the extra overhead caused by duplicate values and a slightly different implementation than the typical algorithm. However, it does run close enough to its O(nlogn**)** time to be considered valid. When comparing the theoretical and the experimental results we can see that Heap Sort performs close to our theoretical result.

**iv. Quick Sort - FP**



*Figure D: Quick Sort - FP Graph* (x - sample size, y - # of comparisons)

In Figure D above, the orange line represents the worst-case time complexity for Quick Sort, which is O(nlogn). We can see that the blue line, our Quick Sort, runs close to this nlogn bound, following roughly the same curve. The theoretical results and the experimental results align for the Quick Sort - FP algorithm. Based on our experimental results, we can see this theoretical value holds true. Quick Sort - FP algorithm achieves maximum efficiency for random data only. When given ordered or reversed data, this algorithm is less efficient in comparison to the random input type, with a O(n2) complexity. Overall, the experimental results coincide with the average Big-O complexity O(nlogn).

## v. Quick Sort - RP

*Figure E: Quick Sort - RP Graph* (x - sample size, y - # of comparisons)

On average, the Quick Sort algorithm using a random pivot will run at O(nlogn), with its worst-case being O(n2). In Figure E above, we depict O(nlogn) with the green line. We can assume that the blue line is confined within O(n2) because the graphed n2 goes off the above graph and is not visible within the dataset. Therefore, we can conclude that since our experimental values created a line that falls within the average and O(n2), it does indeed match our hypothesis. Although they have the same complexity, the Quick Sort algorithm using a random pivot performs slightly more efficiently than the Quick Sort algorithm using the first element as a pivot.