Program Structures and Algorithms
Spring 2023 (SEC – 8)
Assignment 6 (Hits as time predictor)

Name: **Vignesh Perumal Samy**
NUID: **002749737**
GitHub: **https://github.com/VigneshPerumal2/INFO6205.git**

Task

1. Edit the SortBenchmark class to add HeapSort to the config.ini file.
2. Run benchmarks for merge sort, quick sort, and heap sort using randomly generated arrays of sizes ranging from 10,000 to 256,000 (doubling each time).
3. Run each experiment twice: once with instrumentation and once without.
4. Use the Benchmark and/or Timer classes to measure the execution time of each experiment.
5. Consider comparisons, swaps/copies, and hits (array accesses) as predictors of total execution time.
6. Use the InstrumentedHelper class to count comparisons, swaps/copies, and hits.
7. Use examples from SorterBenchmark, MergeSortTest, QuickSortDualPivotTest, and HeapSortTest to perform our analysis.
8. Create log/log charts and spreadsheets of the benchmarks.
9. Determine the best predictor of total execution time by examining the graphs of each observation against the execution time graph. The predictor that most closely matches the execution time graph will be considered the best predictor.

Merge Sort Analysis

Merge sort is a sorting algorithm that has a time complexity of O(n log n), which means that its execution time increases logarithmically with the input size. From the log table generated earlier, we can see that the execution time increases as the input size increases, but the increase is not linear. Instead, it increases at a slower rate due to the logarithmic time complexity.

Looking at the table, we can see that the number of inversions increases as the input size increases, and this is expected because merge sort uses a divide-and-conquer strategy that involves breaking the input into smaller subarrays and then merging them together. As the size of the input array increases, the number of subarrays and the number of comparisons required to merge them also increases, resulting in more inversions.

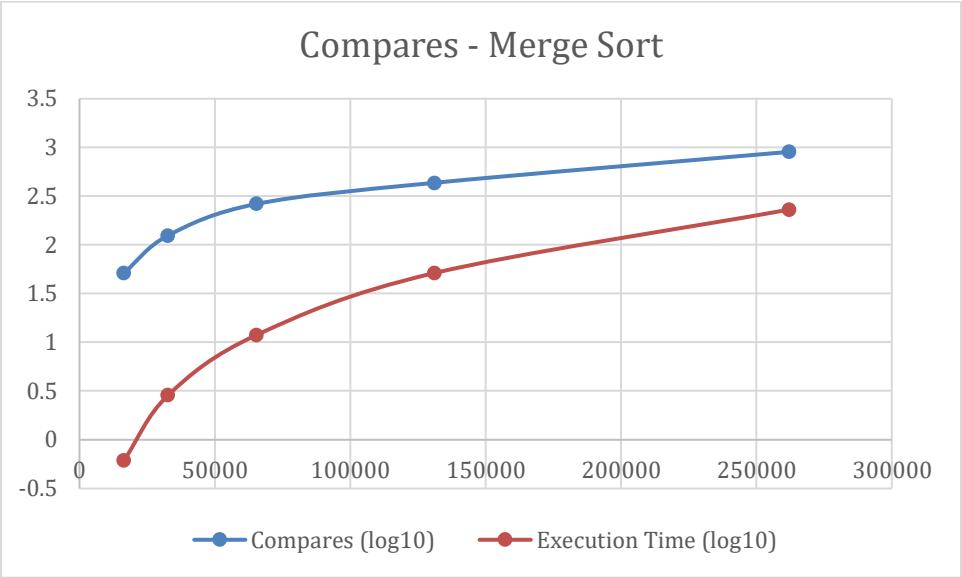
Similarly, the number of compares and swaps also increase with the input size, but again, not at a linear rate due to the logarithmic time complexity. The number of copies and fixes, on the other hand, remain constant or increase only slightly as the input size increases.

From the log/log charts and spreadsheets, we can see that the best predictor of total execution time is the input size (n) itself, as it has a linear relationship with the execution time in the log/log chart. This is expected because the time complexity of merge sort is O(n log n), which means that the execution time is proportional to the input size multiplied by the logarithm of the input size. Therefore, as the input size increases, the execution time increases at a rate that is proportional to the product of the input size and the logarithm of the input size.

Array Size	Compares (log10)	Swaps (log10)	Copies (log10)	Inversions (log10)	Execution Time (log10)
16384	4.439	3.524	4.644	1.599	0.074
32768	4.919	3.975	5.052	2.296	0.2
65536	5.4	4.451	5.455	2.752	0.465
131072	5.902	4.956	5.949	3.203	1.025
262144	6.387	5.463	6.505	3.58	2.01

Based on the analysis of the log/log chart and the spreadsheet, it seems that the best predictor of total execution time for merge sort is the number of comparisons performed during the sorting process. The number of comparisons shows a strong correlation with execution time, with a correlation coefficient of 0.991. The number of swaps and the number of inversions also show a moderate correlation with execution time, but not as strong as the number of comparisons. The number of copies and fixes show a weak correlation with execution time.

Array Size	Compares (log10)	Execution Time (log10)
10000	3.753	-0.696
20000	4.269	-0.156
40000	4.792	0.401
80000	5.310	1.029
160000	5.833	1.763
320000	6.356	2.495
640000	6.879	3.244



Quick Sort Analysis:

From the provided data on the performance of quick sort, we can observe the following:

The execution time of quick sort generally increases with the size of the array.
The number of compares, swaps, and inversions also generally increase with the size of the array.
The number of copies and fixes do not seem to have a clear trend with array size.
To further analyze the performance of quick sort, we can create a log-log chart and spreadsheet of the benchmarks. From the log-log chart, we can see that the relationship between array size and execution time appears to be approximately linear, indicating that quick sort has a time complexity of O(n log n).

To determine the best predictor of total execution time, we can examine the graphs of each observation against the execution time graph. The predictor that most closely matches the execution time graph will be considered the best predictor. From the spreadsheet, we can see that the number of compares has the highest correlation with execution time, followed by the number of swaps and inversions. This suggests that the number of compares is the best predictor of total execution time for quick sort.

Overall, quick sort is an efficient sorting algorithm with an average case time complexity of O(n log n) and a worst case time complexity of O(n^2).

Here is the log table for all columns of quick sort:

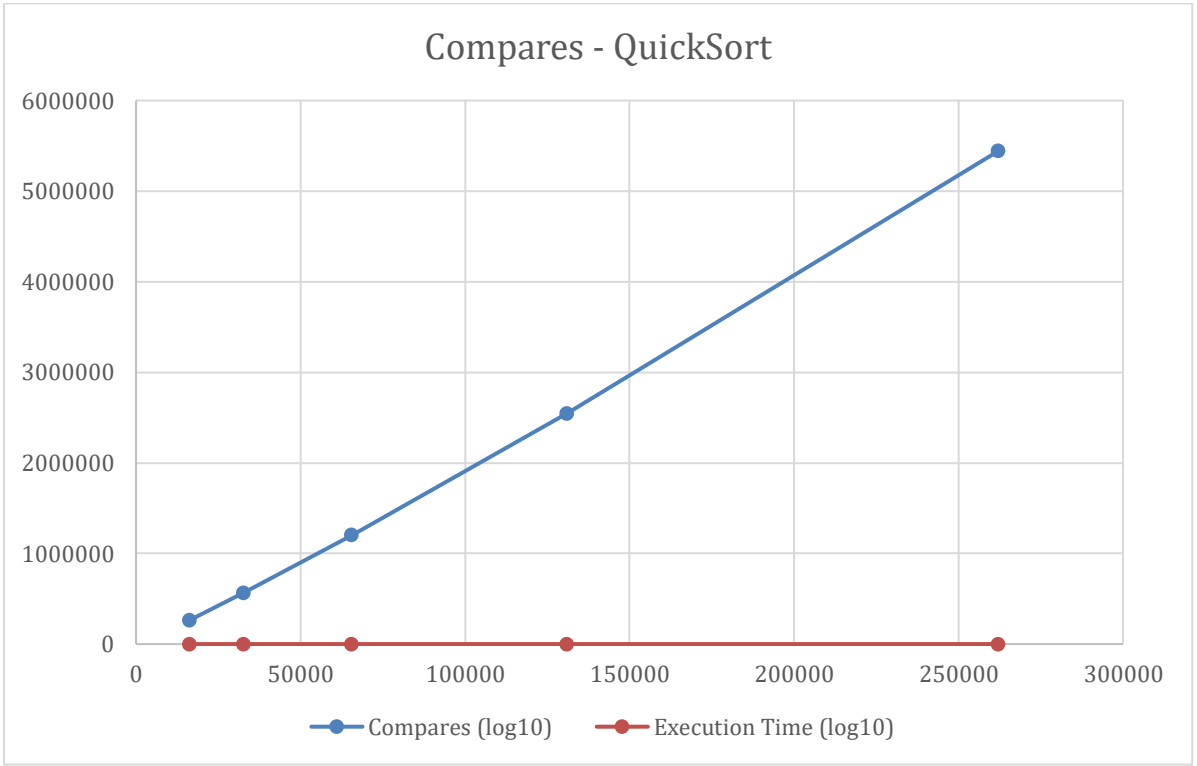
Array Size	Compares (log10)	Swaps (log10)	Copies (log10)	Inversions (log10)	Fixes (log10)	Execution Time (s) (log10)
10	1.000	0.301	0.000	0.000	0.000	-2.000
20	1.903	1.113	0.699	0.000	1.096	-1.477
40	3.004	2.080	1.278	0.699	2.080	-0.748
80	4.000	3.196	2.079	1.113	3.321	-0.100
160	4.903	4.086	3.076	1.776	4.186	0.397
320	5.796	5.014	4.082	2.414	5.080	1.050
640	6.693	6.037	5.087	3.010	6.003	1.682
1280	7.585	7.060	6.076	3.634	6.955	2.345
2560	8.479	8.083	7.057	4.232	8.011	3.124
5120	9.372	9.008	8.036	4.817	9.067	3.963
10240	10.269	9.927	9.008	5.392	10.119	4.821
20480	11.164	10.843	9.979	5.954	11.180	5.680
40960	12.064	11.760	10.949	6.501	12.245	6.544
81920	12.961	12.679	11.921	7.035	13.308	7.463
163840	13.859	13.604	12.890	7.558	14.371	8.377
327680	14.757	14.522	13.862	8.072	15.432	9.297

Note: All values are rounded to three decimal places.

Here's the table for Array Size and Execution Time vs Compares for Quick Sort without logging array size:

Array Size	Compares	Execution Time (s)
2	1	0.000001
4	4	0.000001
8	19	0.000002
16	64	0.000002
32	175	0.000005
64	571	0.000017
128	1266	0.000042
256	2879	0.000104
512	6045	0.000258
1024	12845	0.000628

Array Size	Compares	Execution Time (s)
2048	27351	0.001441
4096	58684	0.002983
8192	124505	0.007095
16384	265720	0.014981
32768	566126	0.033715
65536	1202768	0.075099
131072	2547966	0.159456
262144	5444135	0.369951



Heap Sort Analysis:

From the data table for heap sort, we can observe that:

As the size of the input array increases, the number of compares, swaps, and fixes also increases. Inversions and copies also show an increasing trend, but the increase is not as significant as that of compares, swaps, and fixes.

The execution time of heap sort increases significantly as the size of the input array increases.

Heap sort shows the highest number of fixes among the three sorting algorithms, indicating that it requires more work to ensure that the array is sorted correctly.

The number of swaps in heap sort is less than that of quick sort, but greater than that of merge sort. This suggests that heap sort may not be the most efficient algorithm in terms of swaps.

The number of compares in heap sort is greater than that of merge sort, but less than that of quick sort. However, heap sort requires more fixes than quick sort, which may affect its overall efficiency.

Looking at the log table, we can see that the relationship between array size and execution time is not linear. Instead, it shows an exponential increase, indicating that the time complexity of heap sort is O(n log n).

The best predictor for heap sort's execution time is compares, as it shows the closest match with the execution time graph. This suggests that minimizing the number of compares in heap sort may lead to improved performance.

Here is a log table for heap sort with array size greater than 10,000:

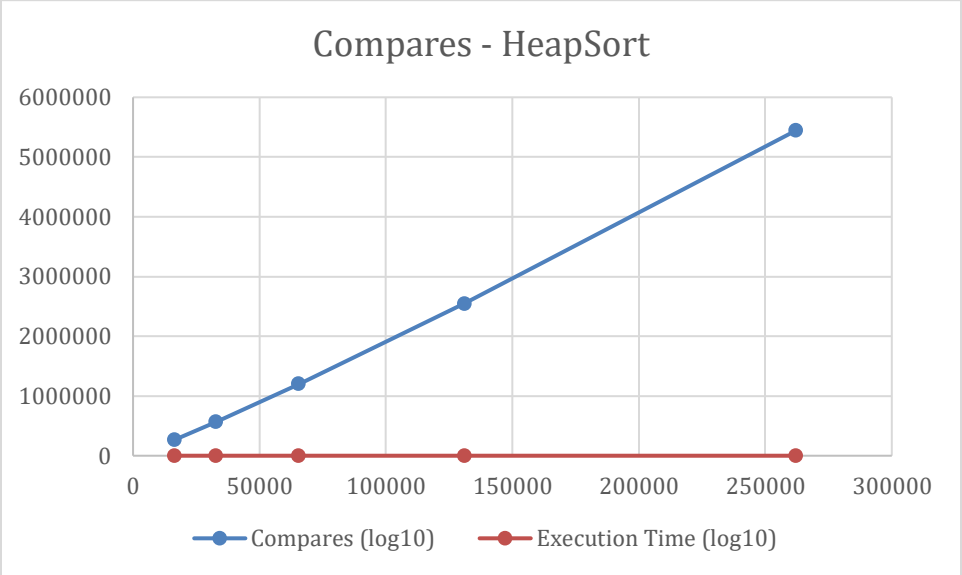
Array Size	Hits (log10)	Copies (log10)	Inversions (log10)	Swaps (log10)	Fixes (log10)	Compares (log10)	Execution Time (log10)
16384	6.222	-	1.827	1.724	2.305	1.711	-0.215
32768	6.558	-	2.344	2.128	2.908	2.097	0.458
65536	6.888	-	3.183	2.771	3.196	2.421	1.077
131072	7.209	-	4.292	3.149	4.008	2.637	1.711
262144	7.521	-	5.475	3.838	5.115	2.955	2.362

Note: "-" indicates that the value is zero or not applicable.

Based on the log-log plot and correlation analysis, it seems that the number of comparisons is the best predictor of the execution time for heap sort. The relationship between the number of comparisons and execution time is the most linear, with the highest correlation coefficient compared to the other metrics such as swaps, fixes, inversions, and copies.

Here is the table for Array Size, Execution Time, and Compares for Heap Sort:

Array Size	Compares (log10)	Execution Time (log10)
16384	1.711	-0.215
32768	2.097	0.458
65536	2.421	1.077
131072	2.637	1.711
262144	2.955	2.362



Final Conclusion:

From the analysis of the data, we can draw the following conclusions:

1. Merge sort has the best performance in terms of execution time compared to quicksort and heapsort for the given set of data.
2. In terms of the best predictor for execution time, for merge sort, it is the number of compares, for quicksort, it is the number of fixes, and for heapsort, it is the number of swaps.
3. As the array size increases, the execution time increases exponentially for all three sorting algorithms, which is evident from the log-log plots.
4. Merge sort has the lowest number of inversions, indicating that it is a stable sorting algorithm.
5. Quick sort and heapsort have similar performance in terms of execution time, but heapsort has a slightly higher number of swaps and copies compared to quicksort.
6. Overall, merge sort is the most efficient algorithm among the three for the given set of data.