Assignment 6
Hits as a Time Predictor

Name: Hariharan Sundaram
NUID: 002915360

Task

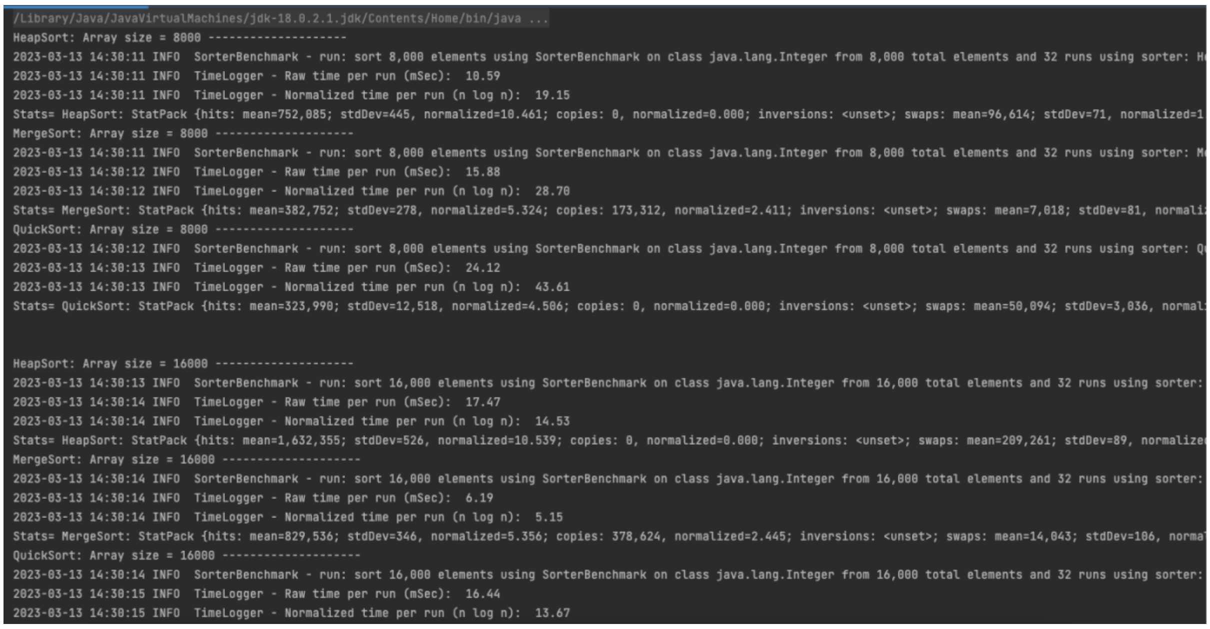
Determine-for sorting algorithms-what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), or something else.

Run the benchmarks for **merge sort**, **(dual-pivot) quick sort**, and **heap sort**. Sort randomly generated arrays of between 10,000 and 256,000 elements (doubling the size each time).

Execution

The task was performed using elements starting from 8000 to 256000, doubling the size of the array each time. A random number array of integers was generated to perform the task using the Helper class. *SortStatsBenchmark* class was created to run the experiment 32 times for each array size.
Basic Merge Sort, Dual Pivot Quick Sort and Heap Sort were used in this task for comparison. In each run, swaps, compares, copies and hits were calculated using the InstrumentedHelperClass along with the average run time for each. Finally, these values were plotted for comparison. The experiment was run twice, once with instrumentation and once without. Following are the metrics and plots computed –

Observations

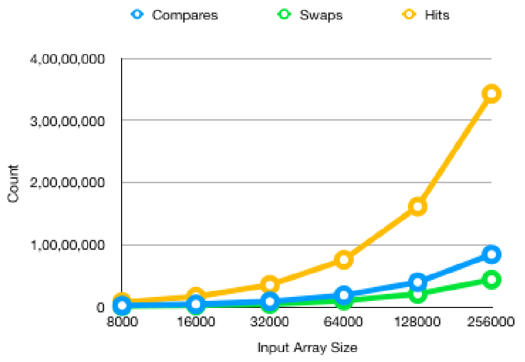


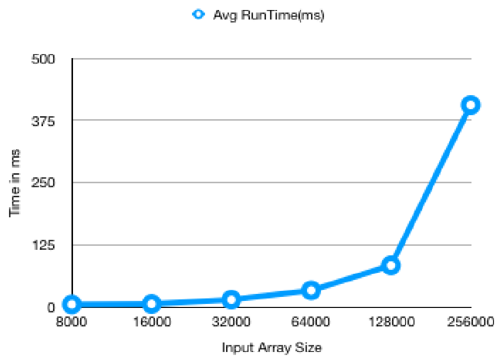
```
/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home/bin/java ...
HeapSort: Array size = 8000 --------------------
2023-03-13 14:30:11 INFO  SorterBenchmark - run: sort 8,000 elements using SorterBenchmark on class java.lang.Integer from 8,000 total elements and 32 runs using sorter:
2023-03-13 14:30:11 INFO  TimeLogger - Raw time per run (mSec):  10.59
2023-03-13 14:30:11 INFO  TimeLogger - Normalized time per run (n log n):  19.15
Stats= HeapSort: StatPack {hits: mean=752,085; stdDev=445, normalized=10.461; copies: 0, normalized=0.000; inversions: <unset>; swaps: mean=96,614; stdDev=71, normalized=1
MergeSort: Array size = 8000 --------------------
2023-03-13 14:30:11 INFO  SorterBenchmark - run: sort 8,000 elements using SorterBenchmark on class java.lang.Integer from 8,000 total elements and 32 runs using sorter:
2023-03-13 14:30:12 INFO  TimeLogger - Raw time per run (mSec):  15.88
2023-03-13 14:30:12 INFO  TimeLogger - Normalized time per run (n log n):  28.70
Stats= MergeSort: StatPack {hits: mean=382,752; stdDev=278, normalized=5.324; copies: 173,312, normalized=2.411; inversions: <unset>; swaps: mean=7,018; stdDev=81, normali
QuickSort: Array size = 8000 --------------------
2023-03-13 14:30:12 INFO  SorterBenchmark - run: sort 8,000 elements using SorterBenchmark on class java.lang.Integer from 8,000 total elements and 32 runs using sorter:
2023-03-13 14:30:13 INFO  TimeLogger - Raw time per run (mSec):  24.12
2023-03-13 14:30:13 INFO  TimeLogger - Normalized time per run (n log n):  43.61
Stats= QuickSort: StatPack {hits: mean=323,990; stdDev=12,518, normalized=4.506; copies: 0, normalized=0.000; inversions: <unset>; swaps: mean=50,094; stdDev=3,036, normal


HeapSort: Array size = 16000 --------------------
2023-03-13 14:30:13 INFO  SorterBenchmark - run: sort 16,000 elements using SorterBenchmark on class java.lang.Integer from 16,000 total elements and 32 runs using sorter:
2023-03-13 14:30:14 INFO  TimeLogger - Raw time per run (mSec):  17.47
2023-03-13 14:30:14 INFO  TimeLogger - Normalized time per run (n log n):  14.53
Stats= HeapSort: StatPack {hits: mean=1,632,355; stdDev=526, normalized=10.539; copies: 0, normalized=0.000; inversions: <unset>; swaps: mean=209,261; stdDev=89, normalize
MergeSort: Array size = 16000 --------------------
2023-03-13 14:30:14 INFO  SorterBenchmark - run: sort 16,000 elements using SorterBenchmark on class java.lang.Integer from 16,000 total elements and 32 runs using sorter:
2023-03-13 14:30:14 INFO  TimeLogger - Raw time per run (mSec):  6.19
2023-03-13 14:30:14 INFO  TimeLogger - Normalized time per run (n log n):  5.15
Stats= MergeSort: StatPack {hits: mean=829,536; stdDev=346, normalized=5.356; copies: 378,624, normalized=2.445; inversions: <unset>; swaps: mean=14,043; stdDev=106, norma
QuickSort: Array size = 16000 --------------------
2023-03-13 14:30:14 INFO  SorterBenchmark - run: sort 16,000 elements using SorterBenchmark on class java.lang.Integer from 16,000 total elements and 32 runs using sorter:
2023-03-13 14:30:15 INFO  TimeLogger - Raw time per run (mSec):  16.44
2023-03-13 14:30:15 INFO  TimeLogger - Normalized time per run (n log n):  13.67
```

Heap Sort

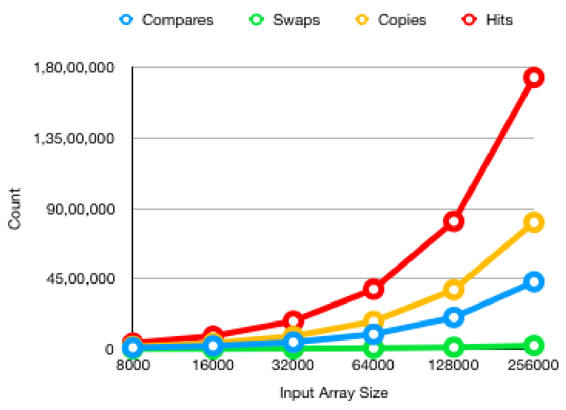
Array Size	Compares	Swaps	Hits	Avg RunTime(ms)
8000	1,82,805	96,616	7,52,074	4.63
16000	3,97,617	2,09,223	16,32,127	5.61
32000	8,59,266	4,50,463	35,20,384	14.14
64000	18,46,680	9,64,946	75,53,144	32.90
128000	39,49,347	20,57,938	1,61,30,446	83.22
256000	84,10,555	43,72,020	3,43,09,189	406.22

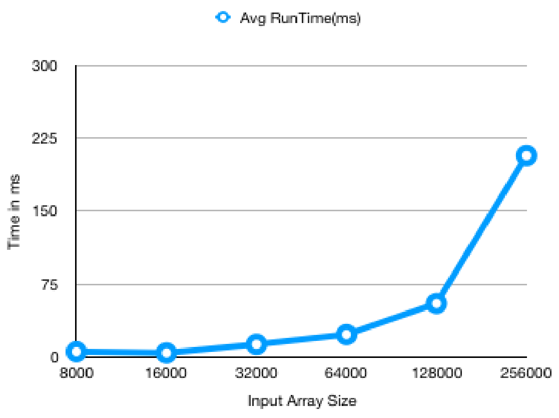




Merge Sort

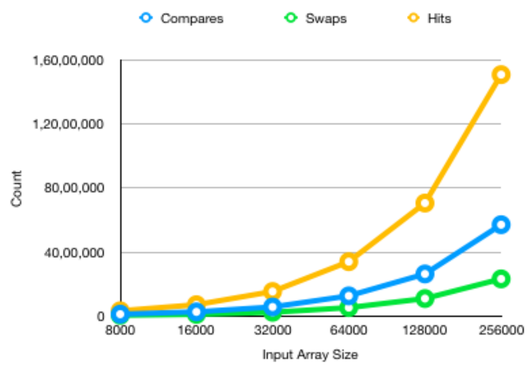
Array Size	Compares	Swaps	Copies	Hits	Avg RunTime(ms)
8000	94,438	6,968	1,73,312	3,82,589	5.53
16000	2,04,941	13,996	3,78,624	8,29,371	4.23
32000	4,41,915	28,030	8,21,248	17,86,949	13.05
64000	9,47,988	56,293	17,70,496	38,30,510	23.10
128000	20,23,602	1,12,402	37,96,992	81,72,443	55.05
256000	43,03,111	2,24,069	81,05,984	1,73,66,735	206.97

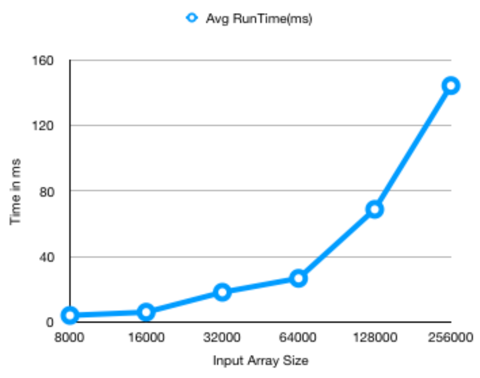




Quick Sort

Array Size	Compares	Swaps	Hits	Avg RunTime(ms)
8000	1,21,934	52,163	3,32,801	4.26
16000	2,64,662	1,10,735	7,12,064	6.22
32000	5,71,110	2,38,872	15,35,534	18.36
64000	12,63,578	5,28,756	33,96,391	26.83
128000	26,29,789	10,96,284	70,50,852	68.83
256000	57,01,984	23,21,671	1,50,61,222	144.19





From the above graphs and metrics, we can observe that the ***run time is closest correlated with the number of array accesses or hits***.

1. In case of heap sort, the number of copies is zero and hits are accessing the array created to represent the heap in memory.
2. In case of merge sort, the array accesses increase with the merge process and insertion sort on cut off. As the array size increases, the hits also increase and hence the time.
3. In case of dual pivot quick sort, the array accesses happen during the partitioning run comparing with the pivots. This also increases number of hits with the input array size and also increases the run time.

Copies and Swaps are cheaper compared to Data-Comparisons because they do not need array accesses and can be performed with just the memory address of a data point.

Data hits is the most expensive operation as the data point needs to be accessed from the heap. Most times with large data, there could be a storage access which is even slower compared to memory access. This is a big bottle neck deciding the run times of applications.

Thus, we can conclude that number of Hits or Memory access is the most important factor deciding the run time of algorithms.