Program Structures and Algorithms

Spring 2023 (Section 3)

Assignment 6 –Sorting Benchmarks

Name: Abhinav Choudhary

NUID: 002780326

Task

- To determine the best predictor of total execution time for three sorting algorithms merge sort, dual pivot quick sort, and heap sort. Various predictors include comparisons, swaps, copies, and hits.
- The experiment is to be executed two times, one with instrumentation and one without instrumentation. Execution time is to be noted and results drawn.
- The array size varies between 10,000 and 256,000 elements, in which array size doubles each turn.

Relationship Conclusion

Following are the best indicator for total execution time for each sorting algorithm:

Merge Sort

- If we check how different values change over time (Line Graph), then compares are the best contributor as its line in the graph is closest to time. Hits (array accesses) is the worst and copies are somewhere in the middle.
- If we check how different values add up over time (Stacked Line Graph), then hits are the best contributor with copies close second and compares at worst position.

Dual Pivot Quick Sort

- For DP Quick Sort, in case of a changes over time (Line Graph), the swaps are the best. Hits the worst and compares in the middle.
- In case of values adding over time (Stacked Line Graph), the compares are the worst contributor, followed by swaps at second and hits being the best contributor.

Heap Sort

- The graphs of heap sort were like dual pivot quick sort, in which in case of changes over time (Line Graph) the swaps were the closest to time followed by compares and hits.
- In case of values added over time (Stacked Line Graph), the hits were the closest, with swaps and compares at second and third position respectively.
- The difference in graphs for heap sort and dual pivot quick sort was that in case of heap sort, for changes over time, there was a bigger gap between hits and swaps/compares, and swaps/compares were closer to each other when compared to graph of dual pivot quick sort. For Value added over time, all three indicators (compares, swaps, hits) were a lot closer to each other, when compared to graph of dual pivot quick sort.

Overall, Compares and Hits were the biggest indicator of total execution time for merge and quick sort, whereas swaps were the best contributor for heap sort.

Total execution time for non-instrumented runs

To note the total execution time (in case of non-instrumented runs), in the beginning (i.e., smaller arrays), merge sort was the fastest with heap sort at second and quick sort dual pivot at the end. This trend is quickly changed as array size grows, in which for larger array sizes quick sort dual pivot performed the best with merge sort relatively close and heap sort the farthest.

Evidence to support conclusion

To find the best predictor of total execution time, instrumented runs were conducted for each sorting algorithm: merge sort, dual pivot quick sort, heap sort. Time (in ms), hits, swaps/copies, compares, fixes were noted. Array size doubled each time, starting from 10,000 elements going all the way to 160,0000 elements, number of runs were adjusted to make sure algorithms do not take forever to complete the sorting. Graphs were then created to compare different parameters and find out the best predictor.

Number of fixes was noted as a predictor, but the values and trend were very different from other predictors, because of which fixes will not be taken into consideration for the comparisons of best predictor.

(The basic merge sort algorithm (MergeSortBasic.java) was used to note predictors of total execution time. That is, insurance and no-copy optimizations were not taken into account.)

The algorithms were also conducted without instrumentation, to note total execution time (in ms) for each sorting algorithm.

Following data was gathered as a result of this experiment:

(Check Assignment6.xlsx for better data readability and graphs)

Instrumented								
Size	Runs	Sorting Algorithm	Time (in ms)	Hits	Copies	Swaps	Fixes	Compares
10000	100	Merge Sort	3.05	534464	267232	0	25005672	120445
		Quick Sort - Dual Pivot	174.69	455683	0	70731	28404736	159502
		Heap Sort	245.73	967510	0	124193	75582830	235369
20000	100	Merge Sort	5.79	1148928	574464	0	100011373	260879
		Quick Sort - Dual Pivot	628.39	982608	0	152049	113013955	347906
		Heap Sort	1015.19	2094933	0	268373	302459900	510720
40000	20	Merge Sort	11.9	2457856	1228928	0	399802976	561801
		Quick Sort - Dual Pivot	2301.35	2085666	0	321460	434247626	746796
		Heap Sort	4359.5	4510353	0	576824	1210477412	1101529
80000	1	Merge Sort	26	5235712	2617856	0	1595682	1203696
		Quick Sort - Dual Pivot	8024	4424152	0	687240	1667500984	1568734
		Heap Sort	17013	9661660	0	1233813	549265741	2363204
1600000	1	Merge Sort	54	11111424	5555712	0	2095568262	2567240
		Quick Sort - Dual Pivot	33811	9992374	0	1581992	1911293663	3452518
		Heap Sort	78534	20602230	0	2627426	2103168331	5046263

Non-Instrumented			
Size	Runs	Sorting Algorithm	Time (in ms)
10000	100	Merge Sort	1.93
		Quick Sort - Dual Pivot	2.3
		Heap Sort	2.28
20000	100	Merge Sort	3.22
		Quick Sort - Dual Pivot	2.11
		Heap Sort	4.33
40000	20	Merge Sort	7
		Quick Sort - Dual Pivot	4.7
		Heap Sort	9.9
80000	1	Merge Sort	15
		Quick Sort - Dual Pivot	10
		Heap Sort	21
1600000	1	Merge Sort	34
		Quick Sort - Dual Pivot	30
		Heap Sort	49

Instrumented

In the above excel data, in case of instrumented, you can see that merge sort takes the least time and predictors of merge sort (hits, copies, fixes, compares) are the least in most cases.

Quick sort dual pivot is second where there is a significant increase in time, but certain predictors are either lower or comparable to predictors of merge sort. For example, the number of hits are almost always lowest when compared to other algorithms, and number of compares are slightly higher than merge sort.

Heap sort takes the most time and its predictors are also the highest by a long margin, close to double in most cases.

Non-Instrumented

As mentioned above (in relationship conclusion section), for smaller array sizes (less than or equal to 10,000 elements) merge sort performs the best with heap sort at second and quick sort dual pivot at the last. But as the array size grows (more than or equal to 20,000 elements), the positions change where quick sort dual pivot becomes the best sorting algorithm with least time, followed by merge sort and heap sort.

For larger arrays, time of quick sort dual pivot and merge sort is somewhat close where heap sort take the most time with a more significant gap.

Following are the screenshots from the SortingBenchmark class:

Array Size = 10,000

```
Console × | Debug Shell | Search | Problems | Executables
SortingBenchmarks [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe (03-Mar-2023, 3:49:38 pm) [pid: 7908]
Size: 10000
Runs: 100
Instrumented
2023-03-03 15:49:38 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: merge sort
2023-03-03 15:49:38 INFO  Benchmark_Timer - Begin run: Instrumenting helper for merge sort with 10,000 elements with 100 runs
2023-03-03 15:49:38 INFO  TimeLogger - Raw time per run (mSec):  3.05
merge sort: StatPack {hits: 534,464, normalized=5.803; copies: 267,232, normalized=2.901; inversions: 24,811,115, normalized=269.383; swaps: 0,
normalized=0.000; fixes: mean=25,005,672; stdDev=155,575, normalized=271.496; compares: mean=120,445; stdDev=55, normalized=1.308}
2023-03-03 15:49:38 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: quick sort dual pivot
2023-03-03 15:49:38 INFO  Benchmark_Timer - Begin run: Instrumenting helper for quick sort dual pivot with 10,000 elements with 100 runs
2023-03-03 15:49:57 INFO  TimeLogger - Raw time per run (mSec):  174.69
quick sort dual pivot: StatPack {hits: mean=455,683; stdDev=15,985, normalized=4.948; copies: 0, normalized=0.000; inversions: 24,811,115,
normalized=269.383; swaps: mean=70,731; stdDev=3,525, normalized=0.768; fixes: mean=28,404,736; stdDev=3,309,634, normalized=308.401; compares:
mean=159,502; stdDev=6,371, normalized=1.732}
2023-03-03 15:49:57 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: heap sort
2023-03-03 15:49:57 INFO  Benchmark_Timer - Begin run: Instrumenting helper for heap sort with 10,000 elements with 100 runs
2023-03-03 15:50:23 INFO  TimeLogger - Raw time per run (mSec):  245.73
heap sort: StatPack {hits: mean=967,510; stdDev=428, normalized=10.505; copies: 0, normalized=0.000; inversions: 24,811,115, normalized=269.383;
swaps: mean=124,193; stdDev=70, normalized=1.348; fixes: mean=75,582,830; stdDev=191,944, normalized=820.630; compares: mean=235,369; stdDev=88,
normalized=2.555}
Non-Instrumented
2023-03-03 15:50:23 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:50:23 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:50:23 INFO  TimeLogger - Raw time per run (mSec):  1.93
2023-03-03 15:50:23 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:50:23 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:50:24 INFO  TimeLogger - Raw time per run (mSec):  2.30
2023-03-03 15:50:24 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:50:24 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:50:24 INFO  TimeLogger - Raw time per run (mSec):  2.28
```

Array Size = 20,000

```
SortingBenchmarks [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe (03-Mar-2023, 3:49:38 pm) [pid: 7908]
Size: 20000
Runs: 100
Instrumented
2023-03-03 15:50:24 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: merge sort
2023-03-03 15:50:24 INFO  Benchmark_Timer - Begin run: Instrumenting helper for merge sort with 20,000 elements with 100 runs
2023-03-03 15:50:25 INFO  TimeLogger - Raw time per run (mSec):  5.79
merge sort: StatPack {hits: 1,148,928, normalized=5.801; copies: 574,464, normalized=2.900; inversions: 100,464,160, normalized=507.216; swaps: 0,
normalized=0.000; fixes: mean=100,011,373; stdDev=452,776, normalized=504.930; compares: mean=260,879; stdDev=88, normalized=1.317}
2023-03-03 15:50:25 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: quick sort dual pivot
2023-03-03 15:50:25 INFO  Benchmark_Timer - Begin run: Instrumenting helper for quick sort dual pivot with 20,000 elements with 100 runs
2023-03-03 15:51:32 INFO  TimeLogger - Raw time per run (mSec):  628.39
quick sort dual pivot: StatPack {hits: mean=982,608; stdDev=40,433, normalized=4.961; copies: 0, normalized=0.000; inversions: 100,464,160,
normalized=507.216; swaps: mean=152,049; stdDev=7,854, normalized=0.768; fixes: mean=113,013,955; stdDev=13,199,443, normalized=570.577; compares:
mean=347,906; stdDev=15,037, normalized=1.756}
2023-03-03 15:51:32 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: heap sort
2023-03-03 15:51:32 INFO  Benchmark_Timer - Begin run: Instrumenting helper for heap sort with 20,000 elements with 100 runs
2023-03-03 15:53:20 INFO  TimeLogger - Raw time per run (mSec):  1015.19
heap sort: StatPack {hits: mean=2,094,933; stdDev=619, normalized=10.577; copies: 0, normalized=0.000; inversions: 100,464,160, normalized=507.216;
swaps: mean=268,373; stdDev=99, normalized=1.355; fixes: mean=302,459,900; stdDev=646,390, normalized=1527.037; compares: mean=510,720; stdDev=131,
normalized=2.578}
Non-Instrumented
2023-03-03 15:53:20 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:53:20 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:53:21 INFO  TimeLogger - Raw time per run (mSec):  3.22
2023-03-03 15:53:21 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:53:21 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:53:21 INFO  TimeLogger - Raw time per run (mSec):  2.11
2023-03-03 15:53:21 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements
and 100 runs using sorter: Standard Helper
2023-03-03 15:53:21 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 100 runs
2023-03-03 15:53:21 INFO  TimeLogger - Raw time per run (mSec):  4.33
```

Array Size = 40,000

```
SortingBenchmarks [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (03-Mar-2023, 3:49:38 pm) [pid: 7908]
Size: 40000
Runs: 20
Instrumented
2023-03-03 15:53:21 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: merge sort
2023-03-03 15:53:21 INFO  Benchmark_Timer - Begin run: Instrumenting helper for merge sort with 40,000 elements with 20 runs
2023-03-03 15:53:22 INFO  TimeLogger - Raw time per run (mSec):  11.90
merge sort: StatPack {hits: 2,457,856, normalized=5.799; copies: 1,228,928, normalized=2.899; inversions: 399,334,658, normalized=942.126; swaps:
0, normalized=0.000; fixes: mean=399,802,976; stdDev=1,278,021, normalized=943.231; compares: mean=561,801; stdDev=125, normalized=1.325}
2023-03-03 15:53:22 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: quick sort dual pivot
2023-03-03 15:53:22 INFO  Benchmark_Timer - Begin run: Instrumenting helper for quick sort dual pivot with 40,000 elements with 20 runs
2023-03-03 15:54:13 INFO  TimeLogger - Raw time per run (mSec):  2301.35
quick sort dual pivot: StatPack {hits: mean=2,085,666; stdDev=71,772, normalized=4.921; copies: 0, normalized=0.000; inversions: 399,334,658,
normalized=942.126; swaps: mean=321,460; stdDev=18,526, normalized=0.758; fixes: mean=434,247,626; stdDev=26,613,140, normalized=1024.494;
compares: mean=746,796; stdDev=26,309, normalized=1.762}
2023-03-03 15:54:13 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: heap sort
2023-03-03 15:54:13 INFO  Benchmark_Timer - Begin run: Instrumenting helper for heap sort with 40,000 elements with 20 runs
2023-03-03 15:55:49 INFO  TimeLogger - Raw time per run (mSec):  4359.50
heap sort: StatPack {hits: mean=4,510,353; stdDev=787, normalized=10.641; copies: 0, normalized=0.000; inversions: 399,334,658, normalized=942.126;
swaps: mean=576,824; stdDev=125, normalized=1.361; fixes: mean=1,210,477,412; stdDev=1,479,035, normalized=2855.806; compares: mean=1,101,529;
stdDev=154, normalized=2.599}
Non-Instrumented
2023-03-03 15:55:49 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: Standard Helper
2023-03-03 15:55:49 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 20 runs
2023-03-03 15:55:49 INFO  TimeLogger - Raw time per run (mSec):  7.00
2023-03-03 15:55:49 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: Standard Helper
2023-03-03 15:55:49 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 20 runs
2023-03-03 15:55:49 INFO  TimeLogger - Raw time per run (mSec):  4.70
2023-03-03 15:55:49 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements
and 20 runs using sorter: Standard Helper
2023-03-03 15:55:49 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 20 runs
2023-03-03 15:55:49 INFO  TimeLogger - Raw time per run (mSec):  9.90
```

Array Size = 80,000

```
SortingBenchmarks [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (03-Mar-2023, 3:49:38 pm) [pid: 7908]
Size: 80000
Runs: 1
Instrumented
2023-03-03 15:55:49 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: merge sort
2023-03-03 15:55:49 INFO  Benchmark_Timer - Begin run: Instrumenting helper for merge sort with 80,000 elements with 1 runs
2023-03-03 15:55:49 INFO  TimeLogger - Raw time per run (mSec):  26.00
merge sort: StatPack {hits: 5,235,712, normalized=5.797; copies: 2,617,856, normalized=2.898; inversions: 1,600,304,323, normalized=1771.850;
swaps: 0, normalized=0.000; fixes: 1,595,682,836, normalized=1766.733; compares: 1,203,696, normalized=1.333}
2023-03-03 15:55:49 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: quick sort dual pivot
2023-03-03 15:55:49 INFO  Benchmark_Timer - Begin run: Instrumenting helper for quick sort dual pivot with 80,000 elements with 1 runs
2023-03-03 15:56:18 INFO  TimeLogger - Raw time per run (mSec):  8024.00
quick sort dual pivot: StatPack {hits: 4,424,152, normalized=4.898; copies: 0, normalized=0.000; inversions: 1,600,304,323, normalized=1771.850;
swaps: 687,240, normalized=0.761; fixes: 1,667,500,984, normalized=1846.250; compares: 1,568,734, normalized=1.737}
2023-03-03 15:56:18 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: heap sort
2023-03-03 15:56:18 INFO  Benchmark_Timer - Begin run: Instrumenting helper for heap sort with 80,000 elements with 1 runs
2023-03-03 15:57:09 INFO  TimeLogger - Raw time per run (mSec):  17013.00
heap sort: StatPack {hits: 9,661,660, normalized=10.697; copies: 0, normalized=0.000; inversions: 1,600,304,323, normalized=1771.850; swaps:
1,233,813, normalized=1.366; fixes: 549,265,741, normalized=608.145; compares: 2,363,204, normalized=2.617}
Non-Instrumented
2023-03-03 15:57:09 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 15:57:09 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 15:57:09 INFO  TimeLogger - Raw time per run (mSec):  15.00
2023-03-03 15:57:09 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 15:57:09 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 15:57:09 INFO  TimeLogger - Raw time per run (mSec):  10.00
2023-03-03 15:57:09 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 15:57:09 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 15:57:09 INFO  TimeLogger - Raw time per run (mSec):  21.00
```

Array Size = 160,000

```
Size: 160000
Runs: 1
Instrumented
2023-03-03 15:57:09 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: merge sort
2023-03-03 15:57:09 INFO  Benchmark_Timer - Begin run: Instrumenting helper for merge sort with 160,000 elements with 1 runs
2023-03-03 15:57:09 INFO  TimeLogger - Raw time per run (mSec):  54.00
merge sort: StatPack {hits: 11,111,424, normalized=5.795; copies: 5,555,712, normalized=2.898; inversions: 2,117,702,738, normalized=1104.541;
swaps: 0, normalized=0.000; fixes: 2,095,568,262, normalized=1092.997; compares: 2,567,240, normalized=1.339}
2023-03-03 15:57:09 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: quick sort dual pivot
2023-03-03 15:57:09 INFO  Benchmark_Timer - Begin run: Instrumenting helper for quick sort dual pivot with 160,000 elements with 1 runs
2023-03-03 15:58:58 INFO  TimeLogger - Raw time per run (mSec):  33811.00
quick sort dual pivot: StatPack {hits: 9,992,374, normalized=5.212; copies: 0, normalized=0.000; inversions: 2,117,702,738, normalized=1104.541;
swaps: 1,581,992, normalized=0.825; fixes: -1,911,293,663, normalized=-996.884; compares: 3,452,518, normalized=1.801}
2023-03-03 15:58:58 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: heap sort
2023-03-03 15:58:58 INFO  Benchmark_Timer - Begin run: Instrumenting helper for heap sort with 160,000 elements with 1 runs
2023-03-03 16:02:54 INFO  TimeLogger - Raw time per run (mSec):  78534.00
heap sort: StatPack {hits: 20,602,230, normalized=10.746; copies: 0, normalized=0.000; inversions: 2,117,702,738, normalized=1104.541; swaps:
2,627,426, normalized=1.370; fixes: -2,103,168,331, normalized=-1096.961; compares: 5,046,263, normalized=2.632}
Non-Instrumented
2023-03-03 16:02:54 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 16:02:54 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 16:02:54 INFO  TimeLogger - Raw time per run (mSec):  34.00
2023-03-03 16:02:54 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 16:02:54 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 16:02:54 INFO  TimeLogger - Raw time per run (mSec):  30.00
2023-03-03 16:02:54 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements
and 1 runs using sorter: Standard Helper
2023-03-03 16:02:54 INFO  Benchmark_Timer - Begin run: Helper for Standard Helper with 0 elements instrumented with 1 runs
2023-03-03 16:02:55 INFO  TimeLogger - Raw time per run (mSec):  49.00
```

Graphical Representation

For each sorting algorithm, multiple graphs were created, individual line graphs for each indicator and a combined version containing each predictor. For combined graphs, two versions were created for each sorting algorithm: Line graph (to view changes over time) and Stacked line graph (to view how values add up over time).

Individual graphs for each predictor are also created, but all the predictors (hits, swaps/copies, compares) graphs were very similar to the time graph. It was very difficult to discern which is the best predictor and thus, combined graphs were created to check which predictor is the closest to time.

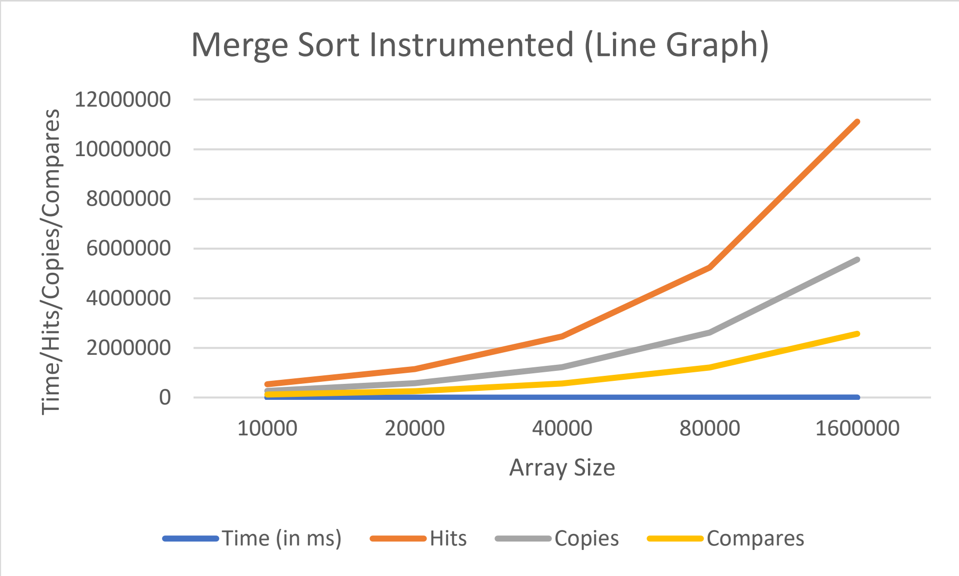
Graphs for non-instrumented runs was not created as they only showed time differences for each array size.

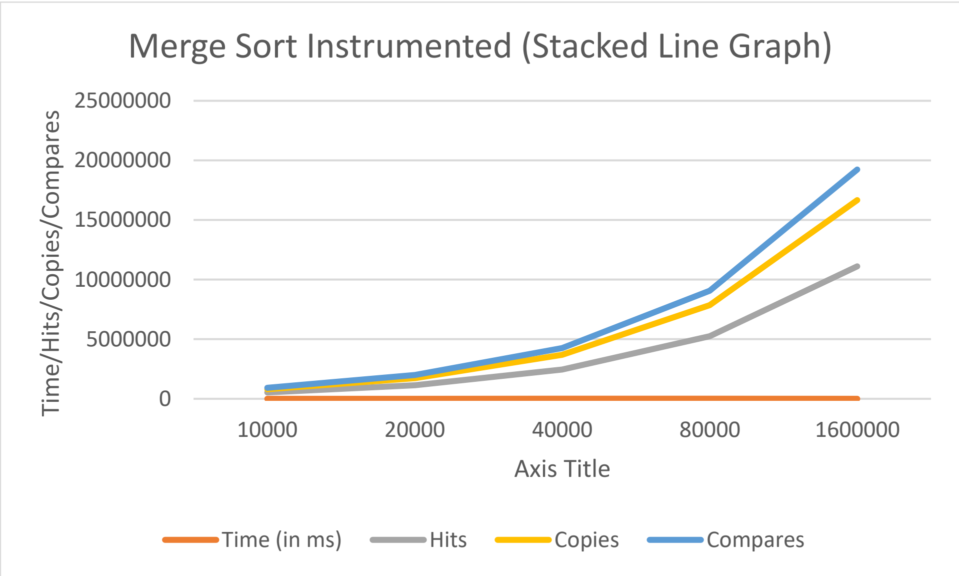
All the graphs were created in Microsoft Excel.

Combined Graphs

Following are combined graphs for each sorting algorithm:

Merge Sort



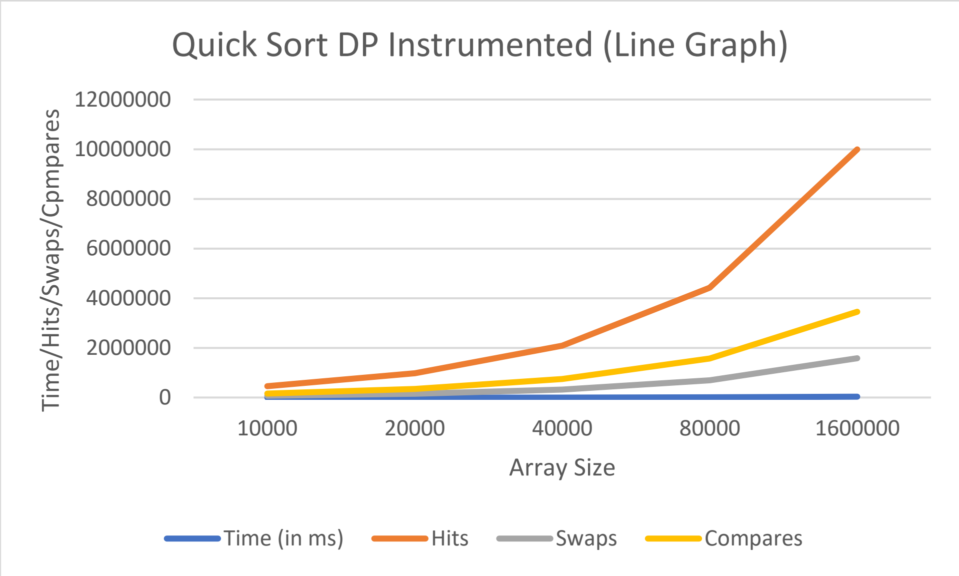


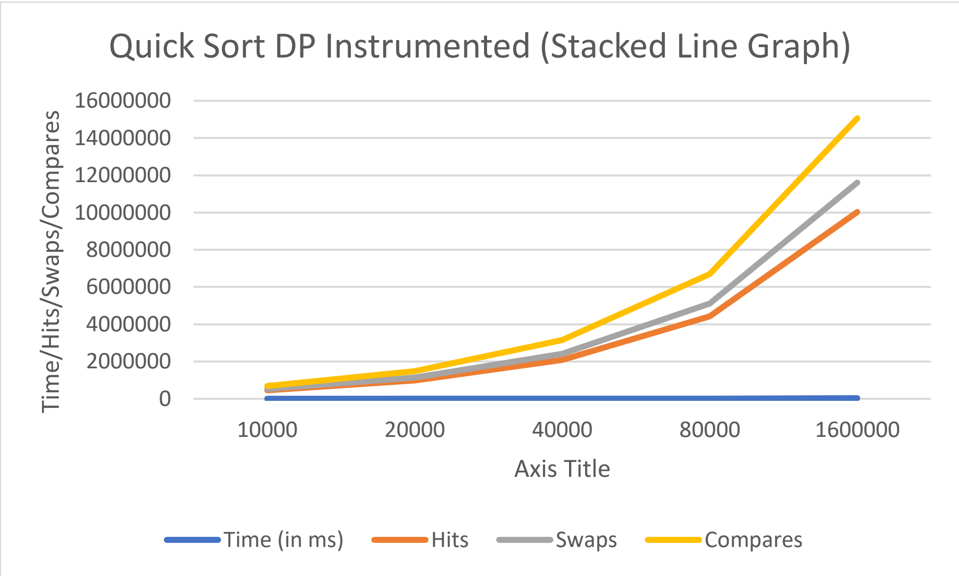
In the graphs provided, we can see that in case of line graph, compares is the best predictor as it is the closest to line for time. Hits is the worst in this case as it is the farthest from the line for time.

In case of stacked line graph, hits are the best predictor as it is closest to the line for time, and compares is the worst as it is the farthest.

Copies is in the middle in both the cases.

Dual Pivot Quick Sort

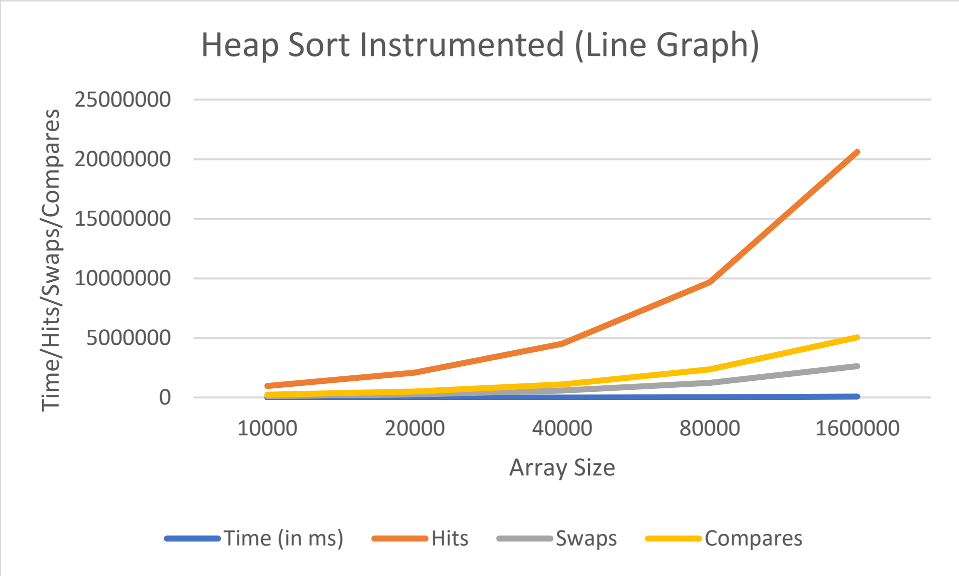


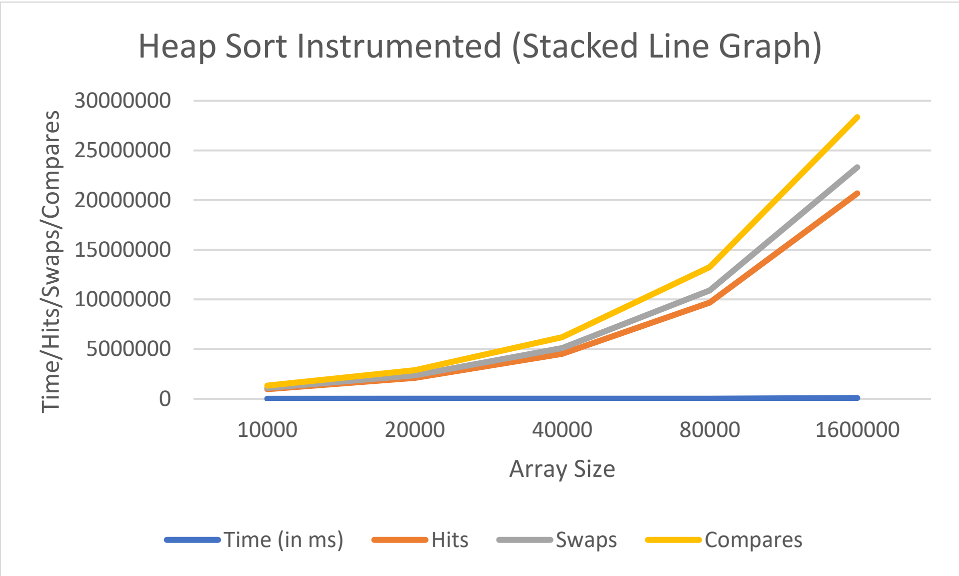


For dual pivot quick sort, in case of line graph, swaps are the best predictor as it is the closest to time and hits is the worst as it is the farthest.

In case of stacked line graph, hits is the best predictor with compares being the worst.

Heap Sort





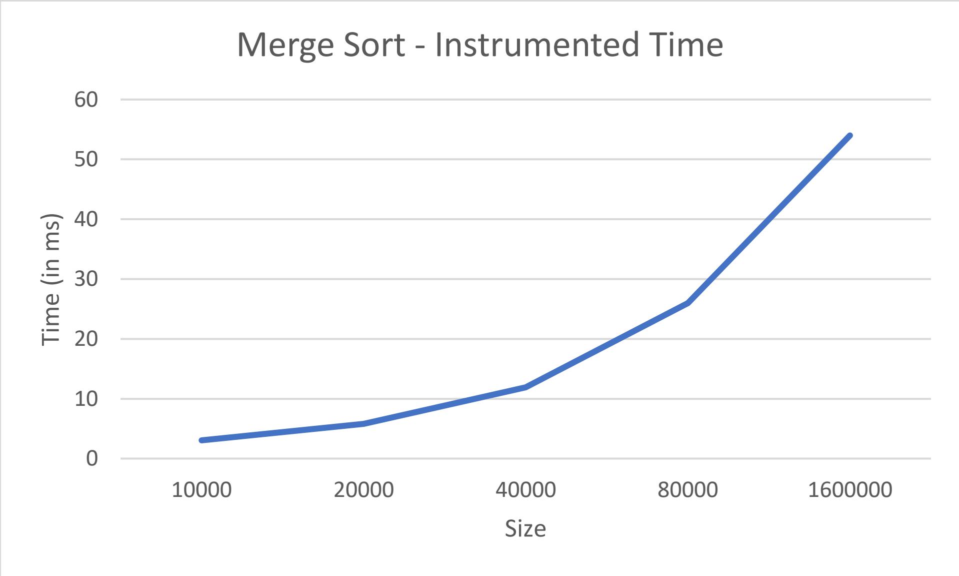
For heap sort, in case of line graph, swaps are the closest to time, and hits the farthest.

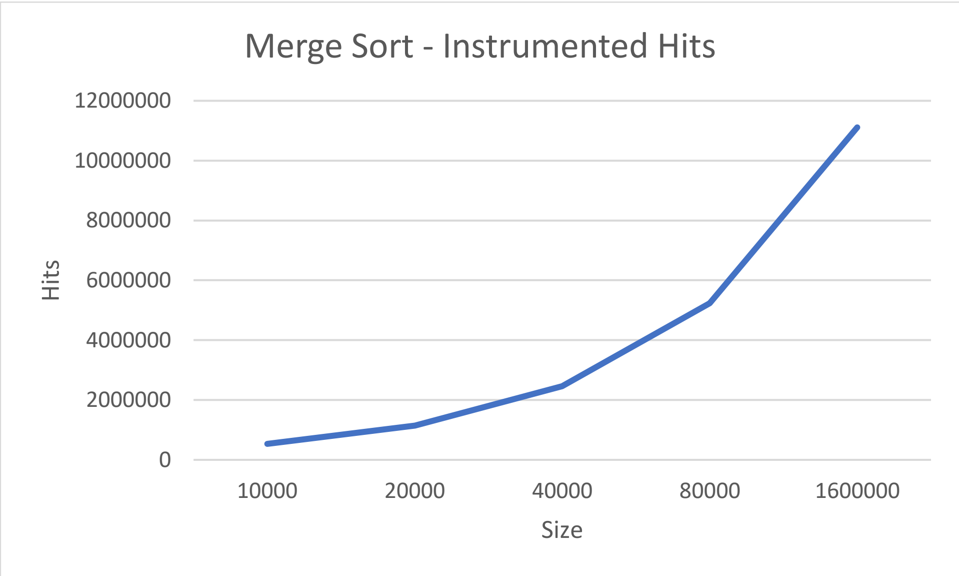
In case of stacked line graph, hits are the closest to time with swaps relatively to close to hits and compares being the farthest.

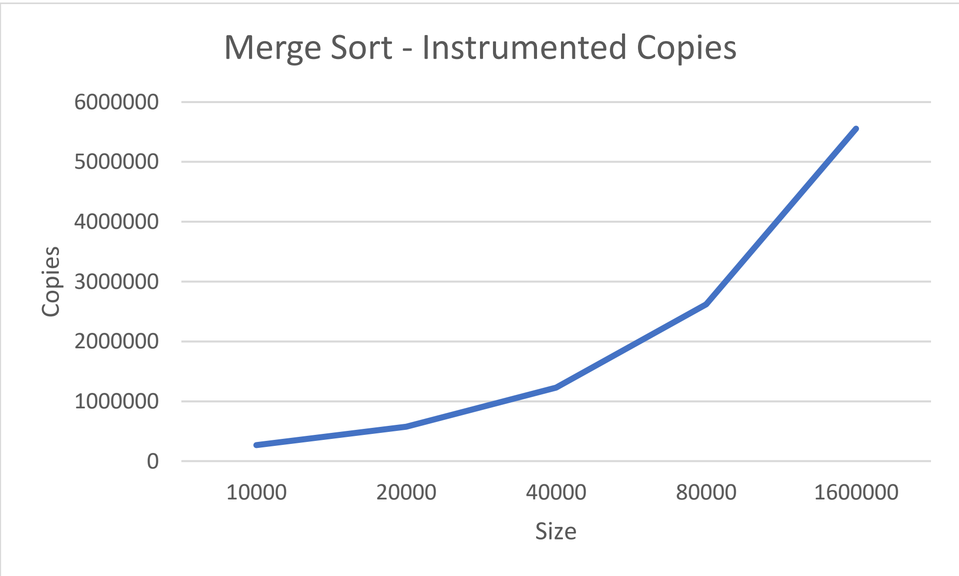
Individual graphs

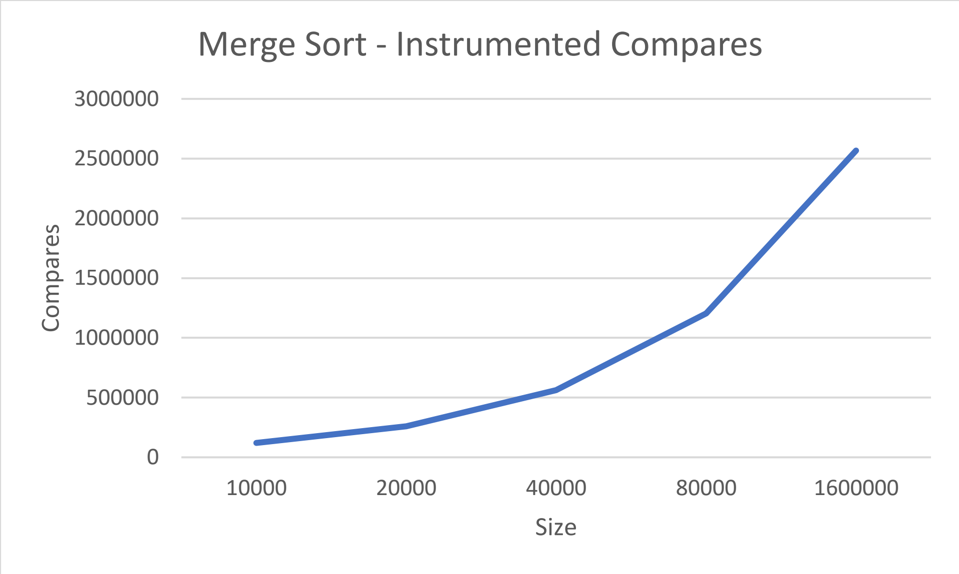
Following are individual graphs for each algorithm:

Merge Sort

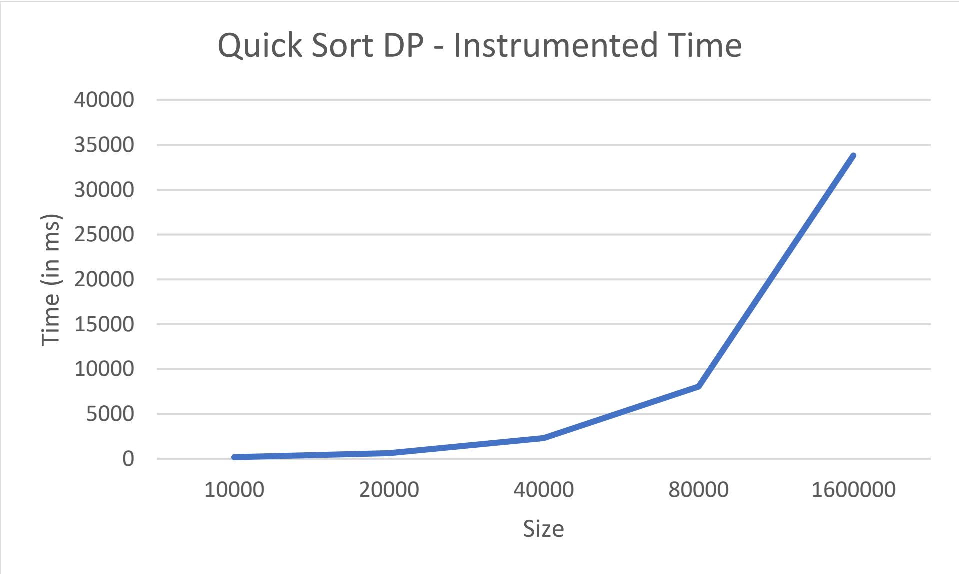

Merge Sort - Instrumented Time
Time (in ms)
60
50
40
30
20
10
0
10000
20000
40000
80000
1600000
Size

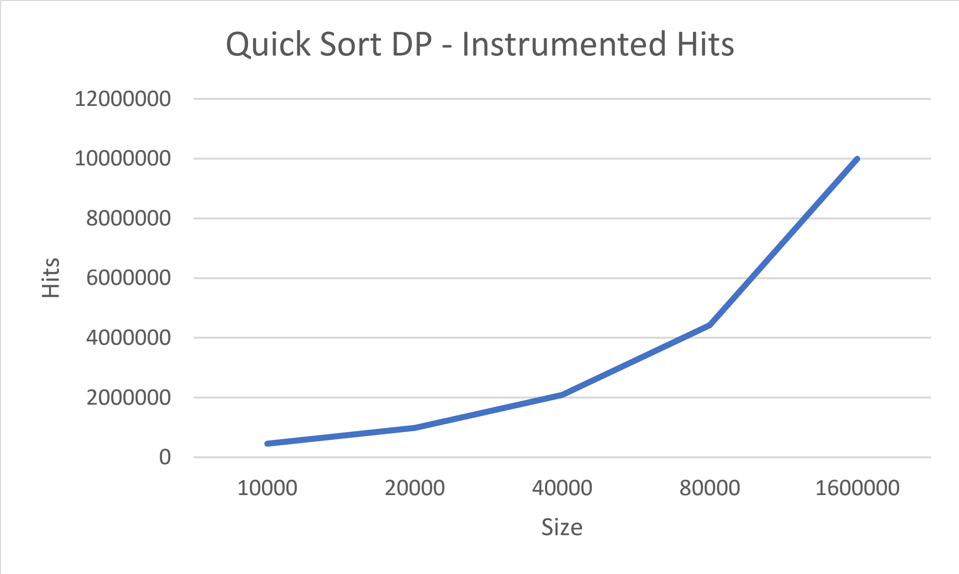

Merge Sort - Instrumented Hits
Hits
12000000
10000000
8000000
6000000
4000000
2000000
0
10000
20000
40000
80000
1600000
Size

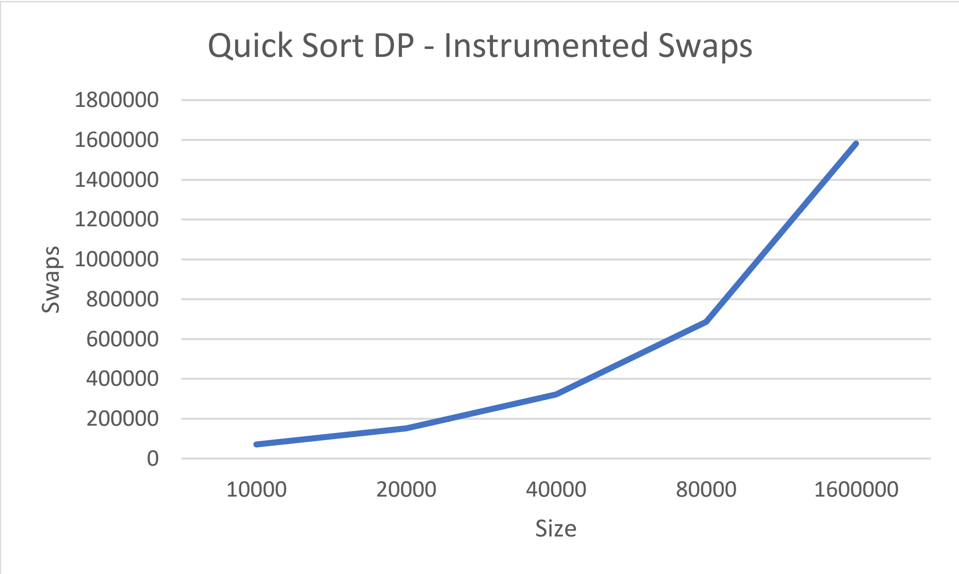

Merge Sort - Instrumented Copies
Copies
6000000
5000000
4000000
3000000
2000000
1000000
0
10000
20000
40000
80000
1600000
Size

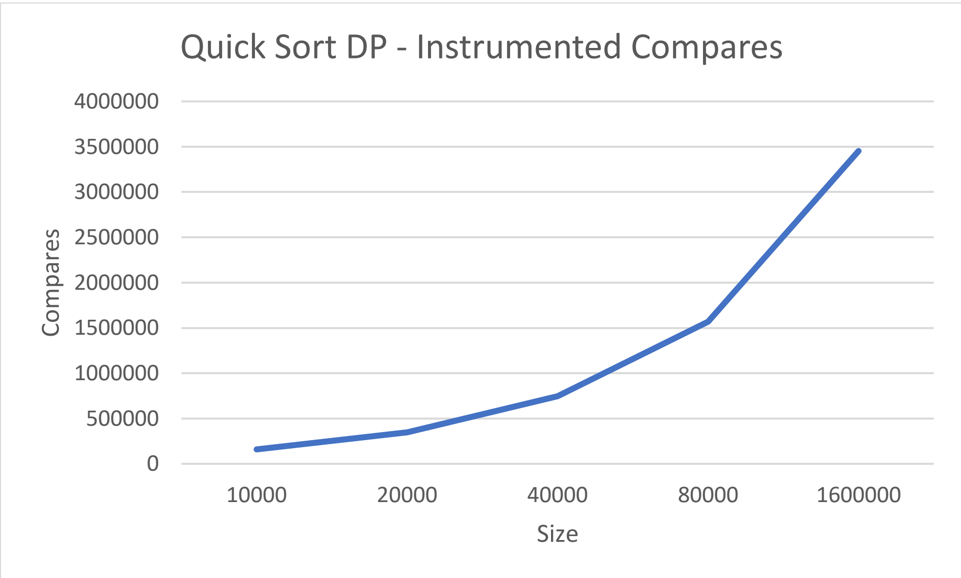


Dual Pivot Quick Sort

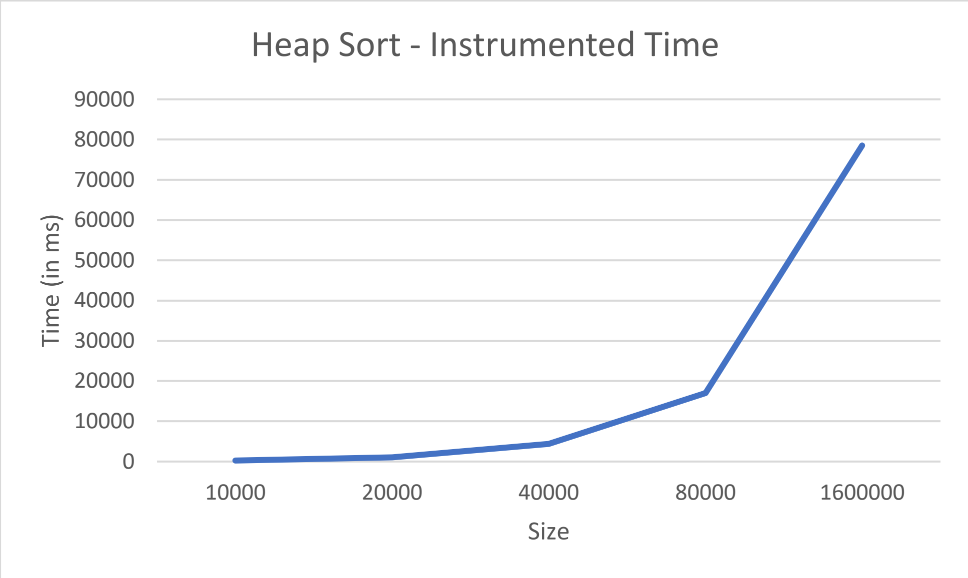



Quick Sort DP - Instrumented Hits
12000000
10000000
8000000
6000000
4000000
2000000
0
Hits
10000
20000
40000
80000
1600000
Size

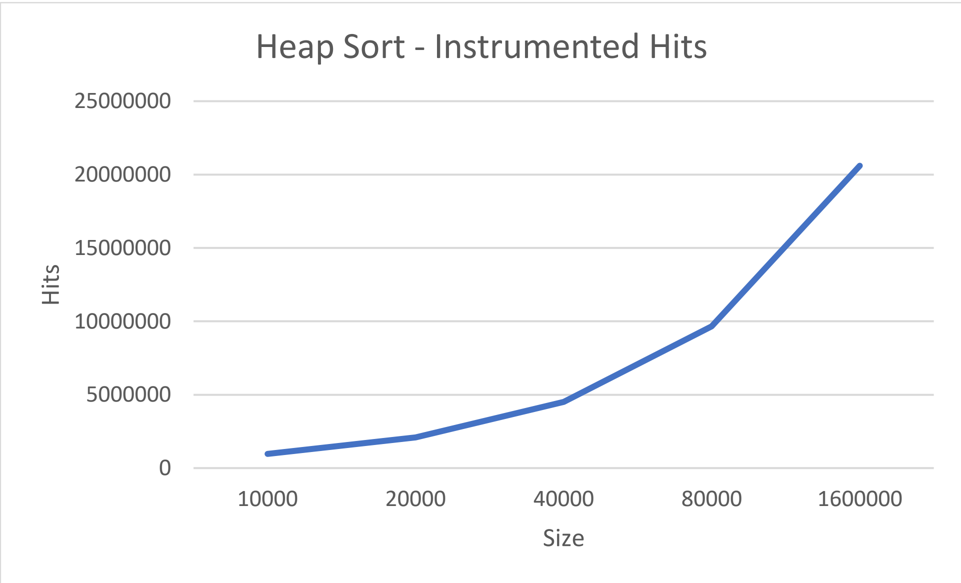

Quick Sort DP - Instrumented Swaps
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
Swaps
10000
20000
40000
80000
1600000
Size


Quick Sort DP - Instrumented Compares
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Compares
10000
20000
40000
80000
1600000
Size

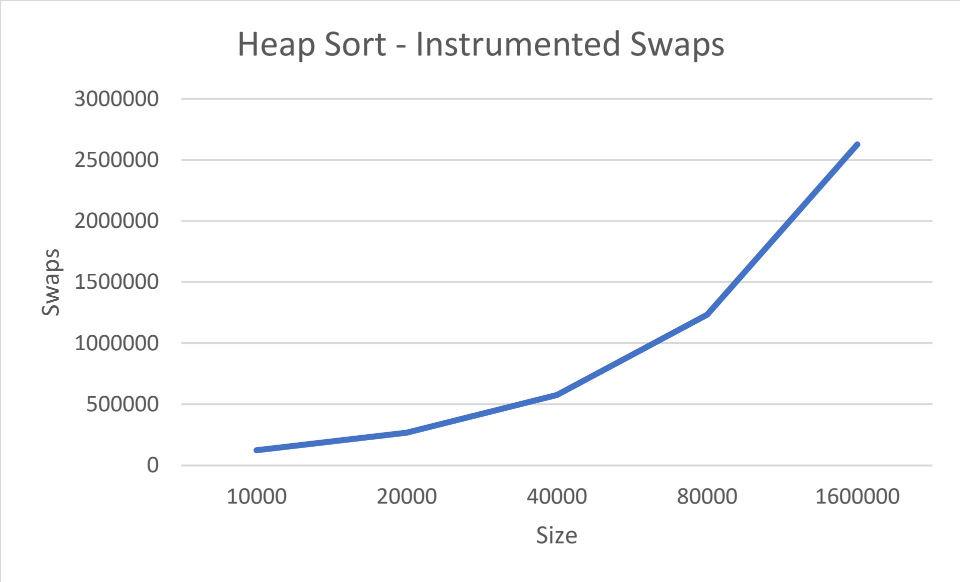
Heap Sort

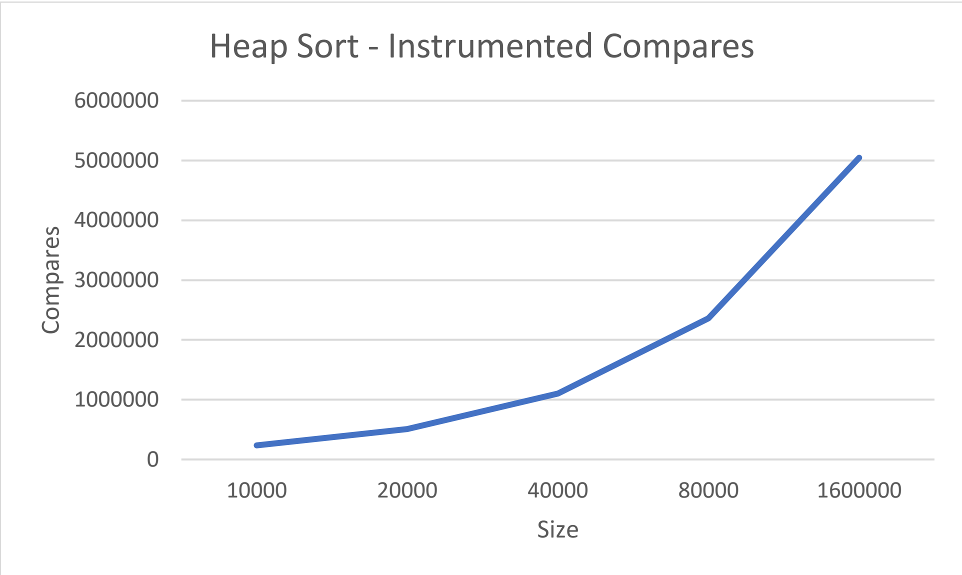


Heap Sort - Instrumented Time
Time (in ms)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
10000
20000
40000
80000
1600000
Size



Heap Sort - Instrumented Hits
Hits
25000000
20000000
15000000
10000000
5000000
0
10000
20000
40000
80000
1600000
Size



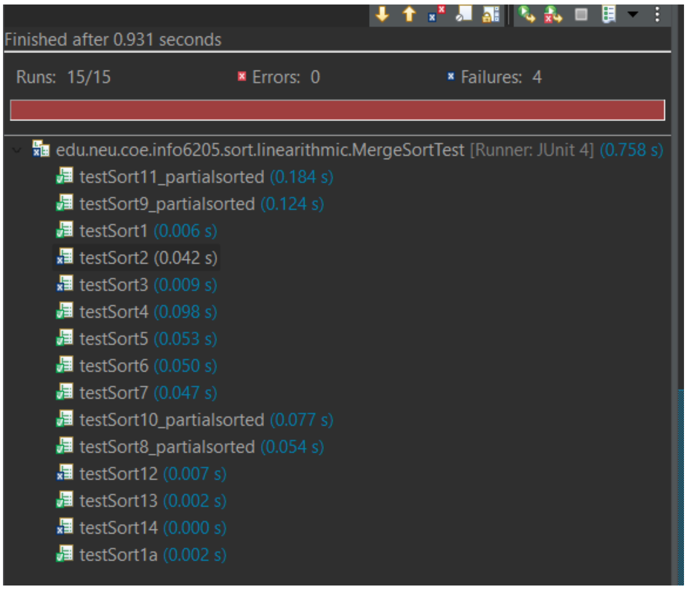


Unit Tests

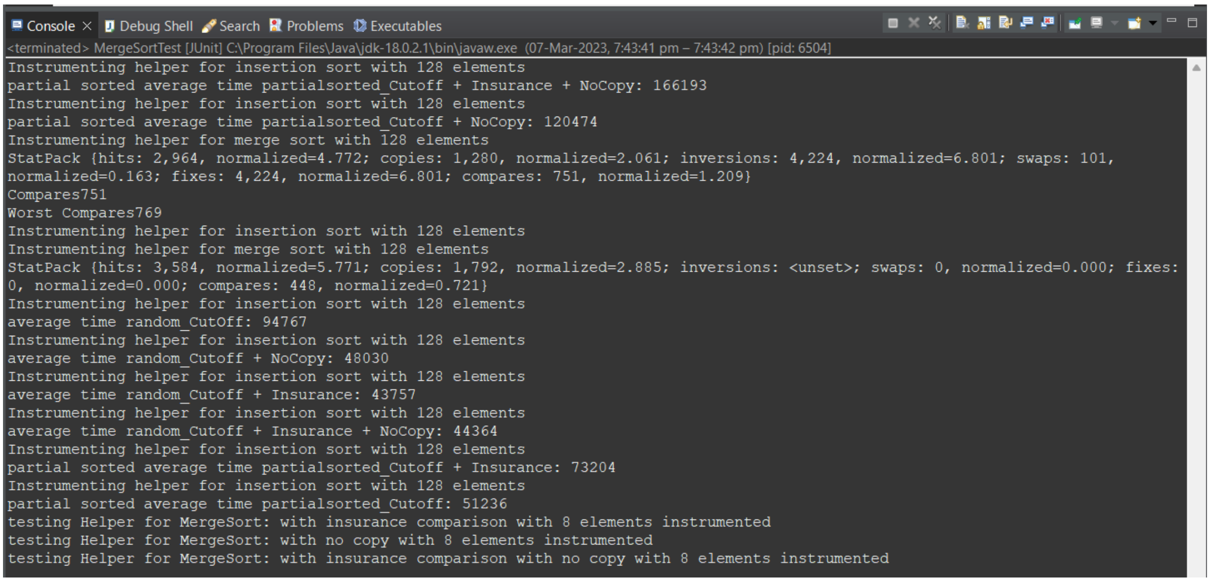
Following are unit test benchmarks and console output for merge sort test, quick sort dual pivot test, and heap sort test.

Merge Sort Test

Benchmark



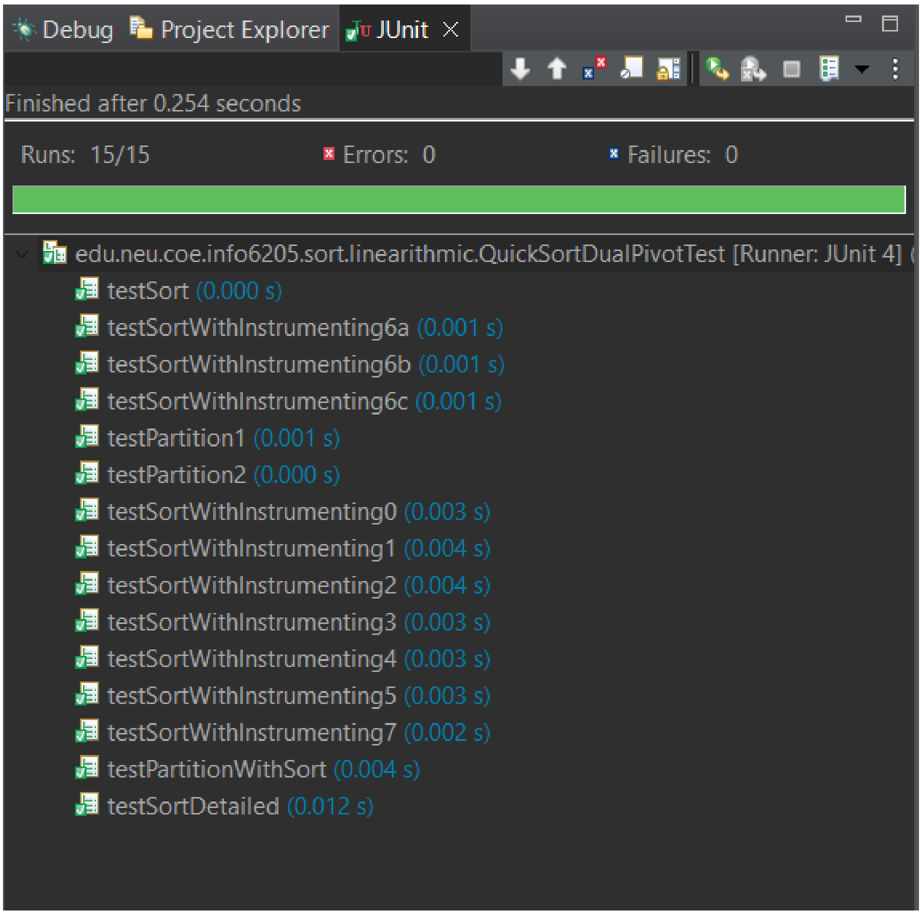
Console



Dual Pivot Quick Sort Test

In QuickSortDualPivotTest.java file, there was an error in testSortDetailed() method at line 250, where the BaseHelper was throwing the error. BaseHelper was replaced with Helper to resolve the error. Apart from this, no change was made to the test file.

Benchmark



Console

```
Console × | Debug Shell | Search | Problems | Executables
<terminated> QuickSortDualPivotTest [JUnit] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (07-Mar-2023, 7:48:47 pm – 7:48:48 pm) [pid: 7372]
Instrumenting helper for quick sort dual pivot with 128 elements
StatPack {hits: 2,619, normalized=4.217; copies: 0, normalized=0.000; inversions: 4,224, normalized=6.801; swaps: 435,
normalized=0.700; fixes: 4,740, normalized=7.632; compares: 950, normalized=1.530}
compares: 950, worstCompares: 1242
```

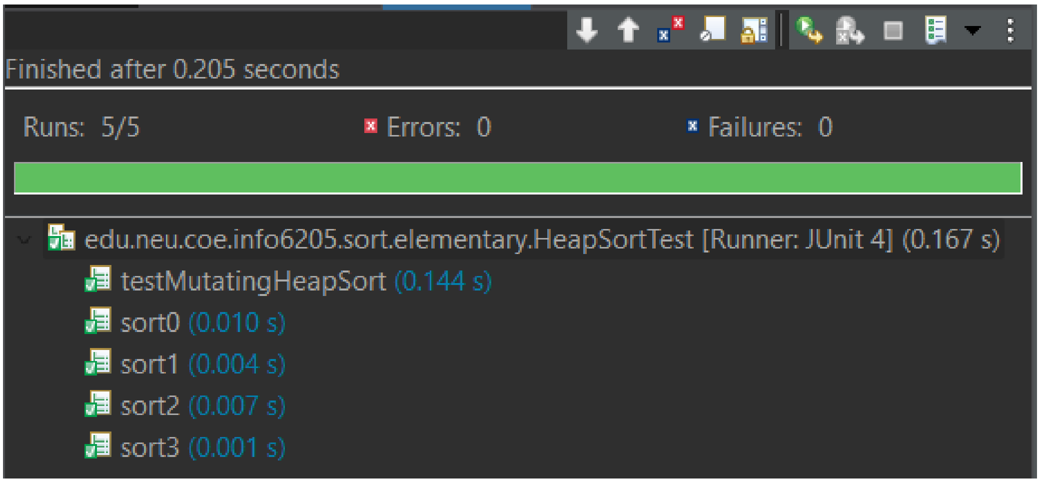
Error correction

```
@Test
public void testSortDetailed() throws Exception {
    int k = 7;
    int N = (int) Math.pow(2, k);
    // NOTE this depends on the cutoff value for quick sort.
    int levels = k - 2;
    final Config config = Config.setupConfig("true", "0", "1", "", "");
    final BaseHelper<Integer> helper = (BaseHelper<Integer>) HelperFactory.create("quick sort dual pivot", N, config);
    System.out.println(helper);
    Sort<Integer> s = new QuickSort_DualPivot<>(helper);
    s.init(N);
    final Integer[] xs = helper.random(Integer.class, r -> r.nextInt(10000));
    assertEquals(Integer.valueOf(1360), xs[0]);
    helper.preProcess(xs);
    Integer[] ys = s.sort(xs);
    assertTrue(helper.sorted(ys));
    helper.postProcess(ys);
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
    final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
    System.out.println(statPack);
    final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
    final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
    final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
    final int swaps = (int) statPack.getStatistics(InstrumentedHelper.SWAPS).mean();
    final int copies = (int) statPack.getStatistics(InstrumentedHelper.COPIES).mean();
    final int worstCompares = round(2.0 * N * Math.log(N));
    System.out.println("compares: " + compares + ", worstCompares: " + worstCompares);
    assertTrue(compares <= worstCompares);
```

```java
@Test
public void testSortDetailed() throws Exception {
    int k = 7;
    int N = (int) Math.pow(2, k);
    // NOTE this depends on the cutoff value for quick sort.
    int levels = k - 2;
    final Config config = Config.setupConfig("true", "0", "1", "", "");
    final Helper<Integer> helper = HelperFactory.create("quick sort dual pivot", N, config);
    System.out.println(helper);
    Sort<Integer> s = new QuickSort_DualPivot<>(helper);
    s.init(N);
    final Integer[] xs = helper.random(Integer.class, r -> r.nextInt(10000));
    assertEquals(Integer.valueOf(1360), xs[0]);
    helper.preProcess(xs);
    Integer[] ys = s.sort(xs);
    assertTrue(helper.sorted(ys));
    helper.postProcess(ys);
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
    final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
    System.out.println(statPack);
    final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
    final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
    final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
    final int swaps = (int) statPack.getStatistics(InstrumentedHelper.SWAPS).mean();
    final int copies = (int) statPack.getStatistics(InstrumentedHelper.COPIES).mean();
    final int worstCompares = round(2.0 * N * Math.log(N));
    System.out.println("compares: " + compares + ", worstCompares: " + worstCompares);
    assertTrue(compares <= worstCompares);
```

Heap Sort Test

Benchmark



Console