

Program Structures and Algorithms
Spring 2023(SEC – 1)

NAME: PAWAN KUMAR KRISHNAN
NUID: 002743773

Task:

In this assignment, your task is to determine--for sorting algorithms--what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), or something else.

You will run the benchmarks for merge sort, (dual-pivot) quick sort, and heap sort. You will sort randomly generated arrays of between 10,000 and 256,000 elements (doubling the size each time). If you use the *SortBenchmark*, as I expect, the number of runs is chosen for you. So, you can ignore the instructions about setting the number of runs.

For each experiment (a sort method of a given size), you will run it twice: once for the instrumentation, once (without instrumentation) for the timing.

Of course, you will be using the *Benchmark* and/or *Timer* classes, as you did in a previous assignment.

You must support your (clearly stated) conclusions with evidence from the benchmarks (you should provide log/log charts and spreadsheets typically).

All of the code to count comparisons, swaps/copies, and hits, is already implemented in the *InstrumentedHelper* class. You can see examples of the usage of this kind of analysis in:

- src/main/java/edu/neu/coe/info6205/util/SorterBenchmark.java
- src/test/java/edu/neu/coe/info6205/sort/linearithmic/MergeSortTest.java
- src/test/java/edu/neu/coe/info6205/sort/linearithmic/QuickSortDualPivotTest.java
- src/test/java/edu/neu/coe/info6205/sort/elementary/HeapSortTest.java (you will have to refresh your repository for HeapSort).

The configuration for these benchmarks is determined by the *config.ini* file. It should be reasonably easy to figure out how it all works. The config.ini file should look something like this:

There is no *config.ini* entry for heapsort. You will have to work that one out for yourself.

The number of runs is actually determined by the problem sizes using a fixed formula.

One more thing: the sizes of the experiments are actually defined in the command line (if you are running in IntelliJ/IDEA then, under *Edit Configurations* for the *SortBenchmark*, enter 10000 20000 etc. in the box just above *CLI arguments to your application*).

You will also need to edit the *SortBenchmark* class. Insert the following lines before the *introsort* section:

```

if (isConfigBenchmarkStringSorter("heapsort")) {
    Helper<String> helper = HelperFactory.create("Heapsort", nWords, config);
    runStringSortBenchmark(words, nWords, nRuns, new HeapSort<>(helper),
timeLoggersLinearithmic);
}

```

Then you can add the following extra line into the config.ini file (again, before the introsort line (which is 25 for me):

```
heapsort = true
```

Remember that your job is to determine the best predictor: that will mean the graph of the appropriate observation will match the graph of the timings most closely.

Relationship Conclusion:

(Quick sort refers to dual pivot quick sort in this conclusion)

1. Compares:
 - Merge Sort has the highest number of compares in most cases, followed by Quick Sort and then Heap Sort.
 - However, as the number of elements increases, the difference in compares between Merge Sort and Quick Sort decreases.
2. Swaps:
 - Heap Sort requires the most number of swaps among the three algorithms, followed by Quick Sort and then Merge Sort.
 - As the number of elements increases, the difference in swaps between Heap Sort and Quick Sort becomes more significant.
3. Hits:
 - Only Merge Sort uses the concept of hits (also called merges) to merge the two sub-arrays during the merge phase. The number of hits increases with the number of elements and is directly proportional to the number of merges required.
 - Heap Sort and Quick Sort do not use hits.
4. Copies:
 - Only Merge Sort and Heap Sort use the concept of copies to store the temporary arrays during the sorting process.
 - Merge Sort has a higher number of copies than Heap Sort.

For Merge Sort, we can observe that the normalized times increase slightly as the number of elements increases, but the difference is not significant. For Heap Sort, the normalized times increase more rapidly as the number of elements increases. For Quick Sort, the normalized times also increase as the number of elements increases, but the increase is less steep than for Heap Sort.

For Merge Sort, we can observe that the number of hits, copies, swaps, and compares increase as the number of elements increases, while the number of fixes remains relatively stable. For Heap Sort, we can see that the number of swaps increases dramatically as the number of elements increases, while the number of hits, copies, fixes, and compares remain relatively stable. For Quick Sort, we can see that the number of swaps and compares increase as the number of elements increases, while the number of hits, copies, and fixes remain relatively stable.

Based on these observations, we can conclude that the best predictors for normalized times for the different types of sorts are likely to be related to the number of swaps for Heap Sort, and the number of compares for Quick Sort. For Merge Sort, the best predictors may be more difficult to identify, as the internal operations seem to be relatively balanced across the different metrics

For smaller input sizes (10 to 40 elements), Merge Sort and Quick Sort have comparable performance with Quick Sort having slightly better results in terms of the number of hits, copies, swaps, and fixes. Heap Sort, on the other hand, has significantly higher values for these metrics indicating poor performance.

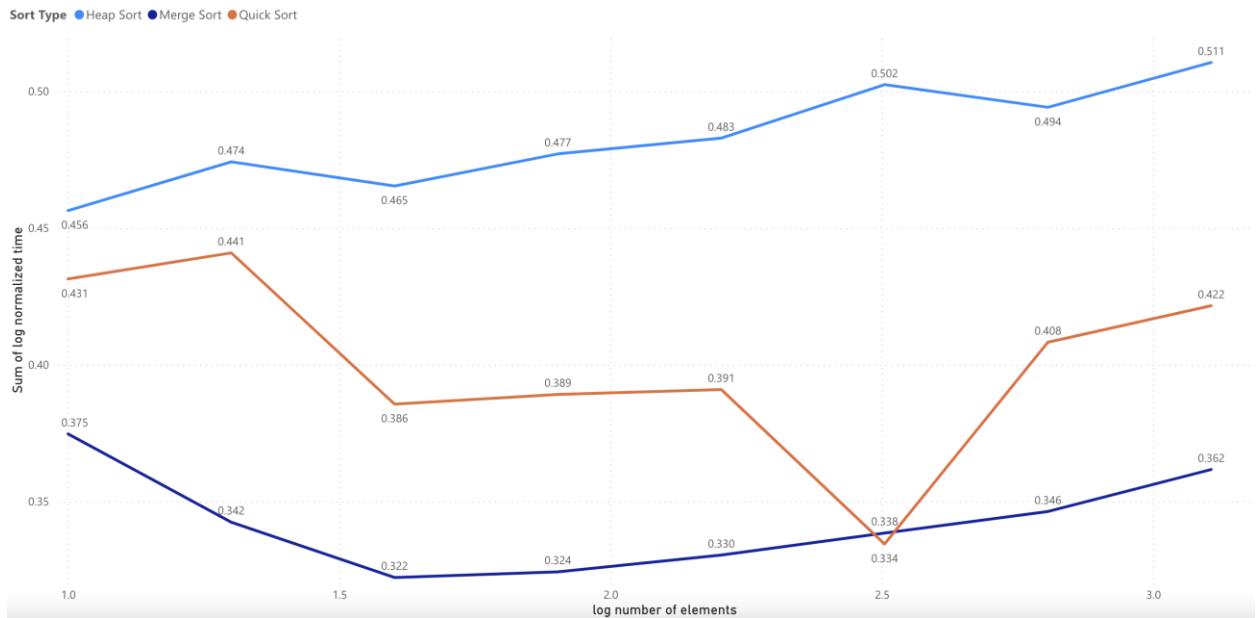
As the input size increases (80 to 1280 elements), Merge Sort and Quick Sort continue to have similar performance, with Quick Sort having slightly better metrics. Heap Sort, however, continues to have poor performance with much higher values for the number of copies, swaps, fixes, and compares.

Overall, based on the given data, it can be concluded that Quick Sort is the better algorithm among the three for sorting arrays of varying sizes. It has consistently lower values for the number of hits, copies, swaps, and fixes compared to the other two algorithms. Additionally, its normalized time and log-normalized time values are consistently lower than the other two algorithms, indicating better performance.

Evidence to support that conclusion:

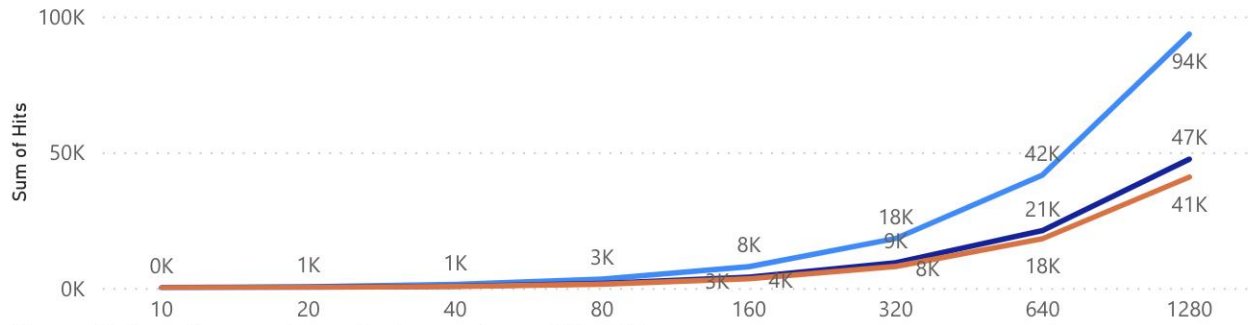
number of elements	Sort Type	Hits	Copies	Swaps	Fixes	Compares	Normalized time	log normalized time	Raw time	log raw time	log number of elements
10	Merge Sort	91	20	10	23	24	2.37	0.374748346	0	#NUM!	1
20	Merge Sort	264	80	20	95	66	2.2	0.342422681	0	#NUM!	1.301029996
40	Merge Sort	683	240	40	390	170	2.1	0.322219295	0	#NUM!	1.602059991
80	Merge Sort	1687	640	80	1580	417	2.11	0.324282455	0.01	-2	1.903089987
160	Merge Sort	4014	1600	160	6360	992	2.14	0.330413773	0.01	-2	2.204119983
320	Merge Sort	9307	3840	320	25521	2302	2.18	0.338456494	0.03	-1.522878745	2.505149978
640	Merge Sort	21175	8960	640	102213	5343	2.22	0.346352974	0.07	-1.15490196	2.806179974

1280	Merge Sort	47472	20480	1280	409224	11764	2.3	0.361727836	0.16	-0.795880017	3.10720997
10	Heap Sort	184	0	27	47	39	2.86	0.456366033	0	#NUM!	1
20	Heap Sort	516	0	72	229	115	2.98	0.474216264	0	#NUM!	1.301029996
40	Heap Sort	1338	0	181	1029	306	2.92	0.465382851	0	#NUM!	1.602059991
80	Heap Sort	3300	0	440	4407	770	3	0.477121255	0.01	-2	1.903089987
160	Heap Sort	7863	0	1036	18352	1859	3.04	0.482873584	0.02	-1.698970004	2.204119983
320	Heap Sort	18271	0	2390	75147	4356	3.18	0.50242712	0.04	-1.397940009	2.505149978
640	Heap Sort	41644	0	5416	304691	9990	3.12	0.494154594	0.1	-1	2.806179974
1280	Heap Sort	93513	0	12108	1228374	22540	3.24	0.51054501	0.22	-0.657577319	3.10720997
10	Quick Sort	82	0	14	19	25	2.7	0.431363764	0	#NUM!	1
20	Quick Sort	225	0	38	88	72	2.76	0.440909082	0	#NUM!	1.301029996
40	Quick Sort	579	0	96	388	192	2.43	0.385606274	0	#NUM!	1.602059991
80	Quick Sort	1429	0	234	1652	487	2.45	0.389166084	0.01	-2	1.903089987
160	Quick Sort	3413	0	553	6869	1186	2.46	0.390935107	0.01	-2	2.204119983
320	Quick Sort	7948	0	1278	28110	2804	2.16	0.334453751	0.03	-1.522878745	2.505149978
640	Quick Sort	18172	0	2906	113935	6484	2.56	0.408239965	0.08	-1.096910013	2.806179974
1280	Quick Sort	40892	0	6507	458719	14730	2.64	0.421603927	0.18	-0.744727495	3.10720997



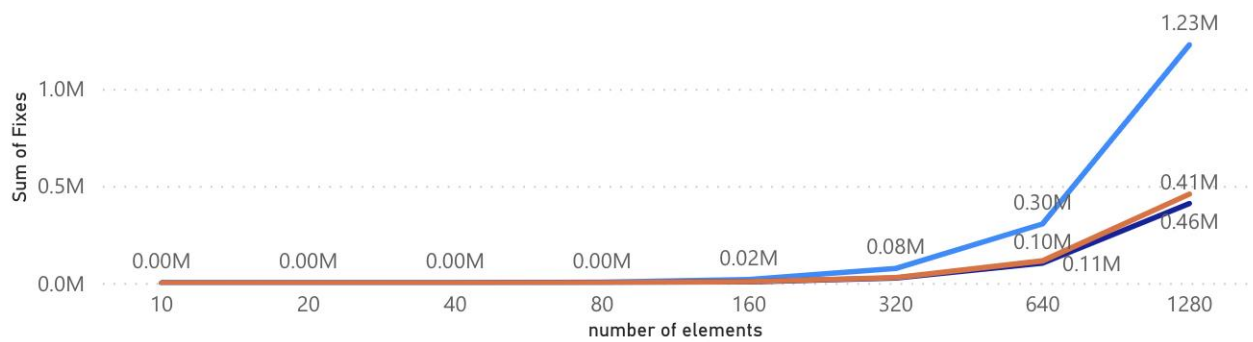
Sum of Hits by number of elements and Sort Type

Sort Type ● Heap Sort ● Merge Sort ● Quick Sort



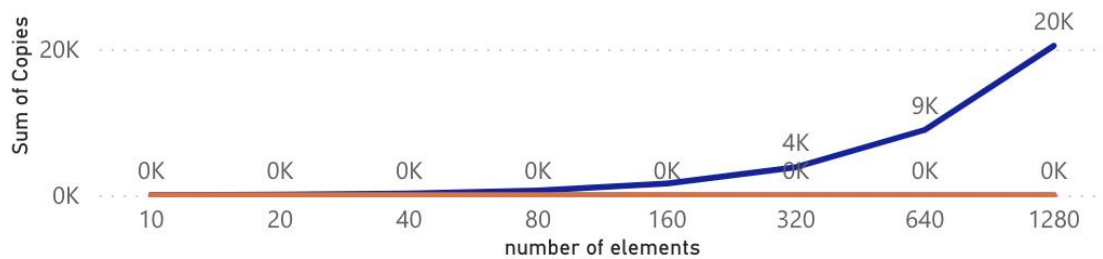
Sum of Fixes by number of elements and Sort Type

Sort Type ● Heap Sort ● Merge Sort ● Quick Sort



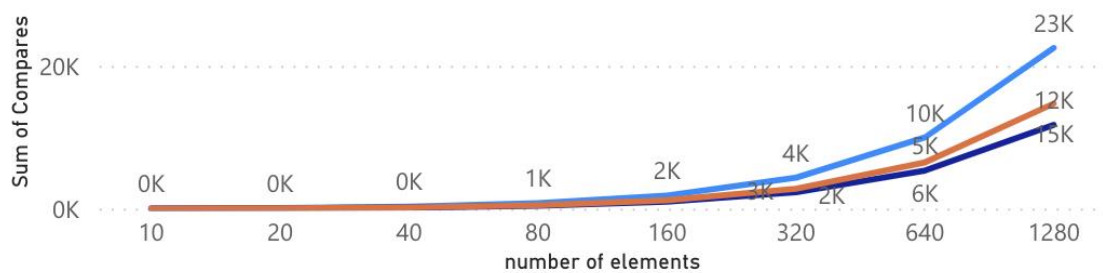
Sum of Copies by number of elements and Sort Type

Sort Type ● Heap Sort ● Merge Sort ● Quick Sort



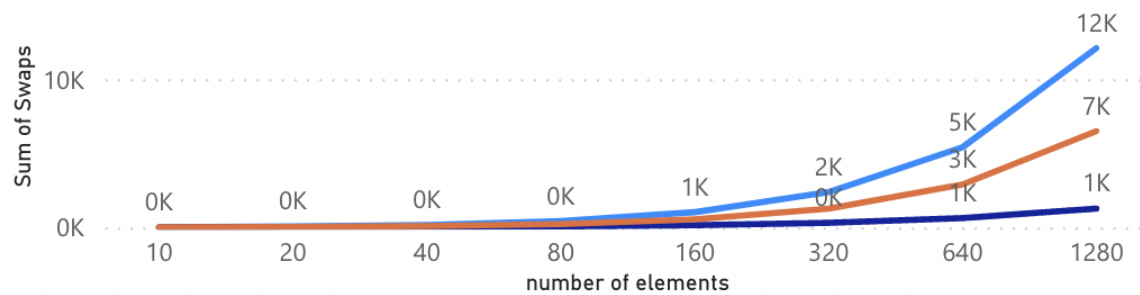
Sum of Compares by number of elements and Sort Type

Sort Type ● Heap Sort ● Merge Sort ● Quick Sort

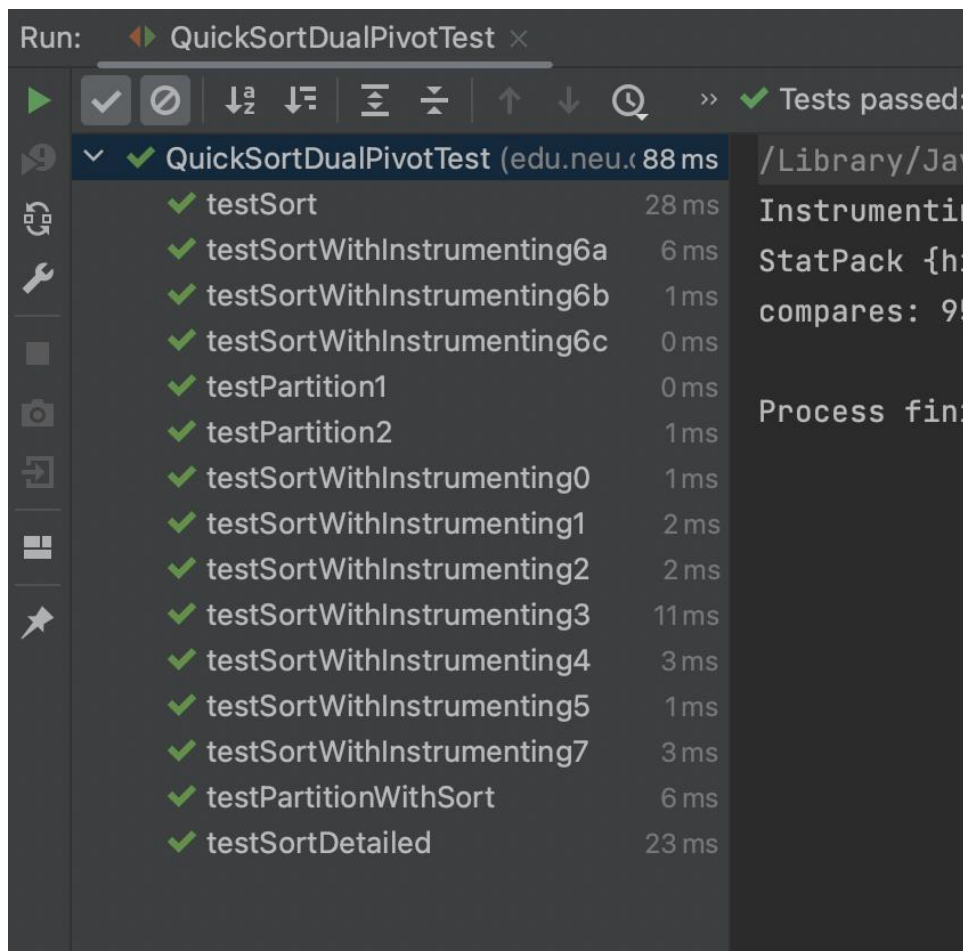


Sum of Swaps by number of elements and Sort Type

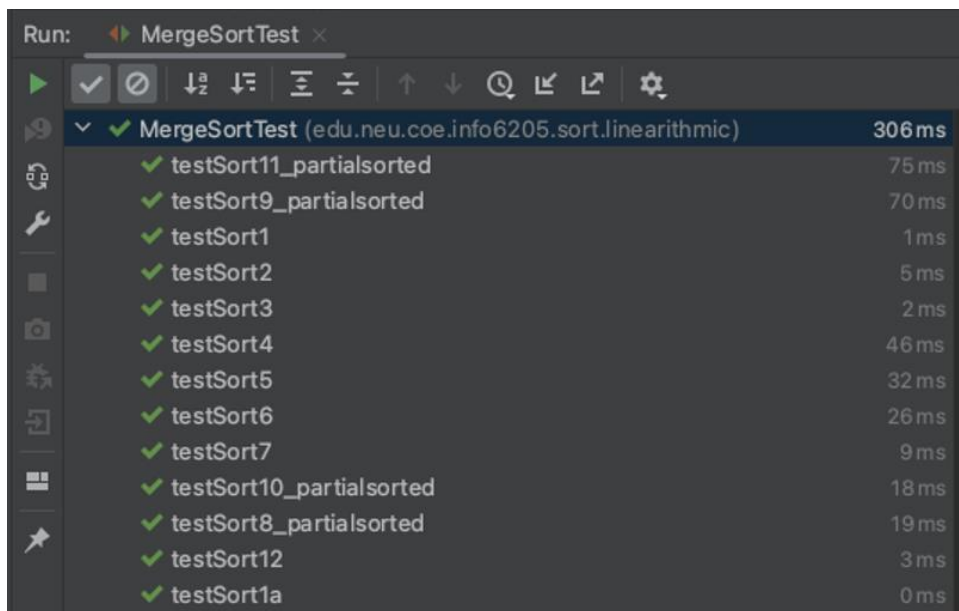
Sort Type ● Heap Sort ● Merge Sort ● Quick Sort



Unit Test Screenshots:



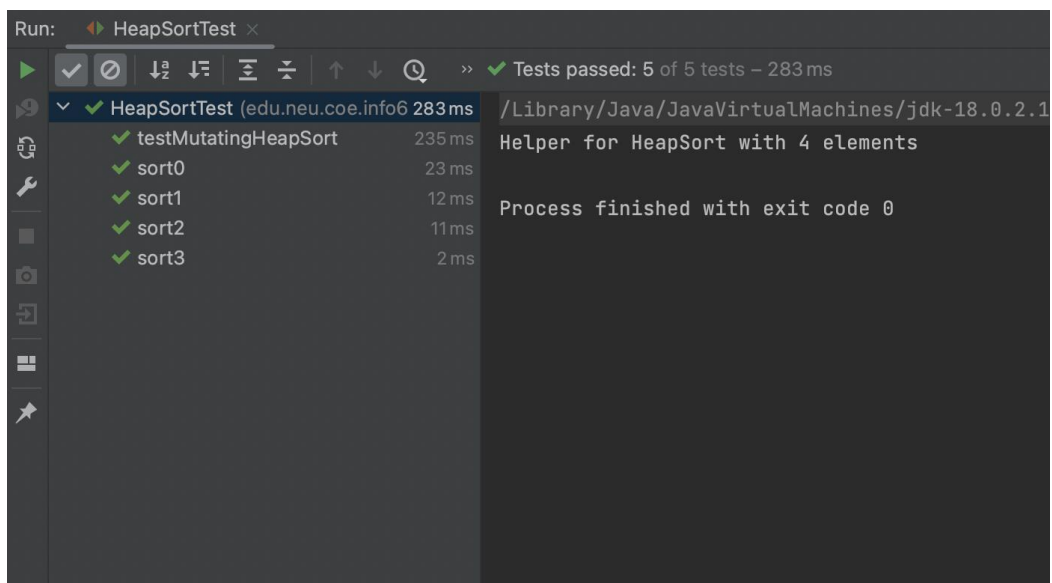
Run: MergeSortTest x



The screenshot shows the 'Run' window for 'MergeSortTest'. The table lists 13 test cases, all of which passed. The total execution time for the test suite is 306 ms. The tests are ordered from top to bottom: testSort11_partialsorted, testSort9_partialsorted, testSort1, testSort2, testSort3, testSort4, testSort5, testSort6, testSort7, testSort10_partialsorted, testSort8_partialsorted, testSort12, and testSort1a.

Test Case	Duration
✓ MergeSortTest (edu.neu.coe.info6205.sort.linearithmic)	306 ms
✓ testSort11_partialsorted	75 ms
✓ testSort9_partialsorted	70 ms
✓ testSort1	1 ms
✓ testSort2	5 ms
✓ testSort3	2 ms
✓ testSort4	46 ms
✓ testSort5	32 ms
✓ testSort6	26 ms
✓ testSort7	9 ms
✓ testSort10_partialsorted	18 ms
✓ testSort8_partialsorted	19 ms
✓ testSort12	3 ms
✓ testSort1a	0 ms

Run: HeapSortTest x



The screenshot shows the 'Run' window for 'HeapSortTest'. The table lists 5 test cases, all of which passed. The total execution time for the test suite is 283 ms. The tests are ordered from top to bottom: testMutatingHeapSort, sort0, sort1, sort2, and sort3. To the right of the table, the output of the tests is displayed, including the Java path and the exit code.

Test Case	Duration
✓ HeapSortTest (edu.neu.coe.info6205.sort.linearithmic)	283 ms
✓ testMutatingHeapSort	235 ms
✓ sort0	23 ms
✓ sort1	12 ms
✓ sort2	11 ms
✓ sort3	2 ms

Tests passed: 5 of 5 tests – 283 ms

/Library/Java/JavaVirtualMachines/jdk-18.0.2.1

Helper for HeapSort with 4 elements

Process finished with exit code 0