Program Structures and Algorithms

Spring 2023(SEC – 8)

NAME: Natarajan Lekshmi Narayana Pillai

NUID: 002766033

Assignment 6

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:

```
[sortbenchmark]
version = 1.0.0 (sortbenchmark)

[helper]
instrument = true
seed = 0
cutoff =

[instrumenting]
# The options in this section apply only if instrument (in [helper]) is set to true.
swaps = true
compares = true
copies = true
fixes = false
hits = true
# This slows everything down a lot so keep this small (or zero)
inversions = 0

[benchmarkstringsorters]
words = 1000 # currently ignored
runs = 20 # currently ignored
mergesort = true
timsort = false
quicksort = false
introsort = false
insertionsort = false
bubblesort = false
quicksort3way = false
quicksortDualPivot = true
randomsort = false

[benchmarkdatesorters]
timsort = false
n = 100000

[mergesort]
insurance = false
nocopy = true

[shellsort]
n = 100000

[operationsbenchmark]
nlargest = 10000000
repetitions = 10
```

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), timeLoggersLin
earithmic);
}
```

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 using evidence:

Heap Sort

N	Time (Without Instrumentation)	Time (With Instrumentation)	Hits	Swaps	Copies	Compares
10000	3.43404	337.19465	967590.2	124206.9	0	235381.3
20000	6.15775	1445.33575	2094863	268364.8	0	510701.7
40000	12.65938	6540.14478	4509677	576721.5	0	1101395.6
80000	29.31918	15120.32781	9660326	1233611	0	2362941.6
160000	64.34579	54840.79588	2.06E+07	2627101	0	5045953.4

Quick Sort

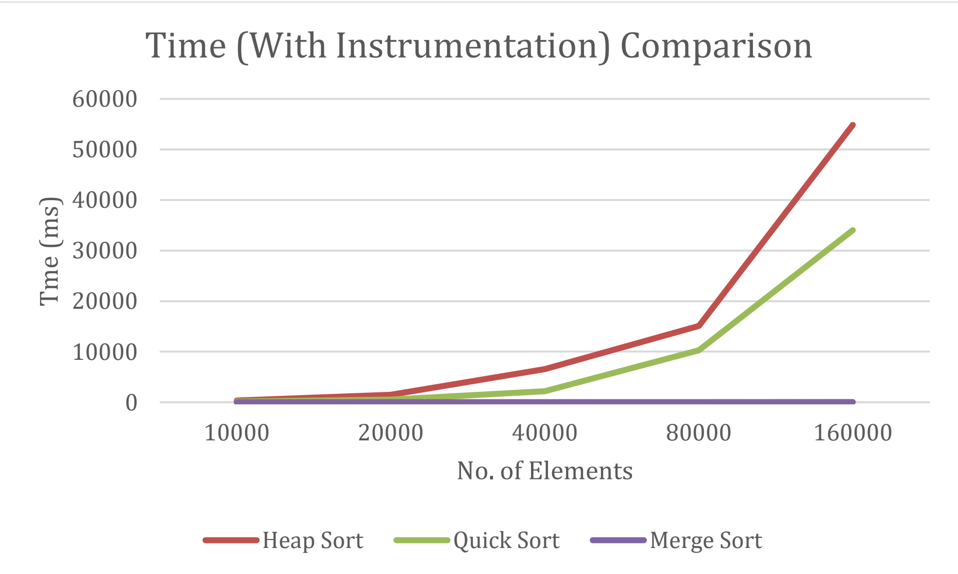
N	Time (Without Instrumentation)	Time (With Instrumentation)	Hits	Swaps	Copies	Compares
10000	11.62793	136.85341	410545.5	63211.4	0	161418.3
20000	6.43024	537.66496	881510.2	137246.1	0	339838.7
40000	10.71557	2167.36082	1940987	301192.9	0	751057.1
80000	21.2767	10268.29565	4150295	643995.9	0	1603625.5
160000	49.8276	34026.81186	8730320	1351509	0	3382965.3

Merge Sort

N	Time (Without Instrumentation)	Time (With Instrumentation)	Hits	Swaps	Copies	Compares
10000	6.41066	3.1032098	259158	0	9789.5	121494.3
20000	6.72007	3.9345603	558088.8	0	19522.2	263023.2
40000	13.09208	7.7190107	1196788	0	39197.1	566170.9
80000	23.98621	17.8479802	2552392	0	78097.9	1212049
160000	51.9062	37.9105399	5425243	0	156310.7	2584173.5

1. If we look at the **times** with instrumentation for the different sorting algorithms:

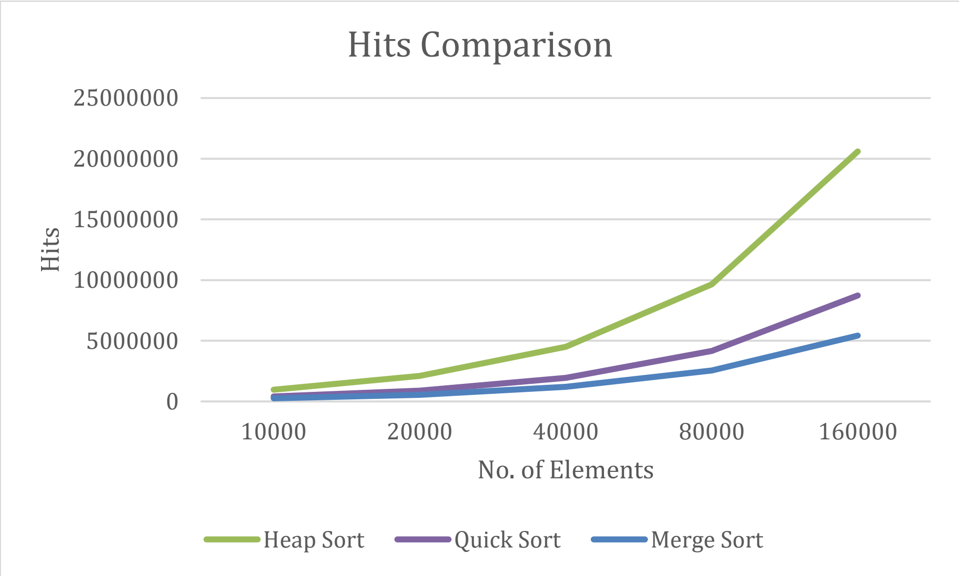
N	Heap Sort	Quick Sort	Merge Sort
10000	337.19465	136.85341	3.1032098
20000	1445.33575	537.66496	3.9345603
40000	6540.14478	2167.36082	7.7190107
80000	15120.3278	10268.2957	17.8479802
160000	54840.7959	34026.8119	37.9105399



We can see that as the number of elements increase, the run time of the algorithm also increases. As seen on the graph, the run time for heap sort is the highest and the run time for merge sort is the lowest.

2. If we look at the number of **hits** for the different sorting algorithms:

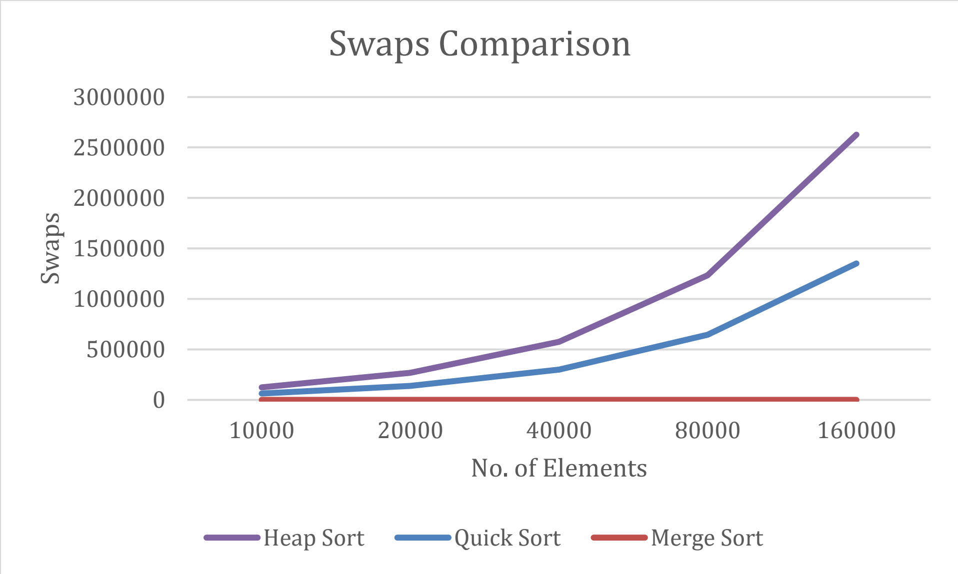
N	Heap Sort	Quick Sort	Merge Sort
10000	967590.2	410545.5	259158
20000	2094862.6	881510.2	558088.8
40000	4509677.2	1940986.8	1196788.4
80000	9660326.4	4150295.2	2552391.6
160000	2.06E+07	8730320.2	5425242.8



We can see that as the as the number of elements increase, the number of hits for each algorithm increases. As seen on the graph, the number of hits for heap sort is the highest and the number of hits for merge sort is the lowest. We can also see that the number of hits for quick sort is just less than half of heap sort.

3. If we look at the number of **swaps** for the different sorting algorithms:

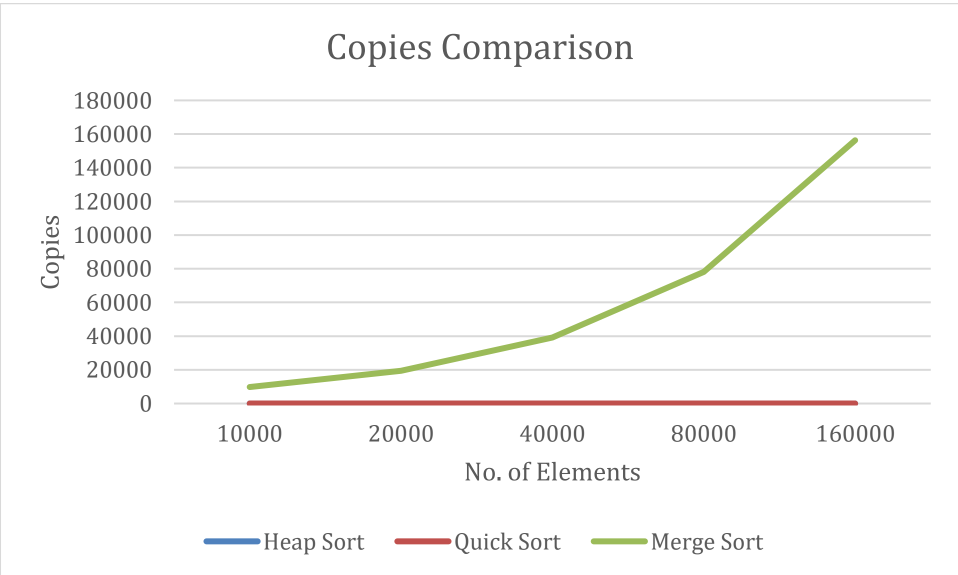
N	Heap Sort	Quick Sort	Merge Sort
10000	124206.9	63211.4	0
20000	268364.8	137246.1	0
40000	576721.5	301192.9	0
80000	1233610.8	643995.9	0
160000	2627100.5	1351508.5	0



We can see that as the number of elements increase, the number of swaps also increases. As seen on the graph, the number of swaps for heap sort is the highest and the number of swaps for quick sort is the lowest. This is not the case for merge sort which does not do any swaps. Therefore, the values of merge sort for number of swaps is 0.

4. If we look at the number of **copies** for the different sorting algorithms:

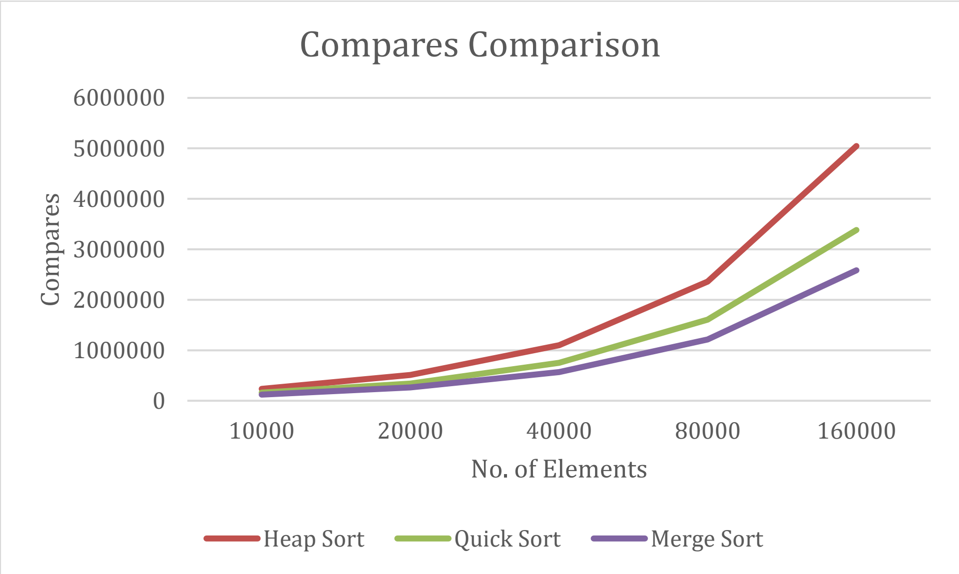
N	Heap Sort	Quick Sort	Merge Sort
10000	0	0	9789.5
20000	0	0	19522.2
40000	0	0	39197.1
80000	0	0	78097.9
160000	0	0	156310.7



We can see that as the number of elements increase, the number of copies increase for merge sort. Heap sort and quick sort do not have any copies. Therefore, the values of heap sort and quick sort for number of copies is 0.

5. If we look at the number of **compares** for the different sorting algorithms:

N	Heap Sort	Quick Sort	Merge Sort
10000	235381.3	161418.3	121494.3
20000	510701.7	339838.7	263023.2
40000	1101395.6	751057.1	566170.9
80000	2362941.6	1603625.5	1212049
160000	5045953.4	3382965.3	2584173.5



We can see that as the number of elements increase, the number of compares for each algorithm increases. As seen on the graph, the number of compares for heap sort is the highest, and the number of compares for merge sort is the lowest.

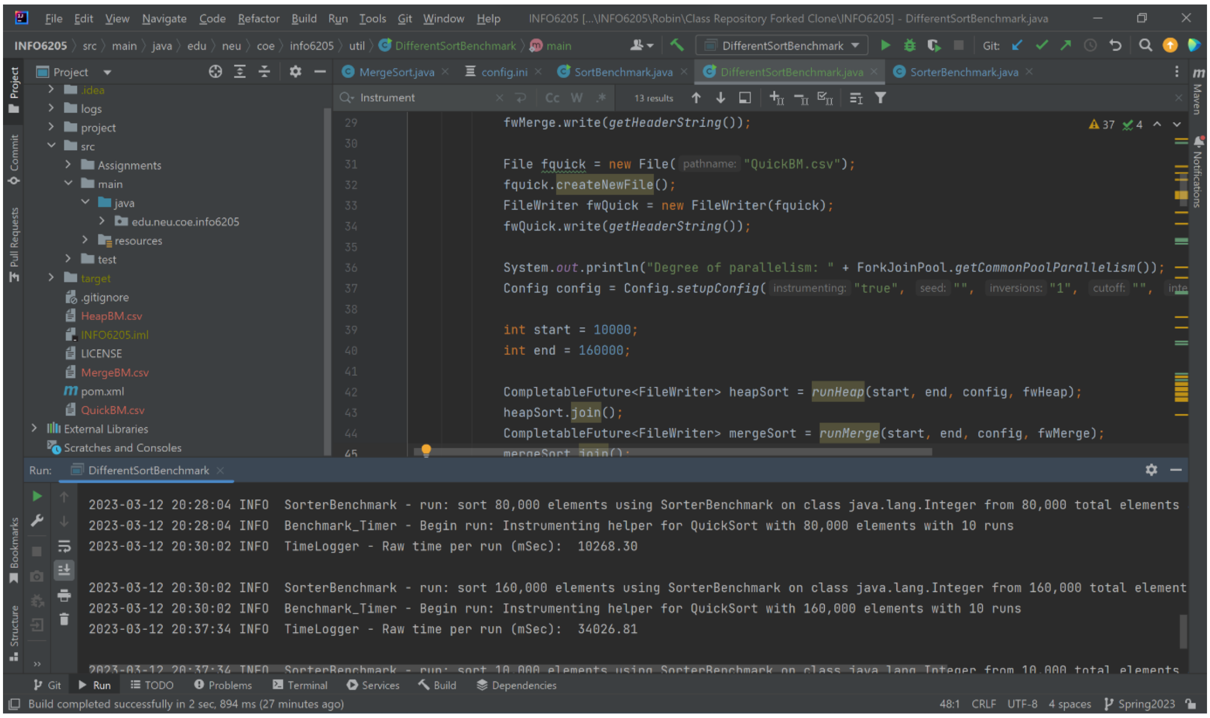
Conclusion:

The best predictor of total execution time among heap sort, quick sort, and merge sort would depend on several factors such as the size of the input, the distribution of the data, and the implementation of the algorithm. In general, an algorithm's total execution time can be significantly impacted by the quantity of comparisons, swaps/copies, and hits (array accesses). Certain algorithms, might be more susceptible to one factor than others.

1. Due to the low number of swaps and copies in heap sort, the number of comparisons and array accesses can serve as reliable indicators of the execution time.
2. Since quick sort can have a large number of swaps, the number of swaps may be a better predictor of total execution time than the number of comparisons or array accesses.
3. Since merge sort has a relatively small number of swaps/copies, the number of comparisons and array accesses can be good predictors of the total execution time.

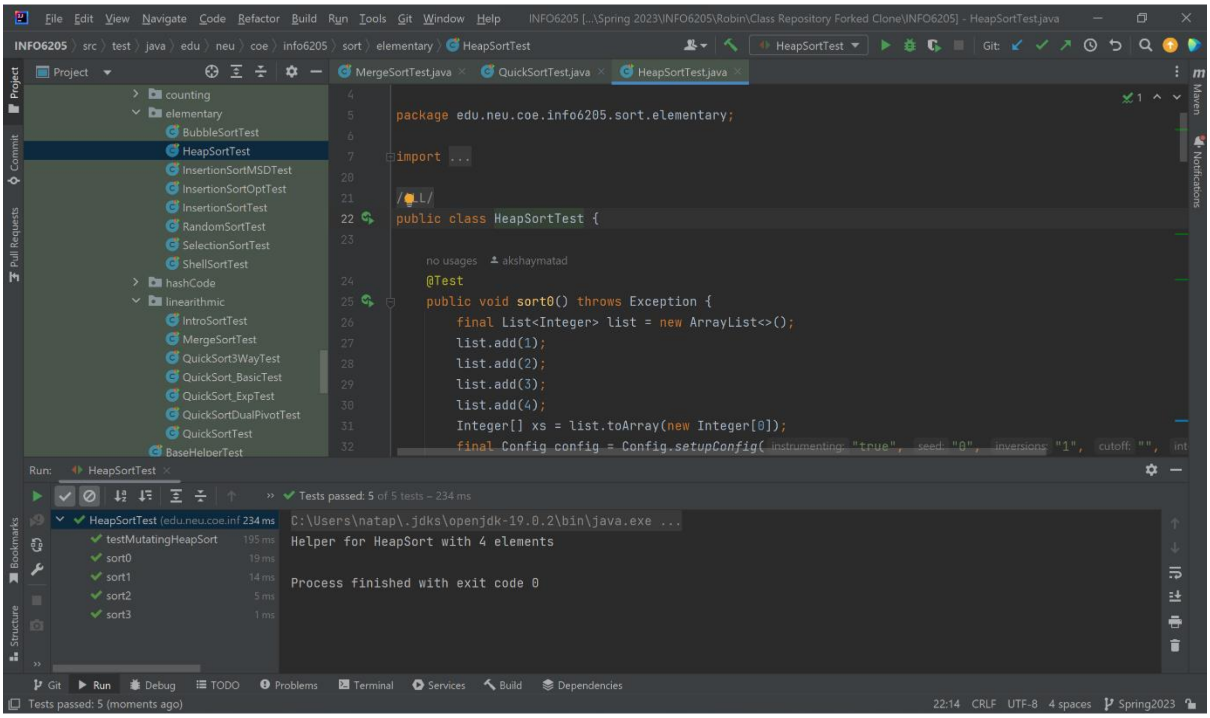
In conclusion, the best predictor of total execution time among heap sort, quick sort, and merge sort would depend on various factors. While the amount of swaps may be a stronger predictor for quick sort, the number of comparisons and hits (array accesses) may be effective predictors for heap sort and merge sort. Though since hits happen even with swaps, comparisons and copies, this suggests that comparing hits will be the easiest way to see which algorithm is the most effective, where more number of hits imply a worse algorithm.

Output Screenshot:

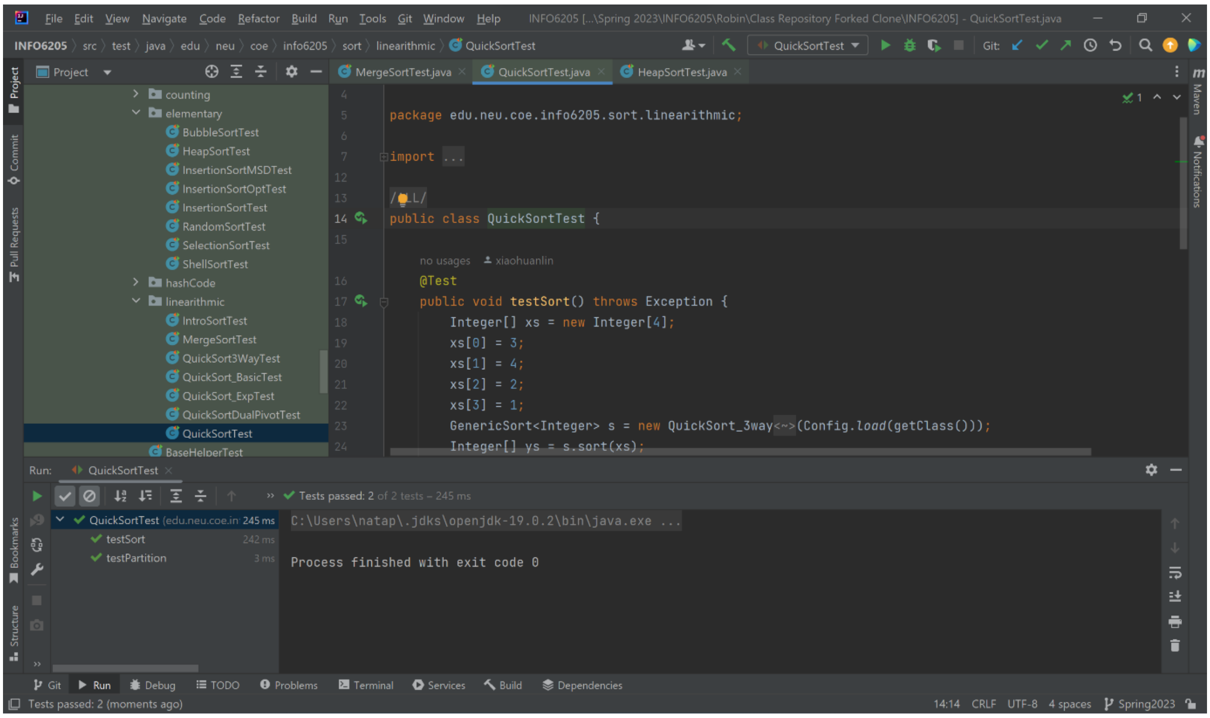


Unit Test Screenshot:

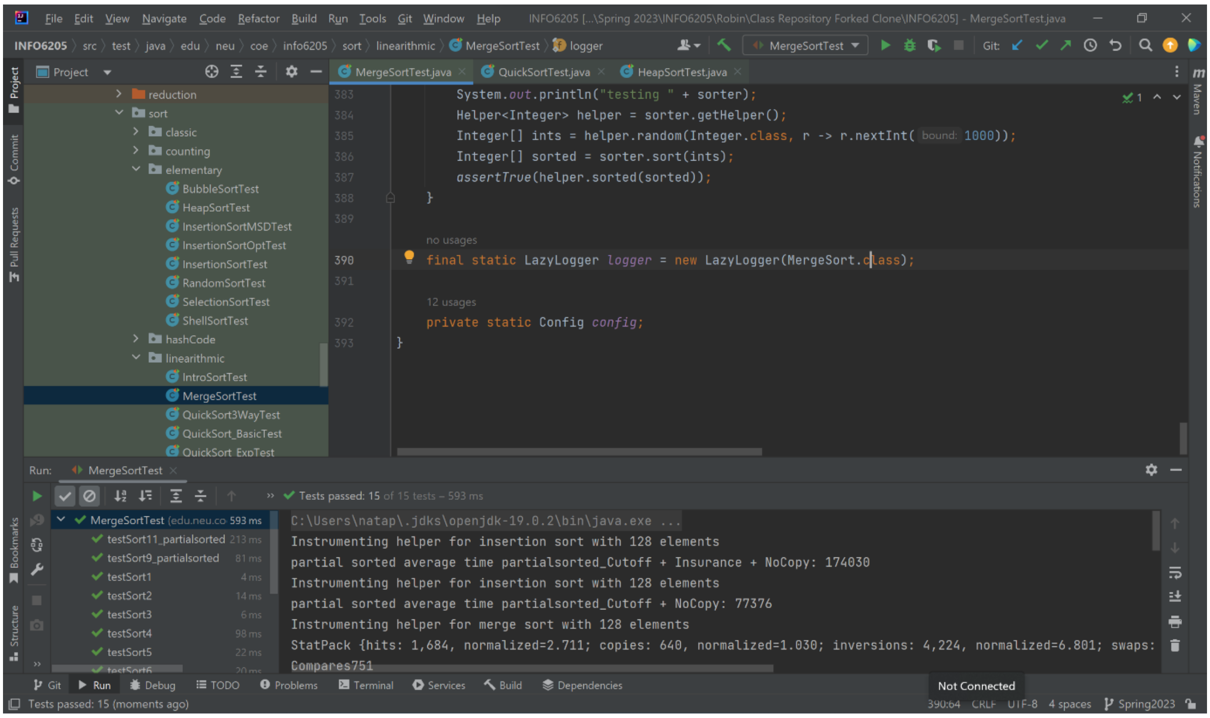
Heap Sort



Quick Sort



Merge Sort



System.out.println("testing " + sorter);
Helper<Integer> helper = sorter.getHelper();
Integer[] ints = helper.random(Integer.class, r -> r.nextInt(bound: 1000));
Integer[] sorted = sorter.sort(ints);
assertTrue(helper.sorted(sorted));
final static LazyLogger logger = new LazyLogger(MergeSort.class);
private static Config config;
Tests passed: 15 of 15 tests – 593 ms
C:\Users\natap\.jdks\openjdk-19.0.2\bin\java.exe ...
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + Insurance + NoCopy: 174030
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + NoCopy: 77376
Instrumenting helper for merge sort with 128 elements
StatPack {hits: 1,684, normalized=2.711; copies: 640, normalized=1.030; inversions: 4,224, normalized=6.801; swaps:
Compares751