Question 6:

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),
timeLoggersLinearithmic);
}
```

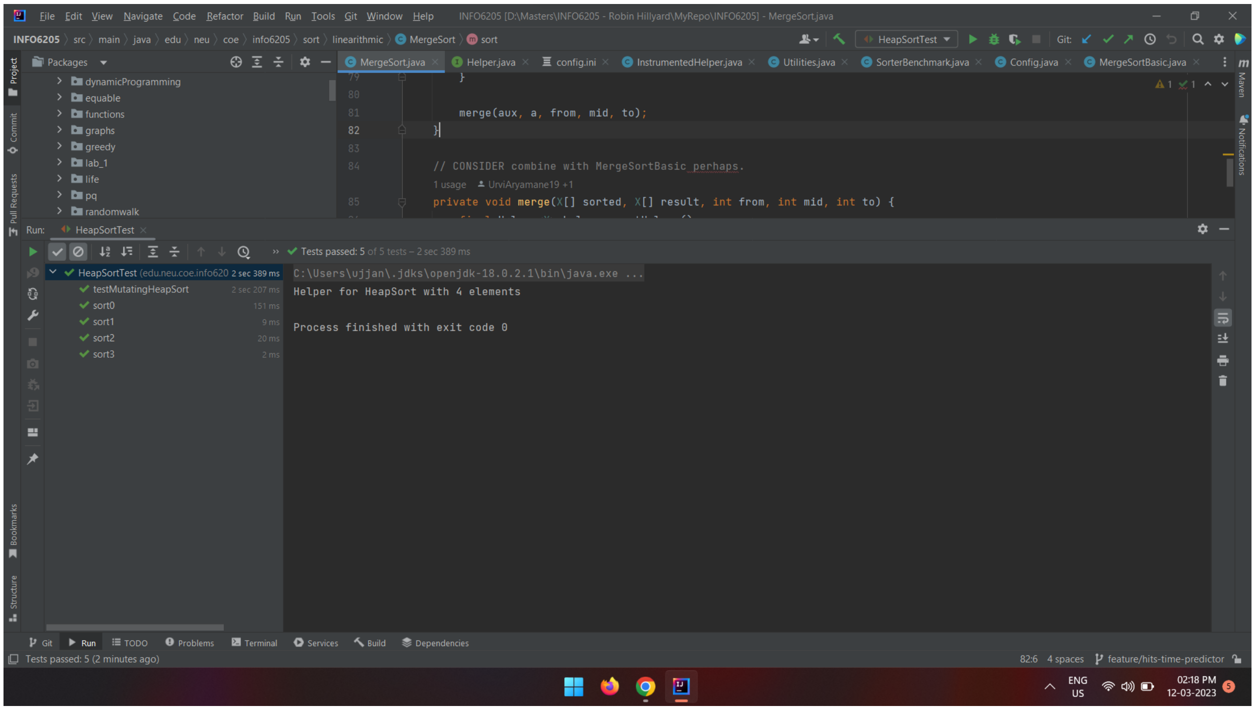
Then you can add the following extra line into the config.ini file (again, before the introsort line (which is 25 for me):

```
heapsort = true
```

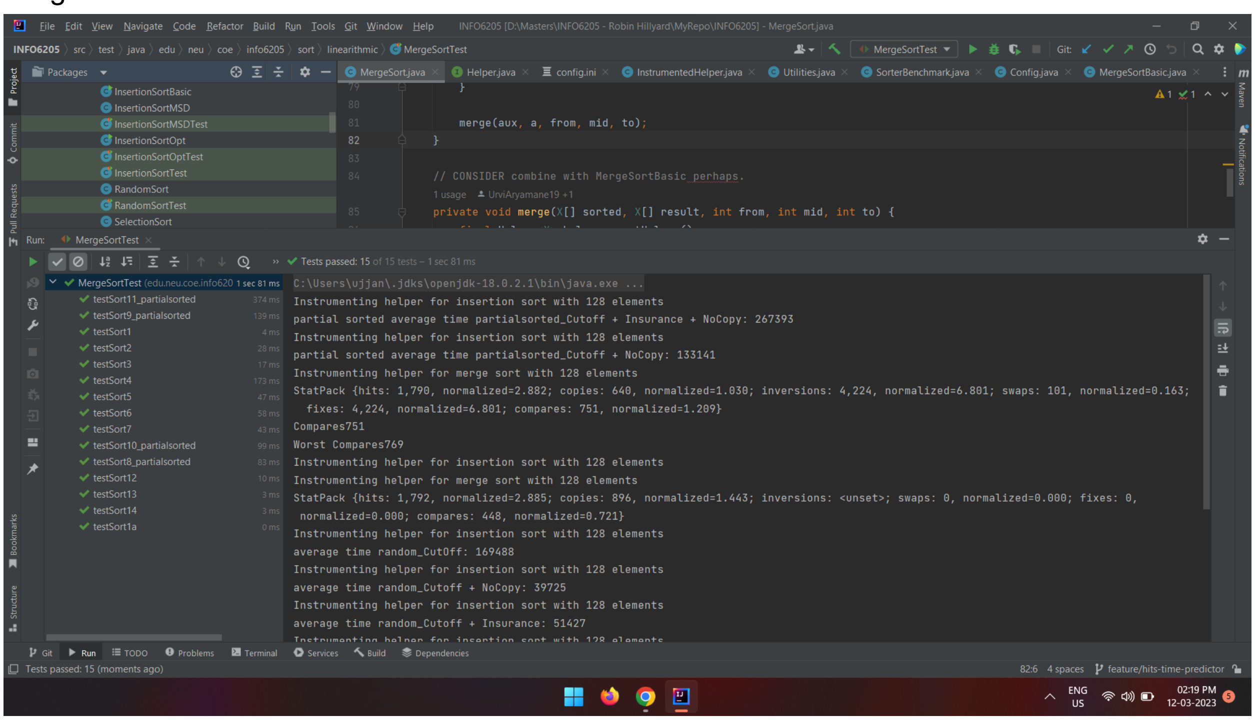
Remember that your job is to determine the best predictor: that will mean the graph of the appropriate observation will match the graph of the timings most closely.

Answer:

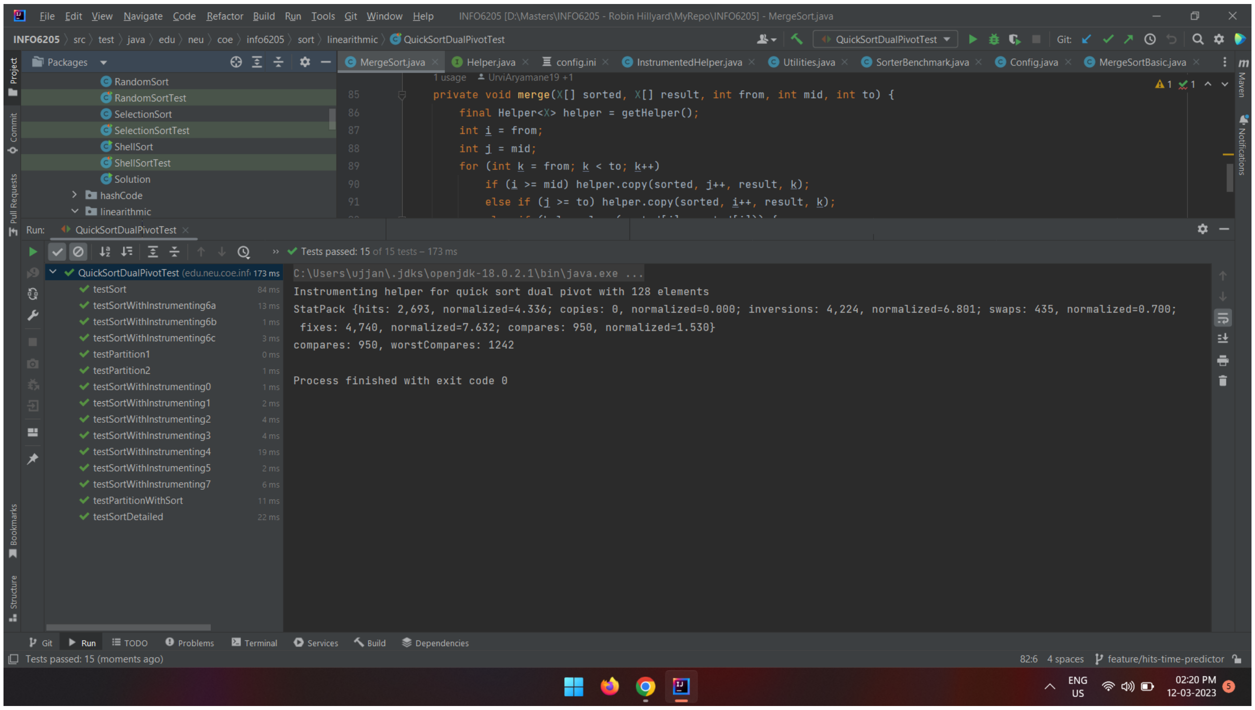
Heap sort test cases



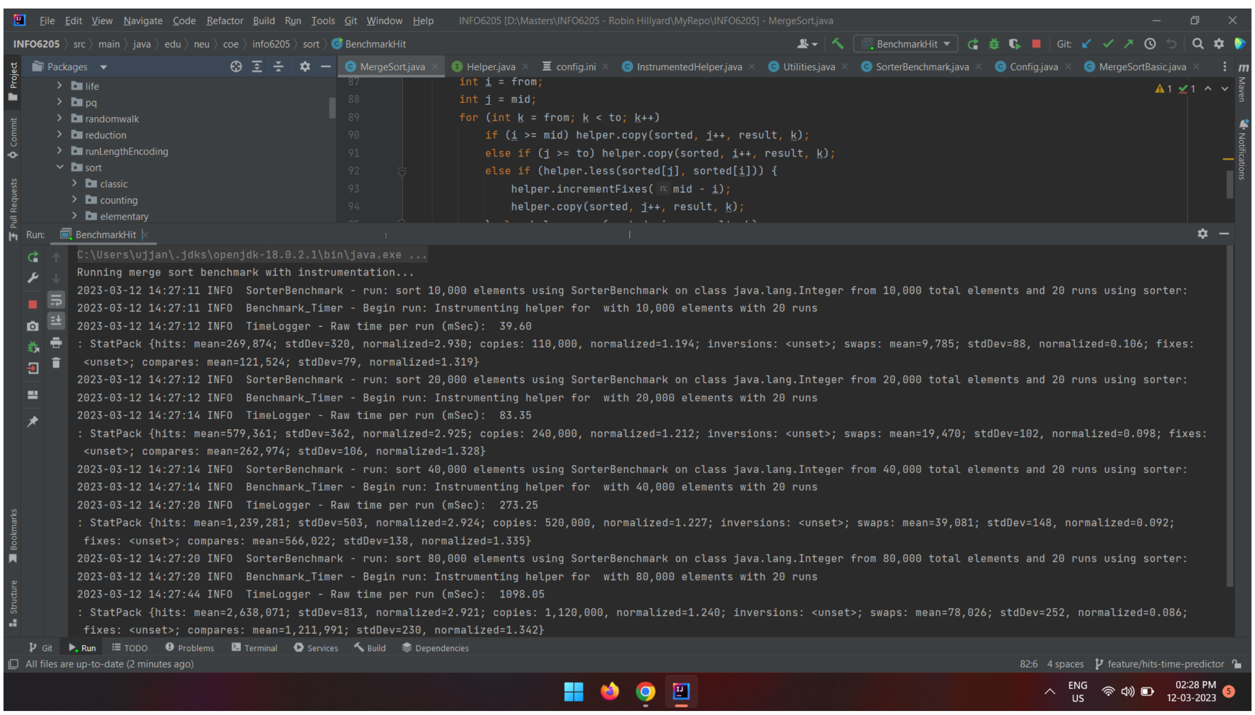
Merge sort test cases



Dual pivot quicksort



Running of the code



To identify the best comparison, first let us see the stats for each type of sorting algorithm.

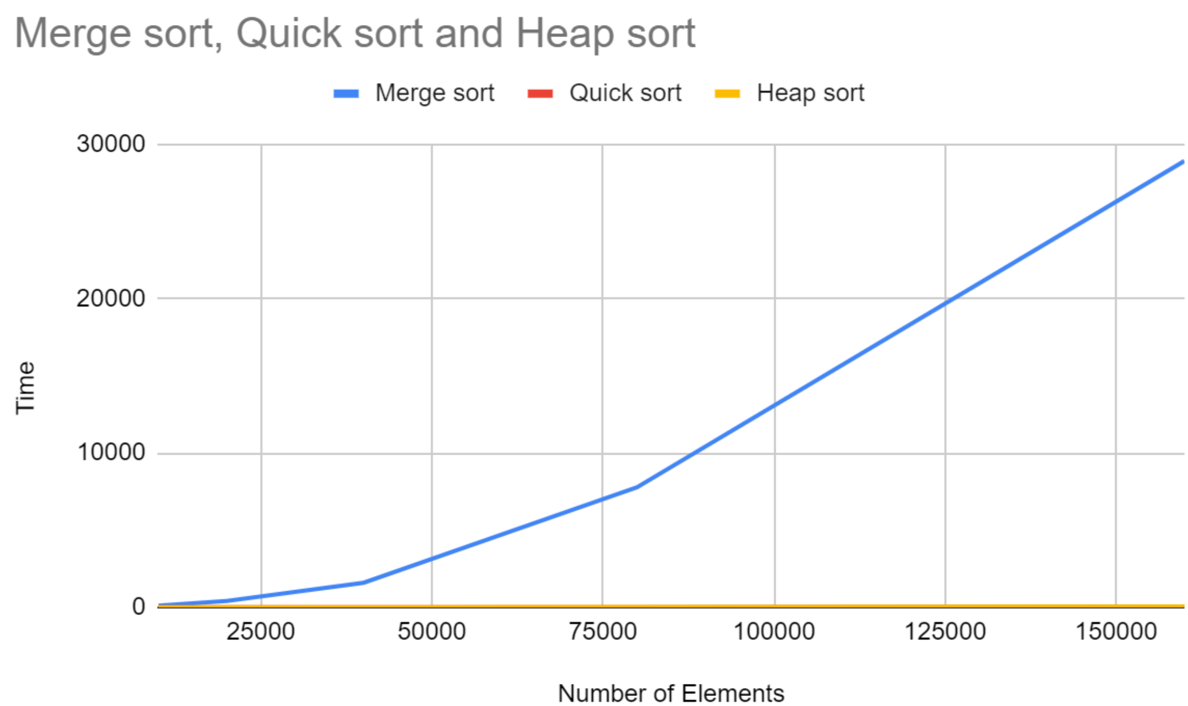
Merge sort						
Number of Elements	With instrumentation	Without instrumentation	Hits	Copies	Swaps	Compares
10000	111.5	105.05	267232	133616	0	120447
20000	409.85	388.05	574464	287232	0	260905
40000	1592.15	1411.3	1228928	614464	0	561766
80000	7777.75	5901.9	2617856	1308928	0	1203594
160000	28915.95	26572.5	5555712	2777856	0	2567220

Quick sort						
Number of Elements	With instrumentation	Without instrumentation	Hits	Copies	Swaps	Compares
10000	3.55	6.1	414739	0	64412	154349
20000	7.05	5.1	903910	0	139071	342099
40000	7.2	6.35	961524	0	305875	727059
80000	16.65	14.15	4199230	0	649695	1578296
160000	35.8	29.9	8970894	0	1386491	3380799

Heap sort						
Number of Elements	With instrumentation	Without instrumentation	Hits	Copies	Swaps	Compares
10000	2.55	5.55	967565	0	124208	235367
20000	4.7	8.4	2095047	0	268396	510731
40000	12.5	12.05	4509795	0	576738	1101421
80000	26.2	26.75	9660730	0	1233662	2363040
160000	59.3	58.8	20599695	0	2627076	5045695

Now let us compare all the run times.

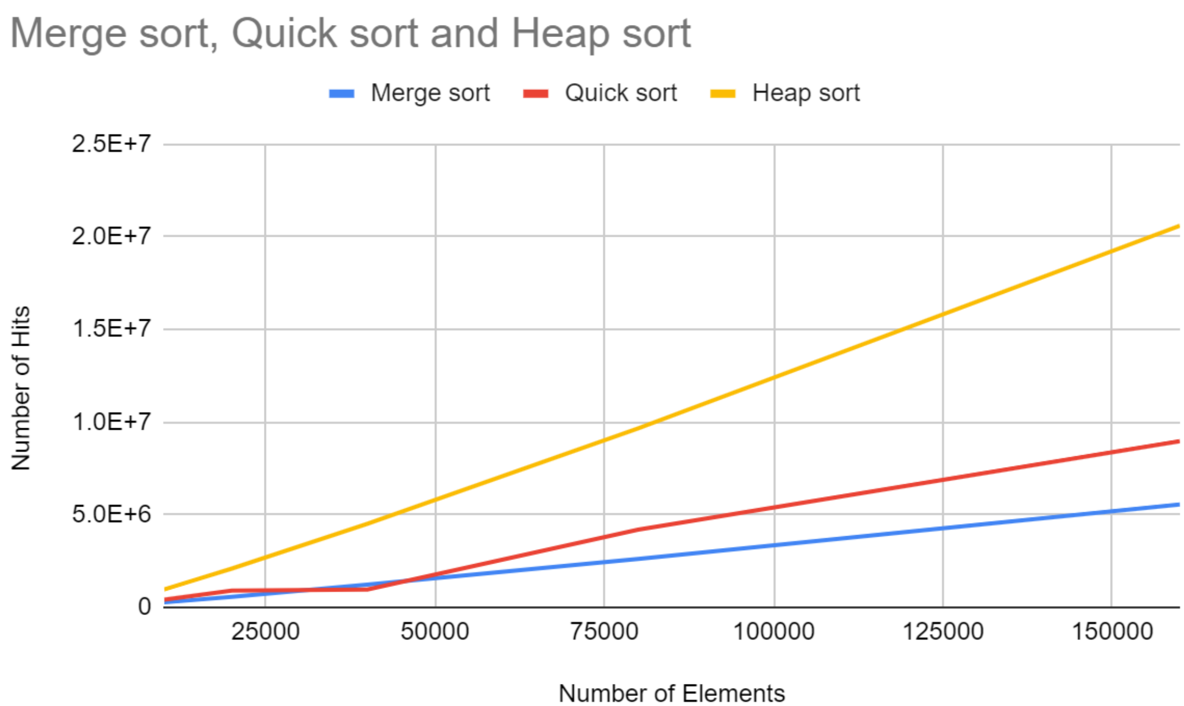
Time			
Number of Elements	Merge sort	Quick sort	Heap sort
10000	111.5	3.55	2.55
20000	409.85	7.05	4.7
40000	1592.15	7.2	12.5
80000	7777.75	16.65	26.2
160000	28915.95	35.8	59.3



We can see that the time increases for every algorithm as the number of elements increases. The effect is predominant for merge sort whose increase is the most for any size of elements under consideration.

Now let us compare all the hits.

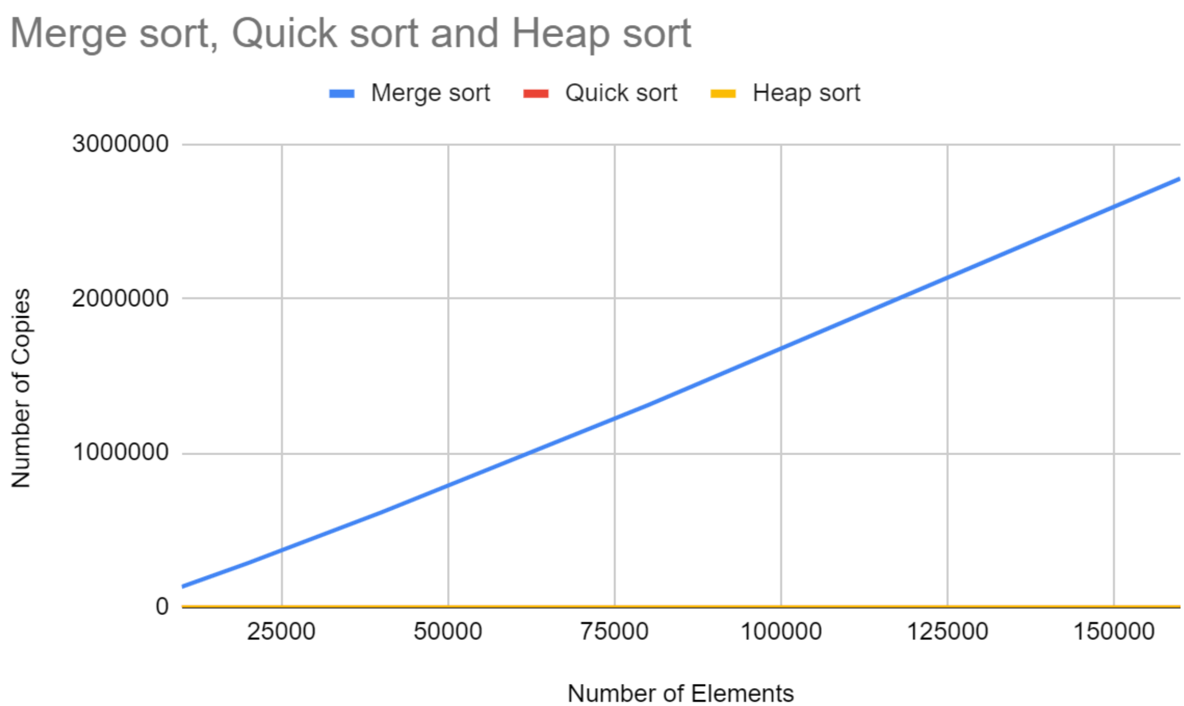
Hits			
Number of Elements	Merge sort	Quick sort	Heap sort
10000	267232	414739	967565
20000	574464	903910	2095047
40000	1228928	961524	4509795
80000	2617856	4199230	9660730
160000	5555712	8970894	20599695



We can see that the number of hits increases as the number of elements increase. The hits is hightest for heap sort. Quick sort seems to perform better compared to merge sort for low numbers but for higher numbers, merge sort seems to perform better.

Now let us compare all the copies.

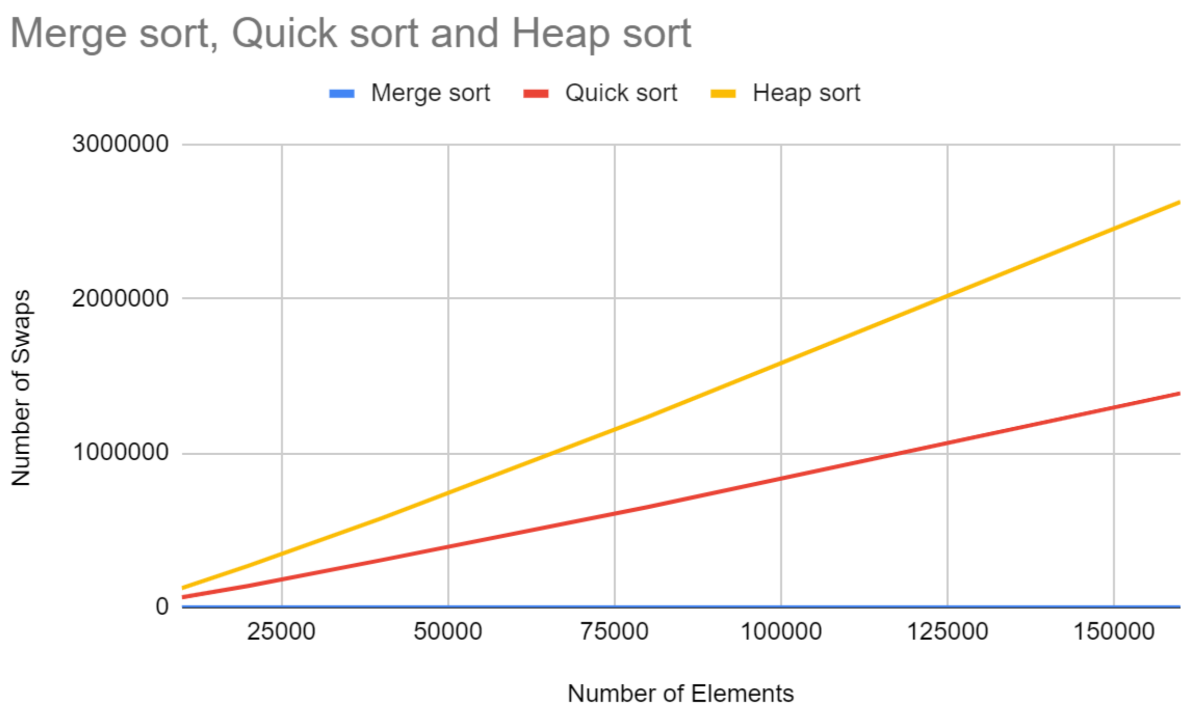
Copies			
Number of Elem	Merge sort	Quick sort	Heap sort
10000	133616	0	0
20000	287232	0	0
40000	614464	0	0
80000	1308928	0	0
160000	2777856	0	0



We can see that the number of copies increases for merge sort as the number of elements increase. Quick sort and heap sort use swaps so the number of copies is 0 for both of them.

Now let us compare all the swaps.

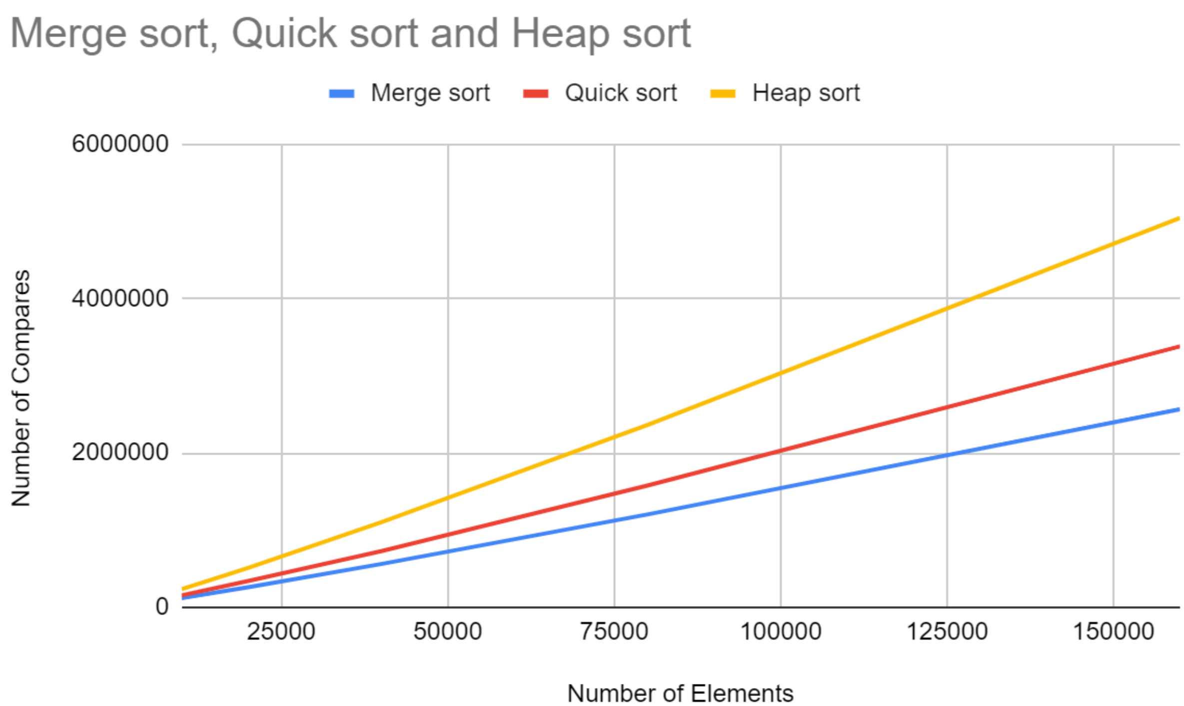
Swaps			
Number of Elem	Merge sort	Quick sort	Heap sort
10000	0	64412	124208
20000	0	139071	268396
40000	0	305875	576738
80000	0	649695	1233662
160000	0	1386491	2627076



We can see that the number of swaps increase linearly as the number of elements increase. Since merge sort does not have any swaps, its value is 0. The number of swaps is the highest for heap sort compared to quick sort.

Now let us compare all the compares.

Compares			
Number of Elem	Merge sort	Quick sort	Heap sort
10000	120447	154349	235367
20000	260905	342099	510731
40000	561766	727059	1101421
80000	1203594	1578296	2363040
160000	2567220	3380799	5045695



We can see that the number of compares is the highest for heap sort followed by quick sort and least for merge sort.

Scenario 1:
From the parameters being considered, compares, copies and swaps also involve array hits. So each time a compare, copy or a swap occurs, array hits occur. This indicates that hits would be a neutral way to determine which algorithm would generally be better. Larger the number of hits would indicate poorer performance of the algorithm, provided that the rest of the operations being performed in the algorithms being compared are exactly the same in terms of time taken. In this situation hits would be the best predictor of time.

Scenario 2:
If the operations being performed by each algorithm do not take the same time, then we will have to compare those operations being performed. In our case, we could split it into 3 parts that is swaps, copies and compares. The operation that takes a lesser amount of time and involves a lesser quantity of the parameter being measured would be a better predictor of the time taken for the completion of the algorithm. In general, swaps are more expensive than copies. This could be explained by the fact that copy involves just 2 hit to complete, example: a[2] = a[1]. Swap on the other hand takes 4 hits to complete, example:
tmp = a[1]
a[1] = a[2]
a[2] = tmp

So, we can say that copy is less expensive than swaps. Comparing comparisons and copy is not that straightforward. Depending on the hardware, comparison could be a more expensive operation than copy.

Based on these observations, we can say that the algorithm with the most amount of swaps has the worst time performance. The next parameter to take into consideration in case of ambiguity is copy followed by comparison. If none of these metrics are available, a general number of hits can be used to identify which would be better. The one with the highest number of hits would indicate the worst performance.

As such the general ranking of the algorithms would be Quick sort followed by Merge sort followed by Heap sort.