Assignment 6 - Program Structures and Algorithms Spring 2023(SEC - 1)

NAME: Paurush Batish **NUID: 002755631**

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")) {
```

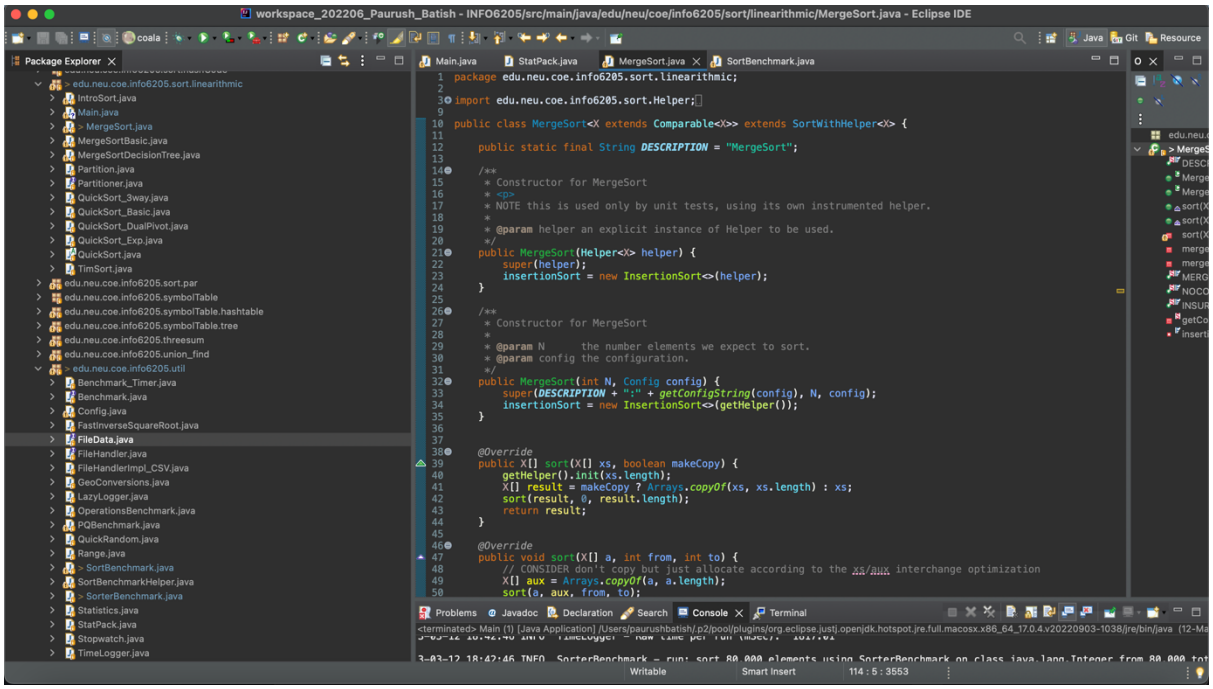
```
    Helper<String> helper = HelperFactory.create("Heapsort", nWords, config
);
    runStringSortBenchmark(words, nWords, nRuns, new HeapSort<>(helper), ti
meLoggersLinearithmic);
}
```

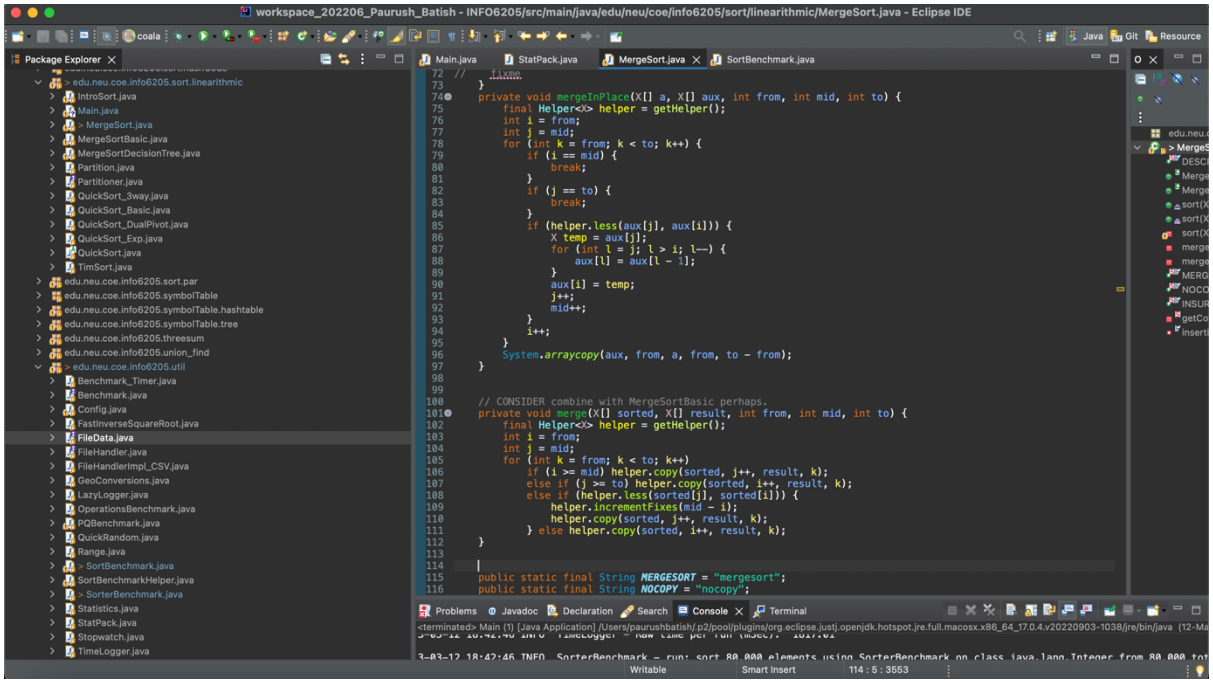
Then you can add the following extra line into the config.ini file (again, before the introsort line (which is 25 for me):

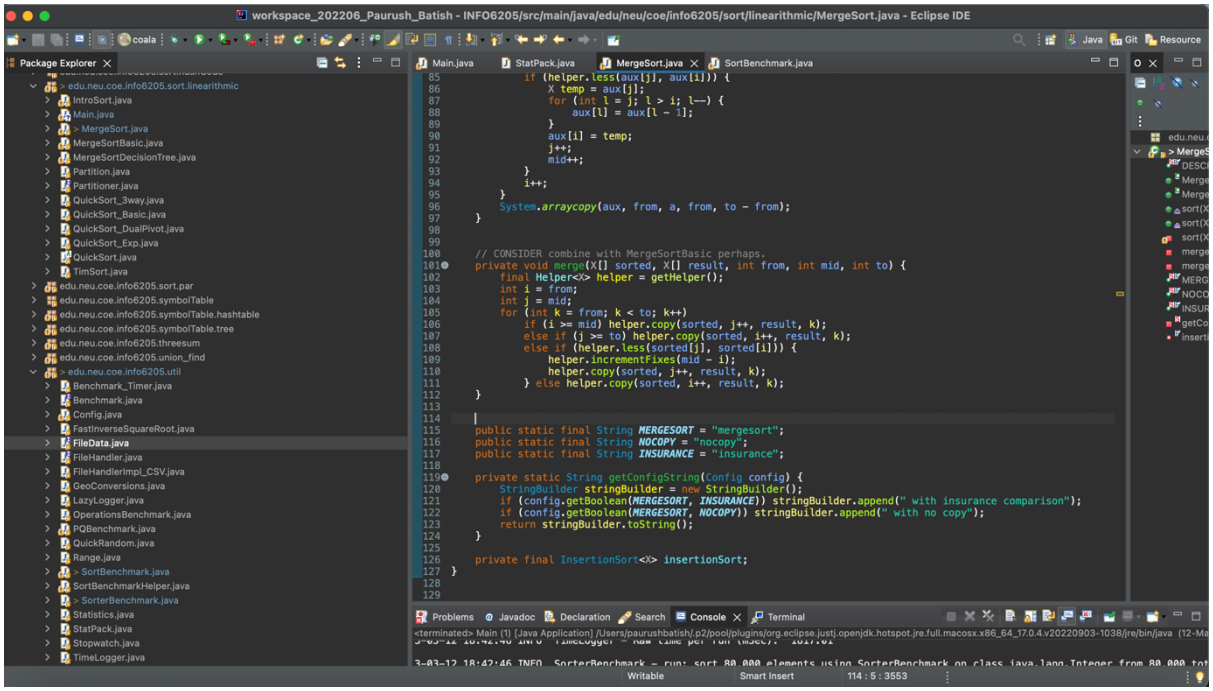
```
heapsort = true
```

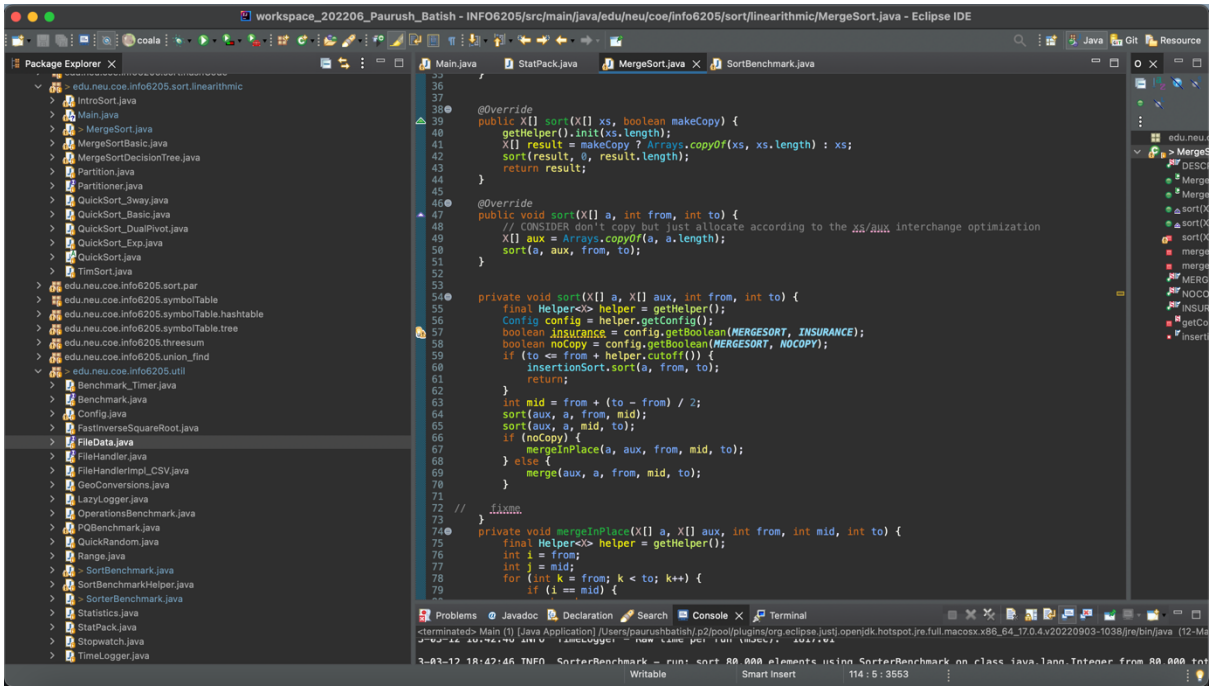
Remember that your job is to determine the best predictor: that will mean the graph of the appropriate observation will match the graph of the timings most closely.

Solution –

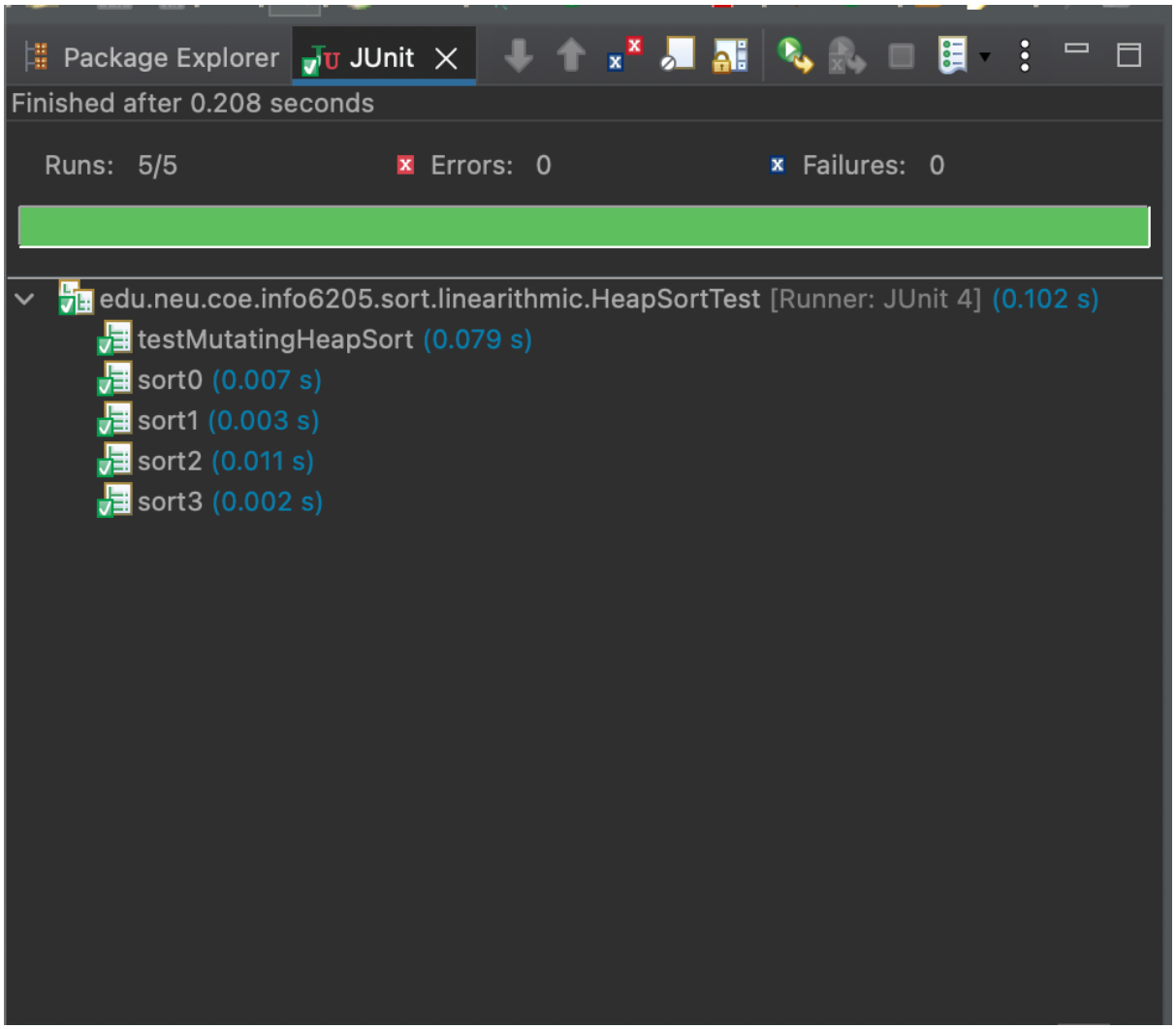


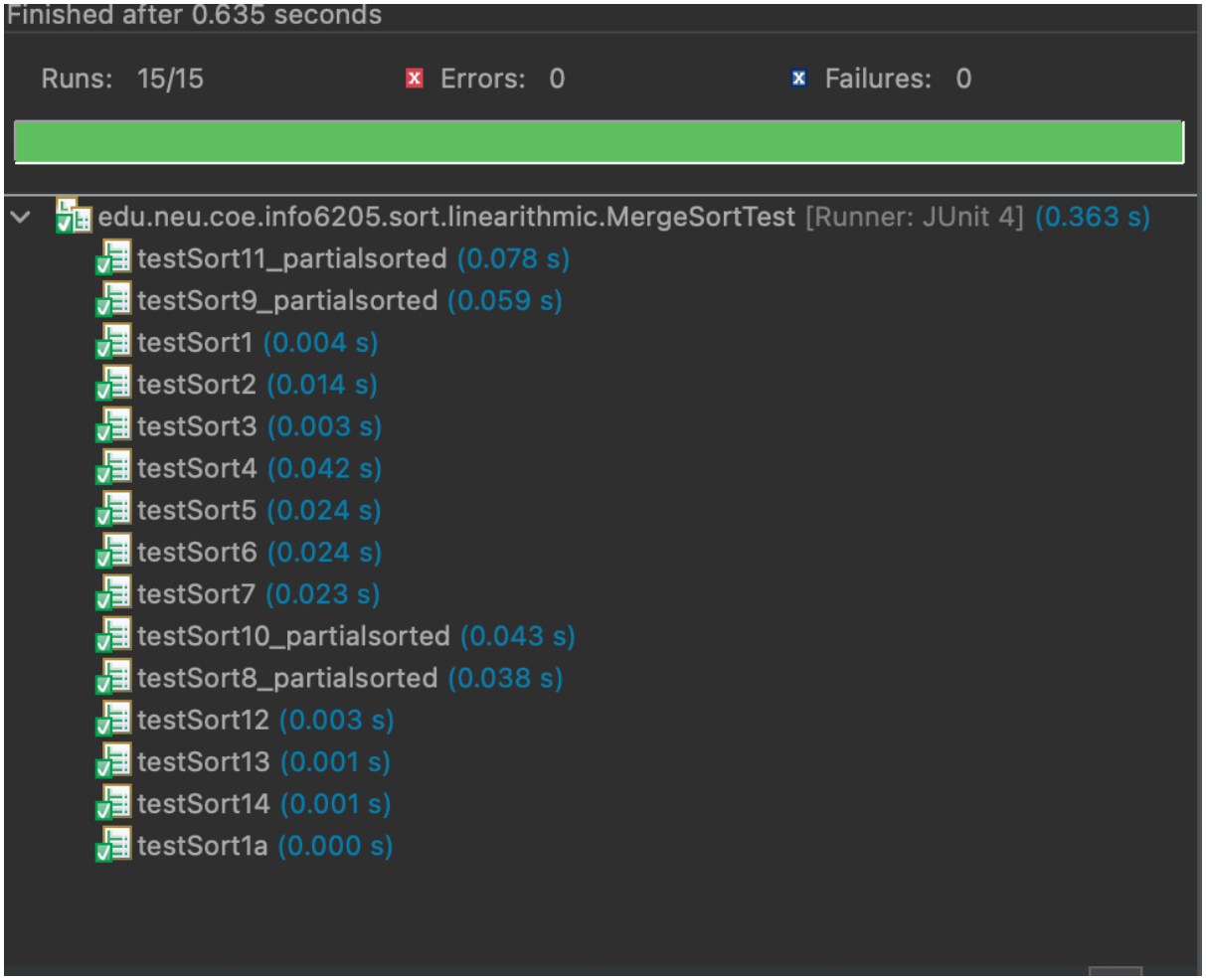

workspace_202206_Paurush_Batish - INFO6205/src/main/java/edu/neu/coe/info6205/sort/linearithmic/MergeSort.java - Eclipse IDE
Package Explorer
Main.java
StatPack.java
MergeSort.java
SortBenchmark.java
Problems
Javadoc
Declaration
Search
Console
Terminal
Writable
Smart Insert
114 : 5 : 3553

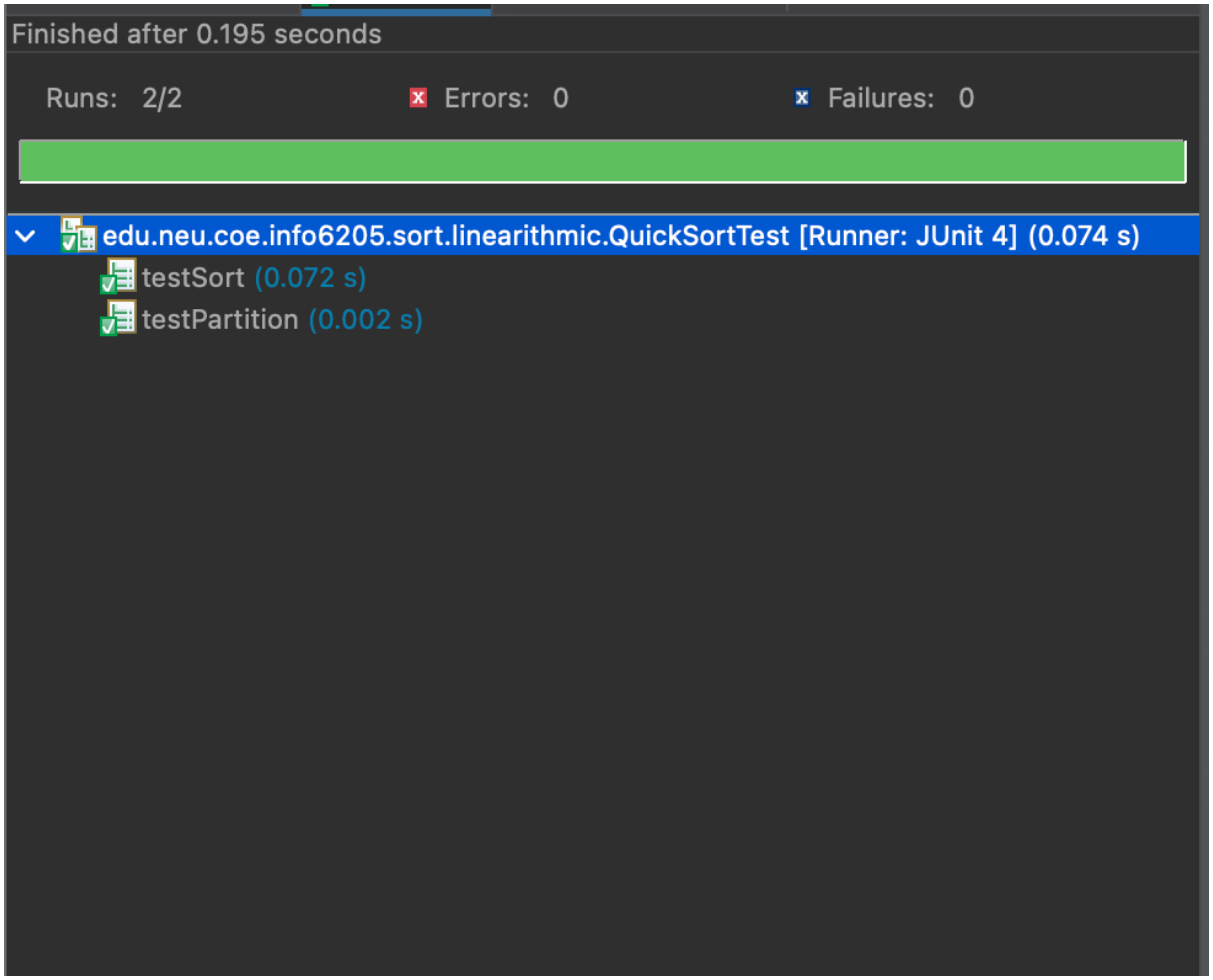

workspace_202206_Paurush_Batish - INFO6205/src/main/java/edu/neu/coe/info6205/sort/linearithmic/MergeSort.java - Eclipse IDE
Package Explorer
Main.java
StatPack.java
MergeSort.java
SortBenchmark.java
Problems
Javadoc
Declaration
Search
Console
Terminal
Writable
Smart Insert
114 : 5 : 3553


workspace_202206_Paurush_Batish - INFO6205/src/main/java/edu/neu/coe/info6205/sort/linearithmic/MergeSort.java - Eclipse IDE
Package Explorer
Main.java
StatPack.java
MergeSort.java
SortBenchmark.java
Problems
Javadoc
Declaration
Search
Console
Terminal

Unit Test –

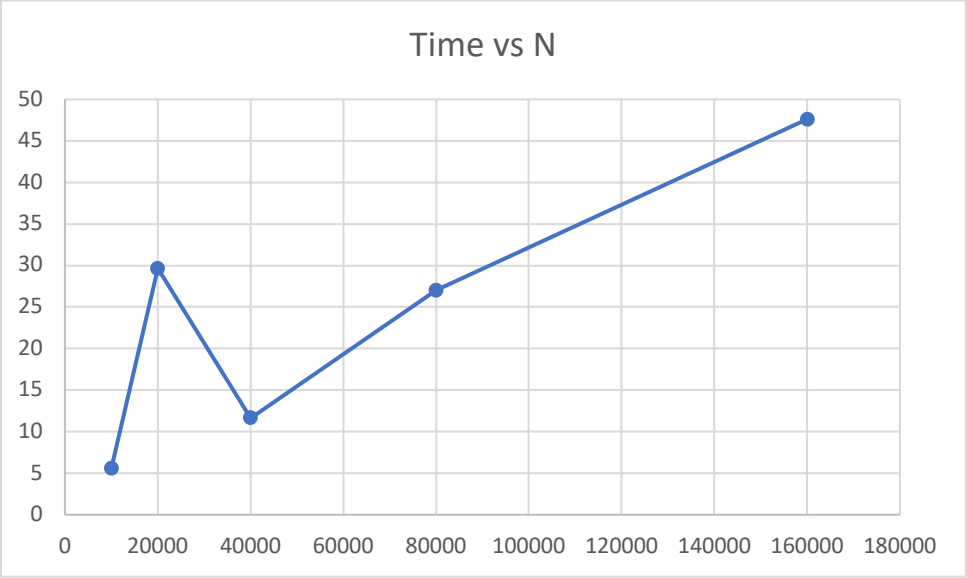

Package Explorer
JUnit
Finished after 0.208 seconds
Runs: 5/5
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.HeapSortTest [Runner: JUnit 4] (0.102 s)
testMutatingHeapSort (0.079 s)
sort0 (0.007 s)
sort1 (0.003 s)
sort2 (0.011 s)
sort3 (0.002 s)

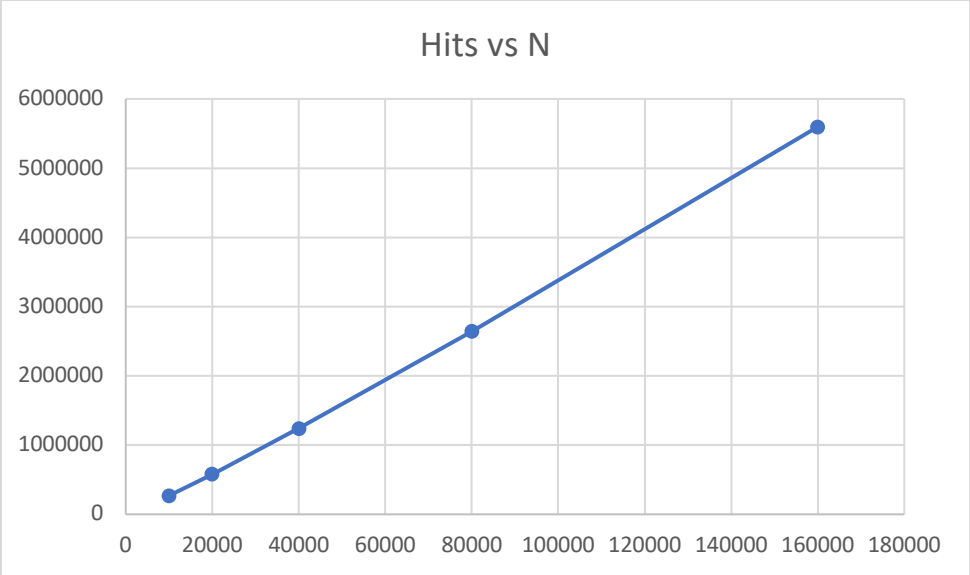

Finished after 0.635 seconds
Runs: 15/15
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.MergeSortTest [Runner: JUnit 4] (0.363 s)
testSort11_partialsorted (0.078 s)
testSort9_partialsorted (0.059 s)
testSort1 (0.004 s)
testSort2 (0.014 s)
testSort3 (0.003 s)
testSort4 (0.042 s)
testSort5 (0.024 s)
testSort6 (0.024 s)
testSort7 (0.023 s)
testSort10_partialsorted (0.043 s)
testSort8_partialsorted (0.038 s)
testSort12 (0.003 s)
testSort13 (0.001 s)
testSort14 (0.001 s)
testSort1a (0.000 s)

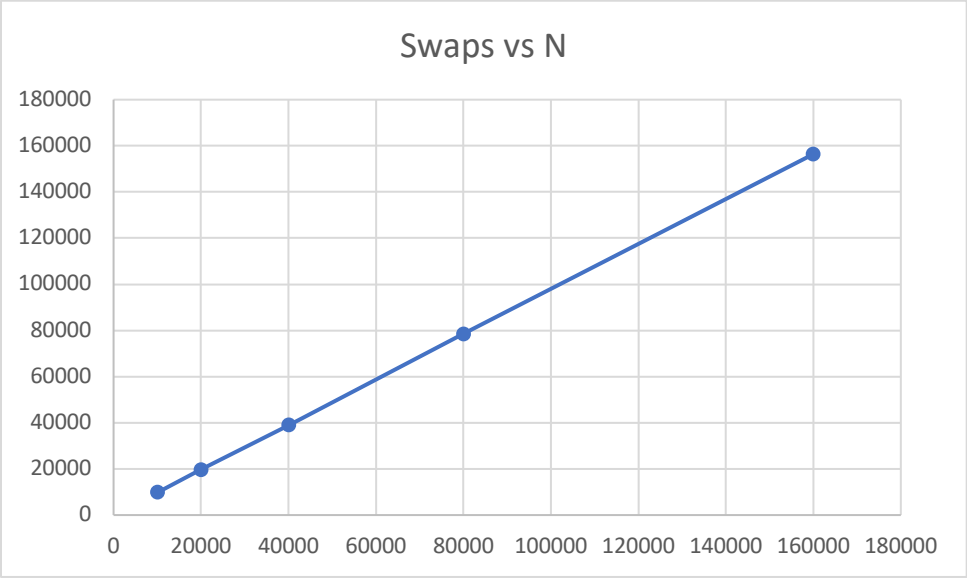


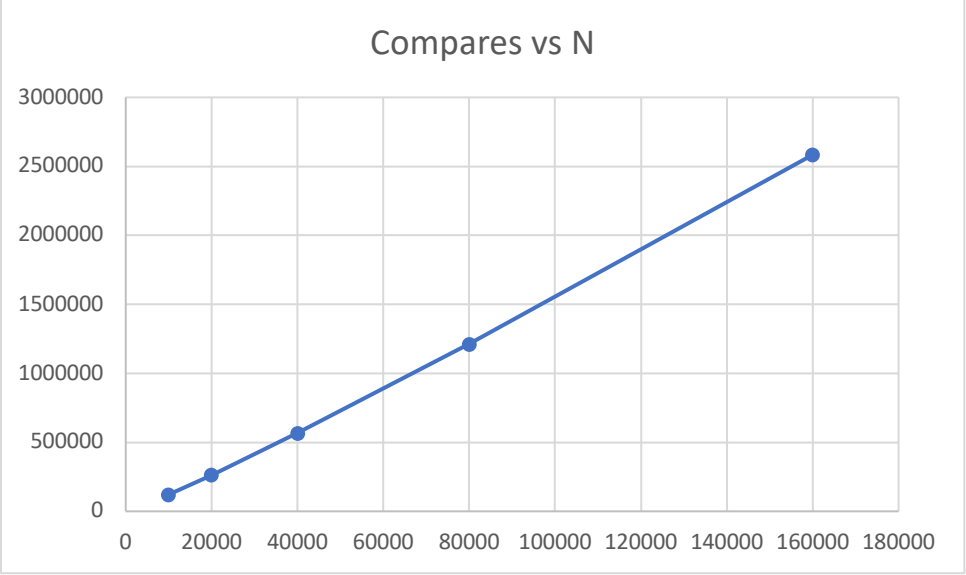
Recorded Observations –

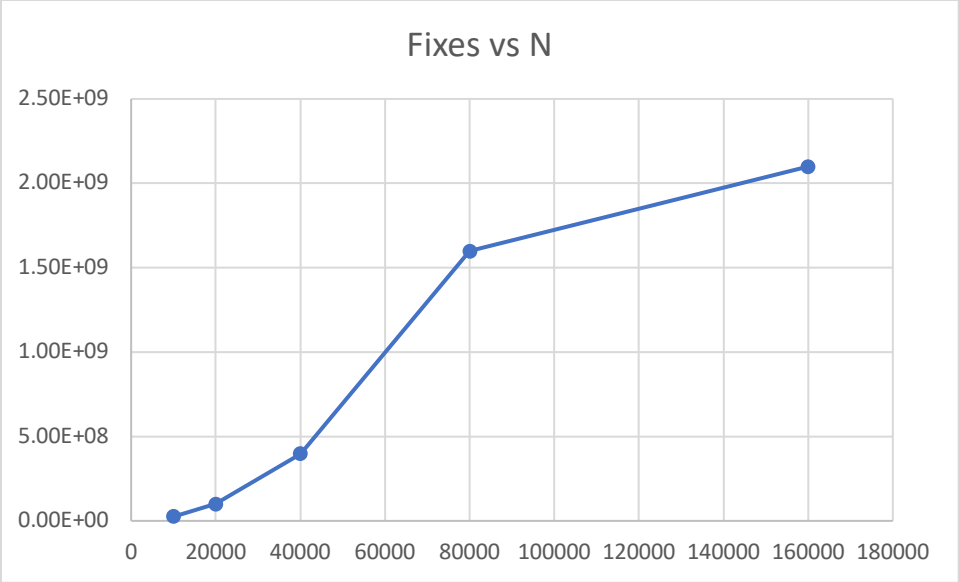
MERGE SORT –



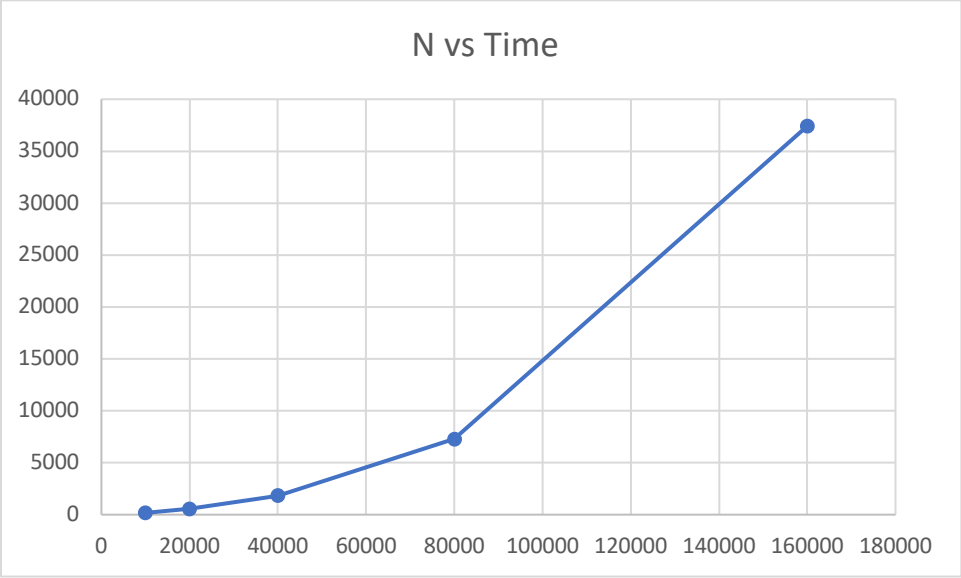

Hits vs N
6000000
5000000
4000000
3000000
2000000
1000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

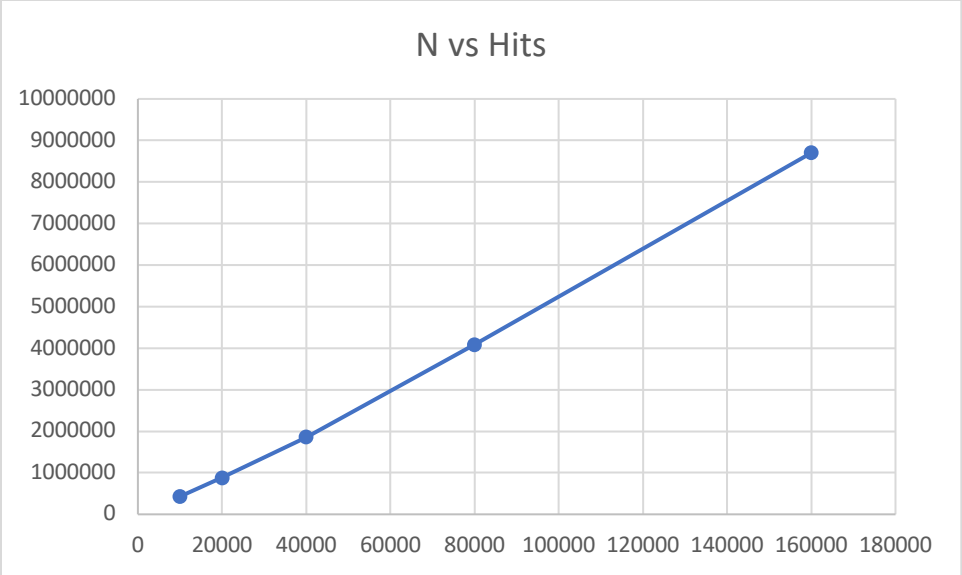

Swaps vs N
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

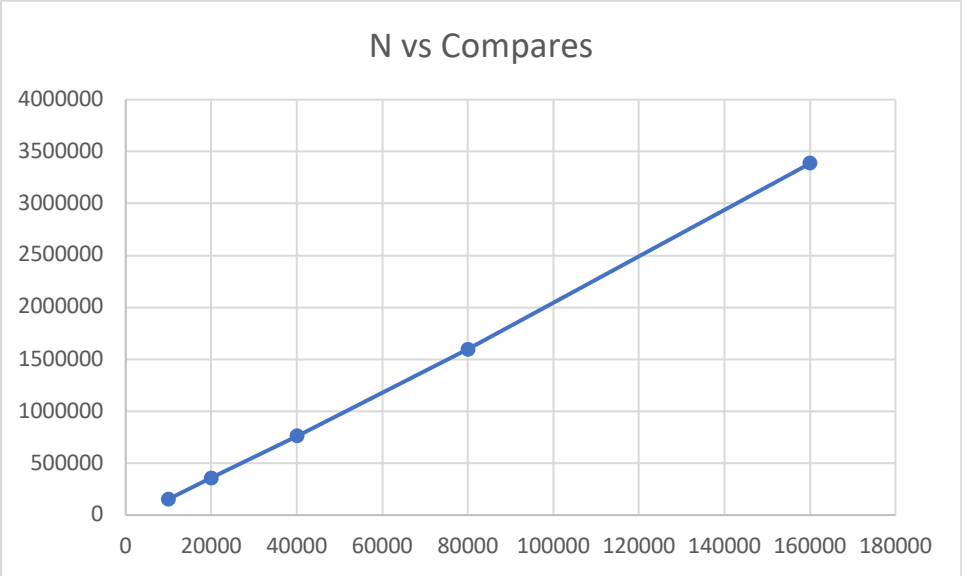

Compares vs N
3000000
2500000
2000000
1500000
1000000
500000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

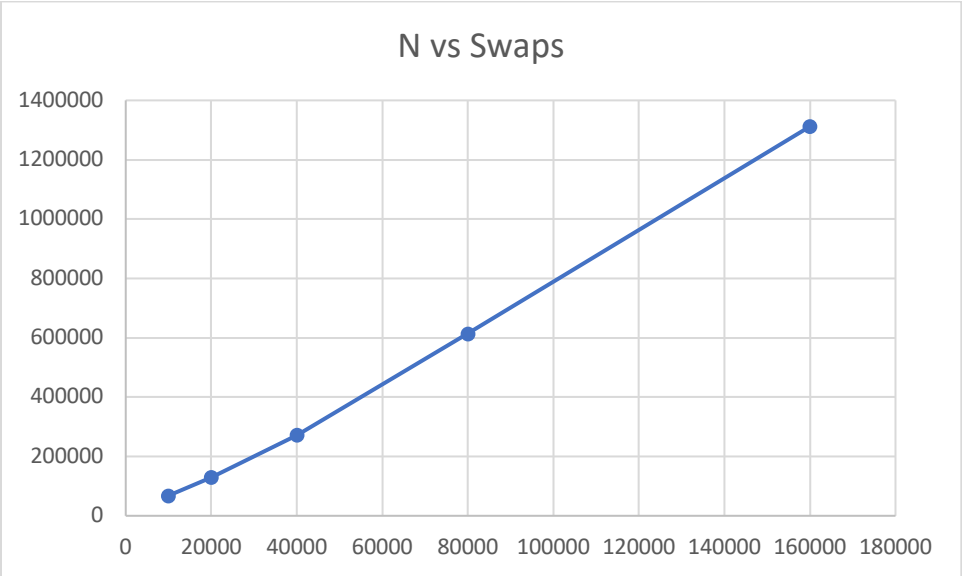

Fixes vs N
2.50E+09
2.00E+09
1.50E+09
1.00E+09
5.00E+08
0.00E+00
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

Quicksort –

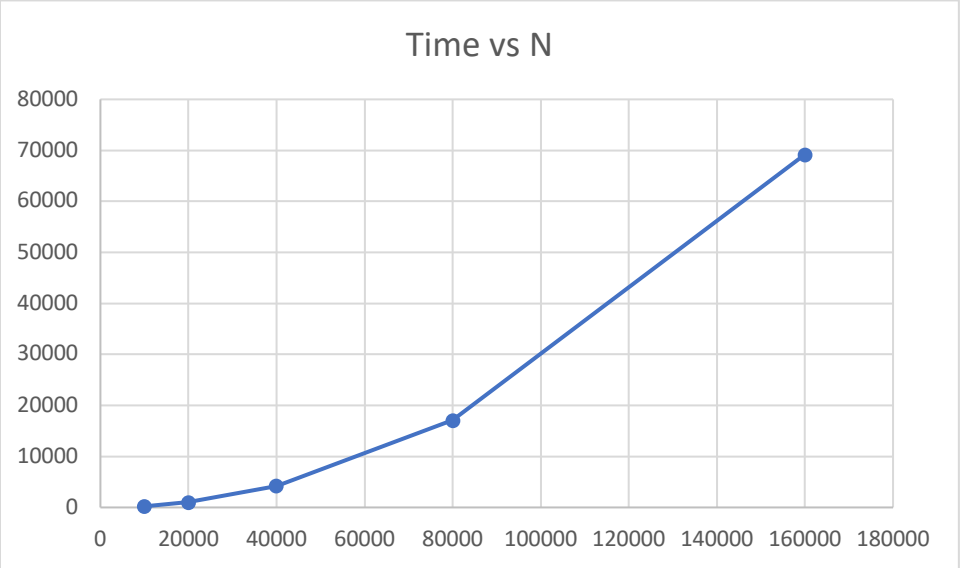

N vs Time
40000
35000
30000
25000
20000
15000
10000
5000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

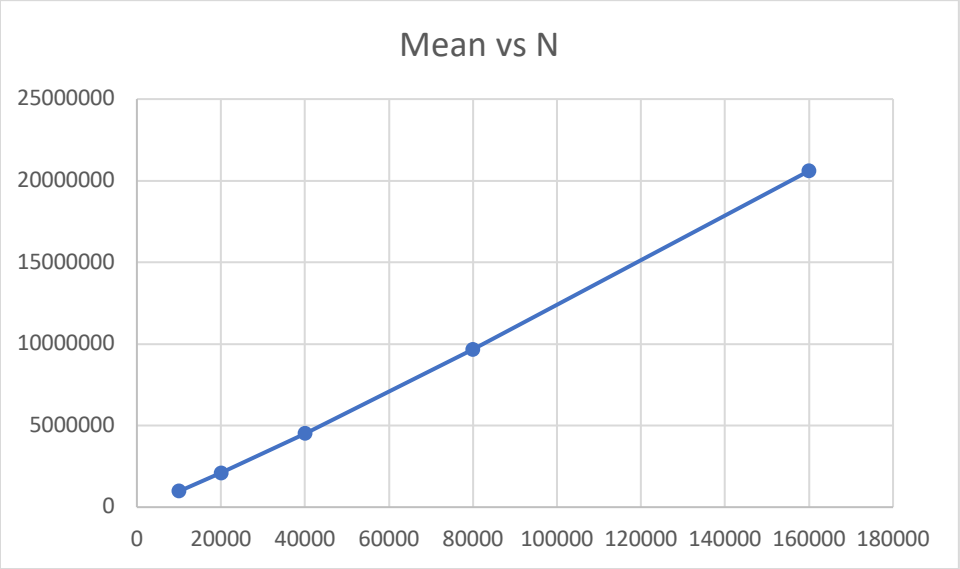

N vs Hits
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

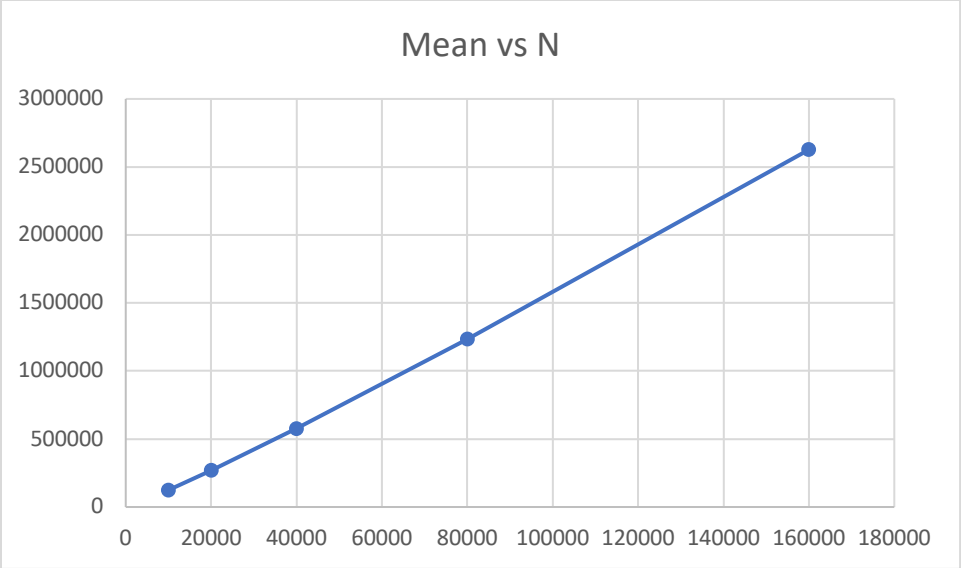

N vs Compares
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

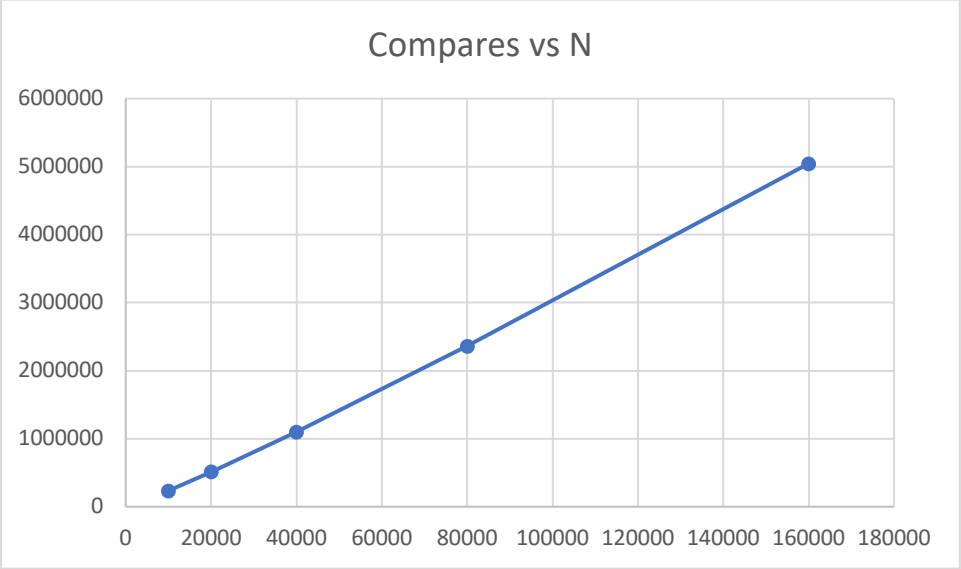

N vs Swaps
1400000
1200000
1000000
800000
600000
400000
200000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

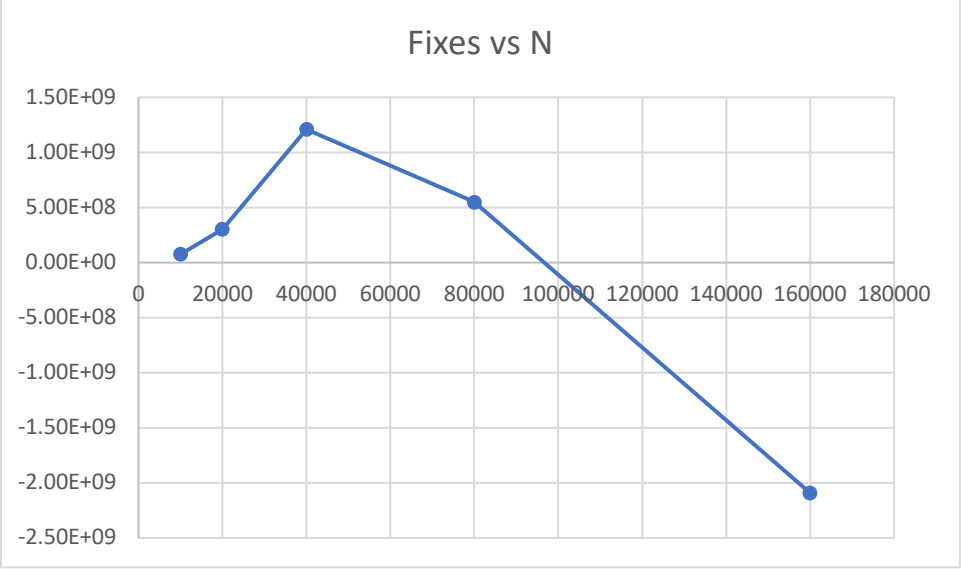
Heap Sort –


Time vs N
80000
70000
60000
50000
40000
30000
20000
10000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000


Mean vs N
25000000
20000000
15000000
10000000
5000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000


Mean vs N
3000000
2500000
2000000
1500000
1000000
500000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

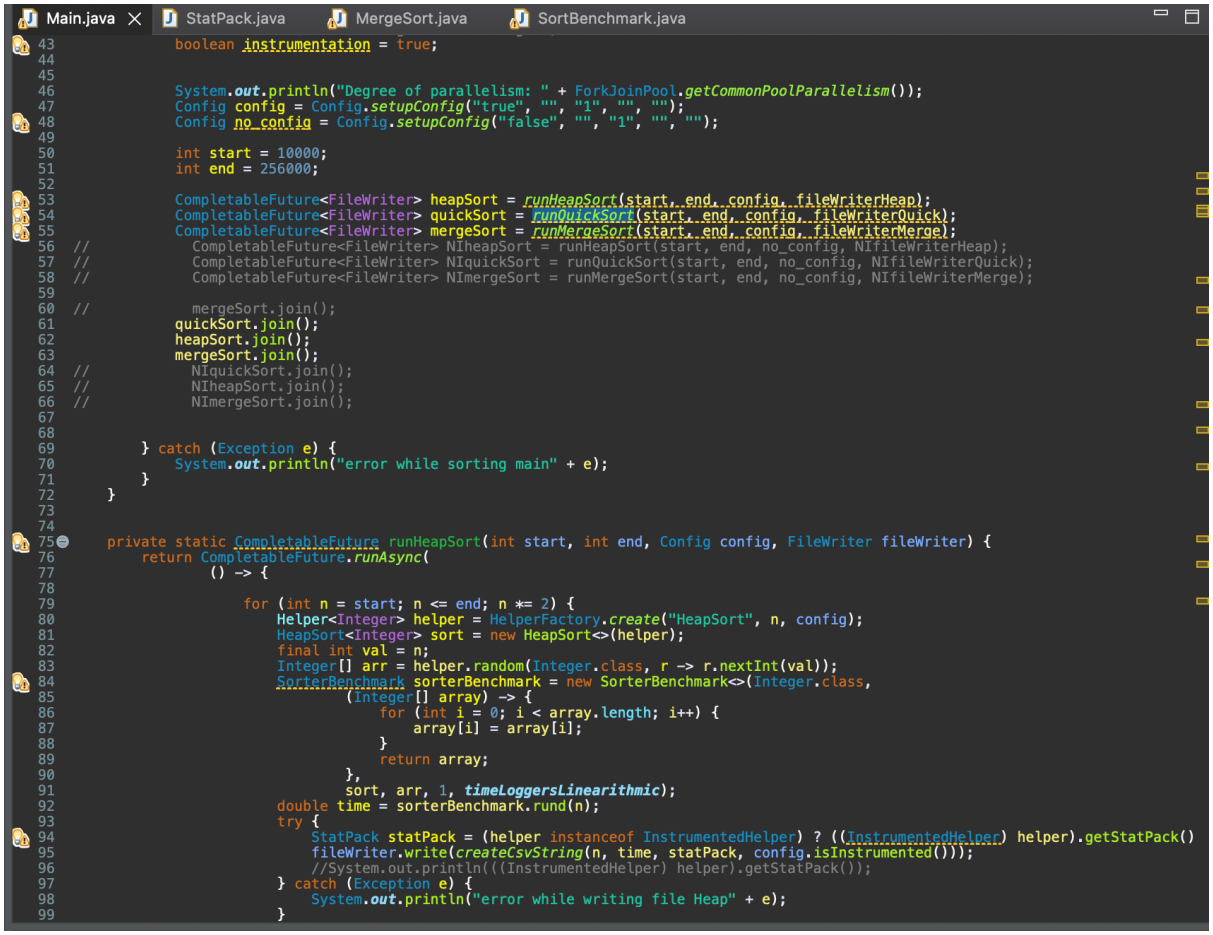

Compares vs N
6000000
5000000
4000000
3000000
2000000
1000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000


Fixes vs N
1.50E+09
1.00E+09
5.00E+08
0.00E+00
-5.00E+08
-1.00E+09
-1.50E+09
-2.00E+09
-2.50E+09
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

Main –

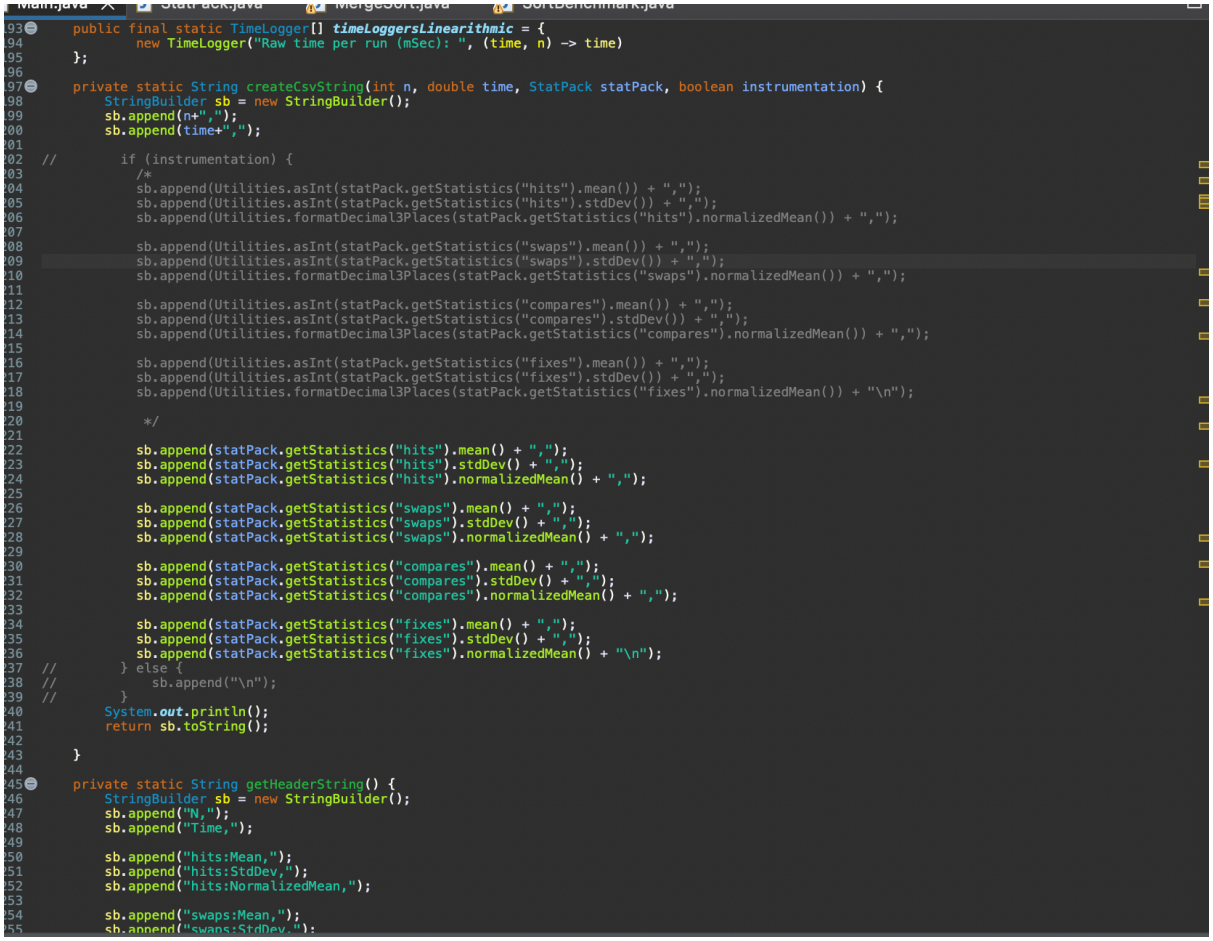
Main.java | StatPack.java | MergeSort.java | SortBenchmark.java

```
package edu.neu.coe.info6205.sort.linearithmic;
import edu.neu.coe.info6205.sort.Helper;
import edu.neu.coe.info6205.sort.HelperFactory;
import edu.neu.coe.info6205.sort.InstrumentedHelper;
import edu.neu.coe.info6205.sort.elementary.HeapSort;
import edu.neu.coe.info6205.util.*;
import java.io.File;
import java.io.FileWriter;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ForkJoinPool;

import static java.util.concurrent.CompletableFuture.runAsync;


public class Main {

    public static void main(String[] args) {
        try {
            File fileHeap = new File("HeapBenchMark.csv");
            File fileMerge = new File("MergeBenchMark.csv");
            File fileQuick = new File("QuickBenchMark.csv");
//            File NIfileHeap = new File("NoInstrumentationHeapBenchMark.csv");
//            File NIfileMerge = new File("NoInstrumentationMergeBenchMark.csv");
//            File NIfileQuick = new File("NoInstrumentationQuickBenchMark.csv");
            fileHeap.createNewFile();
            fileQuick.createNewFile();
            fileMerge.createNewFile();
//            NIfileHeap.createNewFile();
//            NIfileQuick.createNewFile();
//            NIfileMerge.createNewFile();
            FileWriter fileWriterHeap = new FileWriter(fileHeap);
            FileWriter fileWriterMerge = new FileWriter(fileMerge);
            FileWriter fileWriterQuick = new FileWriter(fileQuick);
//            FileWriter NIfileWriterHeap = new FileWriter(NIfileHeap);
//            FileWriter NIfileWriterMerge = new FileWriter(NIfileMerge);
//            FileWriter NIfileWriterQuick = new FileWriter(NIfileQuick);
            fileWriterHeap.write(getHeaderString());
            fileWriterMerge.write(getHeaderString());
            fileWriterQuick.write(getHeaderString());
```


Main.java
StatPack.java
MergeSort.java
SortBenchmark.java
boolean instrumentation = true;
System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
Config config = Config.setupConfig("true", "", "1", "", "");
Config no_config = Config.setupConfig("false", "", "1", "", "");
int start = 10000;
int end = 256000;
CompletableFuture<FileWriter> heapSort = runHeapSort(start, end, config, fileWriterHeap);
CompletableFuture<FileWriter> quickSort = runQuickSort(start, end, config, fileWriterQuick);
CompletableFuture<FileWriter> mergeSort = runMergeSort(start, end, config, fileWriterMerge);
// CompletableFuture<FileWriter> NIheapSort = runHeapSort(start, end, no_config, NIfileWriterHeap);
// CompletableFuture<FileWriter> NIquickSort = runQuickSort(start, end, no_config, NIfileWriterQuick);
// CompletableFuture<FileWriter> NImergeSort = runMergeSort(start, end, no_config, NIfileWriterMerge);
// mergeSort.join();
quickSort.join();
heapSort.join();
mergeSort.join();
// NIquickSort.join();
// NIheapSort.join();
// NImergeSort.join();
} catch (Exception e) {
System.out.println("error while sorting main" + e);
}
}
private static CompletableFuture runHeapSort(int start, int end, Config config, FileWriter fileWriter) {
return CompletableFuture.runAsync(
() -> {
for (int n = start; n <= end; n *= 2) {
Helper<Integer> helper = HelperFactory.create("HeapSort", n, config);
HeapSort<Integer> sort = new HeapSort<>(helper);
final int val = n;
Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
(Integer[] array) -> {
for (int i = 0; i < array.length; i++) {
array[i] = array[i];
}
return array;
},
sort, arr, 1, timeLoggersLinearithmic);
double time = sorterBenchmark.rund(n);
try {
StatPack statPack = (helper instanceof InstrumentedHelper) ? ((InstrumentedHelper) helper).getStatPack()
fileWriter.write(createCsvString(n, time, statPack, config.isInstrumented()));
//System.out.println(((InstrumentedHelper) helper).getStatPack());
} catch (Exception e) {
System.out.println("error while writing file Heap" + e);
}

```
                            sort, arr, 1, timeLoggersLinearithmic);
                    double time = sorterBenchmark.rund(n);
                    try {
                        StatPack statPack = (helper instanceof InstrumentedHelper) ? ((InstrumentedHelper) helper).getStatPack() : null;
                        fileWriter.write(createCsvString(n, time, statPack, config.isInstrumented()));
                        //System.out.println(((InstrumentedHelper) helper).getStatPack());
                    } catch (Exception e) {
                        System.out.println("error while writing file Heap" + e);
                    }
                }
                try {
                    fileWriter.flush();
                    fileWriter.close();
                } catch (Exception e) {
                    System.out.println("error while closing file Heap" + e);
                }

            }
        );
    }


    private static CompletableFuture runMergeSort(int start, int end, Config config, FileWriter fileWriter) {
        return runAsync(
                () -> {
                    for (int n = start; n <= end; n *= 2) {
                        Helper<Integer> helper = HelperFactory.create("MergeSort", n, config);
                        MergeSort<Integer> sort = new MergeSort<>(helper);
                        final int val = n;
                        Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
                        SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
                                (Integer[] array) -> {
                                    for (int i = 0; i < array.length; i++) {
                                        array[i] = array[i];
                                    }
                                    return array;
                                },
                                sort, arr, 1, timeLoggersLinearithmic);
                        double time = sorterBenchmark.rund(n);
                        try {
                            if (helper instanceof InstrumentedHelper) {
                                StatPack statPack = (helper instanceof InstrumentedHelper) ? ((InstrumentedHelper) helper).getStatPack() : null;
                                fileWriter.write(createCsvString(n, time, statPack, config.isInstrumented()));

                            }



                        } catch (Exception e) {
                            System.out.println("error while writing file Merge" + e);
                        }
                    }
                    try {
                        fileWriter.flush();
                        fileWriter.close();
                    } catch (Exception e) {
                        System.out.println("error while closing file Merge" + e);
                    }
                }
        );
    }
```

```
    }


    private static CompletableFuture runQuickSort(int start, int end, Config config, FileWriter fileWriter) {
        return runAsync(
                () -> {
                    for (int n = start; n <= end; n *= 2) {
                        Helper<Integer> helper = HelperFactory.create("QuickSort", n, config);
                        QuickSort_DualPivot<Integer> sort = new QuickSort_DualPivot<>(helper);
                        final int val = n;
                        Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
                        SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
                                (Integer[] array) -> {
                                    for (int i = 0; i < array.length; i++) {
                                        array[i] = array[i];
                                    }
                                    return array;
                                },
                                sort, arr, 1, timeLoggersLinearithmic);
                        double time =sorterBenchmark.rund(n);
                        try {
                            if (helper instanceof InstrumentedHelper) {
                                StatPack statPack = (helper instanceof InstrumentedHelper) ? ((InstrumentedHelper) helper).getStatPack() : null;
                                fileWriter.write(createCsvString(n, time, statPack, config.isInstrumented()));

                            }

                        } catch (Exception e) {
                            System.out.println("error while writing file Quick" + e);
                        }
                    }
                    try {
                        fileWriter.flush();
                        fileWriter.close();
                    } catch (Exception e) {
                        System.out.println("error while closing file Quick" + e);
                    }
                }

        );
    }

    public final static TimeLogger[] timeLoggersLinearithmic = {
            new TimeLogger("Raw time per run (mSec): ", (time, n) -> time)
    };

    private static String createCsvString(int n, double time, StatPack statPack, boolean instrumentation) {
        StringBuilder sb = new StringBuilder();
        sb.append(n+",");
        sb.append(time+",");

//        if (instrumentation) {
          /*
          sb.append(Utilities.asInt(statPack.getStatistics("hits").mean()) + ",");
          sb.append(Utilities.asInt(statPack.getStatistics("hits").stdDev()) + ",");
          sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("hits").normalizedMean()) + ",");

          sb.append(Utilities.asInt(statPack.getStatistics("swaps").mean()) + ",");
          sb.append(Utilities.asInt(statPack.getStatistics("swaps").stdDev()) + ",");
          sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("swaps").normalizedMean()) + ",");
```



```
    public final static TimeLogger[] timeLoggersLinearithmic = {
            new TimeLogger("Raw time per run (mSec): ", (time, n) -> time)
    };

    private static String createCsvString(int n, double time, StatPack statPack, boolean instrumentation) {
        StringBuilder sb = new StringBuilder();
        sb.append(n+",");
        sb.append(time+",");

//        if (instrumentation) {
            /*
            sb.append(Utilities.asInt(statPack.getStatistics("hits").mean()) + ",");
            sb.append(Utilities.asInt(statPack.getStatistics("hits").stdDev()) + ",");
            sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("hits").normalizedMean()) + ",");

            sb.append(Utilities.asInt(statPack.getStatistics("swaps").mean()) + ",");
            sb.append(Utilities.asInt(statPack.getStatistics("swaps").stdDev()) + ",");
            sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("swaps").normalizedMean()) + ",");

            sb.append(Utilities.asInt(statPack.getStatistics("compares").mean()) + ",");
            sb.append(Utilities.asInt(statPack.getStatistics("compares").stdDev()) + ",");
            sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("compares").normalizedMean()) + ",");

            sb.append(Utilities.asInt(statPack.getStatistics("fixes").mean()) + ",");
            sb.append(Utilities.asInt(statPack.getStatistics("fixes").stdDev()) + ",");
            sb.append(Utilities.formatDecimal3Places(statPack.getStatistics("fixes").normalizedMean()) + "\n");

             */

            sb.append(statPack.getStatistics("hits").mean() + ",");
            sb.append(statPack.getStatistics("hits").stdDev() + ",");
            sb.append(statPack.getStatistics("hits").normalizedMean() + ",");

            sb.append(statPack.getStatistics("swaps").mean() + ",");
            sb.append(statPack.getStatistics("swaps").stdDev() + ",");
            sb.append(statPack.getStatistics("swaps").normalizedMean() + ",");

            sb.append(statPack.getStatistics("compares").mean() + ",");
            sb.append(statPack.getStatistics("compares").stdDev() + ",");
            sb.append(statPack.getStatistics("compares").normalizedMean() + ",");

            sb.append(statPack.getStatistics("fixes").mean() + ",");
            sb.append(statPack.getStatistics("fixes").stdDev() + ",");
            sb.append(statPack.getStatistics("fixes").normalizedMean() + "\n");
//        } else {
//            sb.append("\n");
//        }
        System.out.println();
        return sb.toString();

    }

    private static String getHeaderString() {
        StringBuilder sb = new StringBuilder();
        sb.append("N,");
        sb.append("Time,");

        sb.append("hits:Mean,");
        sb.append("hits:StdDev,");
        sb.append("hits:NormalizedMean,");

        sb.append("swaps:Mean,");
        sb.append("swaps:StdDev,");
```

```
            sb.append(statPack.getStatistics("fixes").mean() + ",");
            sb.append(statPack.getStatistics("fixes").stdDev() + ",");
            sb.append(statPack.getStatistics("fixes").normalizedMean() + "\n");
//        } else {
//            sb.append("\n");
//        }
        System.out.println();
        return sb.toString();

    }

    private static String getHeaderString() {
        StringBuilder sb = new StringBuilder();
        sb.append("N,");
        sb.append("Time,");

        sb.append("hits:Mean,");
        sb.append("hits:StdDev,");
        sb.append("hits:NormalizedMean,");

        sb.append("swaps:Mean,");
        sb.append("swaps:StdDev,");
        sb.append("swaps:NormalizedMean,");

        sb.append("compares:Mean,");
        sb.append("compares:StdDev,");
        sb.append("compares:NormalizedMean,");

        sb.append("fixes:Mean,");
        sb.append("fixes:StdDev,");
        sb.append("fixes:NormalizedMean\n");

        return sb.toString();

    }


}
```

```
main [Java Application] /Users/Pranavkapoor1/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java  (12-Mar-2023, 7:51:15 pm) [pid: 80488]
Degree of parallelism: 7
2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  4.93

2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 20,000 elements with 1 runs
2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  22.19

2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 40,000 elements with 1 runs
2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  9.87

2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 80,000 elements with 1 runs
2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  15.03

2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 160,000 elements with 1 runs
2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  32.72

2023-03-12 19:51:16 INFO  TimeLogger - Raw time per run (mSec):  155.01

2023-03-12 19:51:16 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:16 INFO  Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 20,000 elements with 1 runs
2023-03-12 19:51:17 INFO  TimeLogger - Raw time per run (mSec):  203.26

2023-03-12 19:51:17 INFO  SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:17 INFO  Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 20,000 elements with 1 runs
2023-03-12 19:51:19 INFO  TimeLogger - Raw time per run (mSec):  706.28

2023-03-12 19:51:19 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:19 INFO  Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 40,000 elements with 1 runs
2023-03-12 19:51:19 INFO  TimeLogger - Raw time per run (mSec):  895.22

2023-03-12 19:51:19 INFO  SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:19 INFO  Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 40,000 elements with 1 runs
2023-03-12 19:51:28 INFO  TimeLogger - Raw time per run (mSec):  3122.29

2023-03-12 19:51:28 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:28 INFO  Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 80,000 elements with 1 runs
2023-03-12 19:51:31 INFO  TimeLogger - Raw time per run (mSec):  3999.02

2023-03-12 19:51:31 INFO  SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:31 INFO  Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 80,000 elements with 1 runs
2023-03-12 19:52:09 INFO  TimeLogger - Raw time per run (mSec):  13647.90

2023-03-12 19:52:09 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:52:09 INFO  Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 160,000 elements with 1 runs
2023-03-12 19:52:20 INFO  TimeLogger - Raw time per run (mSec):  16313.41

2023-03-12 19:52:20 INFO  SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:52:20 INFO  Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 160,000 elements with 1 runs
```