Program structures and algorithms
Spring 2023 (Section-01)

BY: PRANAV KAPOOR – NUID: 002998253

Assignment 6 (Hits as time predictor)

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).

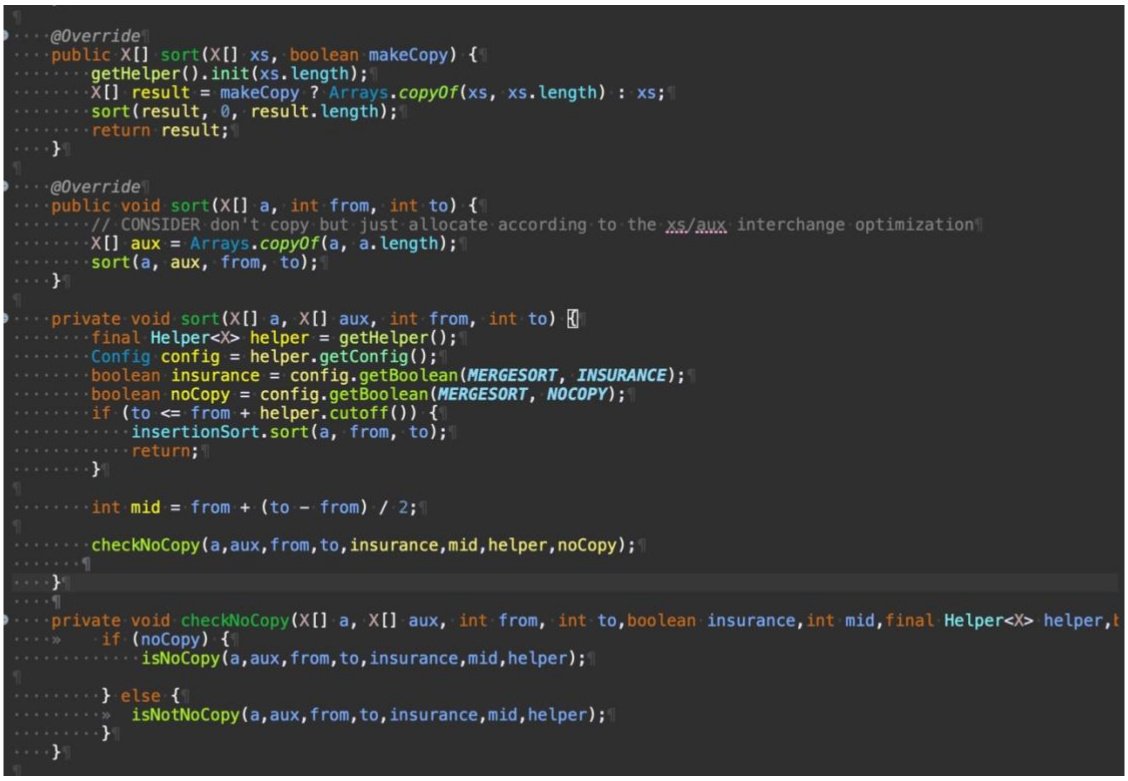
All of the code to count comparisons, swaps/copies, and hits, is already implemented in the *InstrumentedHelper* class. You can see examples of the usage of this kind of analysis in:

- 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.

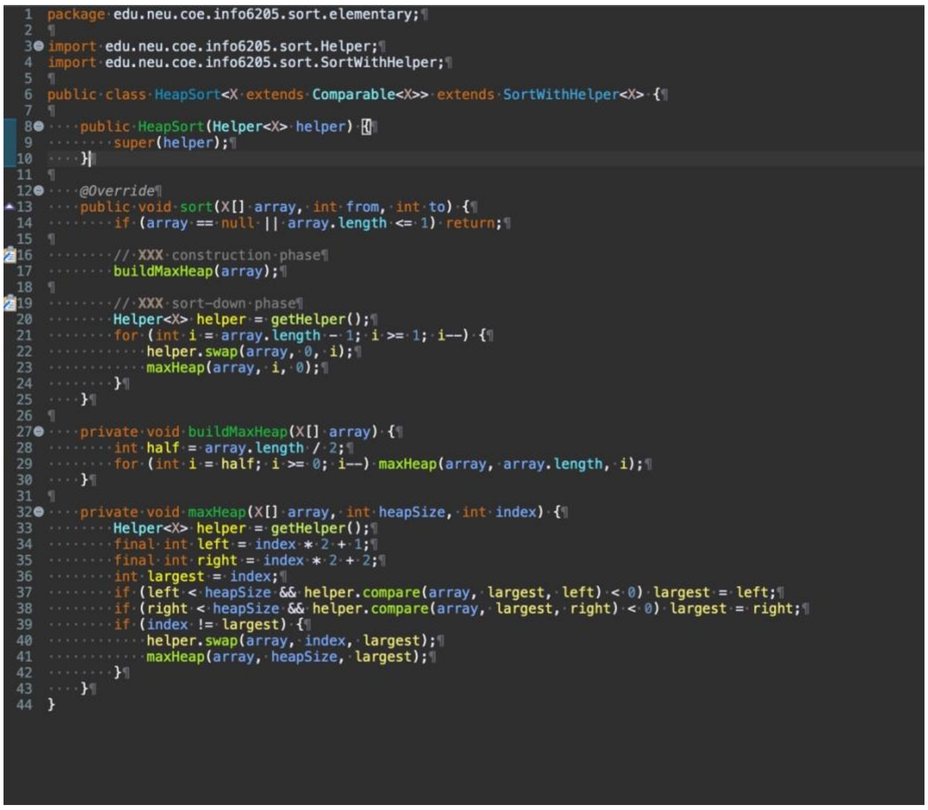
Solution:

MergeSort.java

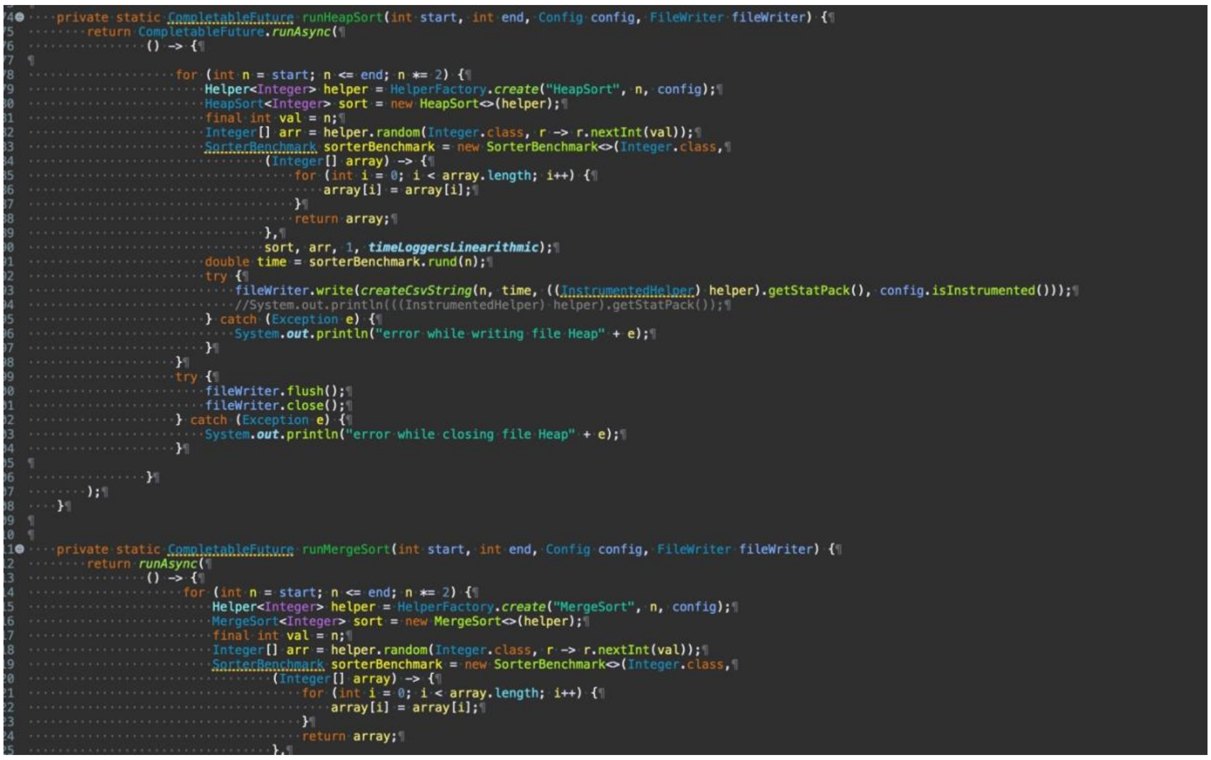


```java
    @Override
    public X[] sort(X[] xs, boolean makeCopy) {
        getHelper().init(xs.length);
        X[] result = makeCopy ? Arrays.copyOf(xs, xs.length) : xs;
        sort(result, 0, result.length);
        return result;
    }

    @Override
    public void sort(X[] a, int from, int to) {
        // CONSIDER don't copy but just allocate according to the xs/aux interchange optimization
        X[] aux = Arrays.copyOf(a, a.length);
        sort(a, aux, from, to);
    }

    private void sort(X[] a, X[] aux, int from, int to) {
        final Helper<X> helper = getHelper();
        Config config = helper.getConfig();
        boolean insurance = config.getBoolean(MERGESORT, INSURANCE);
        boolean noCopy = config.getBoolean(MERGESORT, NOCOPY);
        if (to <= from + helper.cutoff()) {
            insertionSort.sort(a, from, to);
            return;
        }

        int mid = from + (to - from) / 2;

        checkNoCopy(a,aux,from,to,insurance,mid,helper,noCopy);
        
    }
    
    private void checkNoCopy(X[] a, X[] aux, int from, int to,boolean insurance,int mid,final Helper<X> helper,t
         if (noCopy) {
             isNoCopy(a,aux,from,to,insurance,mid,helper);

         } else {
             isNotNoCopy(a,aux,from,to,insurance,mid,helper);
         }
    }
```

HeapSort.java

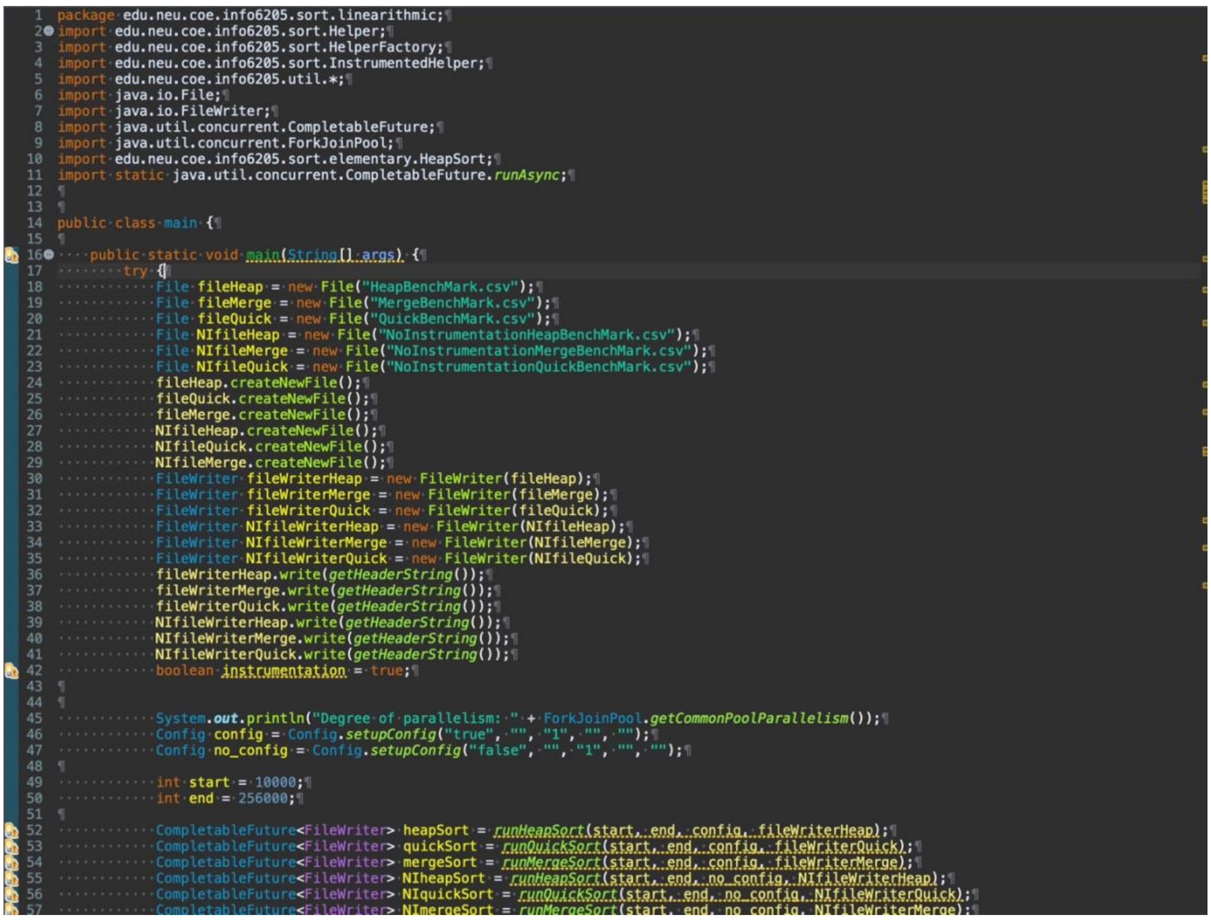


```java
package edu.neu.coe.info6205.sort.elementary;

import edu.neu.coe.info6205.sort.Helper;
import edu.neu.coe.info6205.sort.SortWithHelper;

public class HeapSort<X extends Comparable<X>> extends SortWithHelper<X> {

    public HeapSort(Helper<X> helper) {
        super(helper);
    }

    @Override
    public void sort(X[] array, int from, int to) {
        if (array == null || array.length <= 1) return;

        // XXX construction phase
        buildMaxHeap(array);

        // XXX sort-down phase
        Helper<X> helper = getHelper();
        for (int i = array.length - 1; i >= 1; i--) {
            helper.swap(array, 0, i);
            maxHeap(array, i, 0);
        }
    }

    private void buildMaxHeap(X[] array) {
        int half = array.length / 2;
        for (int i = half; i >= 0; i--) maxHeap(array, array.length, i);
    }

    private void maxHeap(X[] array, int heapSize, int index) {
        Helper<X> helper = getHelper();
        final int left = index * 2 + 1;
        final int right = index * 2 + 2;
        int largest = index;
        if (left < heapSize && helper.compare(array, largest, left) < 0) largest = left;
        if (right < heapSize && helper.compare(array, largest, right) < 0) largest = right;
        if (index != largest) {
            helper.swap(array, index, largest);
            maxHeap(array, heapSize, largest);
        }
    }
}
```



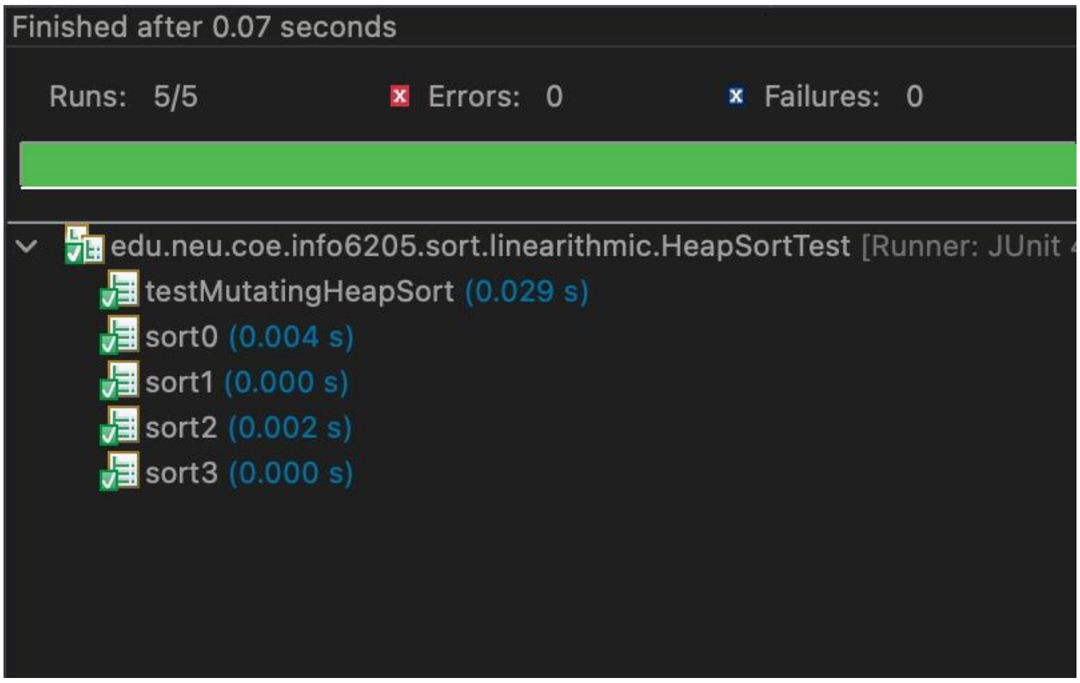
```java
    private static CompletableFuture runHeapSort(int start, int end, Config config, FileWriter fileWriter) {
        return CompletableFuture.runAsync(
                () -> {

                    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 {
                            fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), 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;
                                },
```

Main.java

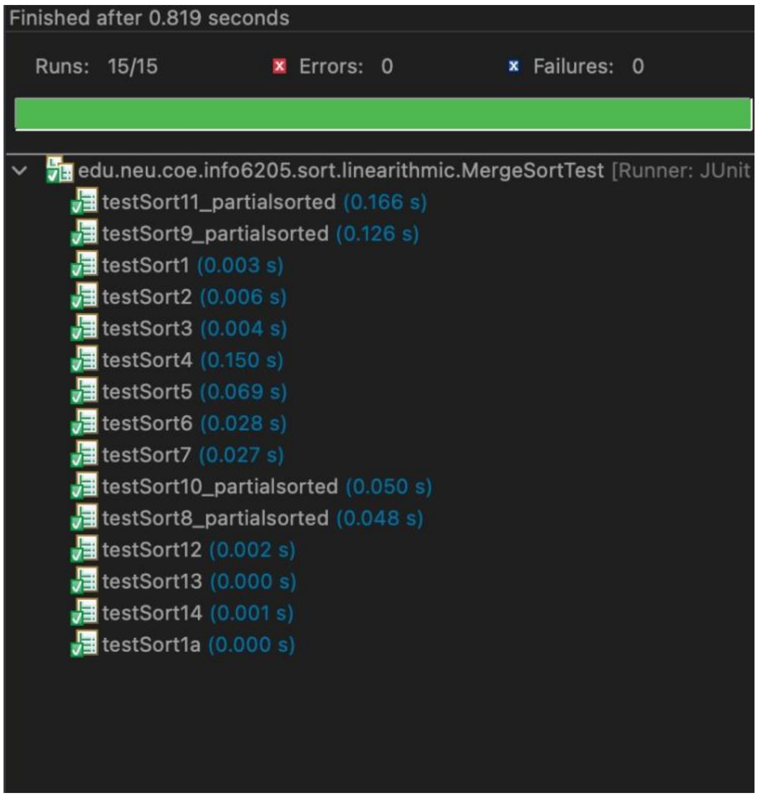


```java
package edu.neu.coe.info6205.sort.linearithmic;
import edu.neu.coe.info6205.sort.Helper;
import edu.neu.coe.info6205.sort.HelperFactory;
import edu.neu.coe.info6205.sort.InstrumentedHelper;
import edu.neu.coe.info6205.util.*;
import java.io.File;
import java.io.FileWriter;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ForkJoinPool;
import edu.neu.coe.info6205.sort.elementary.HeapSort;
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());
            NIfileWriterHeap.write(getHeaderString());
            NIfileWriterMerge.write(getHeaderString());
            NIfileWriterQuick.write(getHeaderString());
            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);
```

Unit Tests

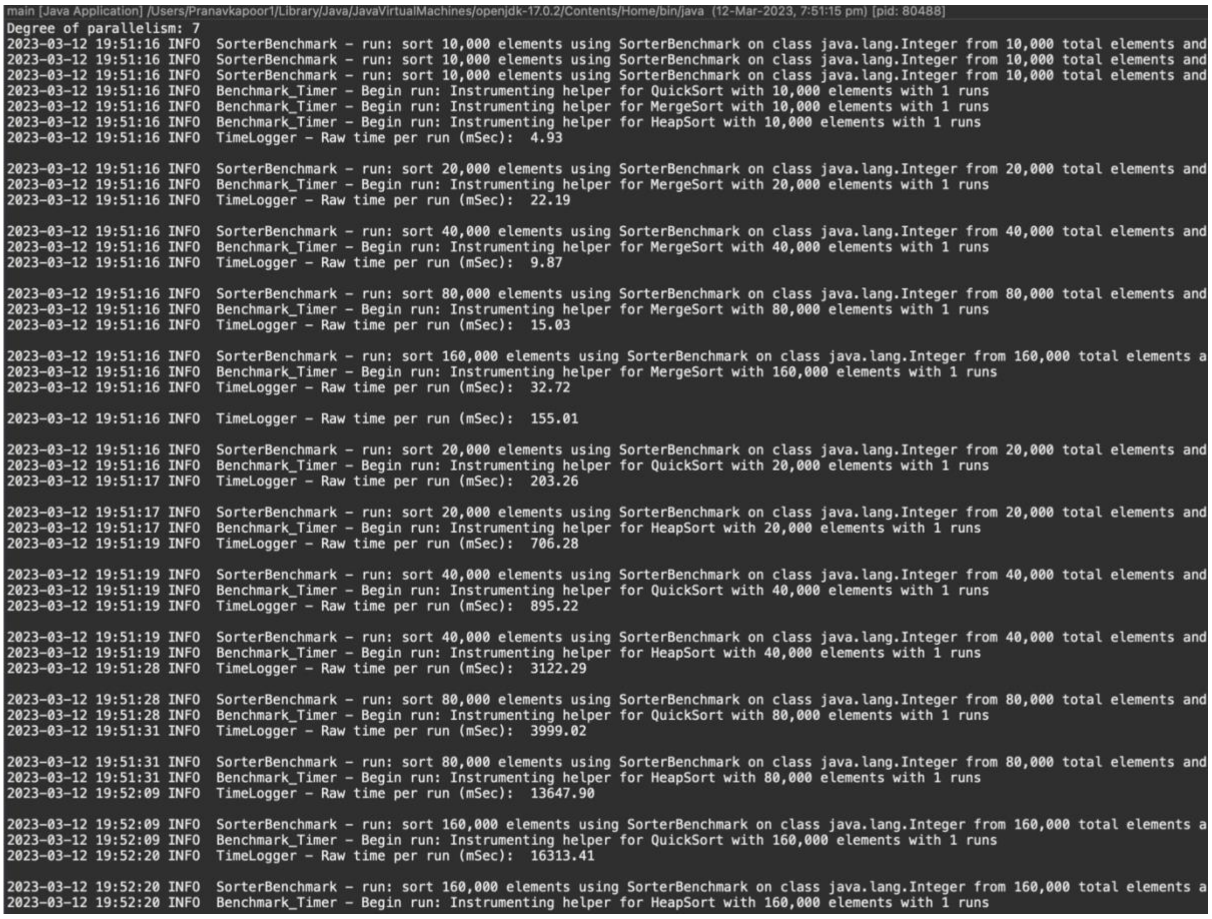


```
Finished after 0.07 seconds

Runs:  5/5          Errors:  0          Failures:  0

edu.neu.coe.info6205.sort.linearithmic.HeapSortTest [Runner: JUnit
    testMutatingHeapSort (0.029 s)
    sort0 (0.004 s)
    sort1 (0.000 s)
    sort2 (0.002 s)
    sort3 (0.000 s)
```

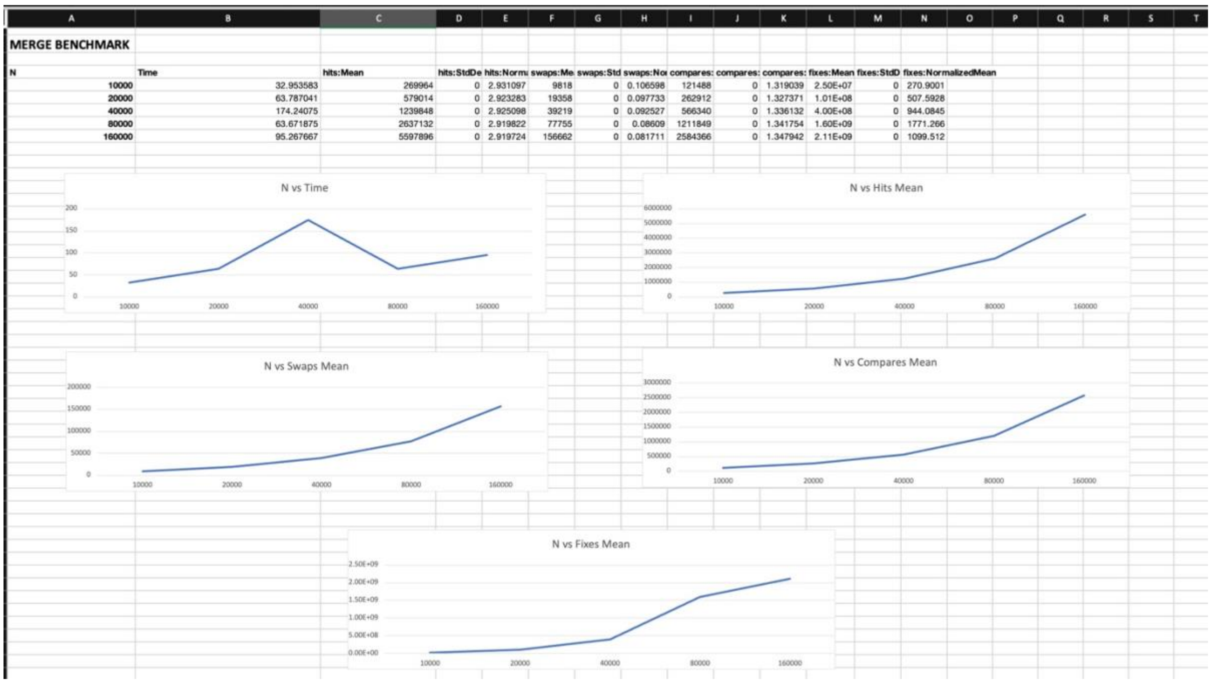


```
Finished after 0.819 seconds

Runs:  15/15          Errors:  0          Failures:  0

edu.neu.coe.info6205.sort.linearithmic.MergeSortTest [Runner: JUnit
    testSort11_partialsorted (0.166 s)
    testSort9_partialsorted (0.126 s)
    testSort1 (0.003 s)
    testSort2 (0.006 s)
    testSort3 (0.004 s)
    testSort4 (0.150 s)
    testSort5 (0.069 s)
    testSort6 (0.028 s)
    testSort7 (0.027 s)
    testSort10_partialsorted (0.050 s)
    testSort8_partialsorted (0.048 s)
    testSort12 (0.002 s)
    testSort13 (0.000 s)
    testSort14 (0.001 s)
    testSort1a (0.000 s)
```

Conclusion


```
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
```

MERGE BENCHMARK

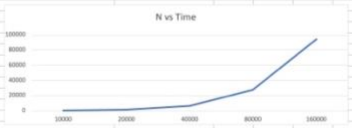

MERGE BENCHMARK

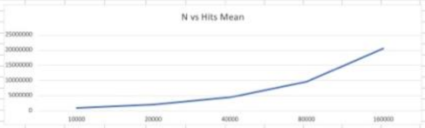
N	Time	hits:Mean	hits:StdDe	hits:Norm	swaps:Me	swaps:Std	swaps:No	compares:	compares:	compares:	fixes:Mean	fixes:StdD	fixes:NormalizedMean
10000	32.953583	269964	0	2.931097	9818	0	0.106598	121488	0	1.319039	2.50E+07	0	270.9001
20000	63.787041	579014	0	2.923283	19358	0	0.097733	262912	0	1.327371	1.01E+08	0	507.5928
40000	174.24075	1239848	0	2.925098	39219	0	0.092527	566340	0	1.336132	4.00E+08	0	944.0845
80000	63.671875	2637132	0	2.919822	77755	0	0.08609	1211849	0	1.341754	1.60E+09	0	1771.266
160000	95.267667	5597896	0	2.919724	156662	0	0.081711	2584366	0	1.347942	2.11E+09	0	1099.512

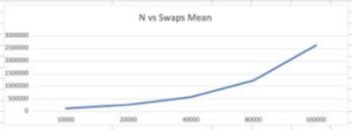
Heap BENCHMARK

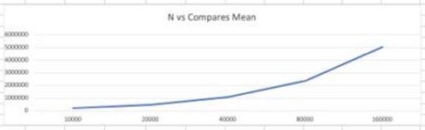
HeapBenchMark

N	Time	hits:Mean	hits:StdDe	hits:Norm	swaps:Me	swaps:Std	swaps:No	compares:	compares:	compares:	fixes:Mean	fixes:StdD	fixes:NormalizedMean
10000	340.997875	967476	0	10.50424	124152	0	1.347963	235434	0	2.556192	7.58E+07	0	822.5351
20000	1503.491374	2095554	0	10.57988	268450	0	1.355331	510877	0	2.579278	3.02E+08	0	1525.805
40000	6796.784334	4510426	0	10.64118	576818	0	1.360852	1101577	0	2.598884	1.21E+09	0	2853.281
80000	27945.96075	9661016	0	10.69664	1233681	0	1.365927	2363146	0	2.616466	5.45E+08	0	603.376
160000	94417.09242	2.06E+07	0	10.74639	2627602	0	1.370492	5046656	0	2.632211	-2.10E+09	0	-1093.03

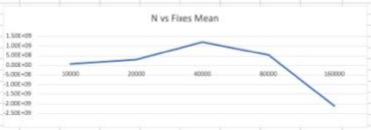






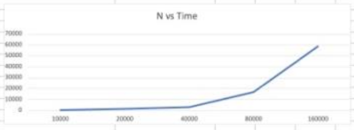


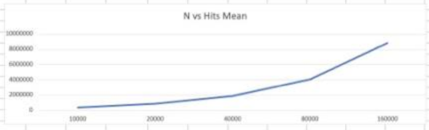
Quick BENCHMARK

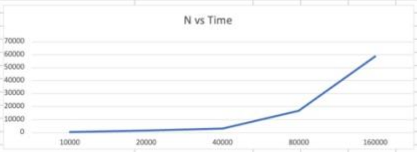


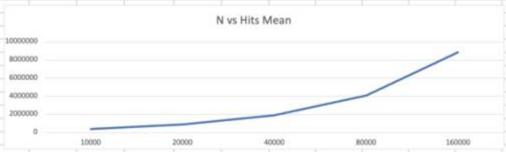
uick Benchmark

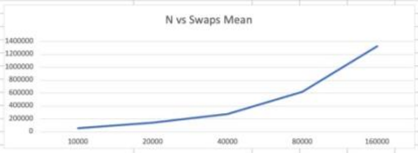
	Time	hits:Mean	hits:StdDe	hits:Norm	swaps:Me	swaps:Std	swaps:No	compares:	compares:	compares:	fixes:Mean	fixes:StdD	fixes:NormalizedMean
10000	268.669417	404760	0	4.394626	61477	0	0.667478	154631	0	1.678885	2.60E+07	0	282.3295
20000	1608.192417	917792	0	4.633681	141874	0	0.716283	341862	0	1.725968	1.33E+08	0	673.4779
40000	3291.927666	1922366	0	4.535322	276910	0	0.653297	797812	0	1.88223	4.21E+08	0	994.0999
80000	16988.87796	4076746	0	4.513756	625184	0	0.692201	1542291	0	1.707618	1.79E+09	0	1985.517
160000	58759.01596	8852071	0	4.617022	1330009	0	0.6937	3463920	0	1.806695	-1.63E+09	0	-852.554

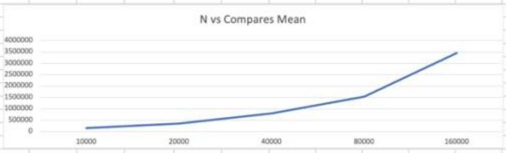


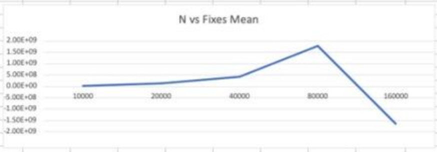



N vs Time
70000
60000
50000
40000
30000
20000
10000
0
10000
20000
40000
80000
160000


N vs Hits Mean
10000000
8000000
6000000
4000000
2000000
0
10000
20000
40000
80000
160000


N vs Swaps Mean
1400000
1200000
1000000
800000
600000
400000
200000
0
10000
20000
40000
80000
160000


N vs Compares Mean
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
10000
20000
40000
80000
160000


N vs Fixes Mean
2.00E+09
1.50E+09
1.00E+09
5.00E+08
0.00E+00
-5.00E+08
-1.00E+09
-1.50E+09
-2.00E+09
10000
20000
40000
80000
160000