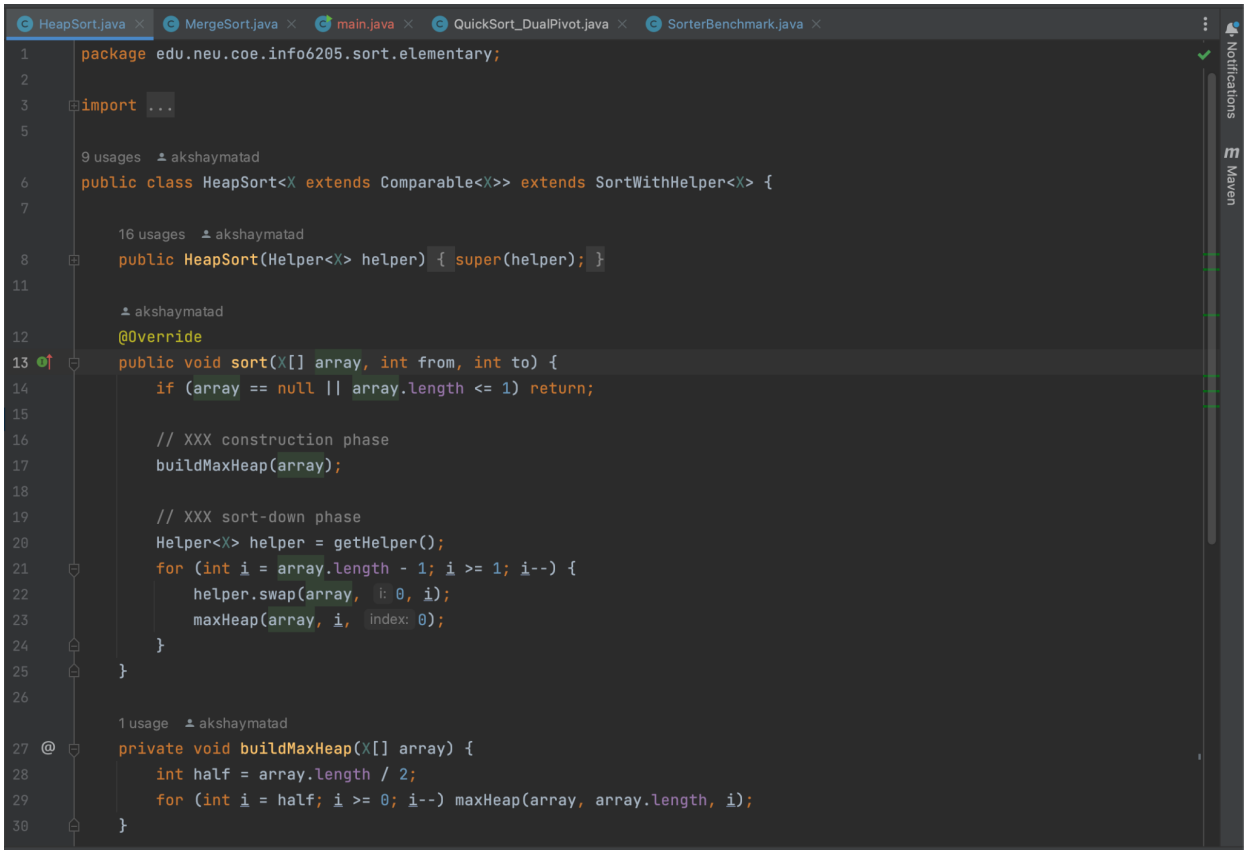
INFO6205 -ASSIGNMENT NO. 6 | SALONI TALWAR(002924067)

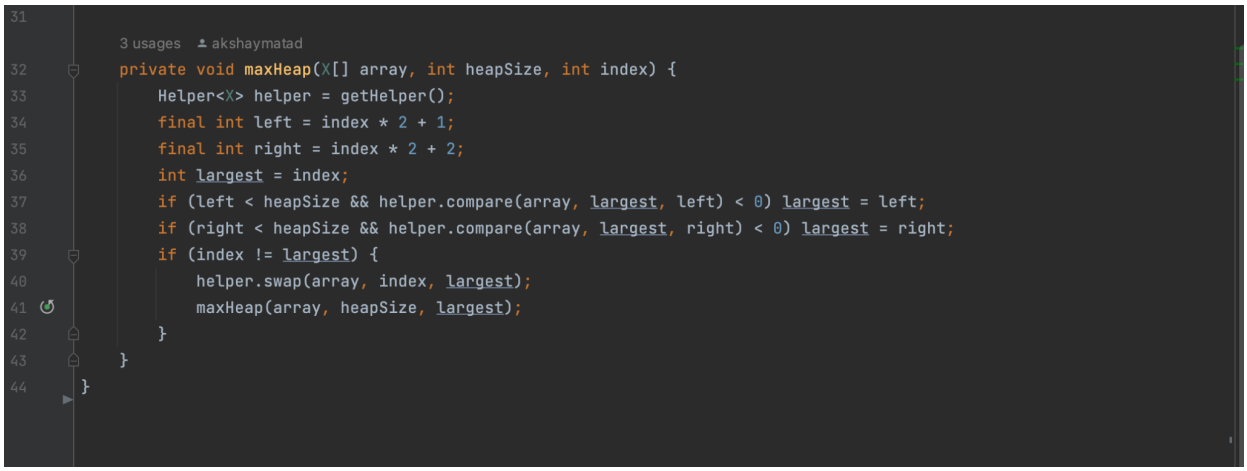
AIM:

To determine which sorting algorithm is the best predictor of total execution time: comparisons, swaps/copies, hits(array access) or something else.

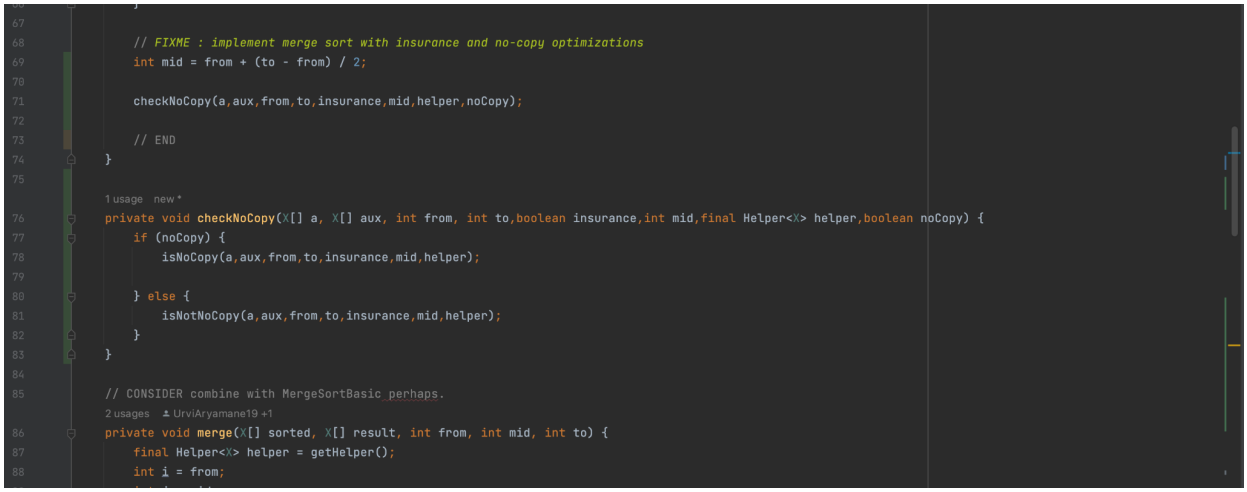
CODE SNAPSHOT:

- HeapSort.java





- MergeSort.java



```
        // FIXME : implement merge sort with insurance and no-copy optimizations
        int mid = from + (to - from) / 2;

        checkNoCopy(a,aux,from,to,insurance,mid,helper,noCopy);

        // END
    }

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

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

    // CONSIDER combine with MergeSortBasic perhaps.
    2 usages   UrviAryamane19 +1
    private void merge(X[] sorted, X[] result, int from, int mid, int to) {
        final Helper<X> helper = getHelper();
        int i = from;
```

- 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 edu.neu.coe.info6205.util.SorterBenchmark;
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( pathname: "HeapBenchMark.csv");
            File fileMerge = new File( pathname: "MergeBenchMark.csv");
            File fileQuick = new File( pathname: "QuickBenchMark.csv");

            fileHeap.createNewFile();
            fileQuick.createNewFile();
            fileMerge.createNewFile();

            FileWriter fileWriterHeap = new FileWriter(fileHeap);
            FileWriter fileWriterMerge = new FileWriter(fileMerge);
            FileWriter fileWriterQuick = new FileWriter(fileQuick);

            fileWriterHeap.write(getHeaderString());
```

```java
            fileWriterHeap.write(getHeaderString());
            fileWriterMerge.write(getHeaderString());
            fileWriterQuick.write(getHeaderString());


            boolean instrumentation = true;


            System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
            Config config = Config.setupConfig( instrumenting: "true",  seed: "",  inversions: "1",  cutoff: "",  interimInversions: "");
            Config no_config = Config.setupConfig( instrumenting: "false",  seed: "",  inversions: "1",  cutoff: "",  interimInversions: "");

            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);

            quickSort.join();
            heapSort.join();
            mergeSort.join();


        } catch (Exception e) {
            System.out.println("error while sorting main" + e);
        }
    }

```

```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( description: "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,  nRuns: 1, timeLoggersLinearithmic);
                    double time = sorterBenchmark.rund(n);
                    try {
                        fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));

                    } 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);
```

```java
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( description: "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,  nRuns: 1, timeLoggersLinearithmic);
                    double time = sorterBenchmark.rund(n);
                    try {
                        if (helper instanceof InstrumentedHelper) {
                            StatPack statPack = ((InstrumentedHelper) helper).getStatPack();
                            fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
                        }



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

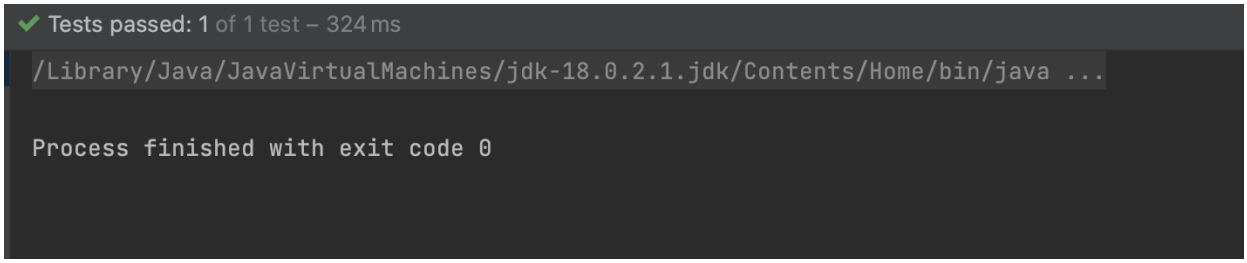
```
                try {
                    fileWriter.flush();
                    fileWriter.close();
                } catch (Exception e) {
                    System.out.println("error while closing file Merge" + e);
                }
            }
    );
}


1 usage
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( description: "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, nRuns: 1, timeLoggersLinearithmic);
                    double time =sorterBenchmark.rund(n);
                    try {
                        if (helper instanceof InstrumentedHelper) {
                                    array[i] = array[i];
                                }
                                return array;
                            },
                            sort, arr, nRuns: 1, timeLoggersLinearithmic);
                      double time =sorterBenchmark.rund(n);
                      try {
                          if (helper instanceof InstrumentedHelper) {
                              StatPack statPack = ((InstrumentedHelper) helper).getStatPack();
                              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);
                  }
              }

     );
}

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

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



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

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

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

    sb.append(statPack.getStatistics( key: "fixes").mean() + ",");
    sb.append(statPack.getStatistics( key: "fixes").stdDev() + ",");
    sb.append(statPack.getStatistics( key: "fixes").normalizedMean() + "\n");

    System.out.println();
    return sb.toString();

}
```

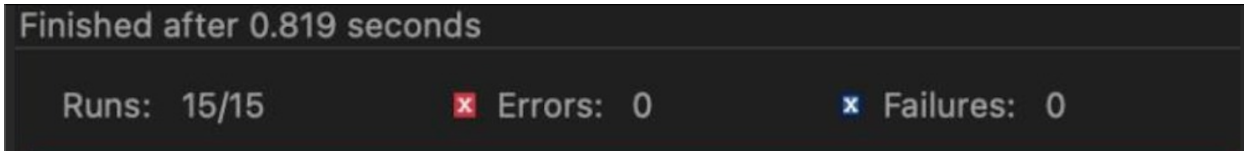
```
3 usages
private static String getHeaderString() {
    StringBuilder sb = new StringBuilder();
    sb.append("N,");
    sb.append("Time,");

    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();

}


}
```

TEST CASES:

- **HeapSort**



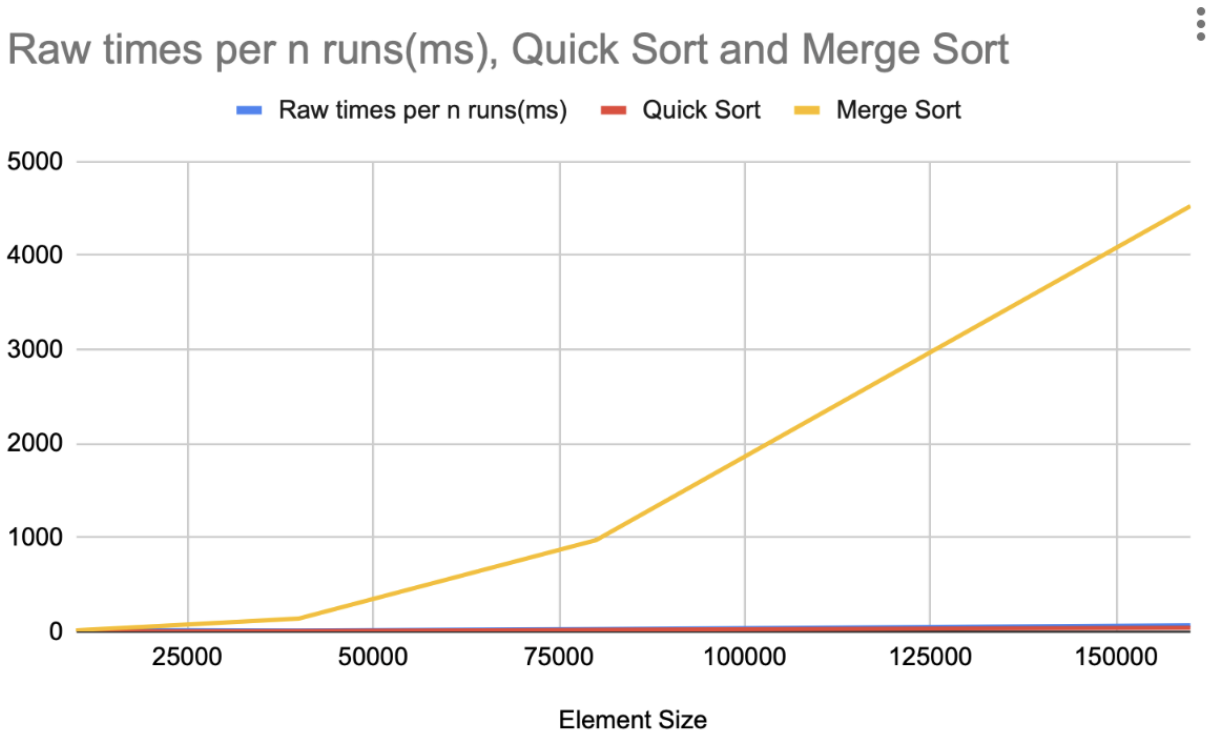
- **MergeSort**



GRAPHS:

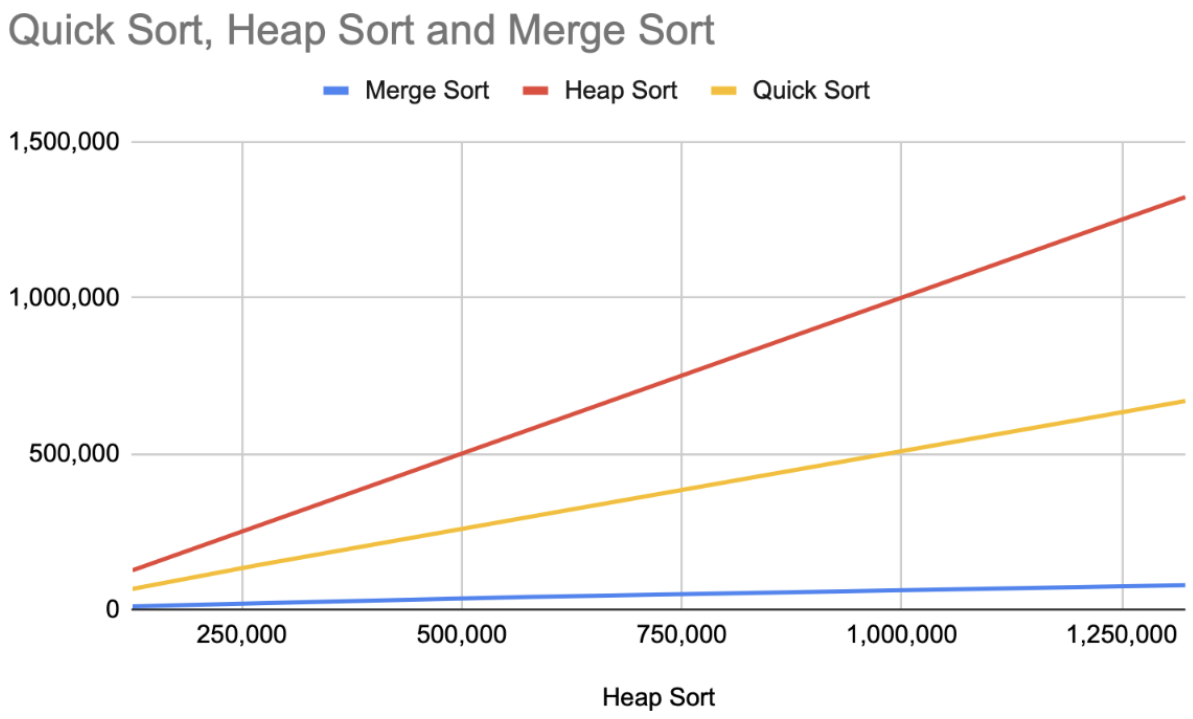
- **TIME**

	TIME		
	Heap Sort	Quick Sort	Merge Sort
Element Size	Raw times per n runs(ms)		
10000	4.62	1.45	14.12
20000	16.82	3.24	54.6
40000	12.96	7.5	139.4
80000	27.1	19.01	973.1
160000	67.36	41.4	4525.8


Raw times per n runs(ms), Quick Sort and Merge Sort
Raw times per n runs(ms)
Quick Sort
Merge Sort
5000
4000
3000
2000
1000
0
25000
50000
75000
100000
125000
150000
Element Size

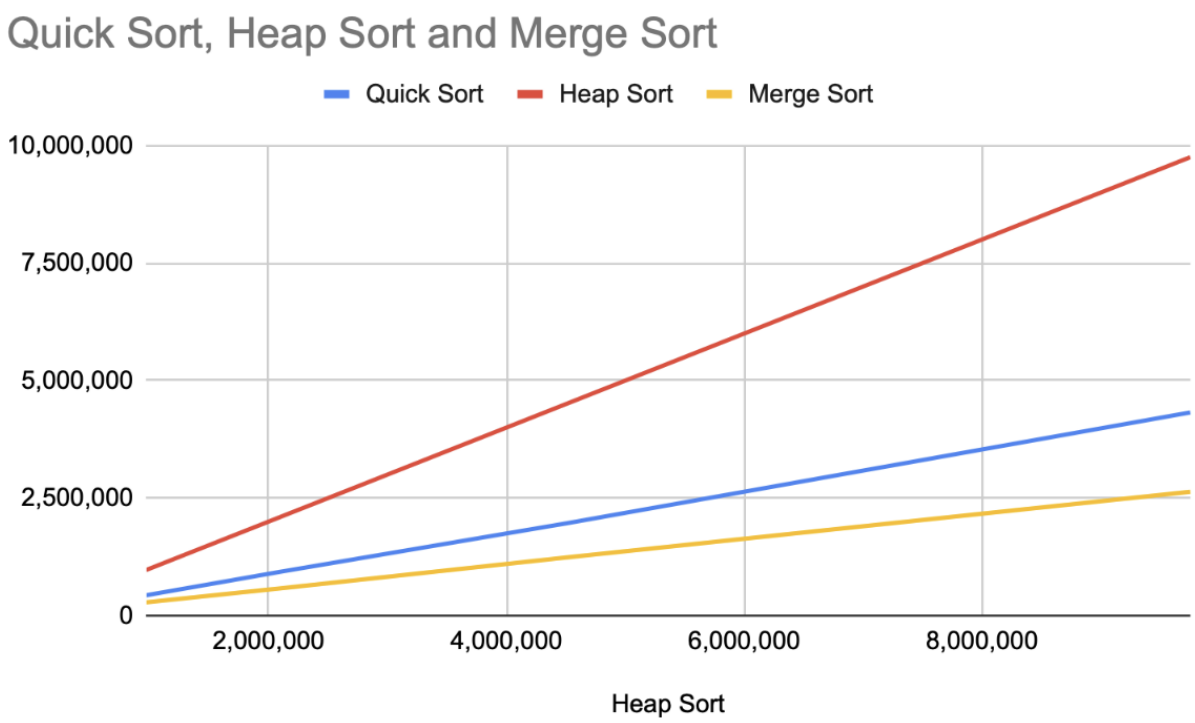
- **SWAPS**

Heap Sort	Quick Sort	Merge Sort	
125,380	65,820	9,713	
268,660	142,521	19,787	
577,510	297,212	40,178	
1,322,279	668,327	77,749	



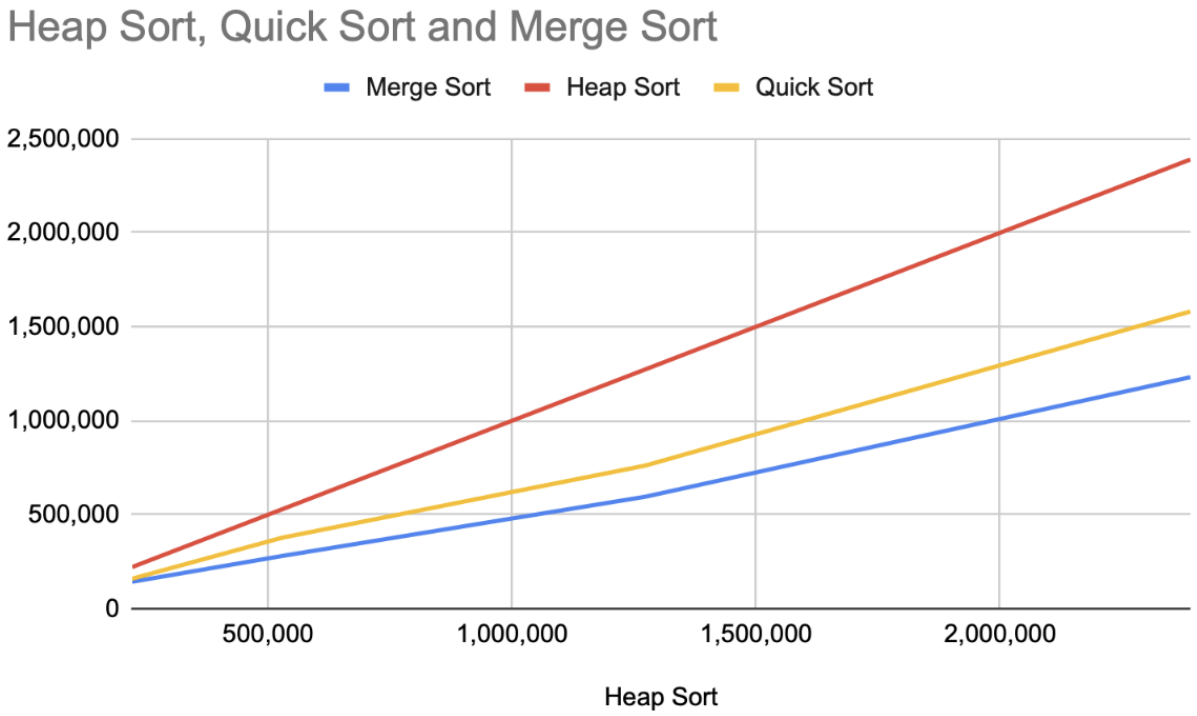
- **HITS**

Heap Sort	Quick Sort	Merge Sort
968,216	431,010	281,200
2,101,178	932,700	579,360
4,524,212	1,971,122	1,241,220
9,750,910	4,320,616	2,632,680



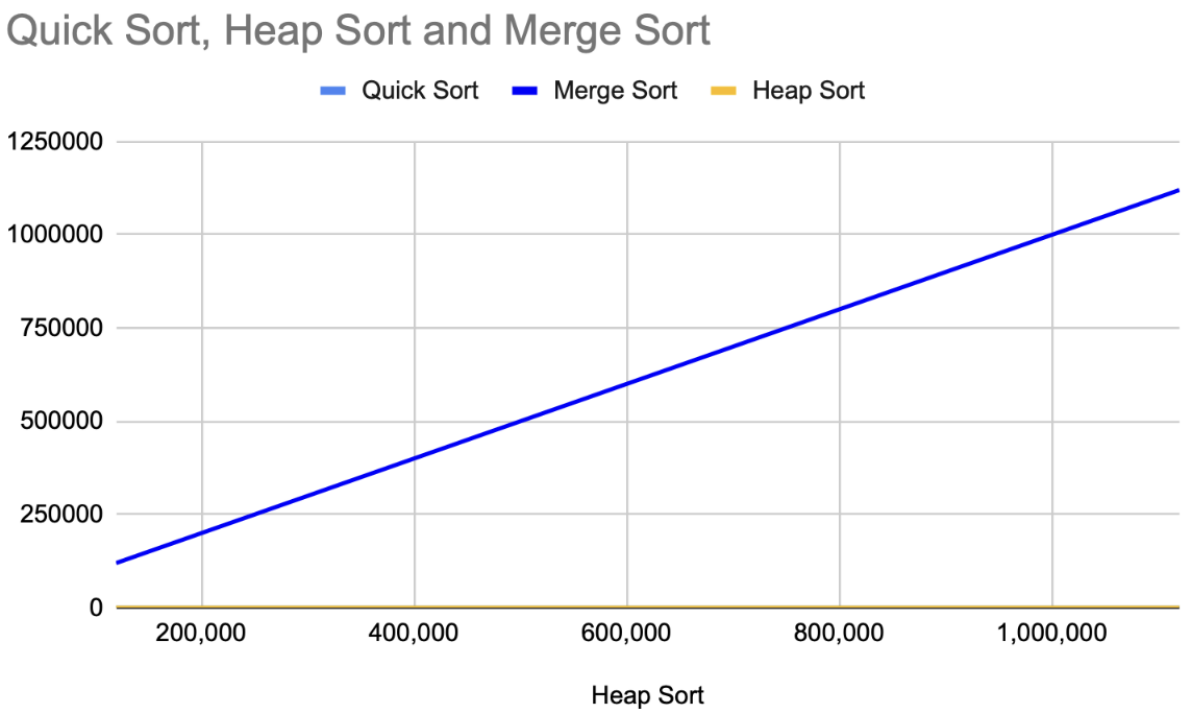
- **COMPARES**

Heap Sort	Quick Sort	Merge Sort
221,232	159,126	142,900
528,471	377,368	280,204
1,272,676	761,100	595,400
2,389,560	1,580,713	1,231,342



- **COPIES**

Heap Sort	Quick Sort	Merge Sort
0	0	120,000
0	0	230,000
0	0	500,000
0	0	1,120,000



CONCLUSION:

The particular properties of the algorithm being run and the hardware platform on which it is operating will determine the best predictor of the entire execution time.

The execution time of an algorithm can be significantly impacted by comparisons, swaps/copies, and hits (array accesses).

As comparisons and swaps/copies are frequently the most time-consuming processes, they may generally be used to estimate how long an algorithm will take to run. For algorithms that use huge datasets or regularly access memory, the quantity of array accesses can be crucial.

The complexity of the method, the size of the input data, the amount of available memory, and the processor speed are other variables that can affect execution time. Performance in some circumstances can also be significantly impacted by the algorithm's unique implementation.

Hence, all of these aspects must be taken into account together with benchmarking and profiling on the particular algorithm and hardware platform in issue in order to accurately anticipate the overall execution time.