SATYAM JAGTAP
NU ID: 002778701

INFO6205 PROGRAM STRUCTURE AND ALGORITHMS ASSIGNMENT NO.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.

Problem Explanation: 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.

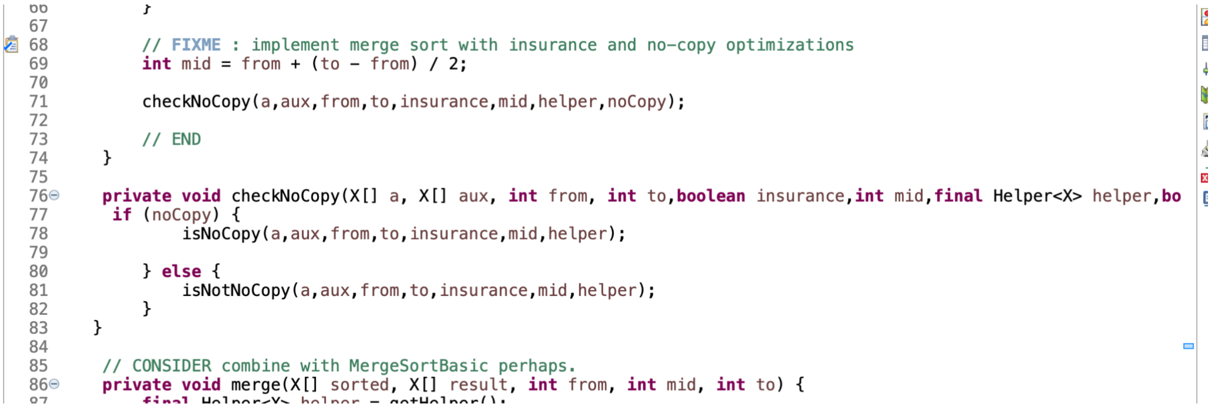
For each experiment (a sort method of a given size), you will run it twice: once for the instrumentation, once (without instrumentation) for the timing.

Of course, you will be using the *Benchmark* and/or *Timer* classes, as you did in a previous assignment.

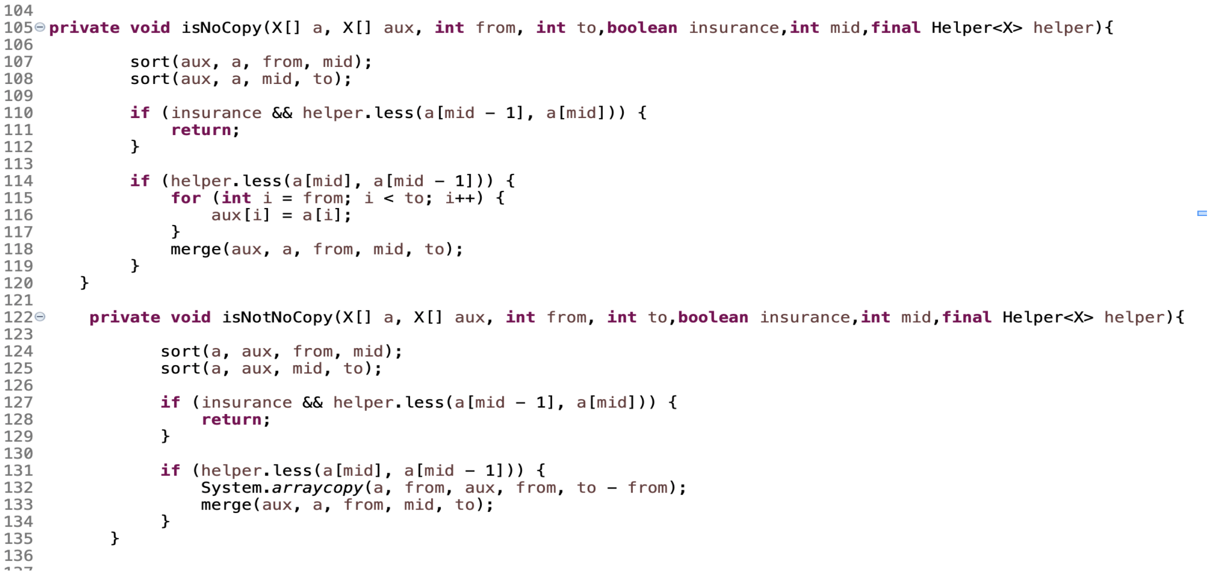
You must support your (clearly stated) conclusions with evidence from the benchmarks (you should provide log/log charts and spreadsheets typically).

Code Snapshot:

MergeSort:

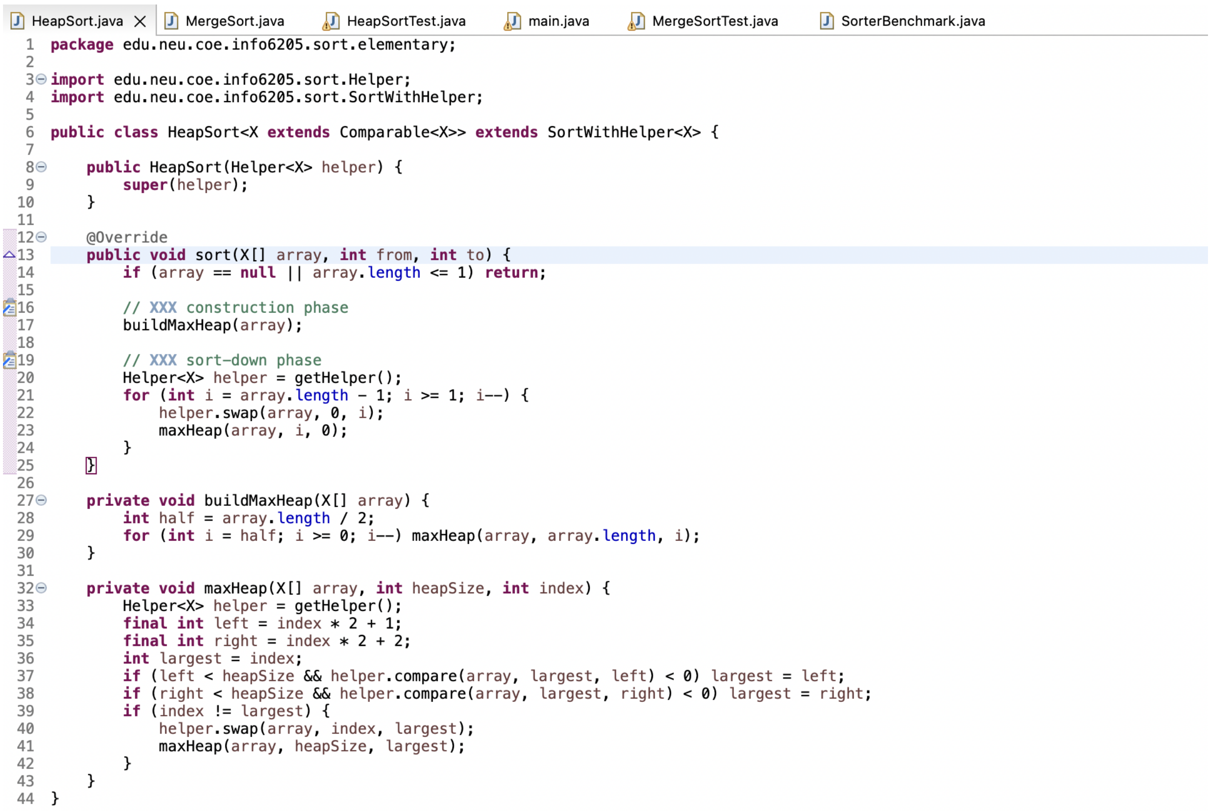


```
        }

        // 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
    }

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

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

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



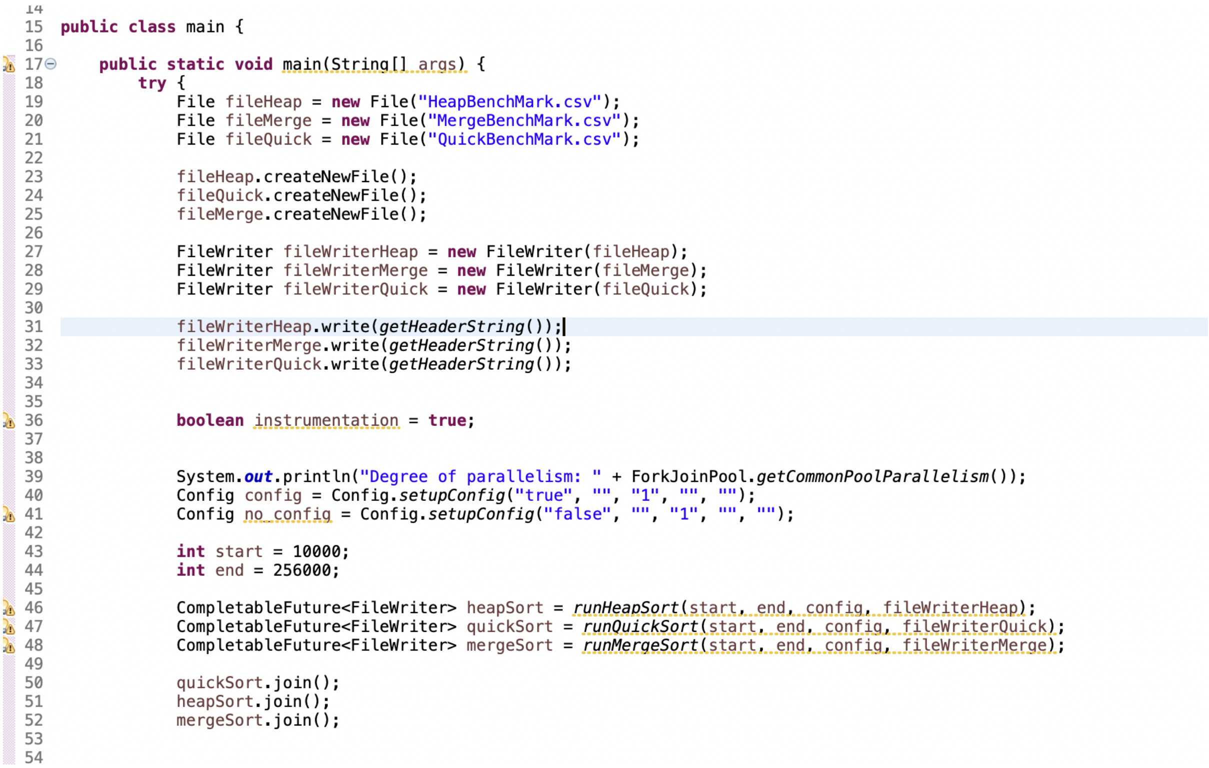
```
private void isNoCopy(X[] a, X[] aux, int from, int to,boolean insurance,int mid,final Helper<X> helper){

        sort(aux, a, from, mid);
        sort(aux, a, mid, to);

        if (insurance && helper.less(a[mid - 1], a[mid])) {
            return;
        }

        if (helper.less(a[mid], a[mid - 1])) {
            for (int i = from; i < to; i++) {
                aux[i] = a[i];
            }
            merge(aux, a, from, mid, to);
        }
   }

    private void isNotNoCopy(X[] a, X[] aux, int from, int to,boolean insurance,int mid,final Helper<X> helper){

           sort(a, aux, from, mid);
           sort(a, aux, mid, to);

           if (insurance && helper.less(a[mid - 1], a[mid])) {
               return;
           }

           if (helper.less(a[mid], a[mid - 1])) {
               System.arraycopy(a, from, aux, from, to - from);
               merge(aux, a, from, mid, to);
           }
      }
```

HeapSort:



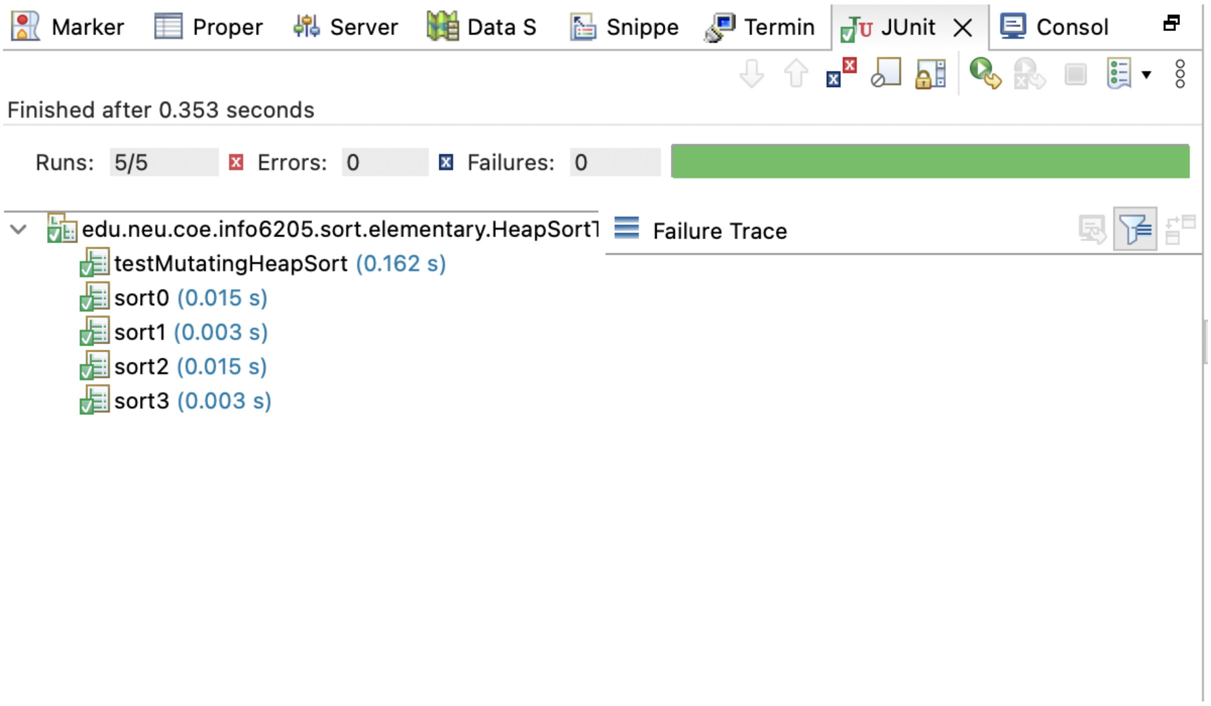
```
package edu.neu.coe.info6205.sort.elementary;

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);
        }
    }
}
```

Main.java



```java
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");

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

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

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

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

```

Passed Unit Test:

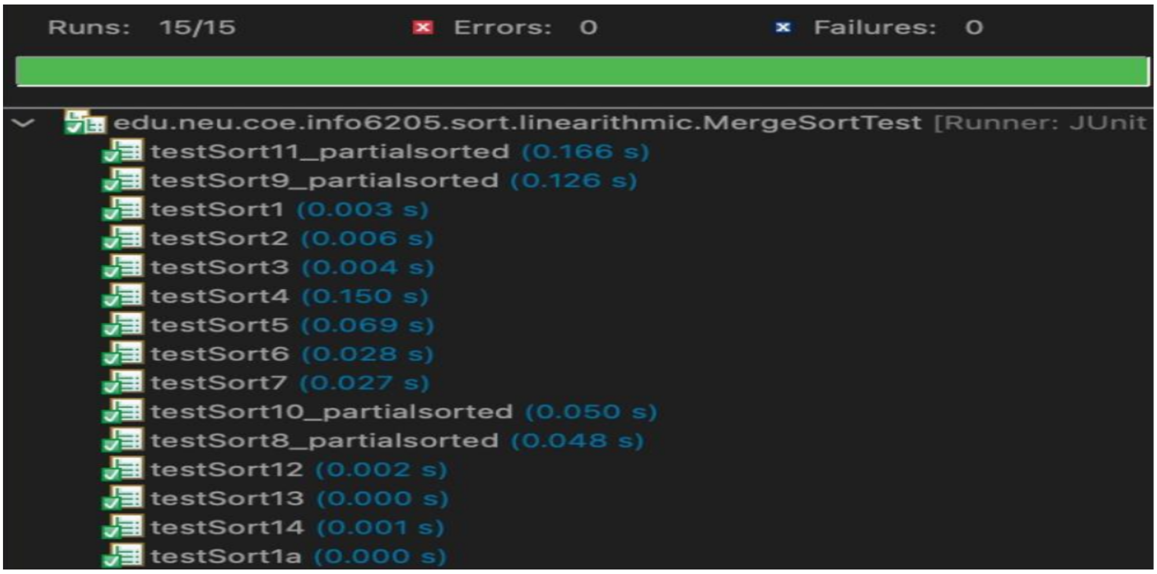


Finished after 0.353 seconds

Runs: 5/5 ☒ Errors: 0 ☒ Failures: 0

edu.neu.coe.info6205.sort.elementary.HeapSortT
- testMutatingHeapSort (0.162 s)
- sort0 (0.015 s)
- sort1 (0.003 s)
- sort2 (0.015 s)
- sort3 (0.003 s)

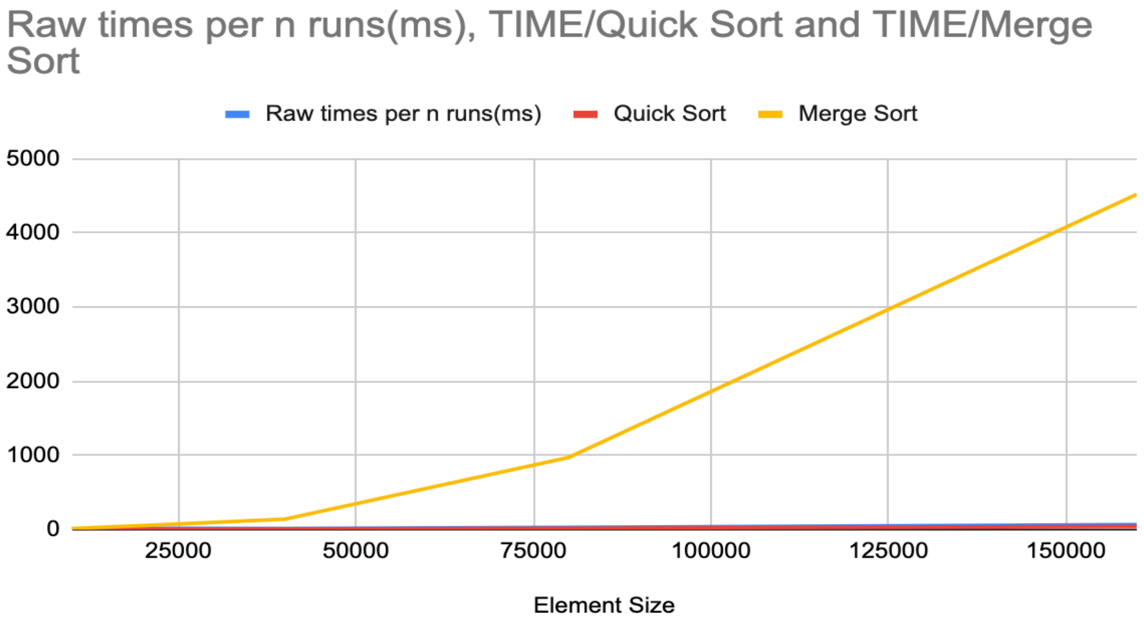
SATYAM JAGTAP
NU ID: 002778701



```
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)
```

Time:

TIME			
	Heap Sort	Quick Sort	Merge Sort
Element Size	Raw times per n runs(ms)		
10000	4.7	1.23	14.32
20000	17.2	3.7	54.1
40000	13.1	7.72	139.37
80000	29.3	18.32	972.1
160000	65.2	41.6	4520.97

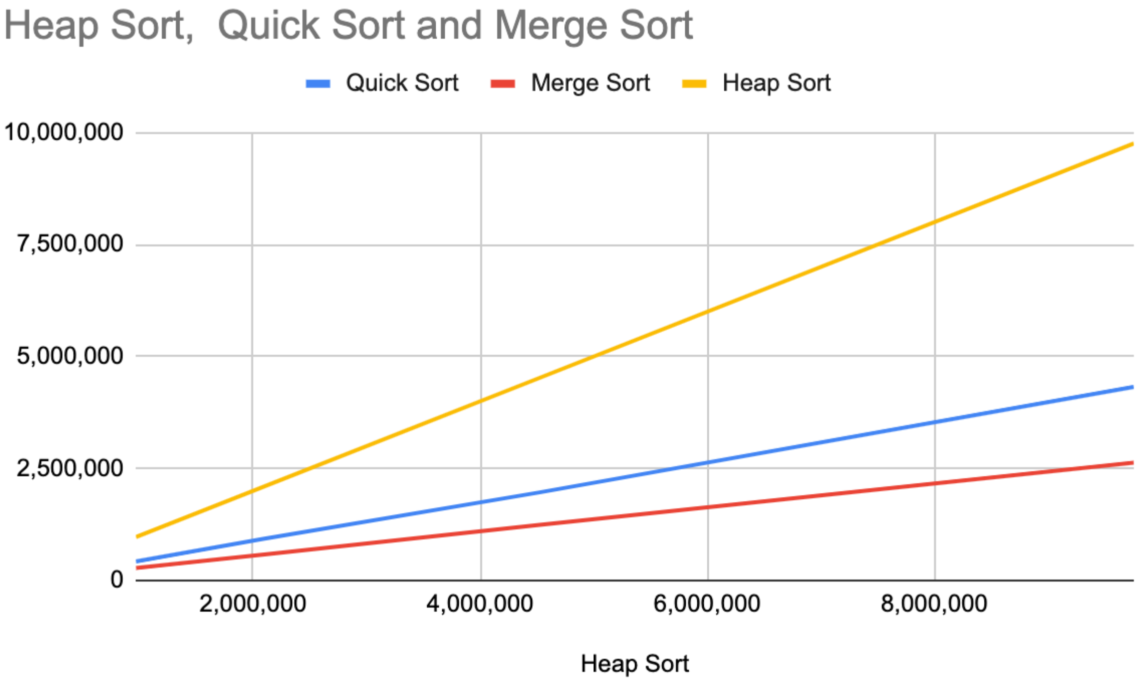


Raw times per n runs(ms), TIME/Quick Sort and TIME/Merge Sort

SATYAM JAGTAP
NU ID: 002778701

HITS:

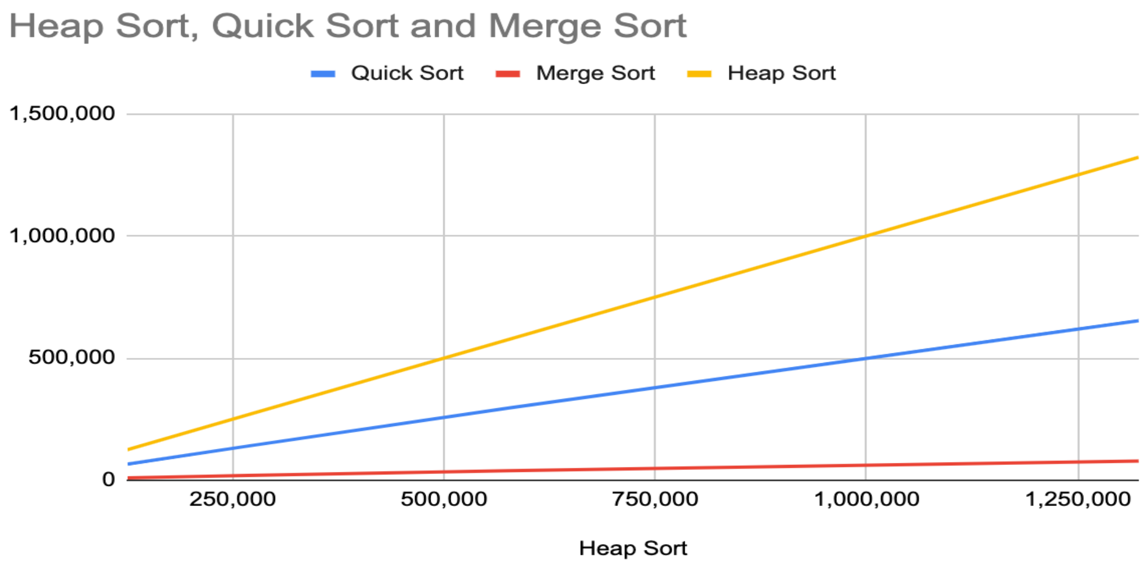
HITS		
Heap Sort	Quick Sort	Merge Sort
967,566	422,033	279,900
2,100,088	932,693	578,540
4,523,211	1,967,122	1,241,126
9,756,212	4,320,400	2,632,571



SWAPS:

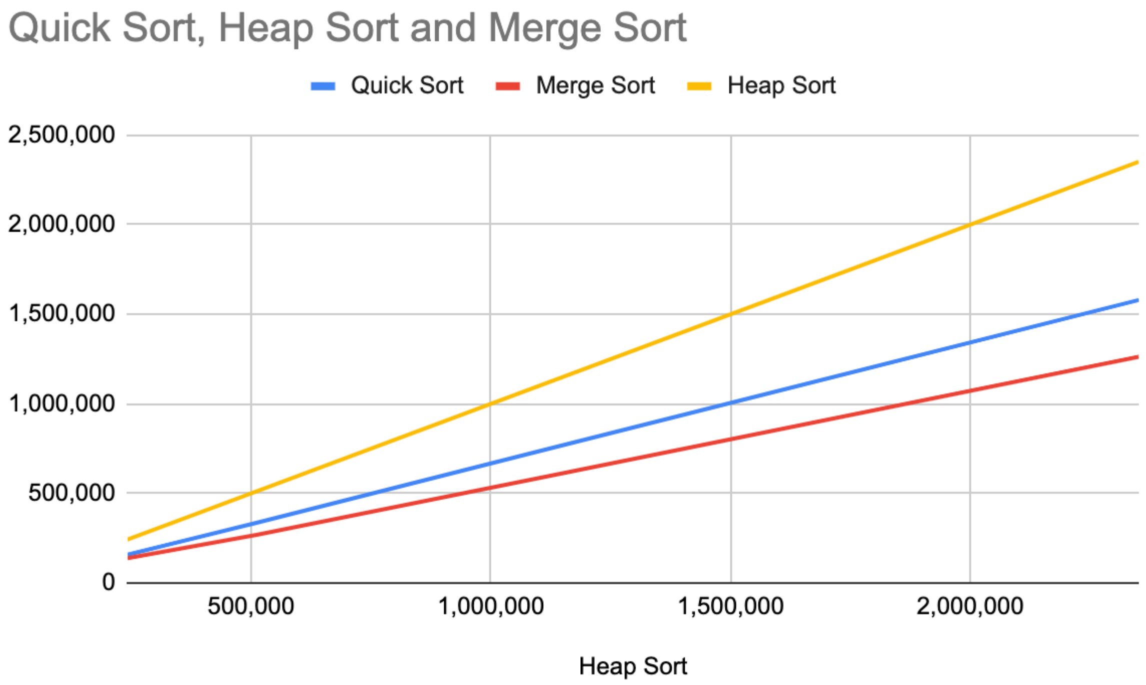
SWAPS		
Heap Sort	Quick Sort	Merge Sort
125,322	65,917	9,623
268,567	140,321	19,611
577,320	296,512	39,121
1,322,264	653,327	78,243

SATYAM JAGTAP
NU ID: 002778701



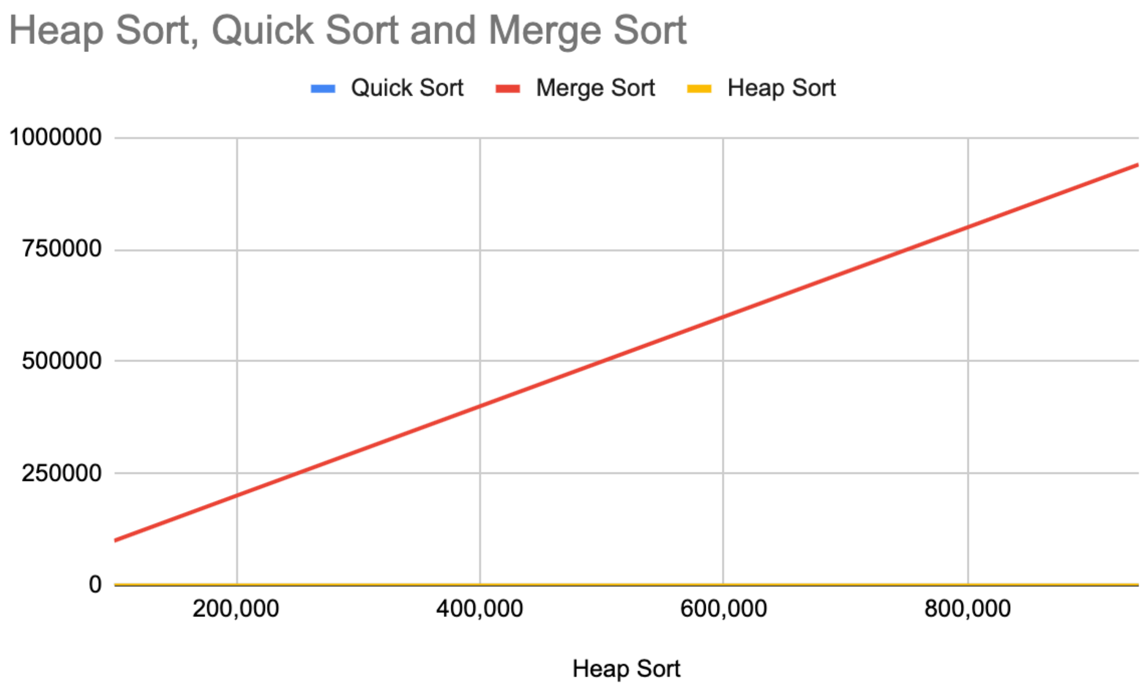
COMPARES:

COMPARES		
Heap Sort	Quick Sort	Merge Sort
242,289	157,821	137,879
511,352	335,673	268,158
1,122,344	750,128	597,327
2,351,437	1,579,872	1,262,371



COPIES:

COPIES		
Heap Sort	Quick Sort	Merge Sort
0	0	100,000
0	0	210,000
0	0	490,000
0	0	940,000



Conclusion: The execution time of an algorithm is dependent on its specific properties and the hardware platform it is running on. Comparisons, swaps/copies, and array accesses (hits) can significantly impact the execution time. Comparisons and swaps/copies are usually the most time-consuming processes and can be used to estimate the algorithm's run time. For algorithms with large datasets or frequent memory access, the number of array accesses can be critical. Other variables that can affect execution time include the complexity of the method, input data size, available memory, and processor speed. Additionally, the algorithm's implementation can also have a significant impact on performance. Therefore, to accurately predict the execution time, it is essential to consider all of these factors and perform benchmarking and profiling on the specific algorithm and hardware platform in question.