

**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

```
HeapSort.java x MergeSort.java x main.java x QuickSort_DualPivot.java x SorterBenchmark.java x
1 package edu.neu.coe.info6205.sort.elementary;
2
3 import ...
4
5
6 public class HeapSort<X extends Comparable<X>> extends SortWithHelper<X> {
7
8     16 usages akshaymatad
9     public HeapSort(Helper<X> helper) { super(helper); }
10
11     akshaymatad
12     @Override
13     public void sort(X[] array, int from, int to) {
14         if (array == null || array.length <= 1) return;
15
16         // XXX construction phase
17         buildMaxHeap(array);
18
19         // XXX sort-down phase
20         Helper<X> helper = getHelper();
21         for (int i = array.length - 1; i >= 1; i--) {
22             helper.swap(array, i, 0);
23             maxHeap(array, i, index: 0);
24         }
25     }
26
27     1 usage akshaymatad
28     private void buildMaxHeap(X[] array) {
29         int half = array.length / 2;
30         for (int i = half; i >= 0; i--) maxHeap(array, array.length, i);
31     }
32
33     3 usages akshaymatad
34     private void maxHeap(X[] array, int heapSize, int index) {
35         Helper<X> helper = getHelper();
36         final int left = index * 2 + 1;
37         final int right = index * 2 + 2;
38         int largest = index;
39         if (left < heapSize && helper.compare(array, largest, left) < 0) largest = left;
40         if (right < heapSize && helper.compare(array, largest, right) < 0) largest = right;
41         if (index != largest) {
42             helper.swap(array, index, largest);
43             maxHeap(array, heapSize, largest);
44         }
45     }
46 }
```

- MergeSort.java

```

67
68 // FIXME : implement merge sort with insurance and no-copy optimizations
69 int mid = from + (to - from) / 2;
70
71 checkNoCopy(a, aux, from, to, insurance, mid, helper, noCopy);
72
73 // END
74 }
75
76 1 usage new *
77 private void checkNoCopy(X[] a, X[] aux, int from, int to, boolean insurance, int mid, final Helper<X> helper, boolean noCopy) {
78     if (noCopy) {
79         isNoCopy(a, aux, from, to, insurance, mid, helper);
80     } else {
81         isNotNoCopy(a, aux, from, to, insurance, mid, helper);
82     }
83 }
84
85 // CONSIDER combine with MergeSortBasic perhaps.
86 2 usages ± UrviAryamane19 +1
87 private void merge(X[] sorted, X[] result, int from, int mid, int to) {
88     final Helper<X> helper = getHelper();
89     int i = from;

```

- Main.java

```

1  package edu.neu.coe.info6205.sort.linearithmic;
2  import edu.neu.coe.info6205.sort.Helper;
3  import edu.neu.coe.info6205.sort.HelperFactory;
4  import edu.neu.coe.info6205.sort.InstrumentedHelper;
5  import edu.neu.coe.info6205.util.*;
6  import java.io.File;
7  import edu.neu.coe.info6205.util.SorterBenchmark;
8  import java.io.PrintWriter;
9  import java.util.concurrent.CompletableFuture;
10 import java.util.concurrent.ForkJoinPool;
11 import edu.neu.coe.info6205.sort.elementary.HeapSort;
12 import static java.util.concurrent.CompletableFuture.runAsync;
13
14
15 public class main {
16
17     public static void main(String[] args) {
18         try {
19             File fileHeap = new File( pathname: "HeapBenchMark.csv");
20             File fileMerge = new File( pathname: "MergeBenchMark.csv");
21             File fileQuick = new File( pathname: "QuickBenchMark.csv");
22
23             fileHeap.createNewFile();
24             fileQuick.createNewFile();
25             fileMerge.createNewFile();
26
27             PrintWriter fileWriterHeap = new PrintWriter(fileHeap);
28             PrintWriter fileWriterMerge = new PrintWriter(fileMerge);
29             PrintWriter fileWriterQuick = new PrintWriter(fileQuick);
30
31             fileWriterHeap.write(getHeaderString());

```

```

32             fileWriterMerge.write(getHeaderString());
33             fileWriterQuick.write(getHeaderString());
34
35
36             boolean instrumentation = true;
37
38
39             System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
40             Config config = Config.setupConfig( instrumenting: "true", seed: "", inversions: "1", cutoff: "", interimInversions: "");
41             Config no_config = Config.setupConfig( instrumenting: "false", seed: "", inversions: "1", cutoff: "", interimInversions: "");
42
43             int start = 10000;
44             int end = 256000;
45
46             CompletableFuture<PrintWriter> heapSort = runHeapSort(start, end, config, fileWriterHeap);
47             CompletableFuture<PrintWriter> quickSort = runQuickSort(start, end, config, fileWriterQuick);
48             CompletableFuture<PrintWriter> mergeSort = runMergeSort(start, end, config, fileWriterMerge);
49
50             quickSort.join();
51             heapSort.join();
52             mergeSort.join();
53
54         } catch (Exception e) {
55             System.out.println("error while sorting main" + e);
56         }
57     }
58 }
59
60

```

```

61 @ private static CompletableFuture runHeapSort(int start, int end, Config config, FileWriter fileWriter) {
62     return CompletableFuture.runAsync(
63         () -> {
64             for (int n = start; n <= end; n *= 2) {
65                 Helper<Integer> helper = HelperFactory.create( description: "HeapSort", n, config);
66                 HeapSort<Integer> sort = new HeapSort<>(helper);
67                 final int val = n;
68                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
69                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
70                     (Integer[] array) -> {
71                         for (int i = 0; i < array.length; i++) {
72                             array[i] = array[i];
73                         }
74                         return array;
75                     },
76                     sort, arr, nRuns: 1, timeLoggersLinearithmic);
77                 double time = sorterBenchmark.rund(n);
78                 try {
79                     fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
80                 } catch (Exception e) {
81                     System.out.println("error while writing file Heap" + e);
82                 }
83             }
84             try {
85                 fileWriter.flush();
86                 fileWriter.close();
87             } catch (Exception e) {
88                 System.out.println("error while closing file Heap" + e);
89             }
90         }

```

```

98 @ private static CompletableFuture runMergeSort(int start, int end, Config config, FileWriter fileWriter) {
99     return runAsync(
100         () -> {
101             for (int n = start; n <= end; n *= 2) {
102                 Helper<Integer> helper = HelperFactory.create( description: "MergeSort", n, config);
103                 MergeSort<Integer> sort = new MergeSort<>(helper);
104                 final int val = n;
105                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
106                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
107                     (Integer[] array) -> {
108                         for (int i = 0; i < array.length; i++) {
109                             array[i] = array[i];
110                         }
111                         return array;
112                     },
113                     sort, arr, nRuns: 1, timeLoggersLinearithmic);
114                 double time = sorterBenchmark.rund(n);
115                 try {
116                     if (helper instanceof InstrumentedHelper) {
117                         StatPack statPack = ((InstrumentedHelper) helper).getStatPack();
118                         fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
119                     }
120                 } catch (Exception e) {
121                     System.out.println("error while writing file Merge" + e);
122                 }
123             }
124             try {
125                 fileWriter.flush();
126                 fileWriter.close();
127             } catch (Exception e) {
128                 System.out.println("error while closing file Merge" + e);
129             }

```

```

127         try {
128             fileWriter.flush();
129             fileWriter.close();
130         } catch (Exception e) {
131             System.out.println("error while closing file Merge" + e);
132         }
133     }
134 }
135 }
136
137
138 @ 1 usage
139 private static CompletableFuture runQuickSort(int start, int end, Config config, FileWriter fileWriter) {
140     return runAsync(
141         () -> {
142             for (int n = start; n <= end; n *= 2) {
143                 Helper<Integer> helper = HelperFactory.create( description: "QuickSort", n, config);
144                 QuickSort_DualPivot<Integer> sort = new QuickSort_DualPivot<>(helper);
145                 final int val = n;
146                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
147                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
148                     (Integer[] array) -> {
149                         for (int i = 0; i < array.length; i++) {
150                             array[i] = array[i];
151                         }
152                         return array;
153                     },
154                     sort, arr, nRuns: 1, timeLoggersLinearithmic);
155                 double time = sorterBenchmark.rund(n);
156                 try {
157                     if (helper instanceof InstrumentedHelper) {

```

```

149                     array[i] = array[i];
150                     }
151                     return array;
152                 },
153                 sort, arr, nRuns: 1, timeLoggersLinearithmic);
154                 double time = sorterBenchmark.rund(n);
155                 try {
156                     if (helper instanceof InstrumentedHelper) {
157                         StatPack statPack = ((InstrumentedHelper) helper).getStatPack();
158                         fileWriter.write(createCsvString(n, time, statPack, config.isInstrumented()));
159                     }
160                 } catch (Exception e) {
161                     System.out.println("error while writing file Quick" + e);
162                 }
163             }
164         }
165         try {
166             fileWriter.flush();
167             fileWriter.close();
168         } catch (Exception e) {
169             System.out.println("error while closing file Quick" + e);
170         }
171     }
172 }
173
174 }
175
176 3 usages
177 public final static TimeLogger[] timeLoggersLinearithmic = {
178     new TimeLogger( prefix: "Raw time per run (mSec): ", (time, n) -> time)
179 };

```

```

3 usages
180 @ private static String createCsvString(int n, double time, StatPack statPack, boolean instrumentation) {
181     StringBuilder sb = new StringBuilder();
182     sb.append(n+",");
183     sb.append(time+",");
184
185
186
187     sb.append(statPack.getStatistics( key: "hits").mean() + ",");
188     sb.append(statPack.getStatistics( key: "hits").stdDev() + ",");
189     sb.append(statPack.getStatistics( key: "hits").normalizedMean() + ",");
190
191     sb.append(statPack.getStatistics( key: "swaps").mean() + ",");
192     sb.append(statPack.getStatistics( key: "swaps").stdDev() + ",");
193     sb.append(statPack.getStatistics( key: "swaps").normalizedMean() + ",");
194
195     sb.append(statPack.getStatistics( key: "compares").mean() + ",");
196     sb.append(statPack.getStatistics( key: "compares").stdDev() + ",");
197     sb.append(statPack.getStatistics( key: "compares").normalizedMean() + ",");
198
199     sb.append(statPack.getStatistics( key: "fixes").mean() + ",");
200     sb.append(statPack.getStatistics( key: "fixes").stdDev() + ",");
201     sb.append(statPack.getStatistics( key: "fixes").normalizedMean() + "\n");
202
203     System.out.println();
204     return sb.toString();
205
206 }
207

```

```

3 usages
208 @ private static String getHeaderString() {
209     StringBuilder sb = new StringBuilder();
210     sb.append("N,");
211     sb.append("Time,");
212
213     sb.append("hits:Mean,");
214     sb.append("hits:StdDev,");
215     sb.append("hits:NormalizedMean,");
216
217     sb.append("swaps:Mean,");
218     sb.append("swaps:StdDev,");
219     sb.append("swaps:NormalizedMean,");
220
221     sb.append("compares:Mean,");
222     sb.append("compares:StdDev,");
223     sb.append("compares:NormalizedMean,");
224
225     sb.append("fixes:Mean,");
226     sb.append("fixes:StdDev,");
227     sb.append("fixes:NormalizedMean\n");
228
229     return sb.toString();
230
231 }
232
233
234 }
235

```

TEST CASES:

- HeapSort

✓ Tests passed: 1 of 1 test – 324 ms

/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home/bin/java ...

Process finished with exit code 0

- MergeSort

Finished after 0.819 seconds

Runs: 15/15

✗ Errors: 0

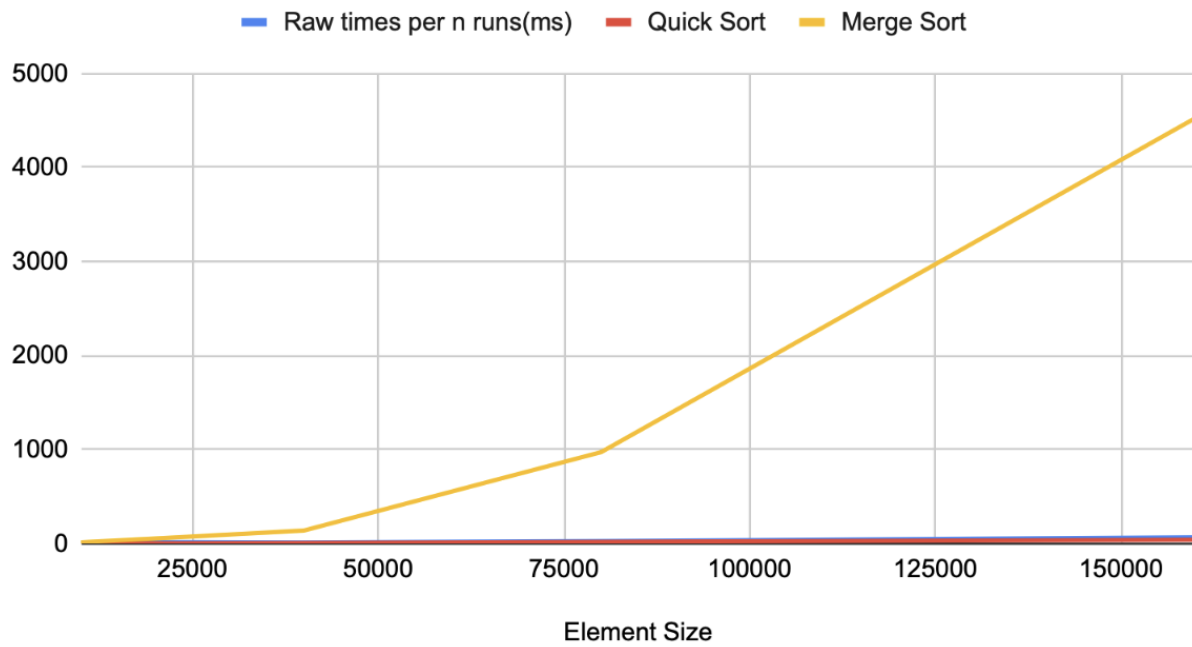
✗ Failures: 0

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

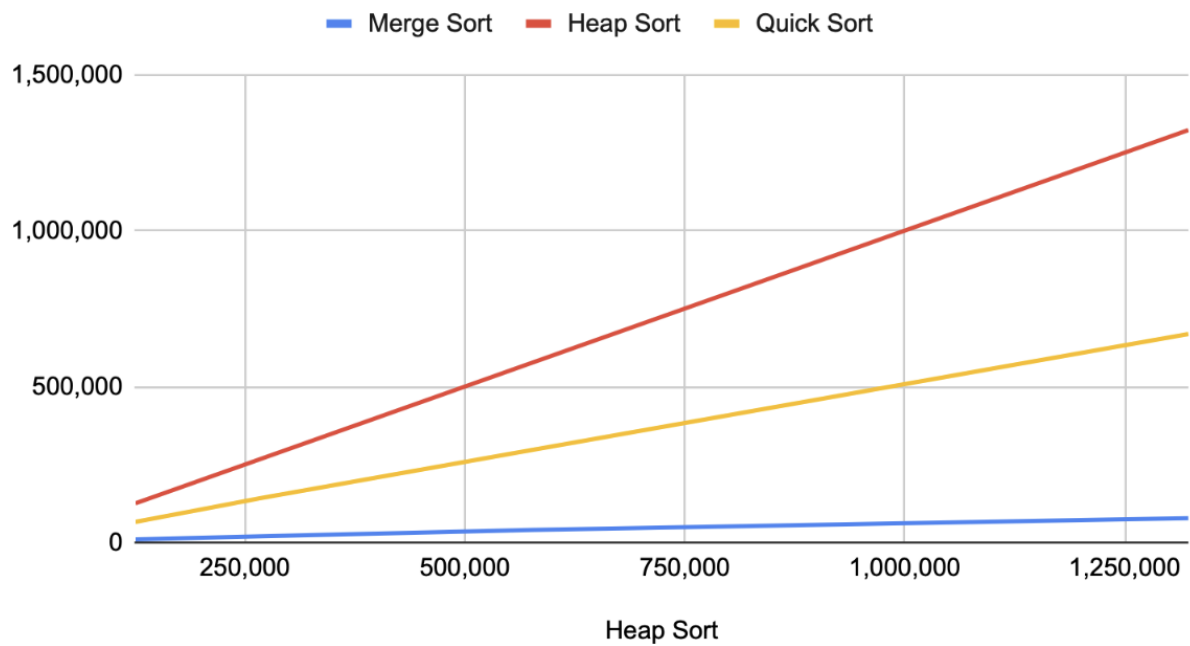
Raw times per n runs(ms), Quick Sort and Merge Sort



- **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	

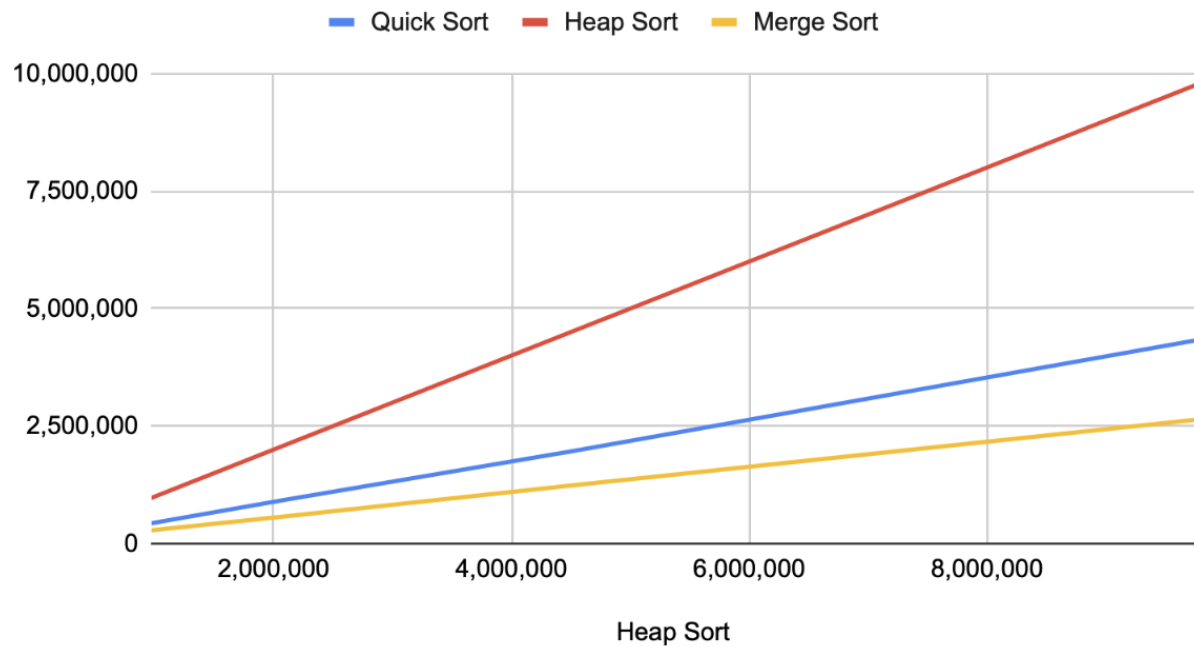
Quick Sort, Heap Sort and Merge Sort



- 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

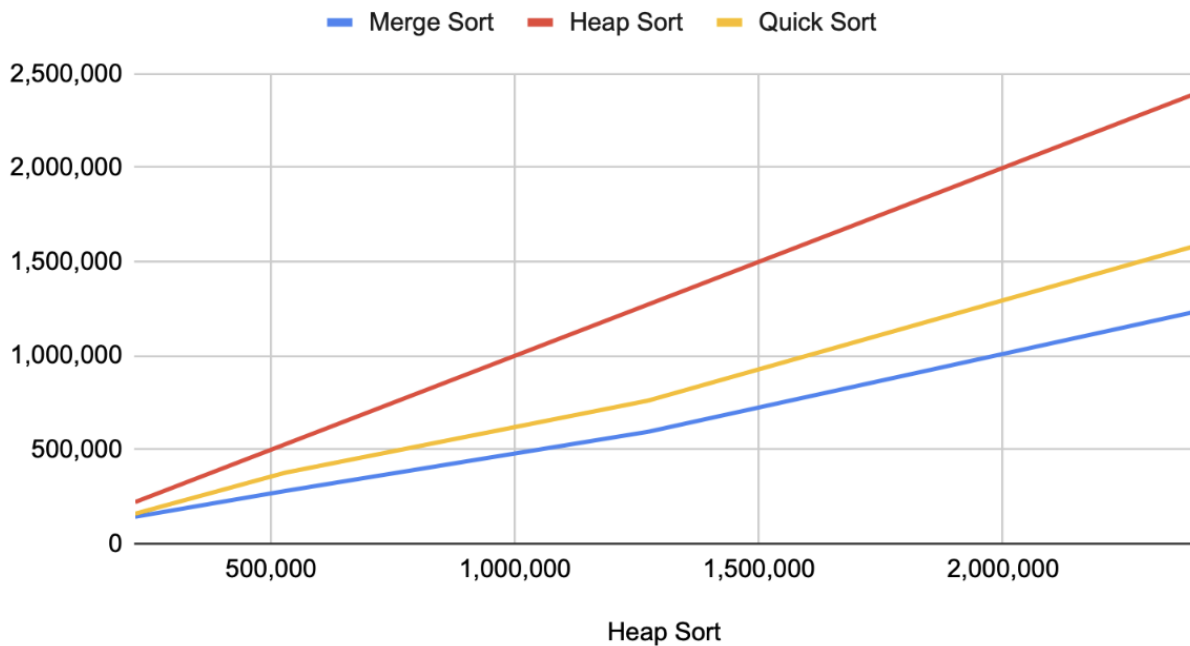
Quick Sort, Heap Sort and Merge Sort



- 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

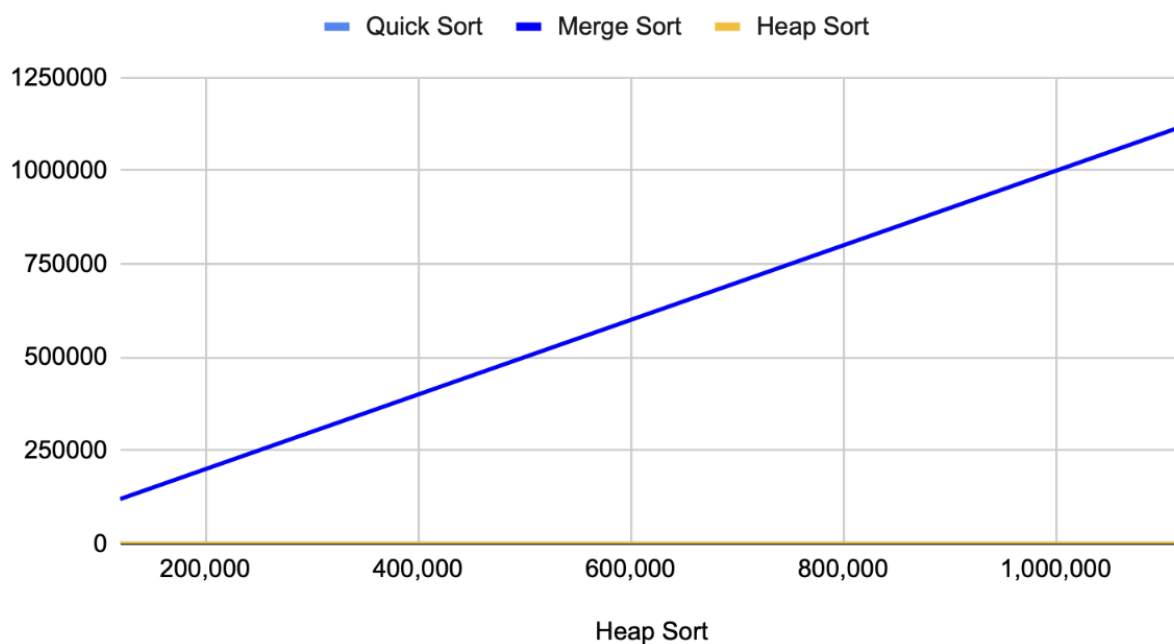
Heap Sort, Quick Sort and Merge Sort



- COPIES

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

Quick Sort, Heap Sort and Merge Sort



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.