Assignment 6 - Program Structures and Algorithms

Spring 2023

Section - 01

NAME: Naman Diwan **NUID: 002724115**

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):

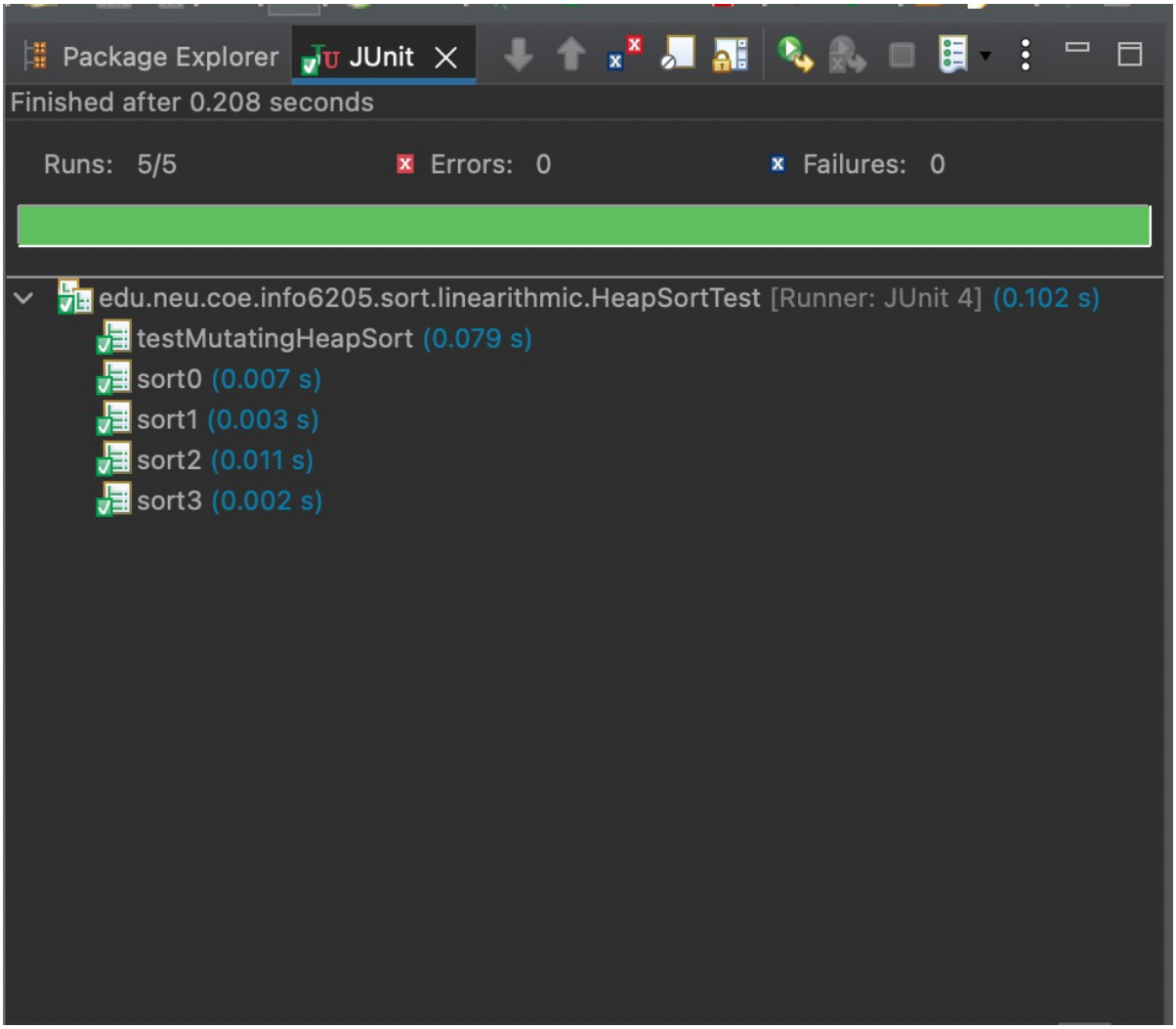
```
heapsort = true
```

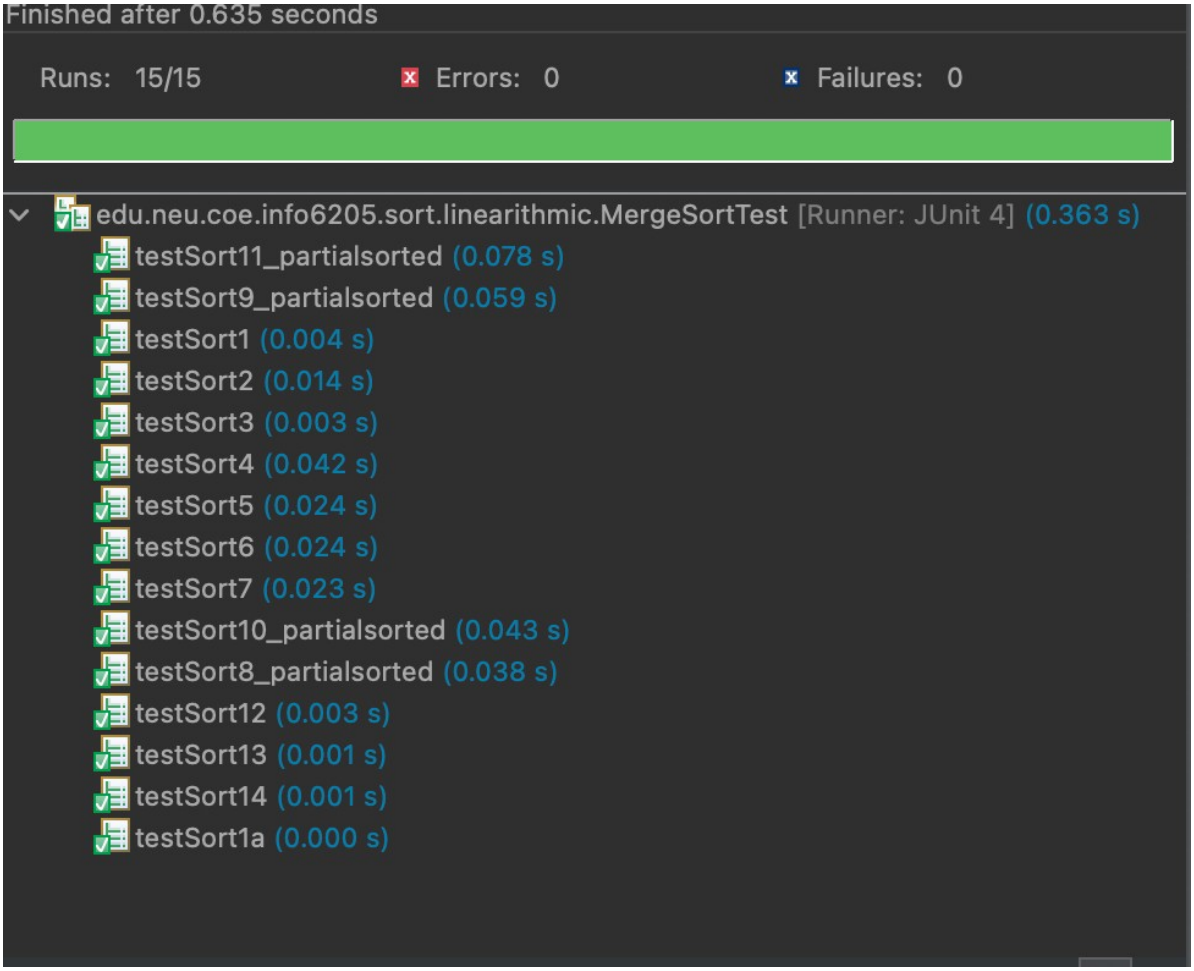
Remember that your job is to determine the best predictor: that will mean the graph of the appropriate observation will match the graph of the timings most closely.

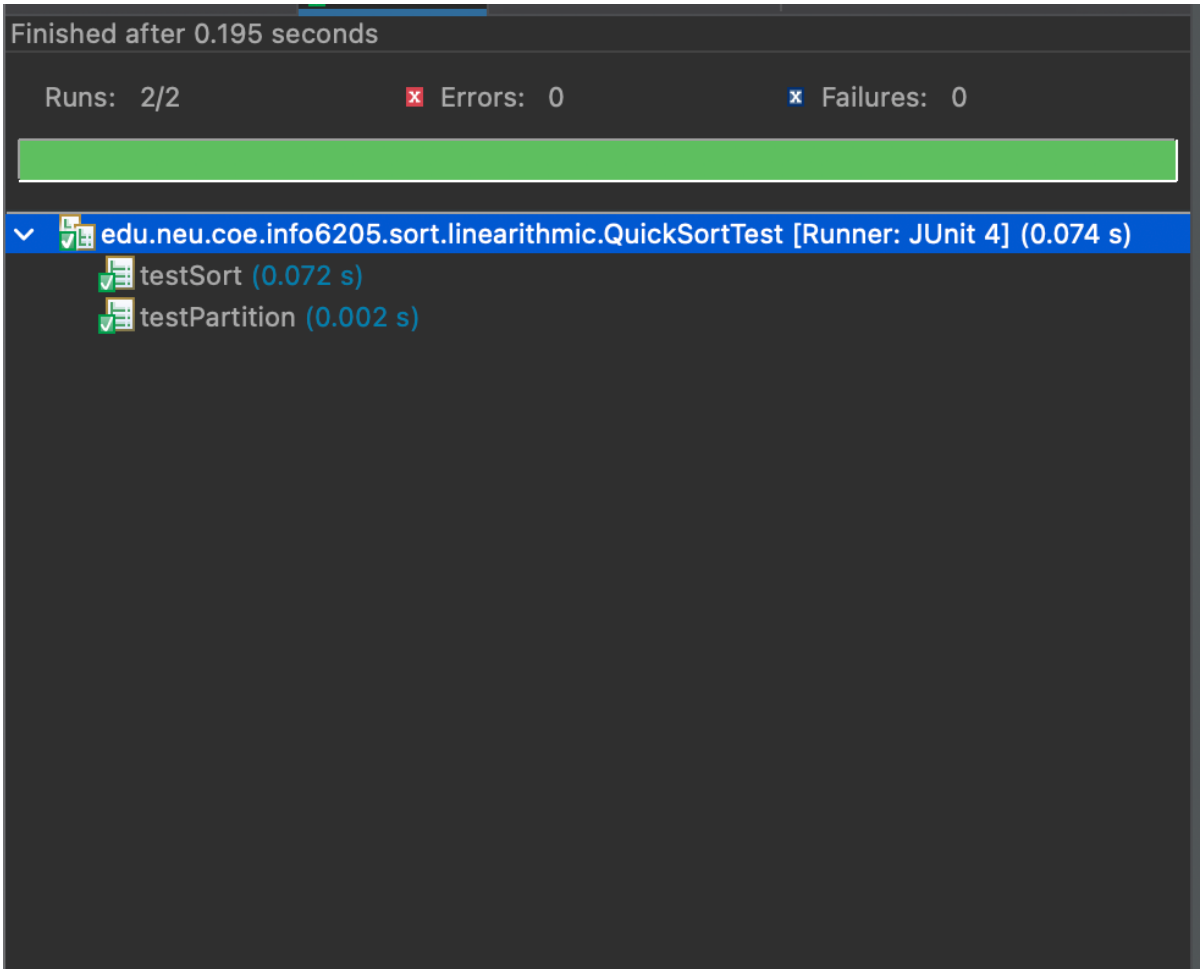
Solution –

Code uploaded to - https://github.com/namandiwan10/INFO-6205-Assignment

Unit Test –

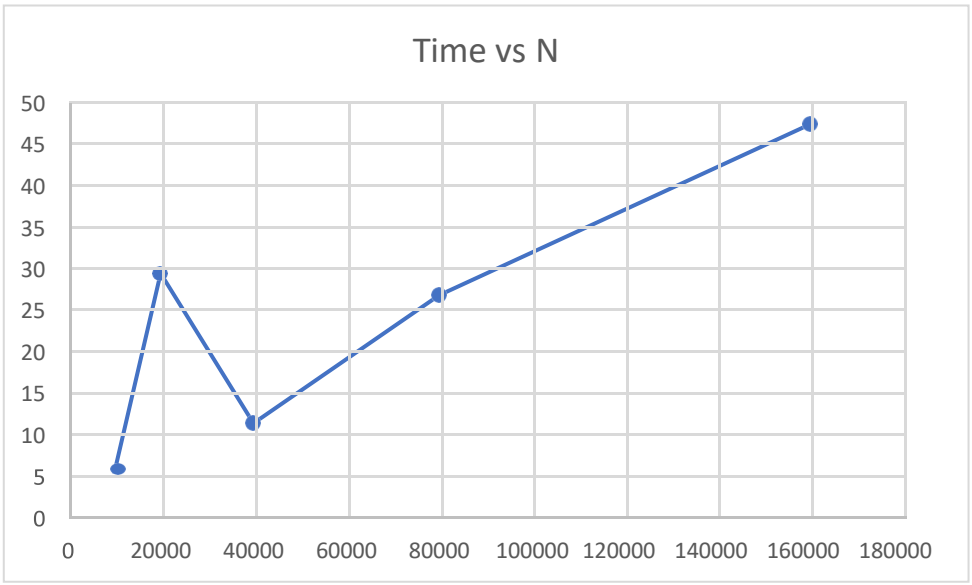


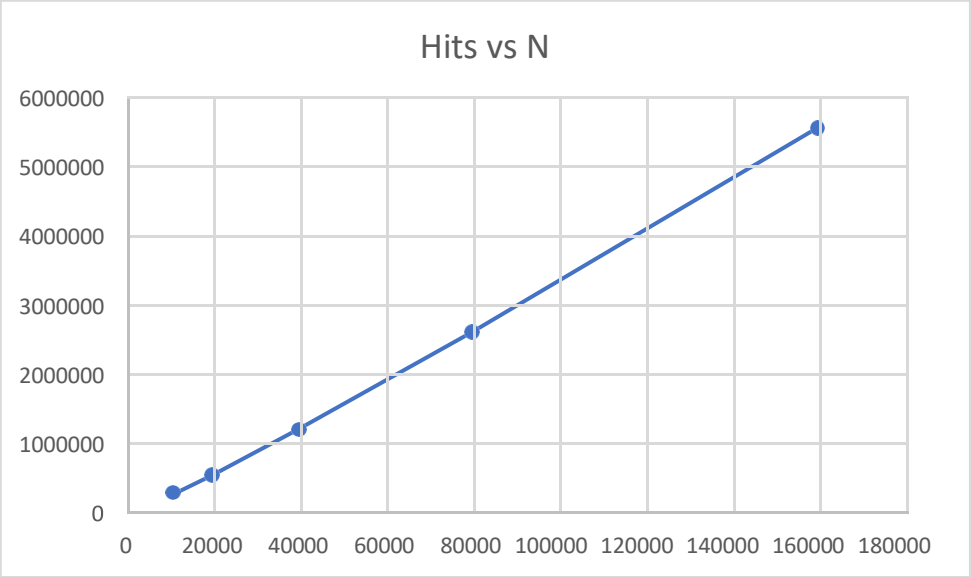

Finished after 0.635 seconds
Runs: 15/15
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.MergeSortTest [Runner: JUnit 4] (0.363 s)
testSort11_partialsorted (0.078 s)
testSort9_partialsorted (0.059 s)
testSort1 (0.004 s)
testSort2 (0.014 s)
testSort3 (0.003 s)
testSort4 (0.042 s)
testSort5 (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)

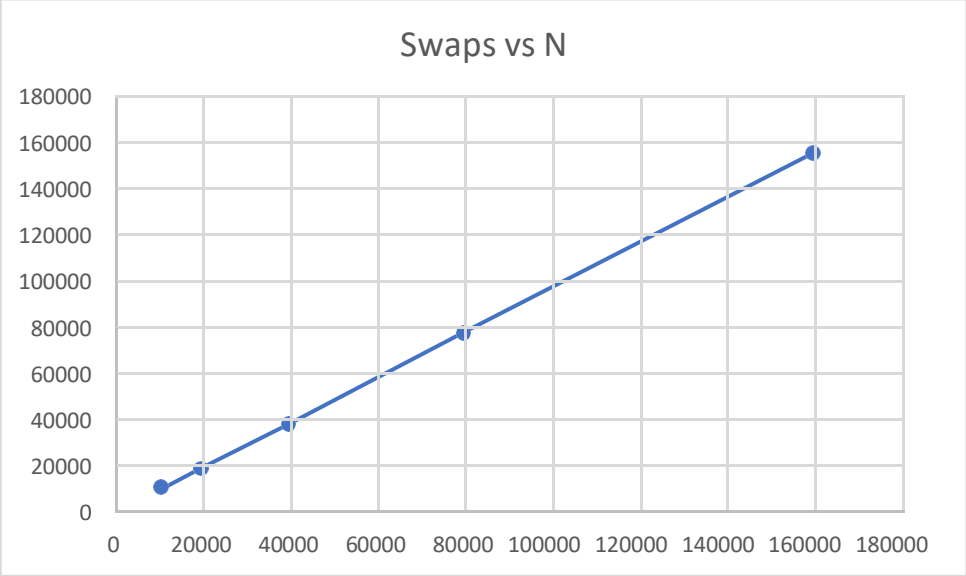


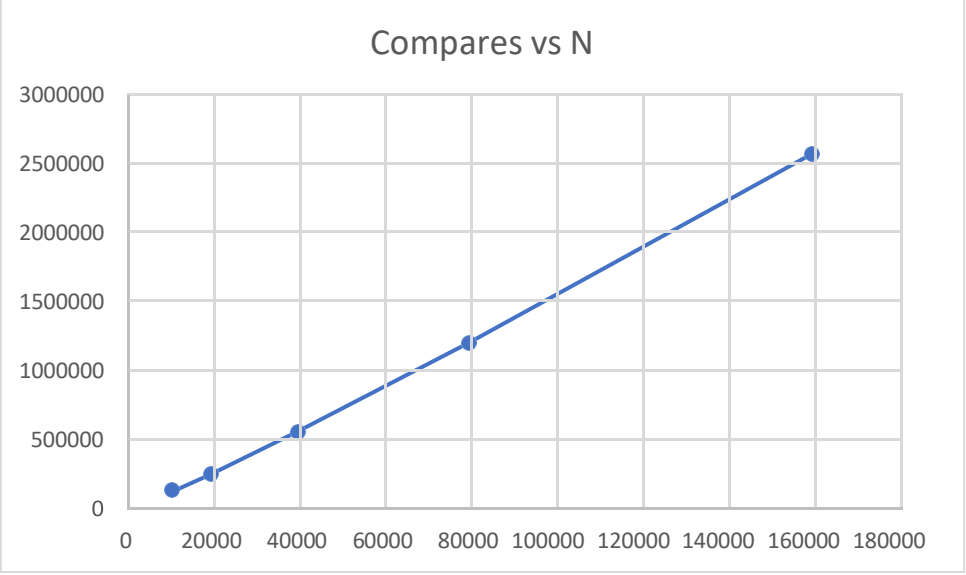
Results–

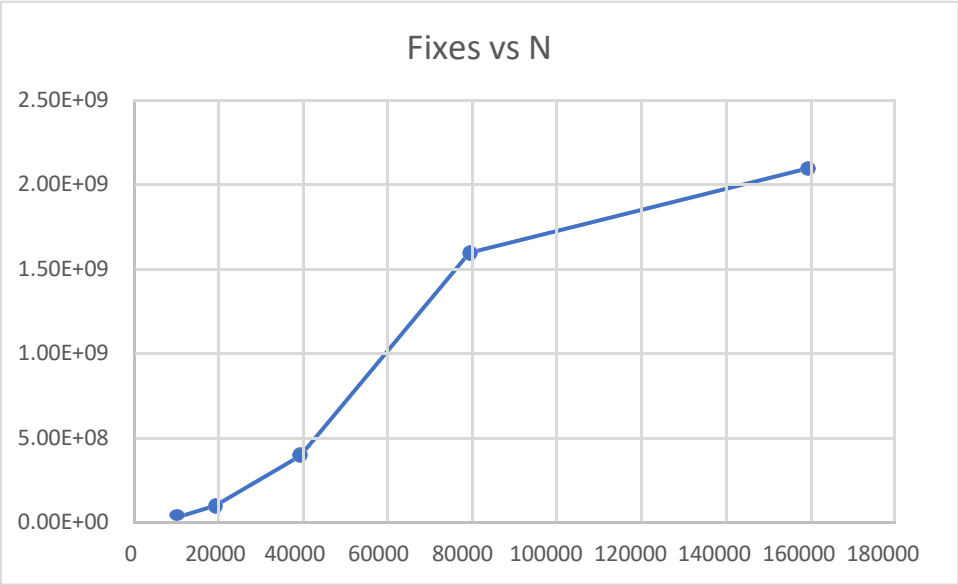
MERGE SORT –



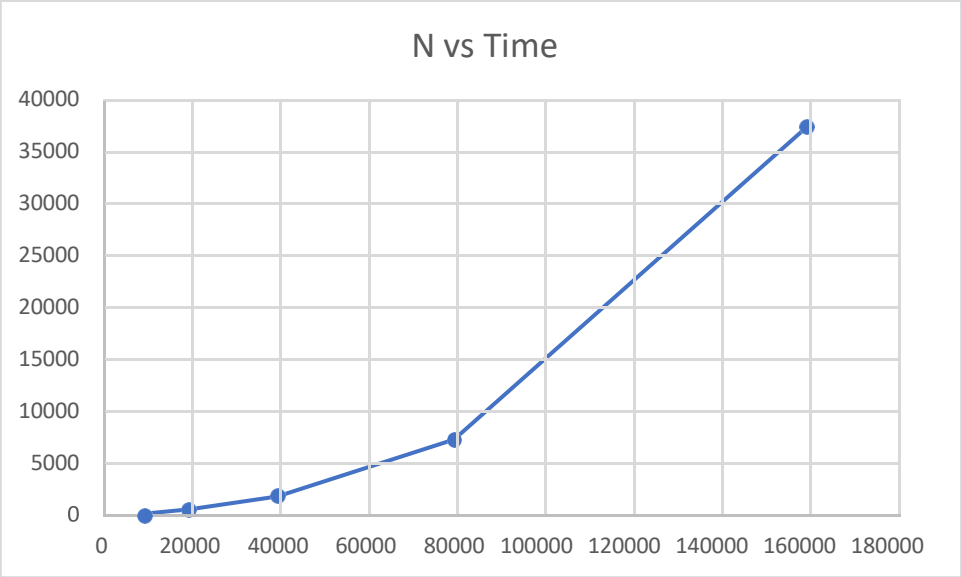

Hits vs N
6000000
5000000
4000000
3000000
2000000
1000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

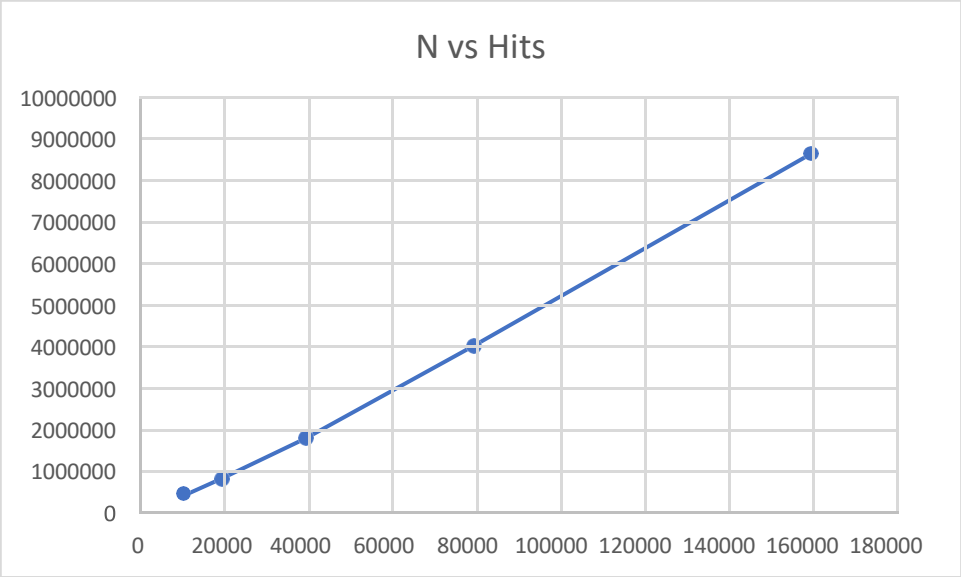

Swaps vs N
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

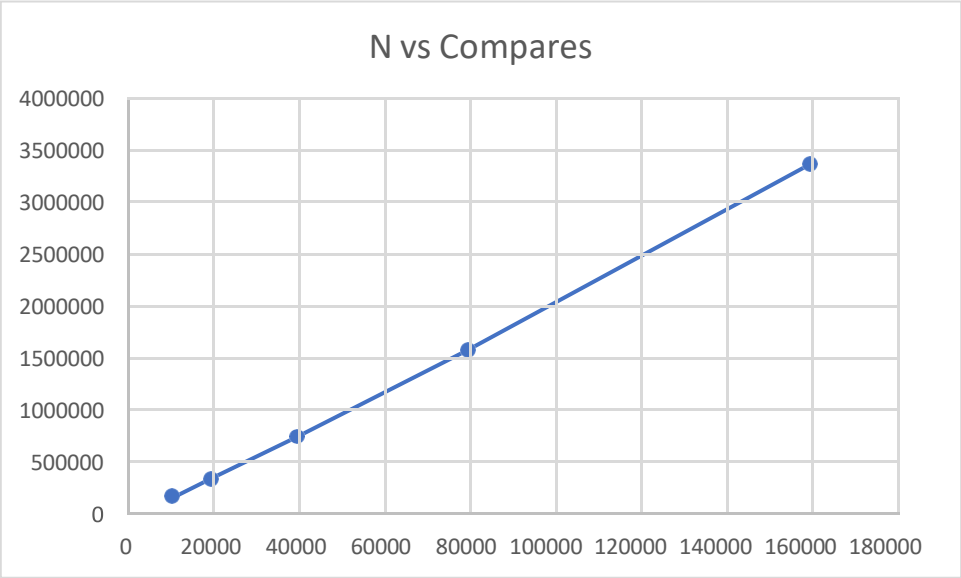

Compares vs N
3000000
2500000
2000000
1500000
1000000
500000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

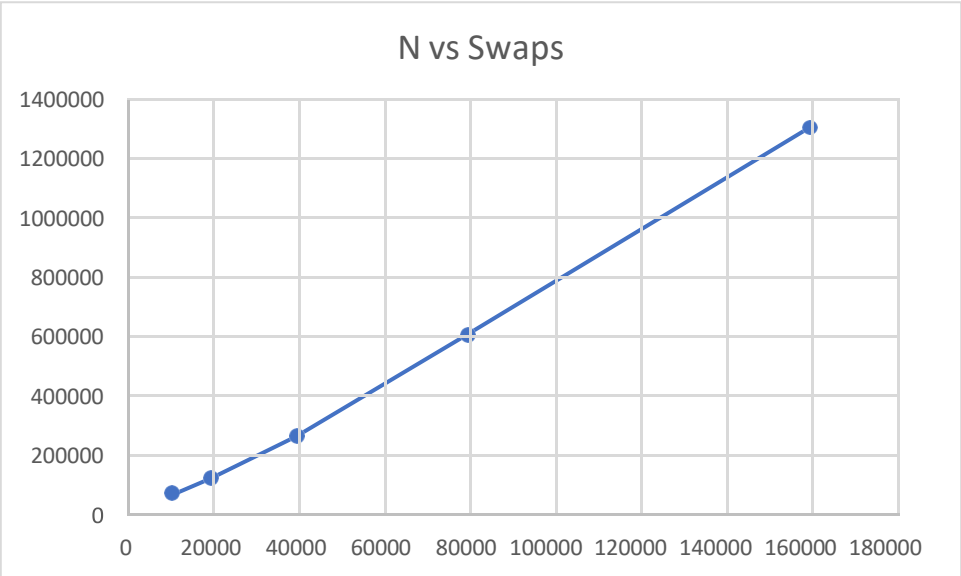

Fixes vs N
2.50E+09
2.00E+09
1.50E+09
1.00E+09
5.00E+08
0.00E+00
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

Quicksort –

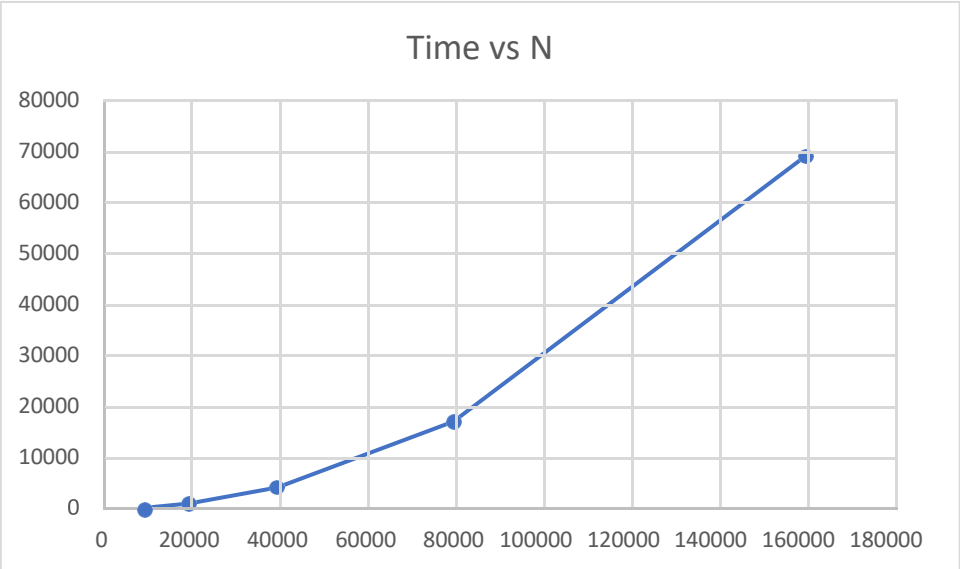

N vs Time
40000
35000
30000
25000
20000
15000
10000
5000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

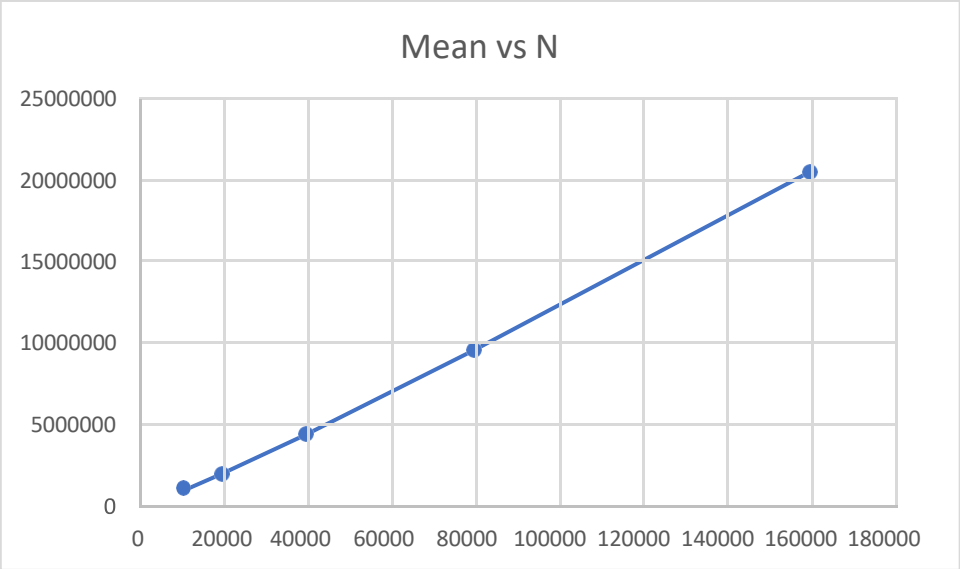

N vs Hits
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

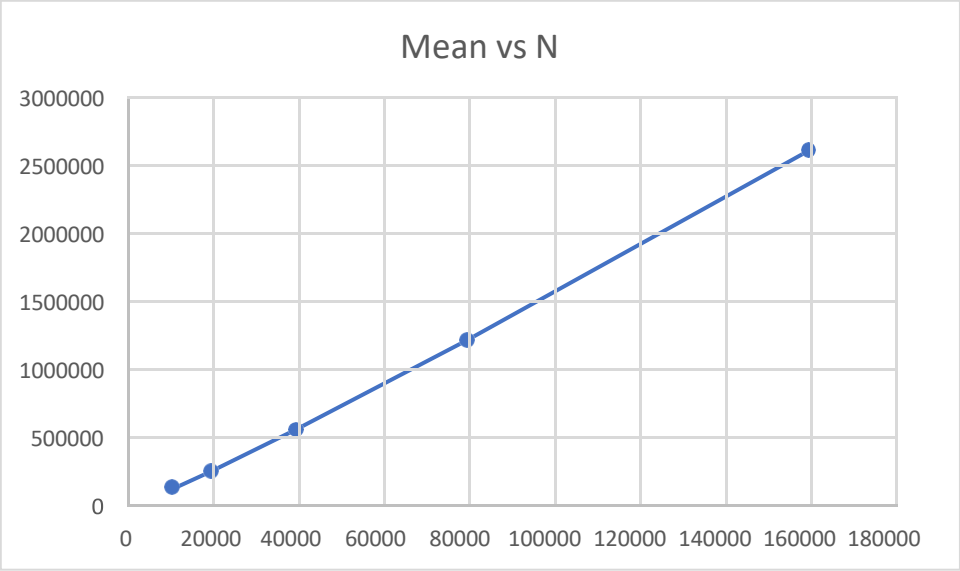

N vs Compares
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

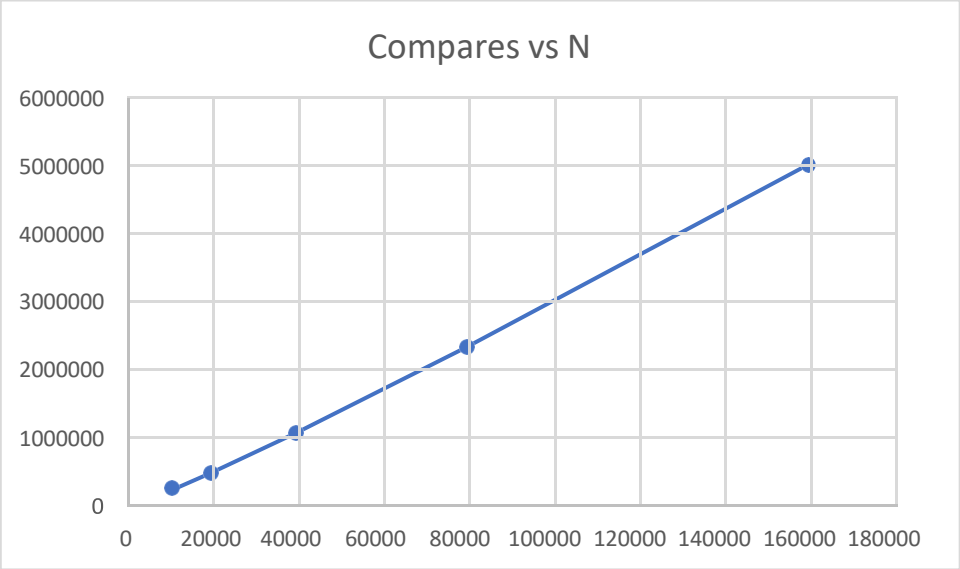

N vs Swaps
1400000
1200000
1000000
800000
600000
400000
200000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

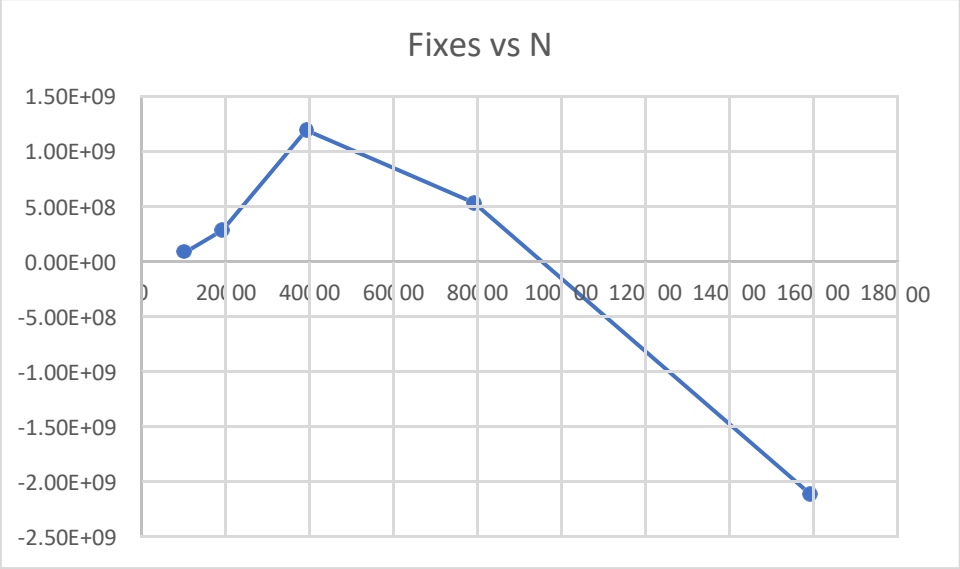
Heap Sort –


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