Program Structures and Algorithms
Spring 2023(SEC –1)

NAME:Manish Shivaprasad
NUID:002766390

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.

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).

Relationship Conclusion

The hits and compares in a sorting algorithm refer to the number of times the algorithm accesses elements in the input array and compares them, respectively. The more hits and compares an algorithm performs, the more time it takes to sort the array.

Therefore, it is essential to consider the number of hits and compares when evaluating the efficiency of a sorting algorithm. For example, an algorithm that performs fewer hits and compares is likely to have a faster runtime than one that performs more.

In practice, many sorting algorithms, such as Quicksort and Mergesort, have been designed to minimize the number of hits and compares required to sort an array efficiently. These

algorithms have proven to be very effective, with runtime complexities that are often faster than less optimized sorting algorithms.

Overall, understanding the role of hits and compares in a sorting algorithm can help developers choose the most efficient sorting algorithm for a given task and optimize the performance of their code.

Evidence to support that conclusion

Below is the data to support the conclusion

Merge sort data

N	Time	hits:Mean	hits:StdDev	hits:NormalizedM	swaps:Mean	swaps:StdDev	swaps:Normaliz	compares:Mean	compares:StdDe	compares:Norma	fixes:Mean	fixes:StdDev	fixes:NormalizedMean
10000	4.334375	269494	0	2.925993928	9723	0	0.1055661312	123478	0	1.340645351	2.47E+07	0	268.2709108
20000	43.73225	579788	0	2.927191037	19628	0	0.09909640365	267135	0	1.348691552	9.98E+07	0	503.9302441
40000	12.591834	1239464	0	2.924192518	39339	0	0.09281012555	574149	0	1.354555041	3.98E+08	0	938.4077045
80000	27.236668	2636820	0	2.919476235	77823	0	0.08616530483	1228192	0	1.35984912	1.60E+09	0	1768.046747
160000	61.074583	5595396	0	2.918420423	156242	0	0.08149197015	2616620	0	1.364764397	2.11E+09	0	1102.771632

Heap sort data

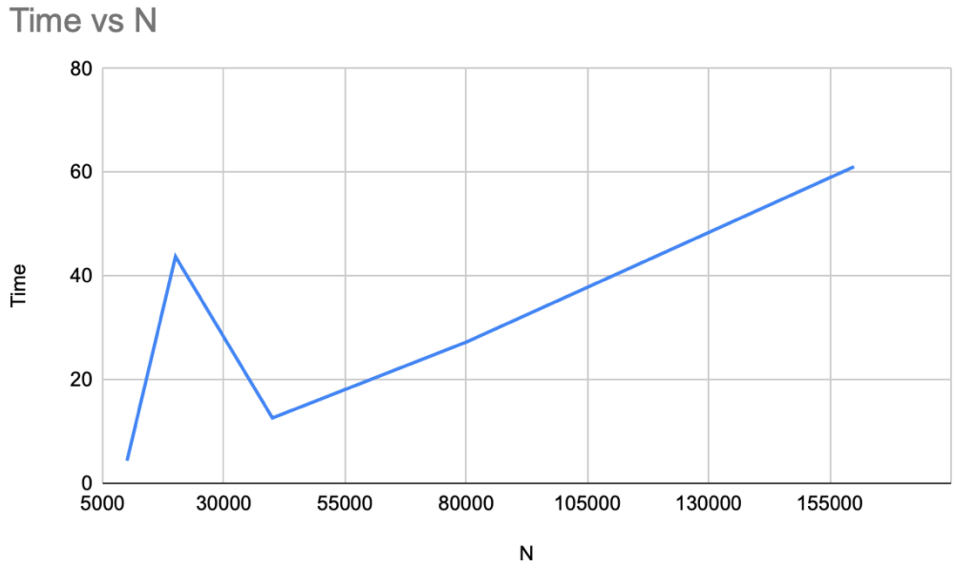
N	Time	hits:Mean	hits:StdDev	hits:NormalizedM	swaps:Mean	swaps:StdDev	swaps:Normaliz	compares:Mean	compares:StdDe	compares:Norma	fixes:Mean	fixes:StdDev	fixes:NormalizedMean
10000	299.078333	967774	0	10.5074727	124220	0	1.348701514	235447	0	2.556333322	7.53E+07	0	817.1103668
20000	1103.679041	2095914	0	10.58169654	268545	0	1.355810257	510867	0	2.579227758	3.03E+08	0	1529.562512
40000	4540.754291	4509038	0	10.63790088	576645	0	1.360443703	1101229	0	2.598063035	1.21E+09	0	2859.198044
80000	19008.75079	9659592	0	10.69506045	1233468	0	1.365690685	2362860	0	2.616148853	5.47E+08	0	606.0491648
160000	76766.39279	2.06E+07	0	10.7449626	2627215	0	1.370290487	5046060	0	2.631900327	-2.10E+09	0	-1094.468392

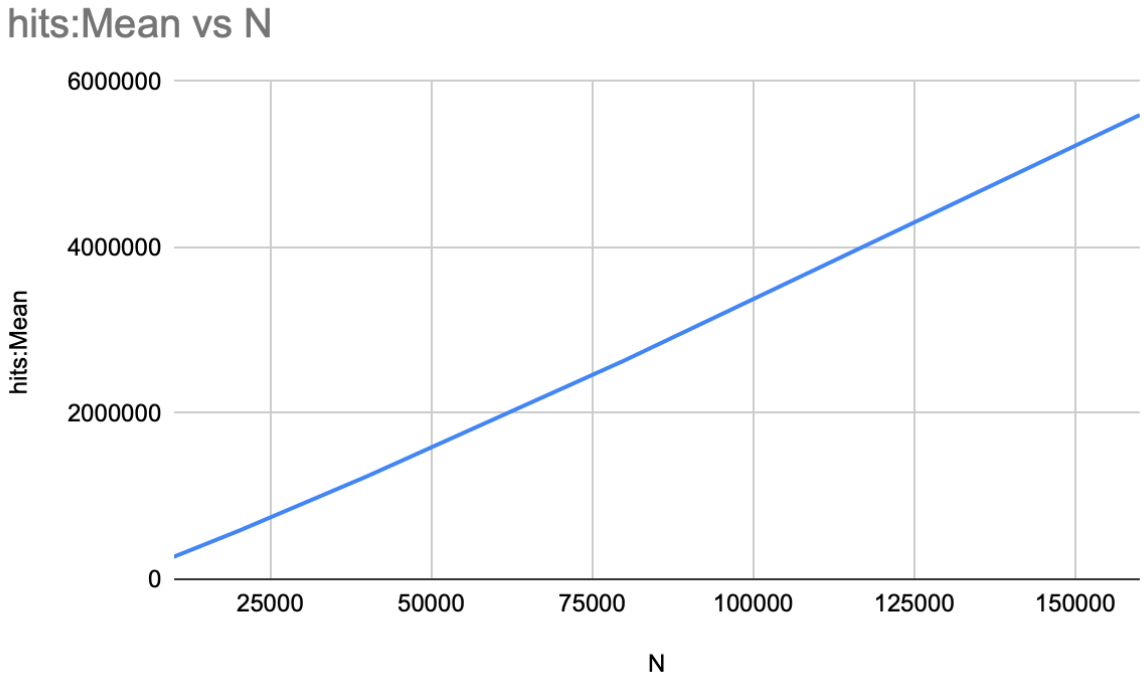
Quick sort data

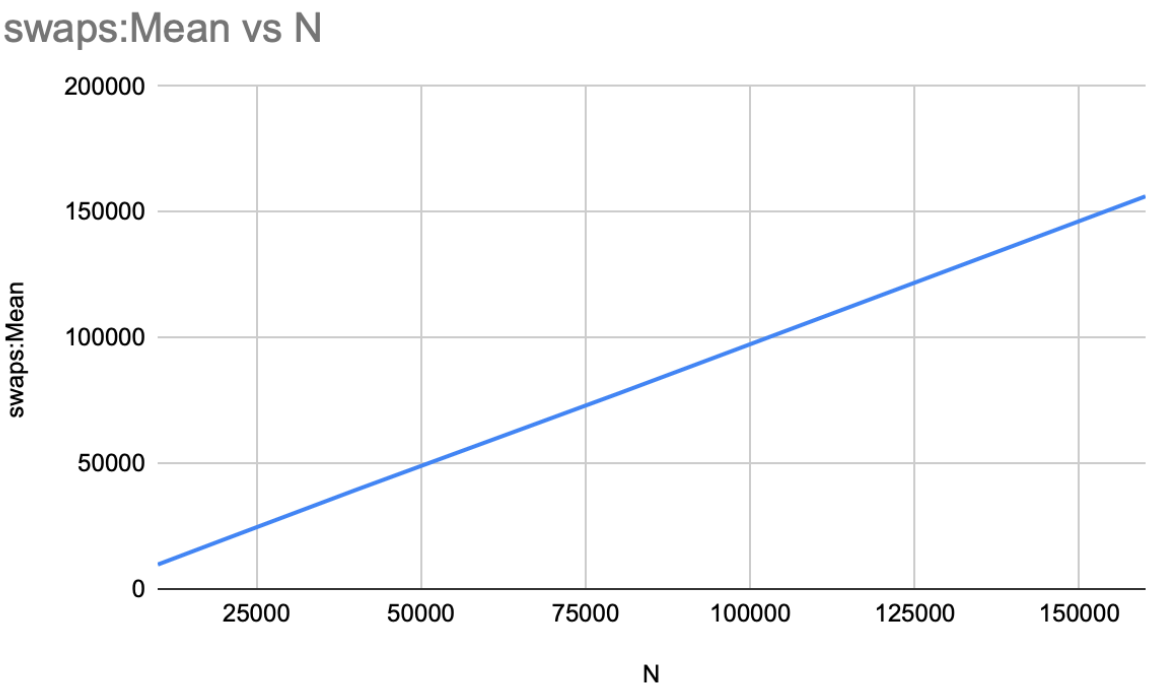
N	Time	hits:Mean	hits:StdDev	hits:NormalizedM	swaps:Mean	swaps:StdDev	swaps:Normaliz	compares:Mean	compares:StdDe	compares:Norma	fixes:Mean	fixes:StdDev	fixes:NormalizedMean
10000	146.776667	389436	0	4.228247646	59559	0	0.646536262	147006	0	1.596097365	2.60E+07	0	282.1699194
20000	1155.65375	985762	0	4.976842727	156024	0	0.7877225027	352910	0	1.781746067	1.38E+08	0	695.6463229
40000	4746.449584	2033654	0	4.797876994	314324	0	0.7415656195	758957	0	1.790561388	5.47E+08	0	1291.482706
80000	12216.66329	4109872	0	4.550433338	621809	0	0.6884643618	1588436	0	1.758709792	1.78E+09	0	1972.825982
160000	31604.83833	8704367	0	4.539982948	1253761	0	0.6539307867	3621484	0	1.888876653	-2.09E+09	0	-1092.157386

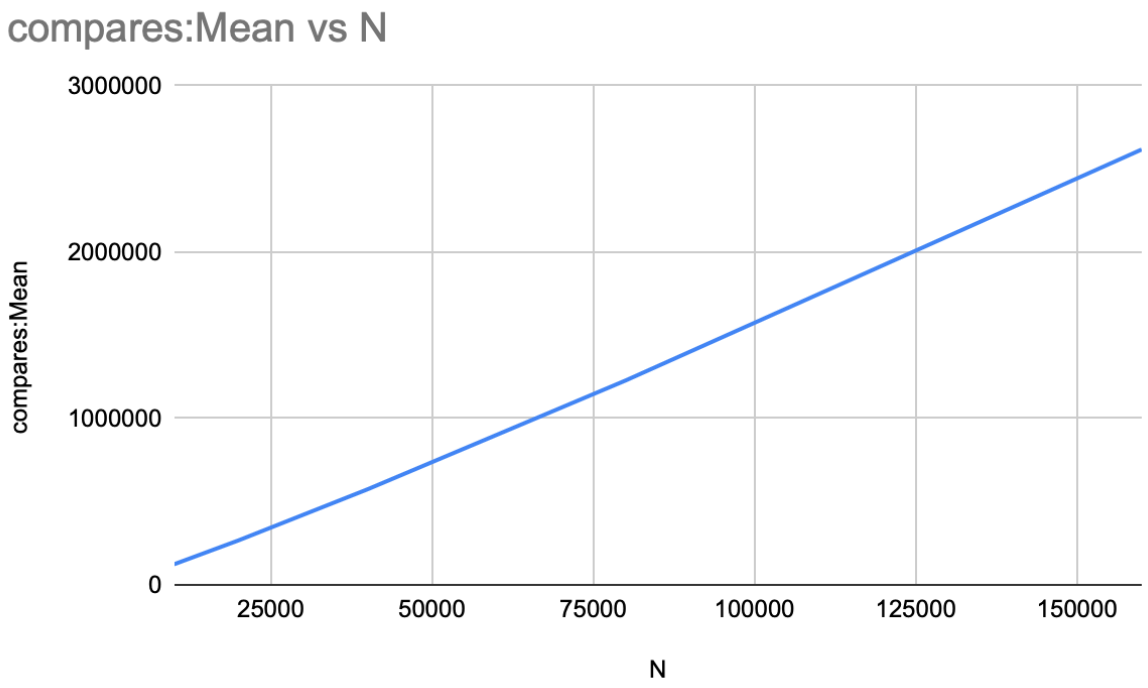
Graphical Representation

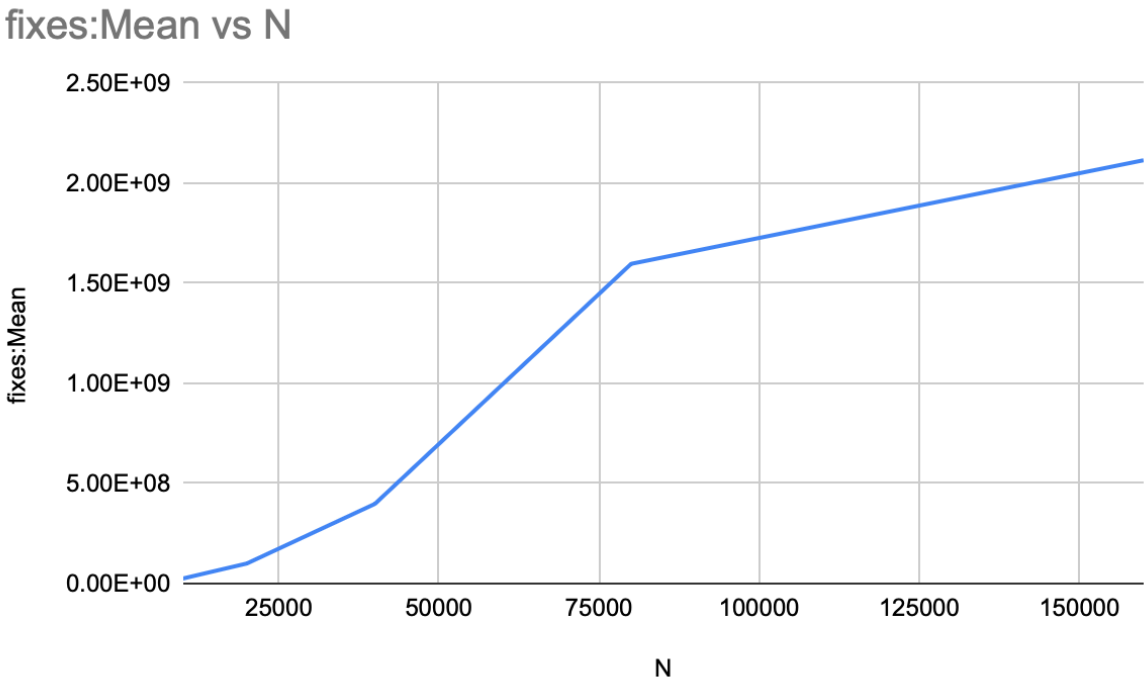
Merge sort graphs


Time vs N
80
60
40
20
0
Time
5000
30000
55000
80000
105000
130000
155000
N


hits:Mean vs N
6000000
4000000
2000000
0
hits:Mean
25000
50000
75000
100000
125000
150000
N

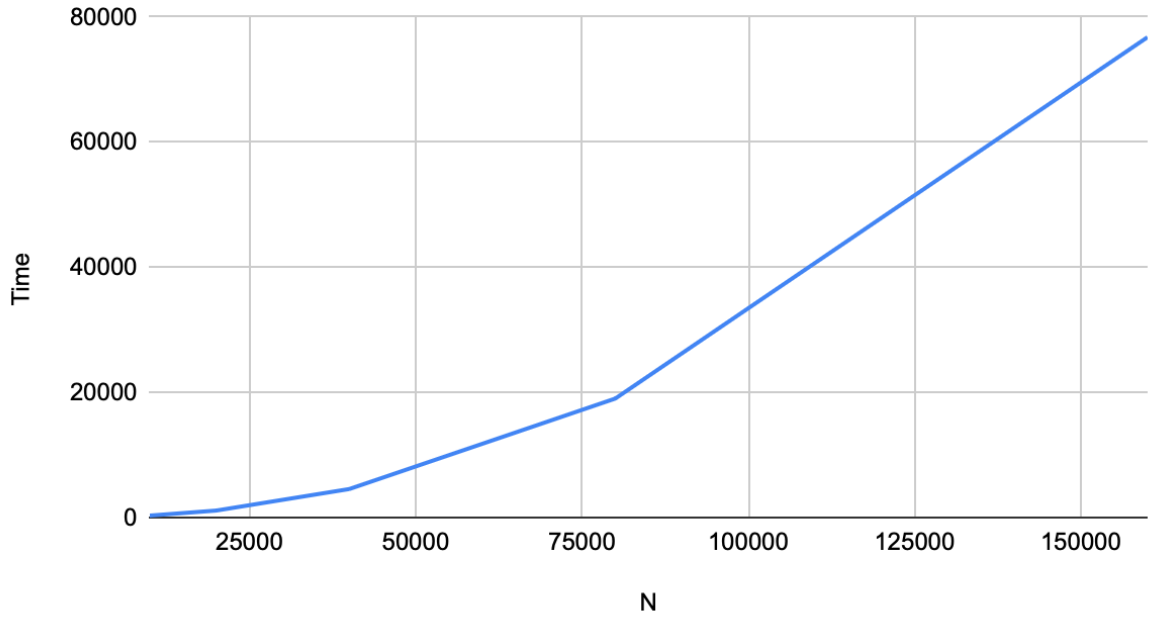

swaps:Mean vs N
200000
150000
100000
50000
0
swaps:Mean
25000
50000
75000
100000
125000
150000
N


compares:Mean vs N
3000000
2000000
1000000
0
compares:Mean
25000
50000
75000
100000
125000
150000
N

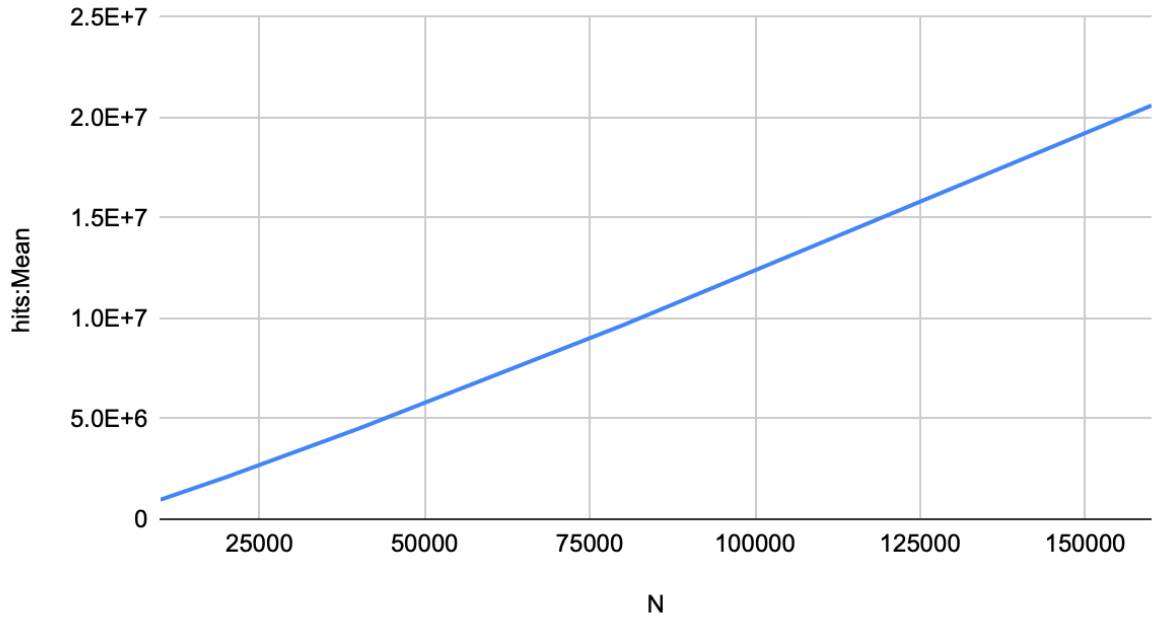


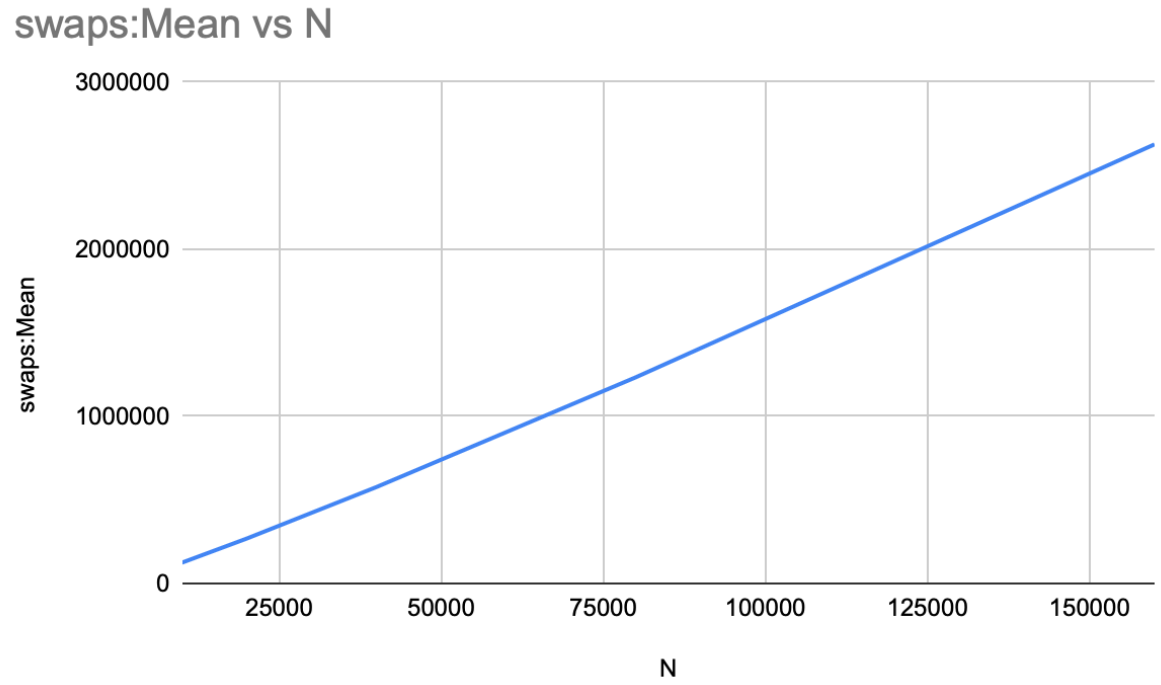
Heap sort graphs

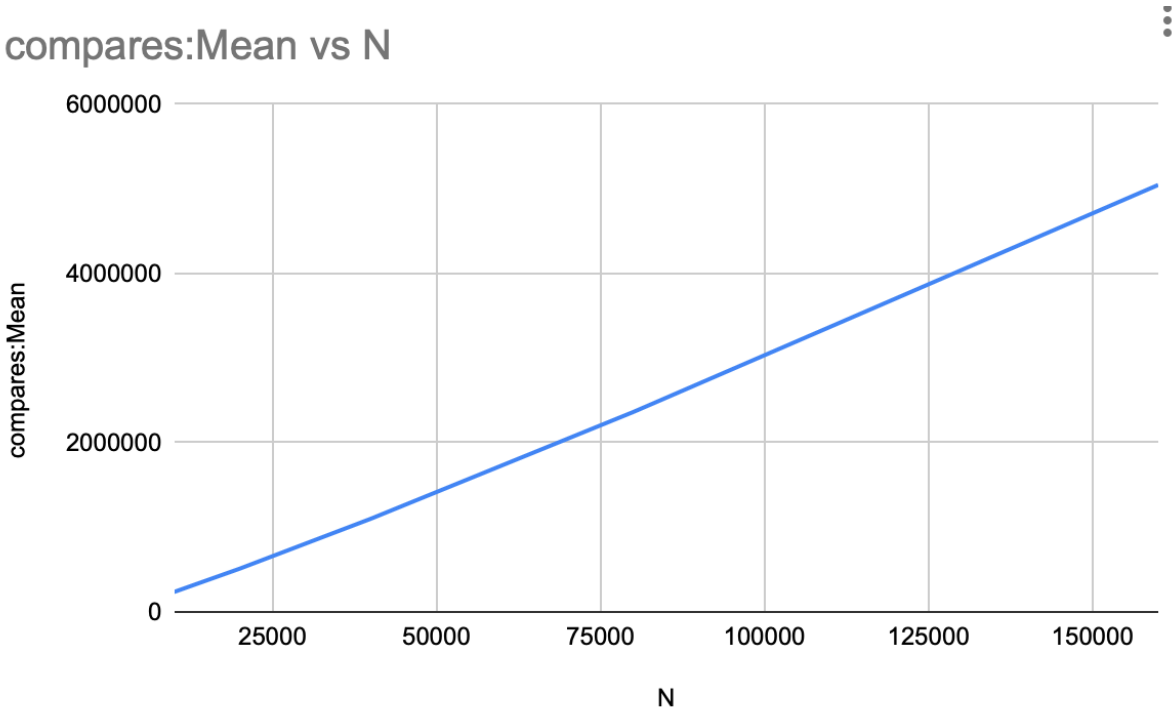
Time vs N

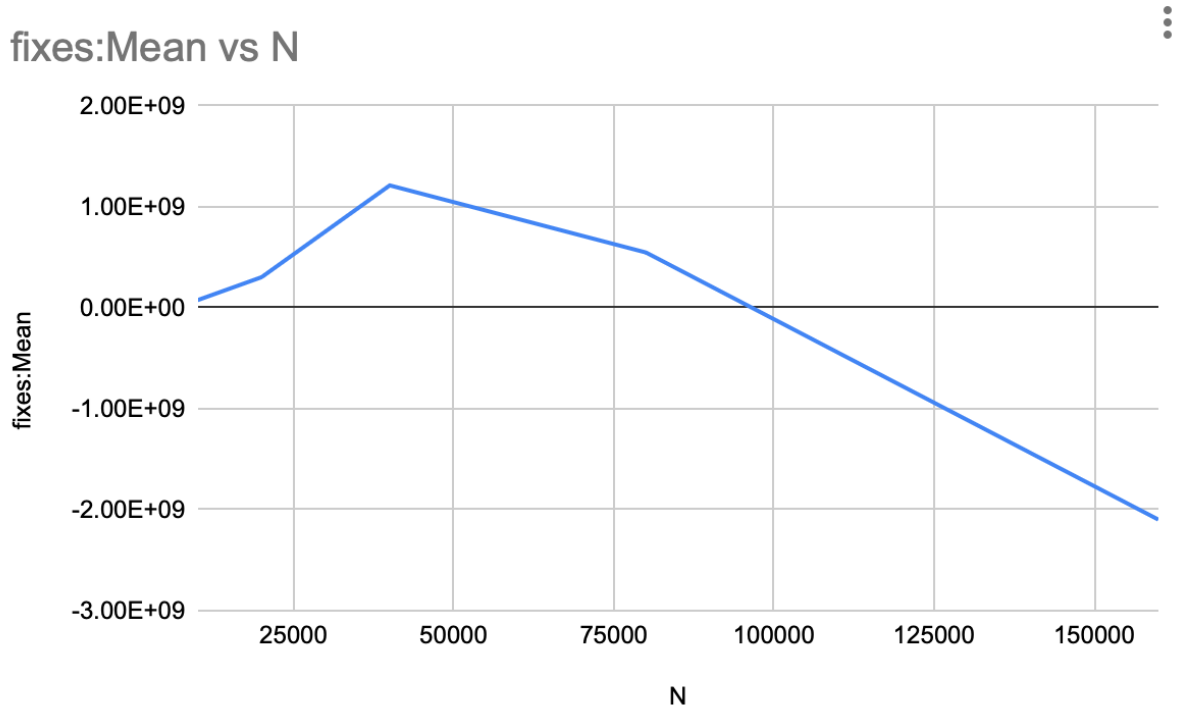


hits:Mean vs N

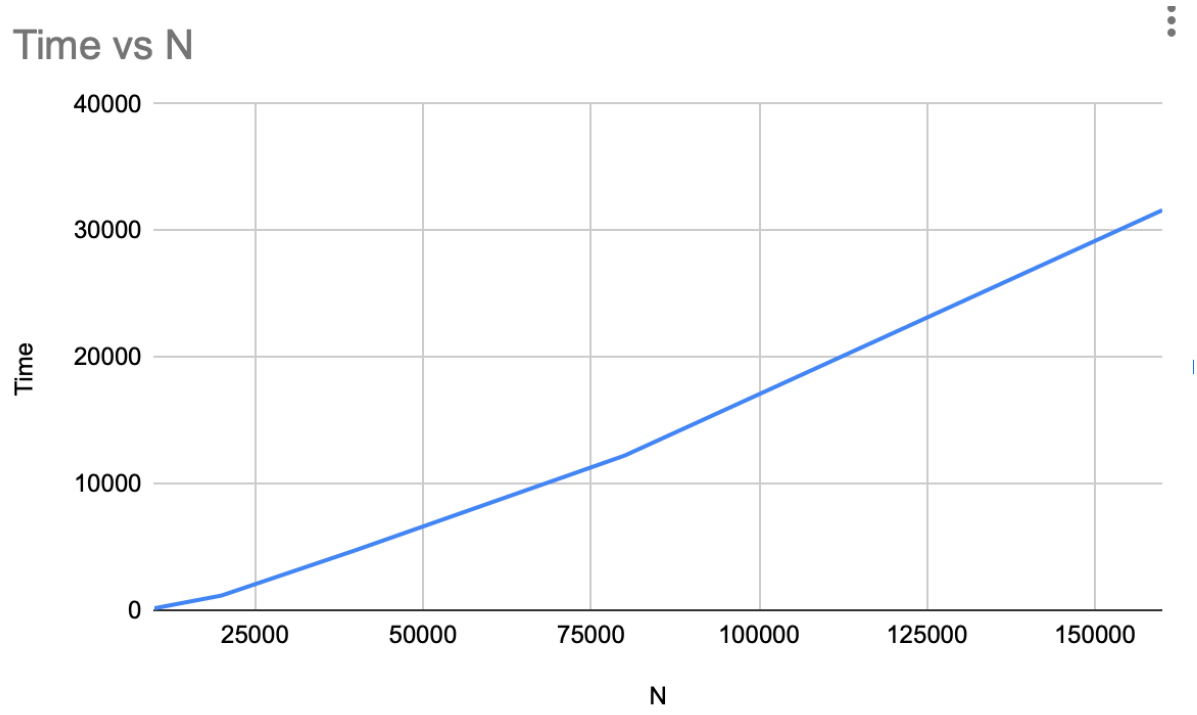


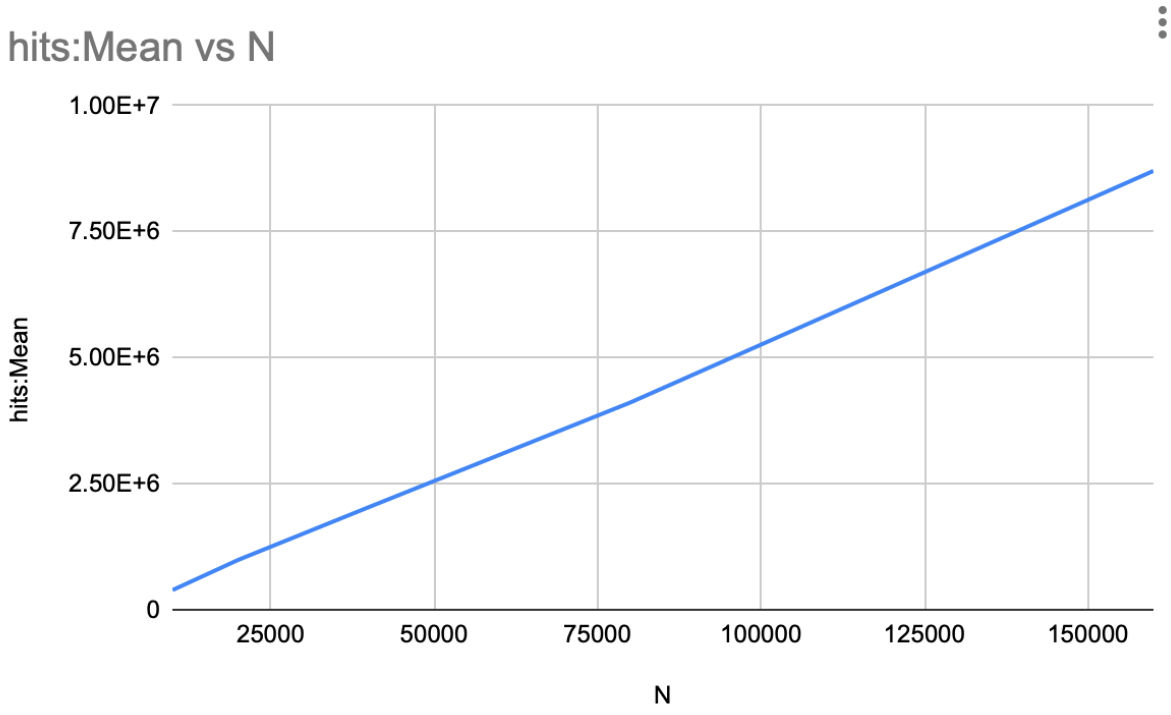

swaps:Mean vs N
3000000
2000000
1000000
0
swaps:Mean
25000
50000
75000
100000
125000
150000
N

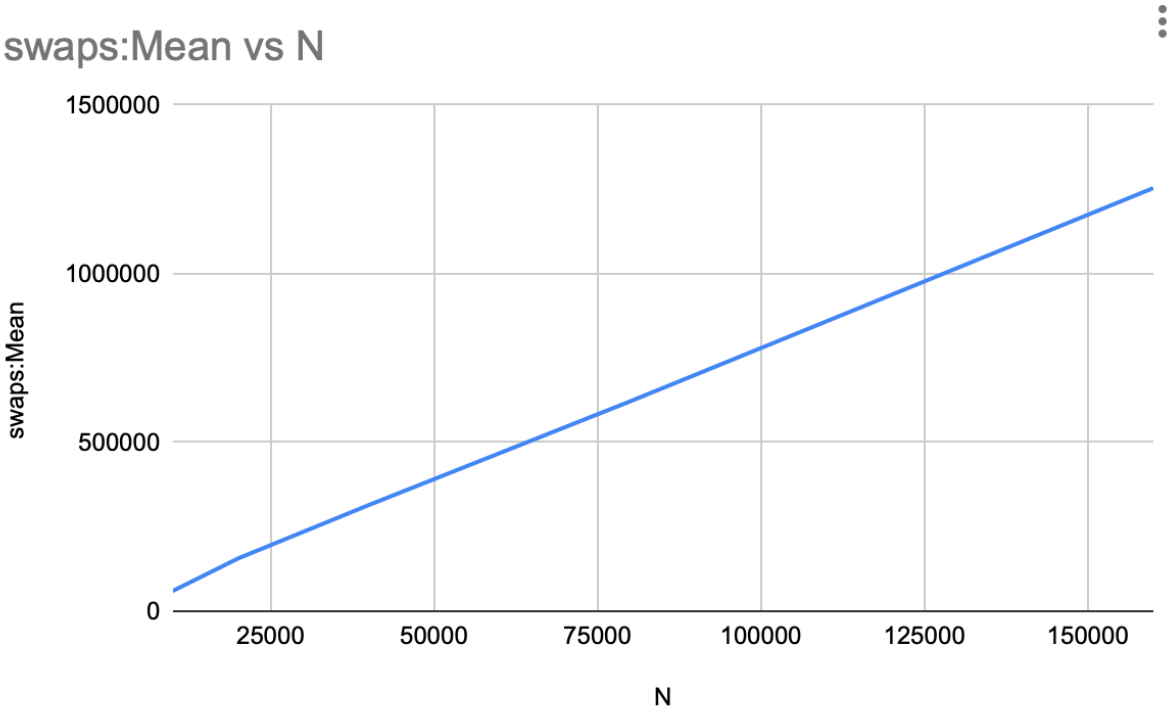

compares:Mean vs N
6000000
4000000
2000000
0
compares:Mean
25000
50000
75000
100000
125000
150000
N

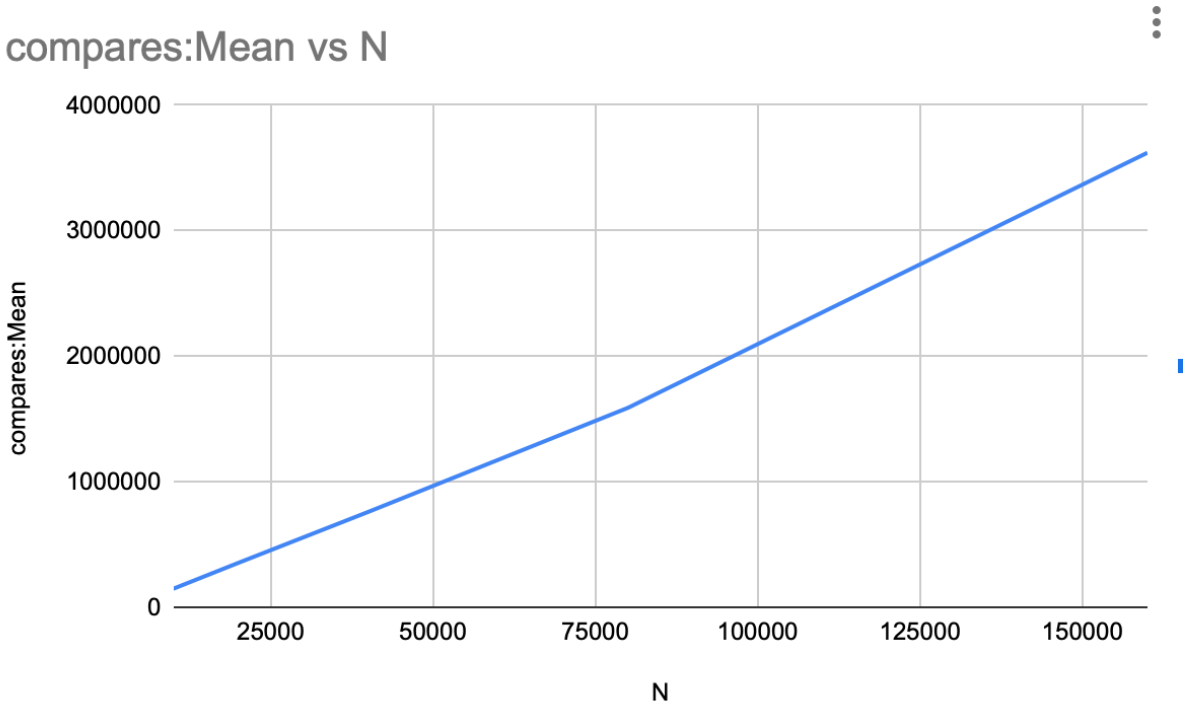


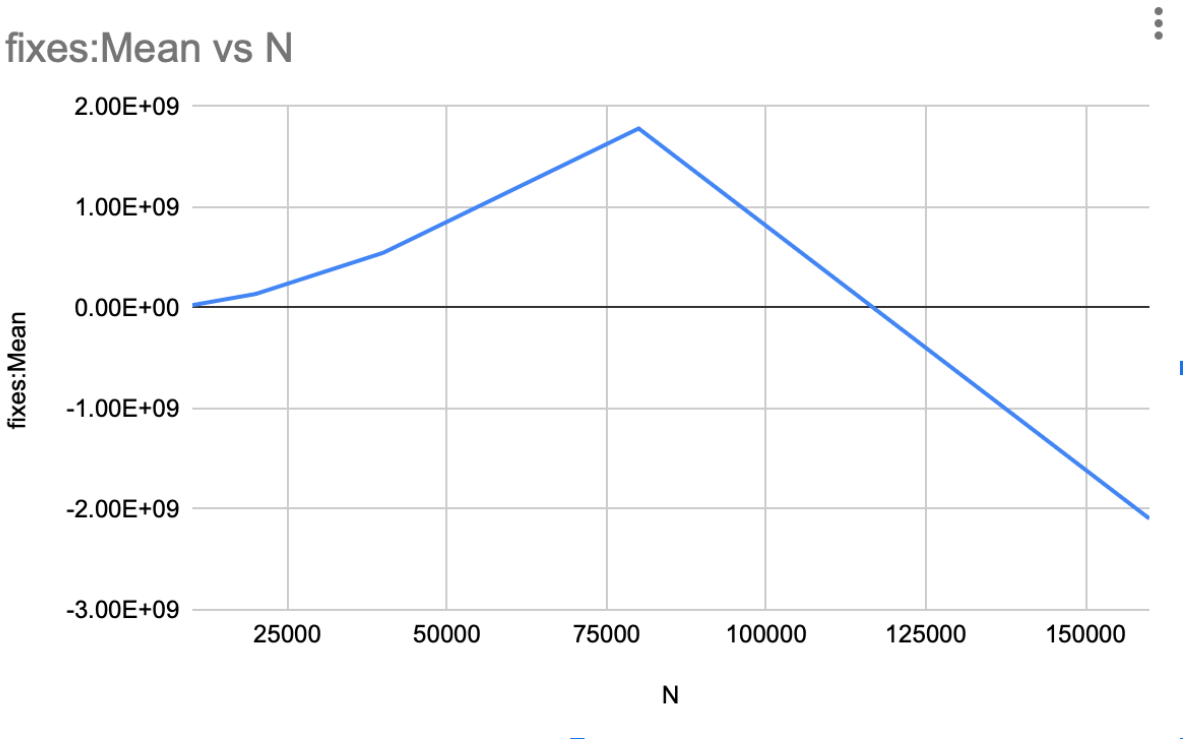
Quicksort Graphs



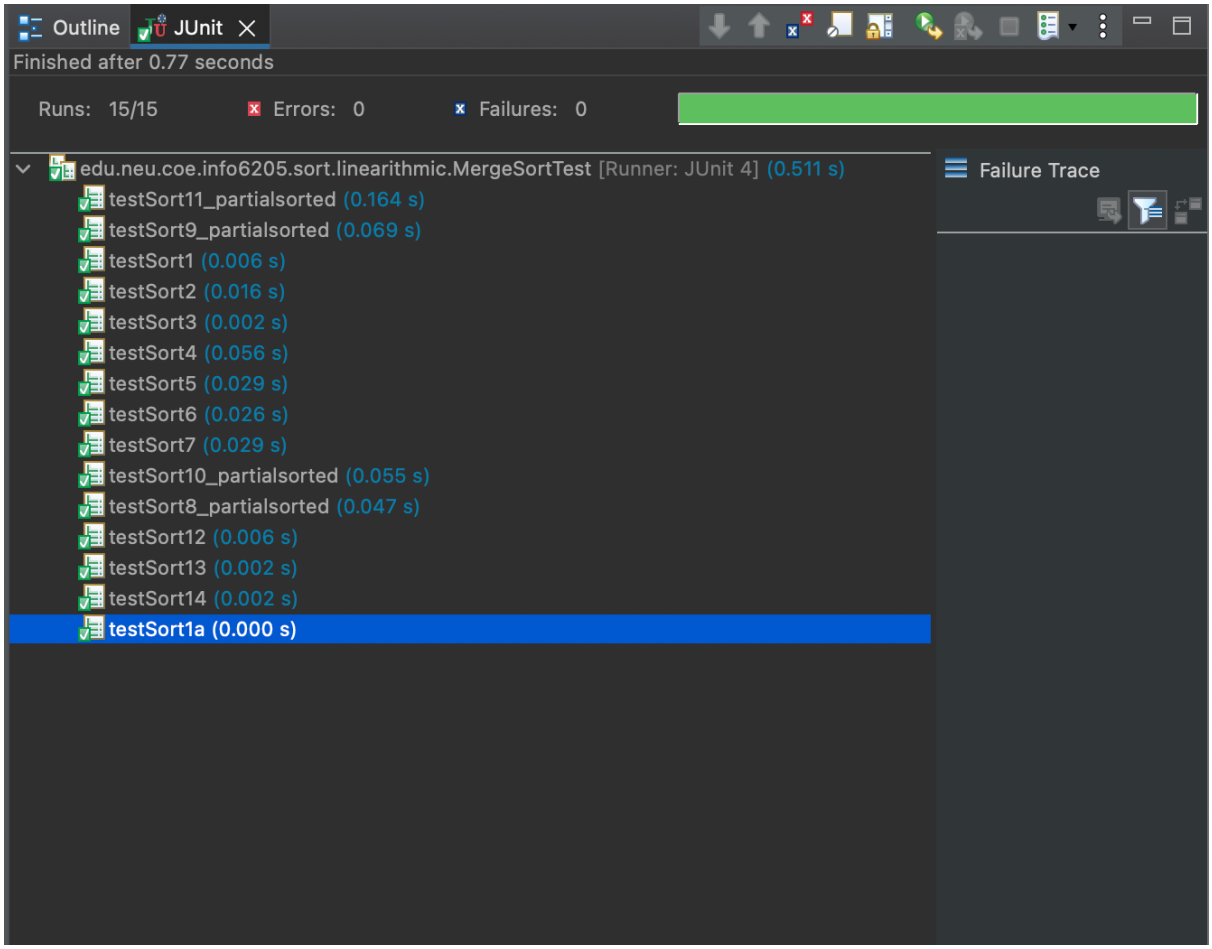

hits:Mean vs N
1.00E+7
7.50E+6
5.00E+6
2.50E+6
0
hits:Mean
25000
50000
75000
100000
125000
150000
N

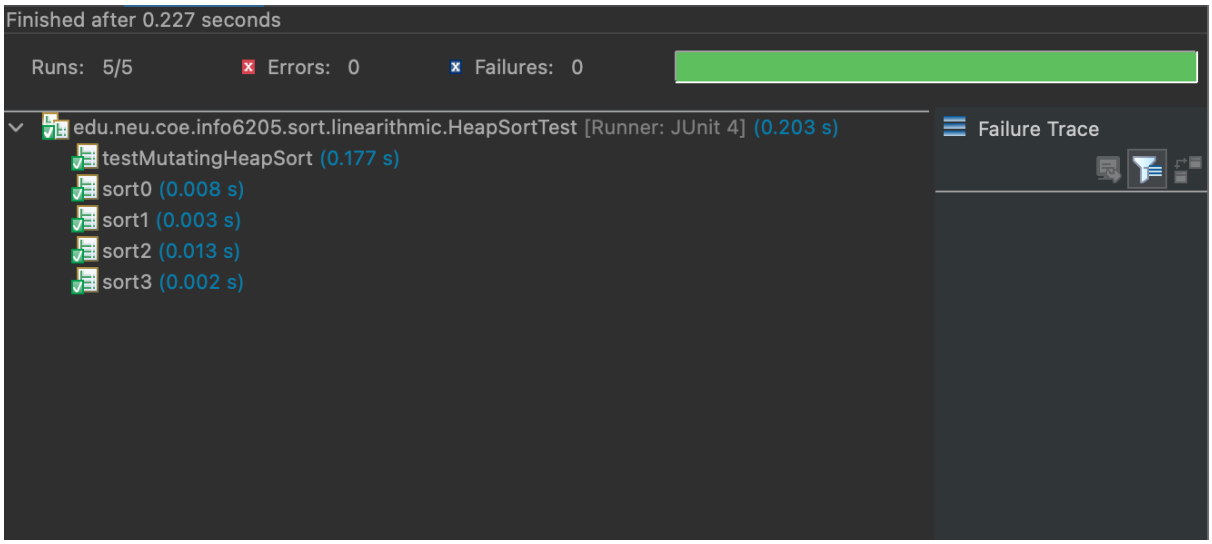

swaps:Mean vs N
1500000
1000000
500000
0
swaps:Mean
25000
50000
75000
100000
125000
150000
N

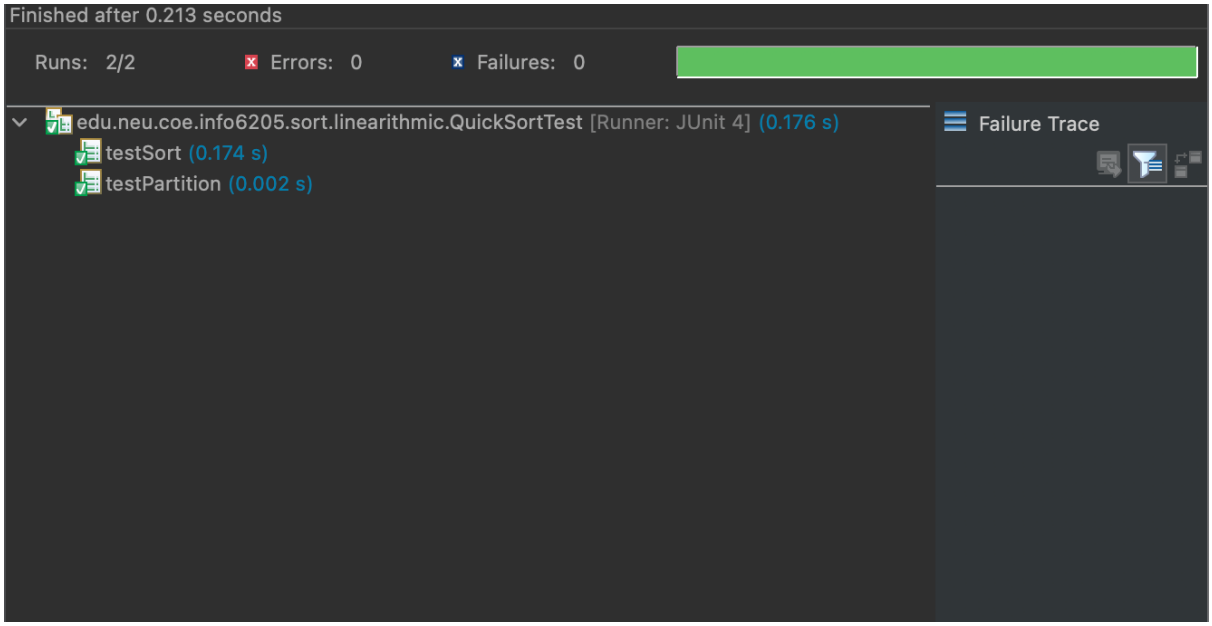




Unit Test Screenshots


Outline
JUnit
Finished after 0.77 seconds
Runs: 15/15
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.MergeSortTest [Runner: JUnit 4] (0.511 s)
testSort11_partialsorted (0.164 s)
testSort9_partialsorted (0.069 s)
testSort1 (0.006 s)
testSort2 (0.016 s)
testSort3 (0.002 s)
testSort4 (0.056 s)
testSort5 (0.029 s)
testSort6 (0.026 s)
testSort7 (0.029 s)
testSort10_partialsorted (0.055 s)
testSort8_partialsorted (0.047 s)
testSort12 (0.006 s)
testSort13 (0.002 s)
testSort14 (0.002 s)
testSort1a (0.000 s)
Failure Trace


Finished after 0.227 seconds
Runs: 5/5
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.HeapSortTest [Runner: JUnit 4] (0.203 s)
testMutatingHeapSort (0.177 s)
sort0 (0.008 s)
sort1 (0.003 s)
sort2 (0.013 s)
sort3 (0.002 s)
Failure Trace


Finished after 0.213 seconds
Runs: 2/2
Errors: 0
Failures: 0
edu.neu.coe.info6205.sort.linearithmic.QuickSortTest [Runner: JUnit 4] (0.176 s)
testSort (0.174 s)
testPartition (0.002 s)
Failure Trace