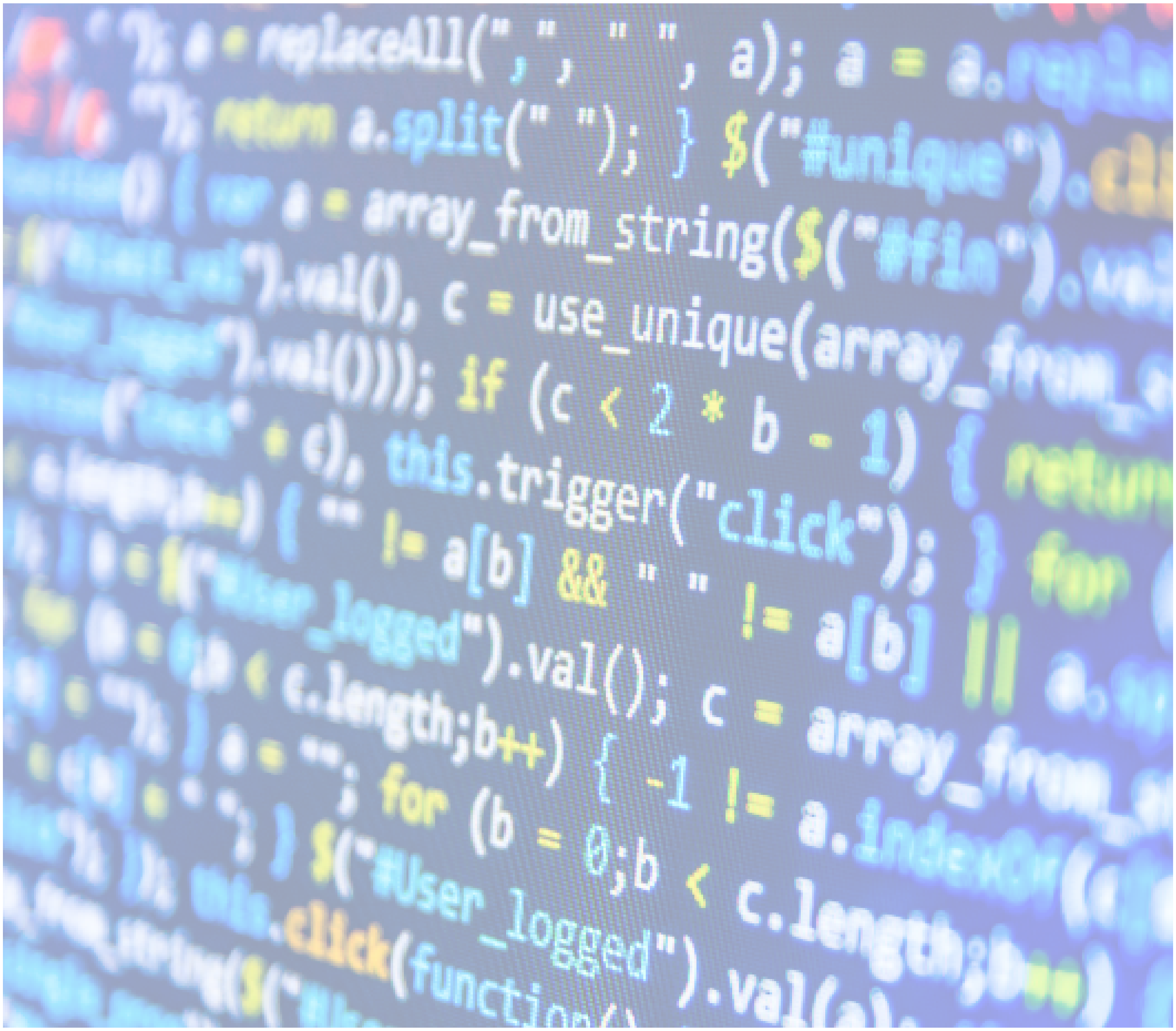


Kelly Xie
kyx203

Project 5

CSCI 102 - Data Structures

Section 1

1 / Selection sort

Selection sort			
N	comparisons	swaps	N^2
10000	49995000	9999	100000000
20000	199990000	19999	400000000
30000	449985000	29999	900000000
40000	799980000	39999	1600000000
50000	1249975000	49999	2500000000
60000	1799970000	59999	3600000000
70000	2449965000	69999	4900000000
80000	3199960000	79999	6400000000
90000	4049955000	89999	8100000000
100000	4999950000	99999	10000000000

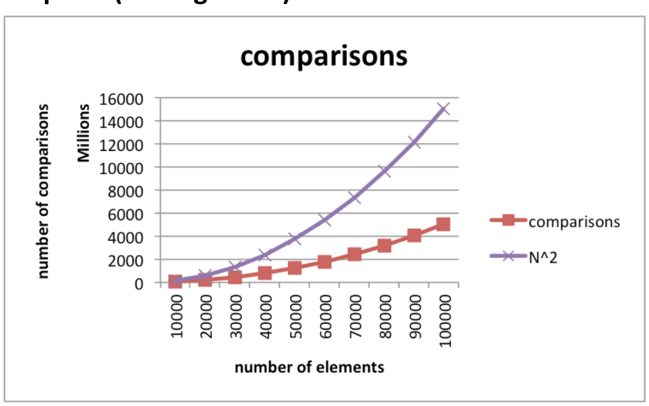
Cases
Random unsorted array
Sorted array
Reverse sorted array

N	comparisons	swaps	N^2
10000	49995000	9999	100000000
20000	199990000	19999	400000000
30000	449985000	29999	900000000
40000	799980000	39999	1600000000
50000	1249975000	49999	2500000000
60000	1799970000	59999	3600000000
70000	2449965000	69999	4900000000
80000	3199960000	79999	6400000000
90000	4049955000	89999	8100000000
100000	4999950000	99999	10000000000

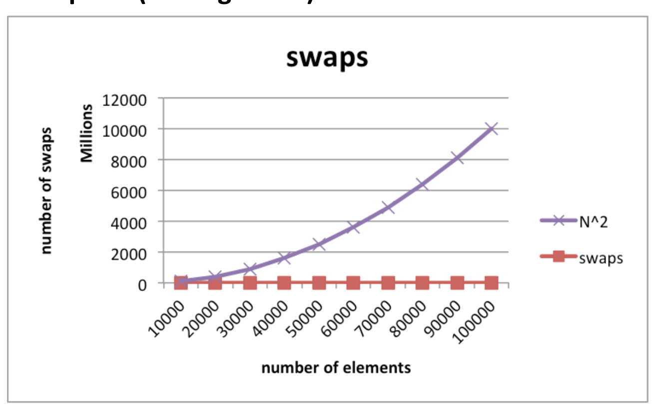
Comparisons: The number of comparisons for this selection sort implementation has a performance of **O(N^2)**. Selection sort requires **(N*(N-1))/2** comparisons in total. This number of comparisons is the same for an array of a particular size, regardless of the order of the data elements in the array, or whether it was sorted or not to begin with. This can be seen by the rough correlation between the "comparisons" and "N^2" lines in Graph A below.

Swaps: The number of swaps performed depends on the arrangement of the array's elements. Graph B below shows that for various-sized arrays of randomly arranged elements, the "swaps" line is not correlated with the "N^2" line. When the selection sort function was called with an array that was either sorted/reverse-sorted/random, the number of data swaps remained constant and equal to **(N − 1)**.

Graph A. (Average case)



Graph B. (Average case)



2 / Short Bubble sort

Short bubble sort			
N	comparisons	swaps	N^2
10000	49995000	24723138	100000000
20000	199990000	100283178	400000000
30000	449985000	225817331	900000000
40000	799980000	399527558	1600000000
50000	1249975000	622304992	2500000000
60000	1799970000	903148623	3600000000
70000	2449965000	1221988687	4900000000
80000	3199960000	1597620163	6400000000
90000	4049955000	2031258778	8100000000
100000	4999950000	2498371476	10000000000

Cases
Random unsorted array
Sorted array
Reverse sorted array

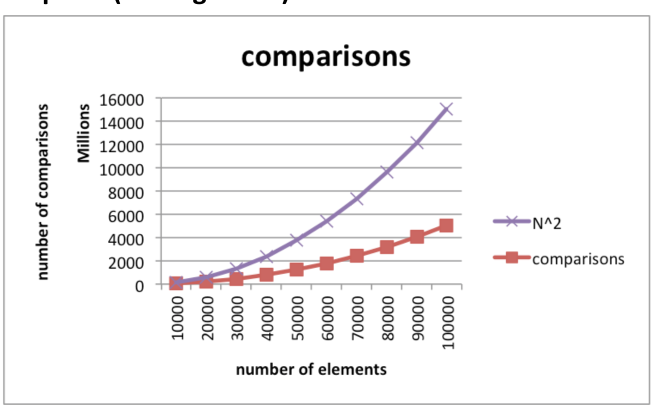
N	comparisons	swaps	N^2
10000	9999	0	100000000
20000	19999	0	400000000
30000	29999	0	900000000
40000	39999	0	1600000000
50000	49999	0	2500000000
60000	59999	0	3600000000
70000	69999	0	4900000000
80000	79999	0	6400000000
90000	89999	0	8100000000
100000	99999	0	10000000000

N	comparisons	swaps	N^2
10000	49995000	49995000	100000000
20000	199990000	199990000	400000000
30000	449985000	449985000	900000000
40000	799980000	799980000	1600000000
50000	1249975000	1249975000	2500000000
60000	1799970000	1799970000	3600000000
70000	2449965000	2449965000	4900000000
80000	3199960000	3199960000	6400000000
90000	4049955000	4049955000	8100000000
100000	4999950000	4999950000	10000000000

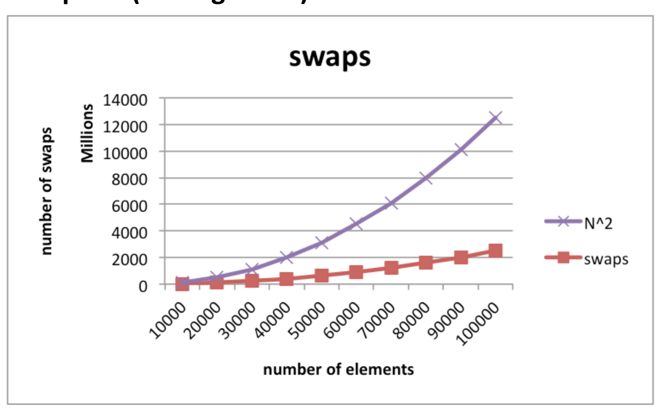
Comparisons: The number of comparisons performed by the short bubble sort implementation has a performance of **O(N)** in the best-case scenario and **O(N^2)** in all other cases. Unlike the other bubble sort method, the number of comparisons changes for this method depends on the arrangement of data in the array. Calls to the short bubble sort method on a sorted array (best-case scenario) return comparisons of **(N-1)**. Calls on a reverse-sorted array (worst-case scenario) or a randomly-generated array return a comparisons equal to **(N*(N-1))/2.**

Swaps: The number of swaps varies depending on the order of the array. Graph B below shows that the "swaps" line roughly correlates with the "N^2" line. With a sorted array of any size, the number of data swaps is **0**. With reverse-sorted arrays, the number of data swaps is maximized and is equal to **(N*(N – 1))/2,** which is equal to the number of comparisons.

Graph A. (Average case)



Graph B. (Average case)



3 / Insertion sort

Insertion sort			
N	comparisons	swaps	N^2
10000	24993946	24983956	100000000
20000	99870366	99850376	400000000
30000	224344004	224314017	900000000
40000	401007807	400967814	1600000000
50000	623912746	623862759	2500000000
60000	901502108	901442119	3600000000
70000	1224064202	1223994214	4900000000
80000	1596653764	1596573774	6400000000
90000	2028542782	2028452790	8100000000
100000	2494443511	2494343529	10000000000

Cases
Random unsorted array
Sorted array
Reverse sorted array

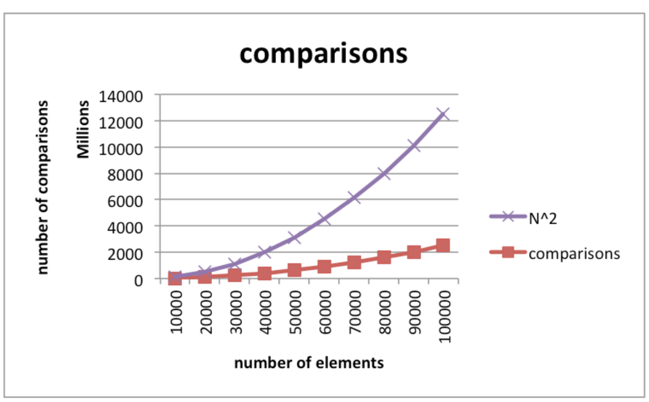
N	comparisons	swaps	N^2
10000	9999	0	100000000
20000	19999	0	400000000
30000	29999	0	900000000
40000	39999	0	1600000000
50000	49999	0	2500000000
60000	59999	0	3600000000
70000	69999	0	4900000000
80000	79999	0	6400000000
90000	89999	0	8100000000
100000	99999	0	10000000000

N	comparisons	swaps	N^2
10000	49995000	49995000	100000000
20000	199990000	199990000	400000000
30000	449985000	449985000	900000000
40000	799980000	799980000	1600000000
50000	1249975000	1249975000	2500000000
60000	1799970000	1799970000	3600000000
70000	2449965000	2449965000	4900000000
80000	3199960000	3199960000	6400000000
90000	4049955000	4049955000	8100000000
100000	4999950000	4999950000	10000000000

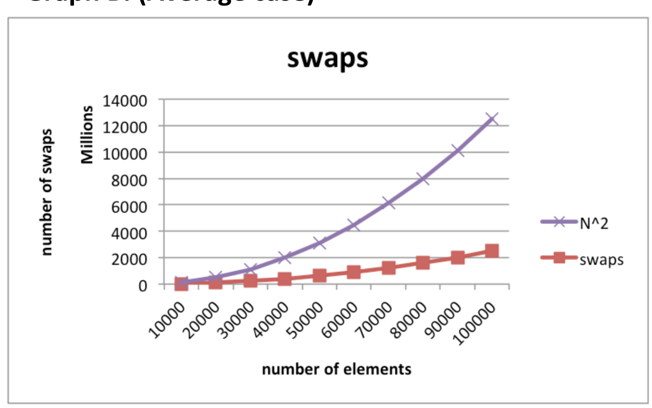
Comparisons: The number of comparisons for the insertion sort implementation has a performance of **O(N^2),** similar to selection and bubble sorts. This number varies based on the data in the array. In the best-case scenario (array is already sorted), there are only **N** comparisons, which gives **O(N)** performance. In the worst-case scenario (reverse-sorted array), the method returns the maximum number of possible comparisons, which is **(N*(N-1))/2**. For a randomly-sorted array, the number of comparisons is between these two possibilities.

Swaps: The number of swaps varies depending on the order of the array. The number of data swaps is **0** for an array that is already sorted in ascending order, and is **(N*(N-1))/2** for an array in reverse-sorted, descending order. For a randomly-sorted array, the number of sorts is between these two cases.

Graph A. (Average case)



Graph B. (Average case)



4 / Merge sort

Merge sort			
N	comparisons	swaps	N log N
10000	120447	267232	132877.1238
20000	260962	574464	285754.2476
30000	408632	894464	446180.2464
40000	561629	1228928	611508.4952
50000	718175	1568928	780482.0237
60000	877239	1908928	952360.4928
70000	1038913	2257856	1126654.711
80000	1203936	2617856	1303016.99
90000	1369468	2977856	1481187.364
100000	1536099	3337856	1660964.047
200000	3272115	7075712	3521928.095
300000	5084838	10951424	5458380.893
400000	6945656	14951424	7443856.19
500000	8837070	18951424	9465784.285
600000	10769127	23102848	11516761.79
700000	12723289	27302848	13591896.78
800000	14690545	31502848	15687712.38
900000	16679980	35702848	17801608.93
1000000	18673917	39902848	19931568.57

Cases
Random unsorted array
Sorted array
Reverse sorted array

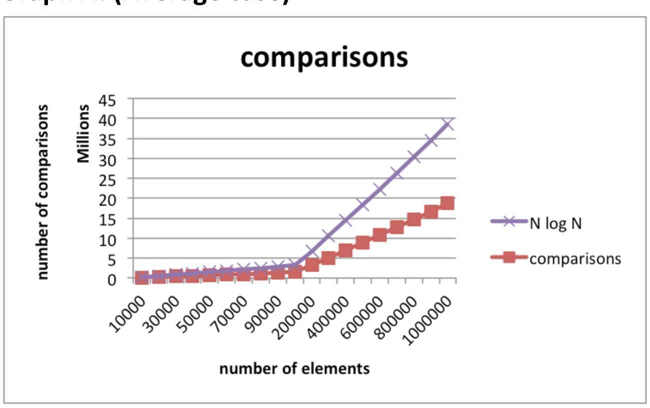
N	comparisons	swaps	N log N
100000	69008	267232	1660964.047
200000	148016	574464	3521928.095
300000	227728	894464	5458380.893
400000	316032	1228928	7443856.19
500000	401952	1568928	9465784.285
600000	485456	1908928	11516761.79
700000	573728	2257856	13591896.78
800000	672064	2617856	15687712.38
900000	765248	2977856	17801608.93
1000000	853904	3337856	19931568.57

N	comparisons	swaps	N log N
100000	64608	267232	1660964.047
200000	139216	574464	3521928.095
300000	219504	894464	5458380.893
400000	298432	1228928	7443856.19
500000	382512	1568928	9465784.285
600000	469008	1908928	11516761.79
700000	555200	2257856	13591896.78
800000	636864	2617856	15687712.38
900000	723680	2977856	17801608.93
1000000	815024	3337856	19931568.57

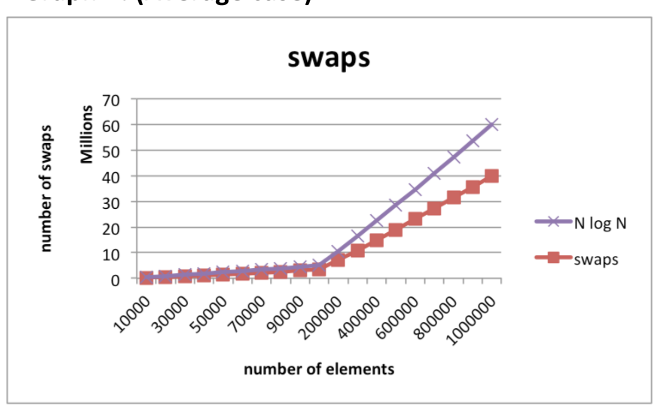
Comparisons: Total number of comparisons performed by the merge sort algorithm varies depending on the order of data in the array. Given the tables above, it appears that on average the number of comparisons in for a sorted and reverse-sorted array are roughly **½** of the comparisons for a randomly-sorted array (average-case).

Swaps: Total number of swaps has a performance of **O(N*log$_2$N)** regardless of the content or size of the array. Graph B below shows the graph for "swaps" roughly correlates to the graph for "N log$_2$N". With merge sort, swaps in the best-case, worst-case, and average-case scenarios all return the same performance.

Graph A. (Average case)



Graph B. (Average case)



5 / Quick sort

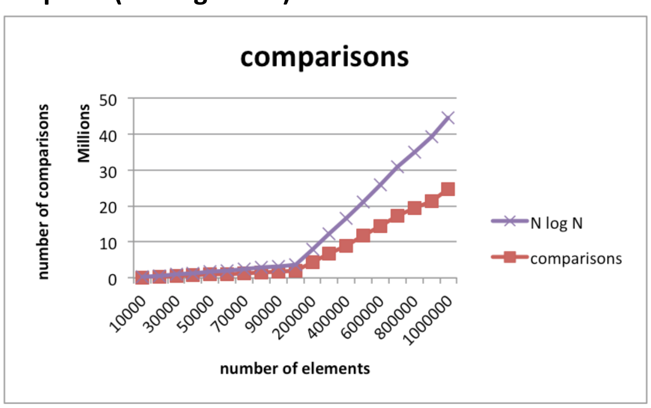
Quick sort			
N	comparisons	swaps	N log N
10000	155945	2470	132877.1238
20000	362181	3086	285754.2476
30000	553670	2485	446180.2464
40000	749568	9885	611508.4952
50000	943497	9988	780482.0237
60000	1132121	5957	952360.4928
70000	1361224	9468	1126654.711
80000	1584236	2978	1303016.99
90000	1750860	20649	1481187.364
100000	1965914	23837	1660964.047
200000	4466254	22990	3521928.095
300000	6730113	47100	5458380.893
400000	9029675	100007	7443856.19
500000	11722351	118684	9465784.285
600000	14429516	130535	11516761.79
700000	17279004	157803	13591896.78
800000	19365220	193219	15687712.38
900000	21410955	38504	17801608.93
1000000	24633869	175120	19931568.57

Cases
Random unsorted array
Sorted array
Reverse sorted array

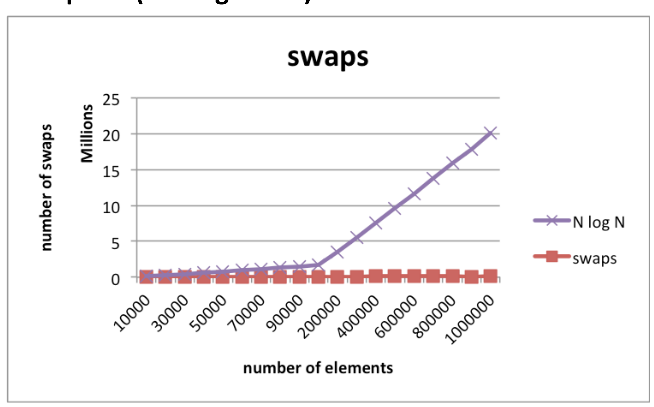
Comparisons: Total number of comparisons performed by the quick sort algorithm has a performance of **O(N*log$_2$N).** Looking at Graph A below, the "comparisons" line roughly correlates to the "N log$_2$N" line. The number of comparisons performed varies based on data in the array. For sorted and reverse-sorted arrays, quick sort performs many more comparisons than for a randomly-sorted array. For pre-sorted arrays, the computational performance increases to **O(N^2),** which is the worst-case scenario.

Swaps: Total number of swaps performed varies depending on the order of the array data. Graph B below shows the results for the average-case scenario, showing that "swaps" is not closely correlated with "N log$_2$N". The number of swaps stays relatively constant for arrays that are randomly sorted.

Graph A. (Average case)



Graph B. (Average case)



6 / Heap sort

Heap sort			
N	comparisons	swaps	N log N
10000	235455	124271	132877.1238
20000	510563	268402	285754.2476
30000	800678	419798	446180.2464
40000	1101648	576674	611508.4952
50000	1409554	737545	780482.0237
60000	1721834	899999	952360.4928
70000	2038112	1064862	1126654.711
80000	2362687	1233344	1303016.99
90000	2690514	1403571	1481187.364
100000	3019080	1574744	1660964.047
200000	6439207	3349970	3521928.095
300000	10001110	5195801	5458380.893
400000	13678085	7098998	7443856.19
500000	17396509	9024194	9465784.285
600000	21202459	10991693	11516761.79
700000	25065692	12988553	13591896.78
800000	28955817	15000480	15687712.38
900000	32867515	17021120	17801608.93
1000000	36792057	19047164	19931568.57

Cases
Random unsorted array
Sorted array
Reverse sorted array

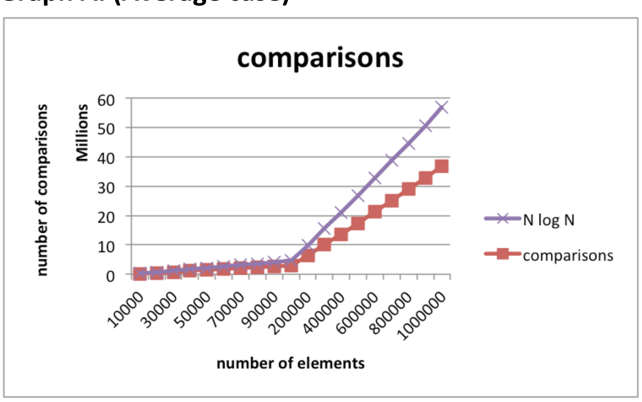
N	comparisons	swaps	N log N
100000	244460	131956	1660964.047
200000	529074	282878	3521928.095
300000	826347	440100	5458380.893
400000	1138114	605202	7443856.19
500000	1455438	773304	9465784.285
600000	1772744	941416	11516761.79
700000	2099178	1114302	13591896.78
800000	2436967	1293150	15687712.38
900000	2774750	1471998	17801608.93
1000000	3112517	1650854	19931568.57

N	comparisons	swaps	N log N
100000	226682	116696	1660964.047
200000	493307	254334	3521928.095
300000	775687	399212	5458380.893
400000	1067779	547628	7443856.19
500000	1366047	698892	9465784.285
600000	1670717	854794	11516761.79
700000	1978603	1012682	13591896.78
800000	2292813	1174074	15687712.38
900000	2608253	1334932	17801608.93
1000000	2926640	1497434	19931568.57

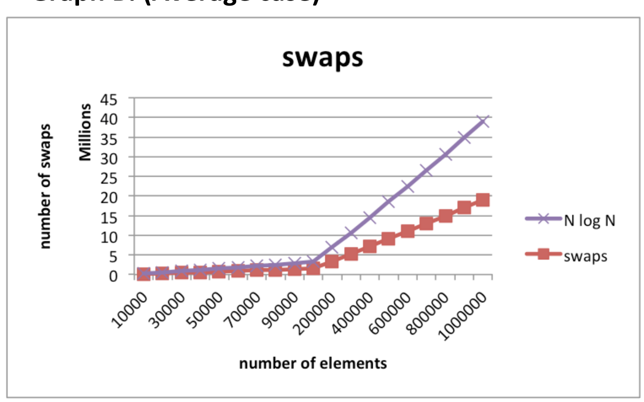
Comparisons: Total number of comparisons for the heap sort algorithm implementation has a performance of **O(N*logN)**. The number of comparisons that this sort method returns is roughly the same regardless of the data in the array. For the best-case, worst-case, and average-case scenarios, the number of comparisons grows at a factor/rate that is approx. N $\log_2 N$.

Swaps: Total number of swaps performed is also consistent among sorted, reverse-sorted, and random-sorted arrays. The tables above show that for all 3 cases, the number of swaps performed is roughly ½ the number of comparisons. Given Graph B below, we see that the heap sort algorithm is more efficient for large arrays, which require fewer and fewer number of swaps relative to N $\log_2 N$.

Graph A. (Average case)



Graph B. (Average case)

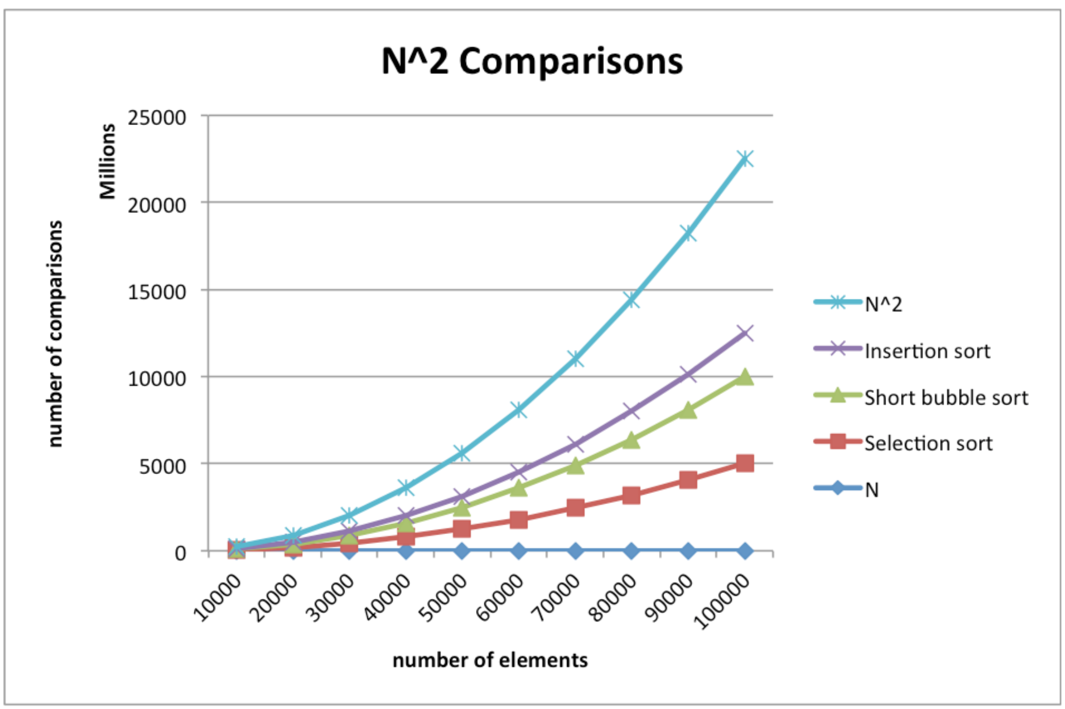


7 / $O(N^2)$ sorts summary

A. Comparisons Analysis

Comparisons chart				
N	Selection sort	Short bubble sort	Insertion sort	N^2
10000	49995000	49995000	24993946	100000000
20000	199990000	199990000	99870366	400000000
30000	449985000	449985000	224344004	900000000
40000	799980000	799980000	401007807	1600000000
50000	1249975000	1249975000	623912746	2500000000
60000	1799970000	1799970000	901502108	3600000000
70000	2449965000	2449965000	1224064202	4900000000
80000	3199960000	3199960000	1596653764	6400000000
90000	4049955000	4049955000	2028542782	8100000000
100000	4999950000	4999950000	2494443511	10000000000

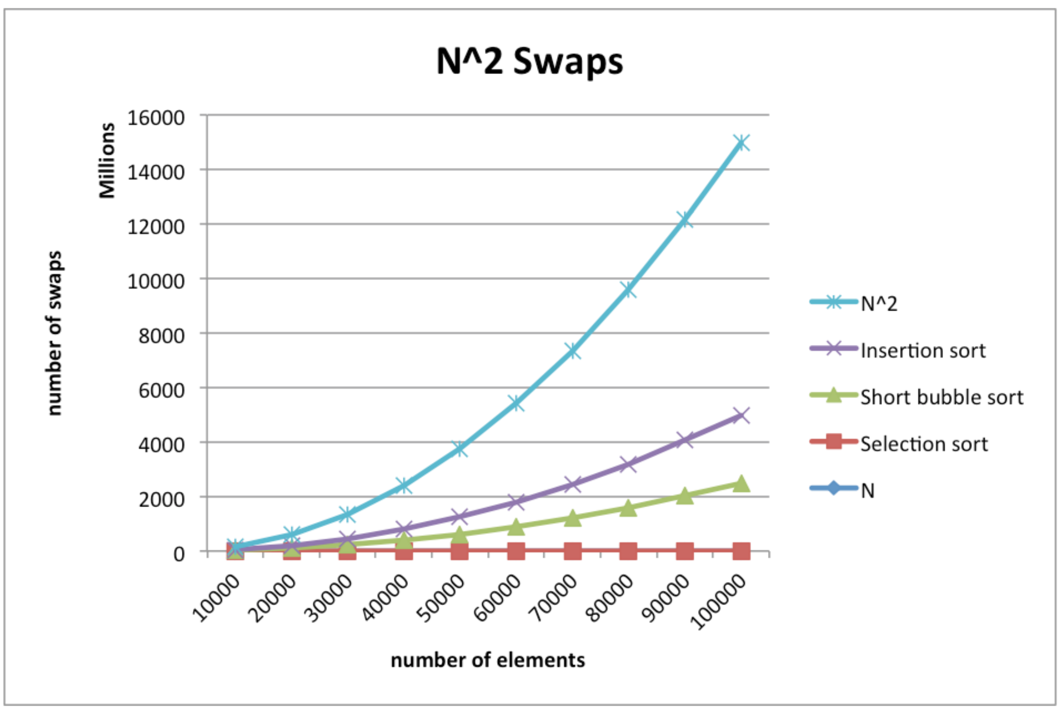
Looking at the graph below, we can see that the order of most efficient to least efficient sorts is **(1) selection, (2) short bubble, and (3) insertion**, when comparing relative performance from the point of view of the number of comparisons performed. With larger arrays, insertion sort is most correlated with $\mathbf{O(N^2)}$ performance. With smaller arrays, however, insertion sort is actually more efficient than both selection and short bubble sorts, requiring fewer total comparisons. We can see this clearly in the table above.



B. Swaps Analysis

Swaps chart				
N	Selection sort	Short bubble sort	Insertion sort	N^2
10000	9999	24723138	24983956	100000000
20000	19999	100283178	99850376	400000000
30000	29999	225817331	224314017	900000000
40000	39999	399527558	400967814	1600000000
50000	49999	622304992	623862759	2500000000
60000	59999	903148623	901442119	3600000000
70000	69999	1221988687	1223994214	4900000000
80000	79999	1597620163	1596573774	6400000000
90000	89999	2031258778	2028452790	8100000000
100000	99999	2498371476	2494343529	10000000000

In the graph below, it appears that the order of most efficient to least efficient sorts is **(1) selection, (2) short bubble, and (3) insertion**, when comparing relative performance from the point of view of the number of swaps performed. As we can see in the graph below, selection sort is requires the fewest number of swaps among all 3 sorts, for any size of array. Selection sort has average performance of **O(N)** from the point of view of total swaps.

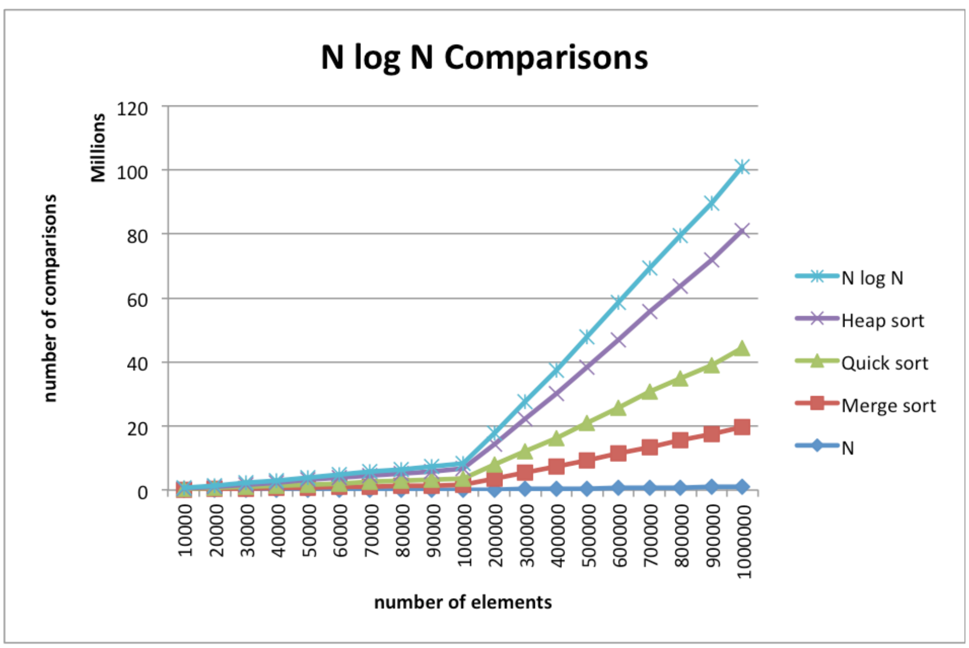


8 / O(N $\log_2$N) sorts summary

A. Comparisons Analysis

Comparisons chart				
N	Merge sort	Quick sort	Heap sort	N^2
10000	120447	155945	235455	132877.1238
20000	260962	362181	510563	285754.2476
30000	408632	553670	800678	446180.2464
40000	561629	749568	1101648	611508.4952
50000	718175	943497	1409554	780482.0237
60000	877239	1132121	1721834	952360.4928
70000	1038913	1361224	2038112	1126654.711
80000	1203936	1584236	2362687	1303016.99
90000	1369468	1750860	2690514	1481187.364
100000	1536099	1965914	3019080	1660964.047
200000	3272115	4466254	6439207	3521928.095
300000	5084838	6730113	10001110	5458380.893
400000	6945656	9029675	13678085	7443856.19
500000	8837070	11722351	17396509	9465784.285
600000	10769127	14429516	21202459	11516761.79
700000	12723289	17279004	25065692	13591896.78
800000	14690545	19365220	28955817	15687712.38
900000	16679980	21410955	32867515	17801608.93
1000000	18673917	24633869	36792057	19931568.57

Looking at the graph below, we can see that the order of most efficient to least efficient sorts is **(1) merge, (2) quick, and (3) heap**, when comparing relative performance from the point of view of the number of comparisons performed. With larger arrays (N > 100,000), heap sort is most correlated with **O(N^2)** performance. With smaller arrays, merge and quick sort both require fewer total comparisons than heap sort. We can see this clearly in the table above.



B. Swaps Analysis

Swaps chart				
N	Merge sort	Quick sort	Heap sort	N^2
10000	267232	2470	124271	132877.1238
20000	574464	3086	268402	285754.2476
30000	894464	2485	419798	446180.2464
40000	1228928	9885	576674	611508.4952
50000	1568928	9988	737545	780482.0237
60000	1908928	5957	899999	952360.4928
70000	2257856	9468	1064862	1126654.711
80000	2617856	2978	1233344	1303016.99
90000	2977856	20649	1403571	1481187.364
100000	3337856	23837	1574744	1660964.047
200000	7075712	22990	3349970	3521928.095
300000	10951424	47100	5195801	5458380.893
400000	14951424	100007	7098998	7443856.19
500000	18951424	118684	9024194	9465784.285
600000	23102848	130535	10991693	11516761.79
700000	27302848	157803	12988553	13591896.78
800000	31502848	193219	15000480	15687712.38
900000	35702848	38504	17021120	17801608.93
1000000	39902848	175120	19047164	19931568.57

In the graph below, it appears that the order of most efficient to least efficient sorts is **(1) quick, (2) merge, and (3) heap**, when comparing relative performance from the point of view of the number of swaps performed. For larger arrays, quick sort and merge sort have similarly correlated performance complexities. For smaller arrays (N < 100,000), quick sort requires the fewest number of swaps among all 3 algorithms -- followed by heap and merge sort, respectively. Heap sort has average performance most correlated to **O(N log$_2$N)** from the point of view of total swaps.