CS 301
Sort Report

By Scott Mikolyski
11/30/05

Abstract

This report compares six different sorting algorithms, the selection sort, insertion sort, bubble sort, merge sort, heap sort, and quick sort. By comparing the efficiency of these different algorithms, evidence reveals which sort performs the fastest. The bubble sort is the least efficient in sorting but the easiest to program. The selection sort is faster than the bubble sort and comes close but doesn't beat the insertion sort. The insertion sort runs fastest when used to sort smaller data sets. Larger data sets are better sorted using the heap sort. Merge sort performs faster than the heap sort but with vary large data sets, memory usage may be an issue. Quick sort is the fastest sorting algorithm, out of the six algorithms compared, to sort large data sets.

Objectives

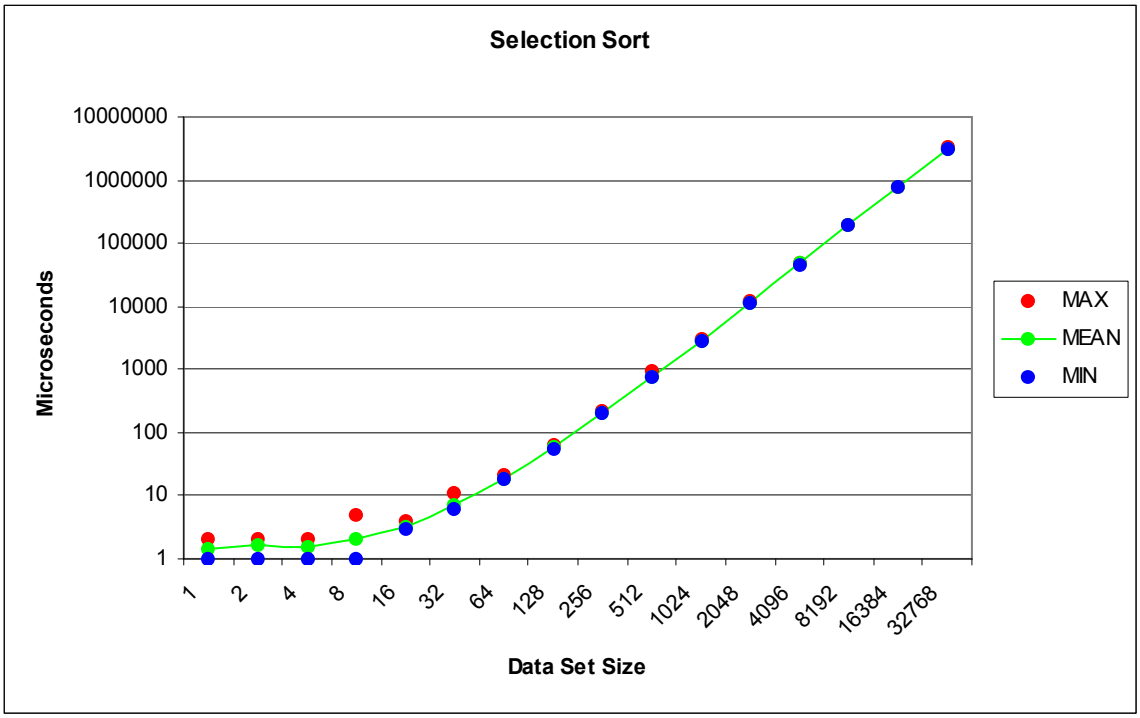
To find the fastest sorting algorithm and under what conditions such performance is achieved. Also, to find the pros and cons of each sorting algorithm and to produce evidence when to use one algorithm over another.

Methodology

Each algorithm will start with sorting a single element data set. The number of elements in the data set will double until the amount of time taken to sort the data set exceeds 1000000 microseconds (1 second). Speed will be measured calling the **gettimeofday()** function, found in the **time.h** C++ library, before and after sorting with the difference being the duration of the sort. The data set will be populated using the **rand()** function found in the C++ library with the seed initialized using the **time()** function also found in the C++ library. Each algorithm will sort the same data set size, populated with different random data, 100 times to calculate the mean running time.

Results: Selection Sort

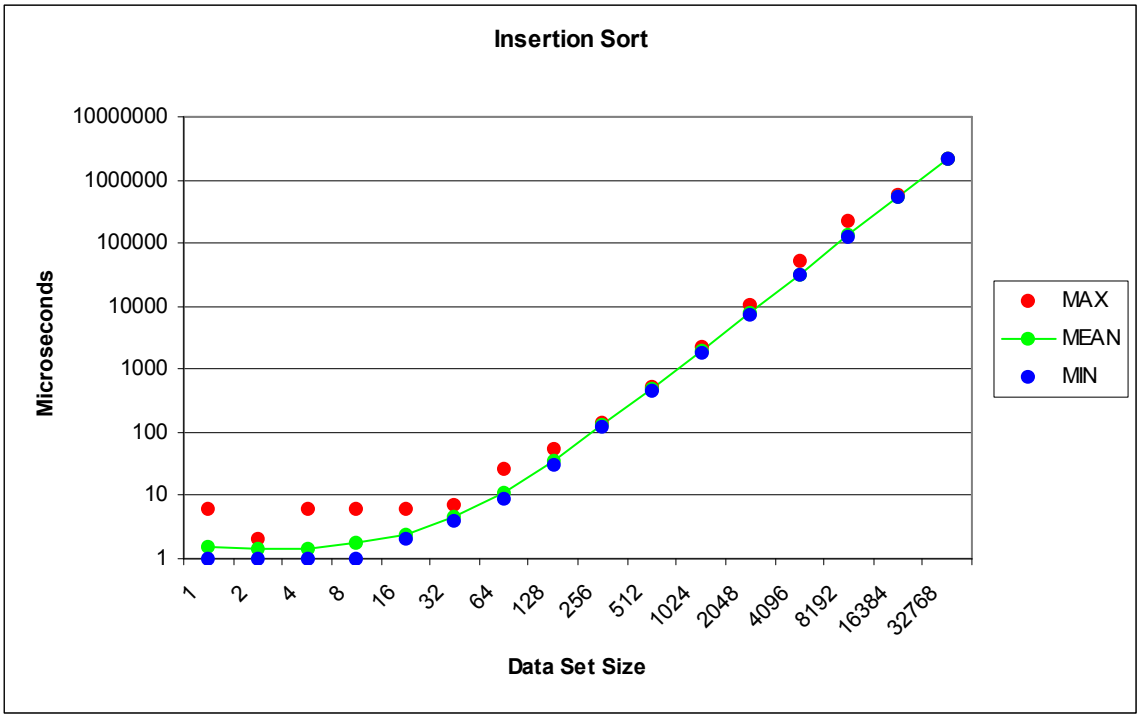
The selection sort works by taking the smallest element from the unsorted list and puts it in the next position of the sorted list. This sorting algorithm has a running time of O(*n*^2).



SIZE	HIGH	AVG	LOW	SAMPLE
1	2	1.43	1	1
2	2	1.61	1	1
4	2	1.53	1	2
8	5	2.01	1	2
16	4	3.17	3	3
32	11	6.92	6	6
64	22	18.65	18	18
128	65	59.09	56	59
256	223	206.52	201	207
512	979	762.28	736	760
1024	3030	2917.53	2829	2929
2048	11830	11410.63	11149	11402
4096	49708	47079.49	46844	46987
8192	191591	189565.67	189077	189125
16384	775832	757828.63	756214	757040
32768	3249174	3033434.43	3023107	3025216

Results: Insertion Sort

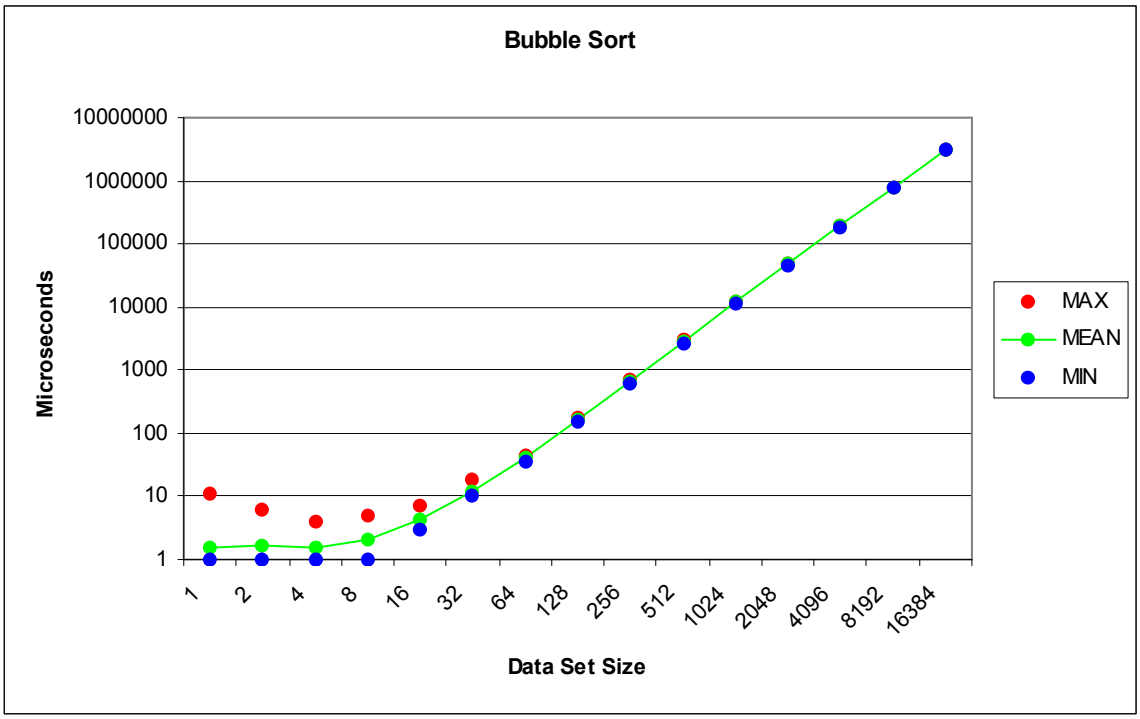
The insertion sort takes the last element from the unsorted list and places the element where it belongs into the sorted list, inserting the unsorted element in position within the sorted elements. This algorithm has a running time of O(*n*^2).



SIZE	HIGH	AVG	LOW	SAMPLE
1	6	1.50	1	2
2	2	1.46	1	2
4	6	1.47	1	2
8	6	1.74	1	2
16	6	2.43	2	3
32	7	4.48	4	5
64	26	11.22	9	11
128	56	35.66	31	35
256	141	128.73	119	131
512	529	498.75	468	485
1024	2196	1949.77	1873	1950
2048	10488	7741.11	7422	7654
4096	52710	31974.89	30788	30947
8192	227379	132066.68	128126	129487
16384	575642	530490.61	522851	531719
32768	2147743	2127822.34	2104476	2126070

Results: Bubble Sort

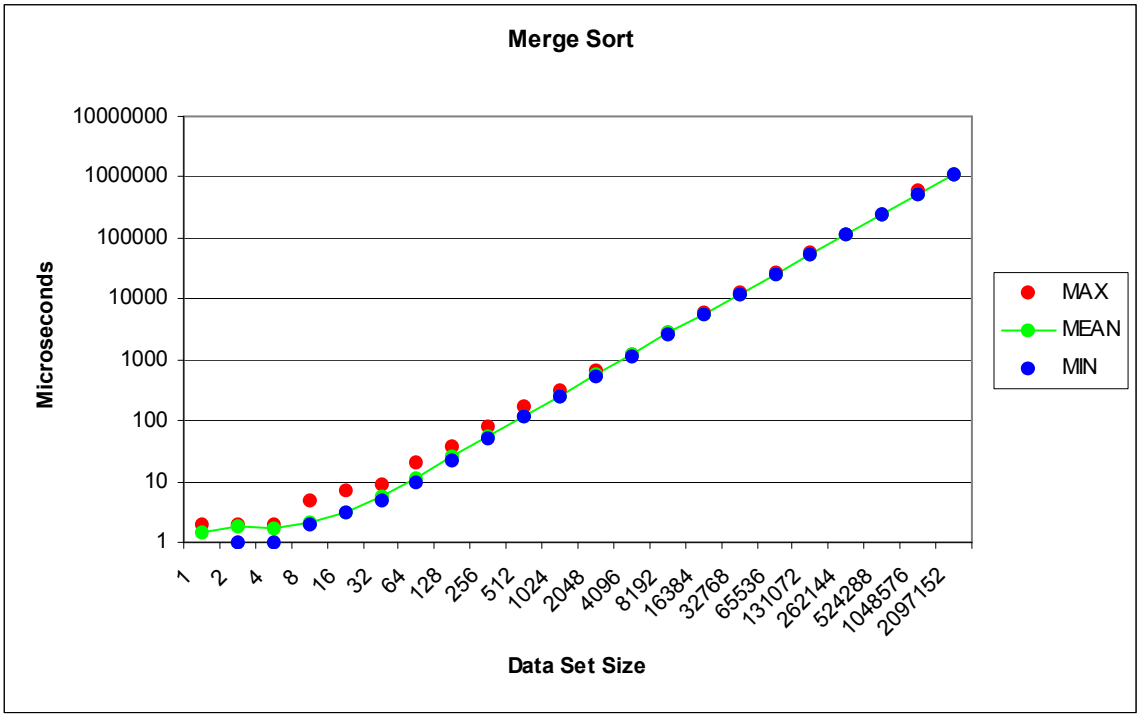
The bubble sort is named after the appearance of how the unsorted lists of elements behave. Every element is compared with its neighboring element; if the smaller element is on the larger side then the two elements switch places. The smaller elements appear to rise to the top of the list while the larger elements sink to the bottom. This sorting algorithm has a running time of O(*n*^2).



SIZE	HIGH	AVG	LOW	SAMPLE
1	11	1.57	1	2
2	6	1.70	1	2
4	4	1.59	1	2
8	5	2.11	1	2
16	7	4.16	3	4
32	19	11.81	10	11
64	46	40.77	36	41
128	177	160.92	149	161
256	732	679.99	632	668
512	3063	2879.55	2723	2965
1024	12260	11786.06	11170	12172
2048	49233	47310.97	46044	46888
4096	197085	190635.46	186157	188202
8192	779270	766523.09	752284	767456
16384	3186614	3075429.22	3045742	3052189

Results: Merge Sort

The merge sort is a divide and conquer algorithm. Taking advantage of recursion, the merge sort splits the unsorted list into two smaller unsorted lists until it only has to sort two elements, returning the sorted list to be merged with the other sorted half. A unique characteristic of this sort is how it uses twice as much space as the data set to be sorted. Because of this trait, the merge sort algorithm may not be a good idea when working with a large number of elements. As a recursive algorithm, the merge sort has a running time of $O(n \log n)$.

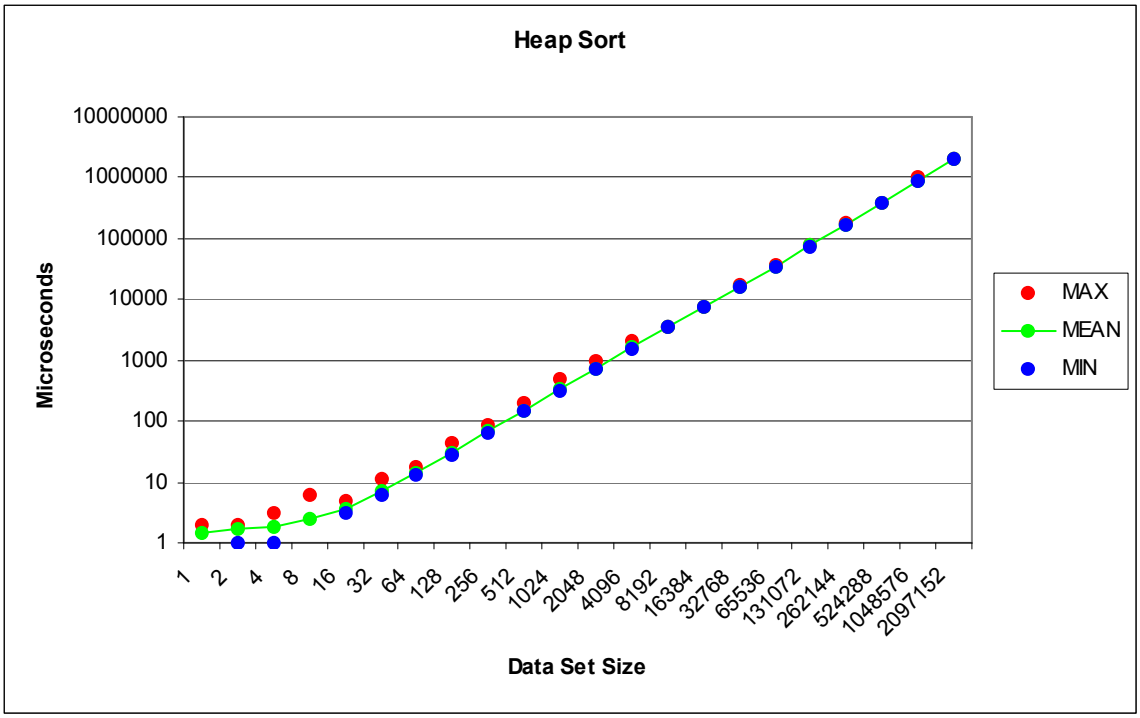


SIZE	HIGH	AVG	LOW	SAMPLE
1	2	1.44	0	2
2	2	1.77	1	2
4	2	1.71	1	2
8	5	2.16	2	2
16	7	3.23	3	3
32	9	5.60	5	6
64	20	11.46	10	12
128	37	24.96	23	25
256	81	54.49	52	54
512	168	118.36	114	118
1024	321	256.21	248	263

SIZE	HIGH	AVG	LOW	SAMPLE
2048	680	559.64	547	553
4096	1260	1202.87	1182	1218
8192	2833	2733.48	2690	2742
16384	5861	5680.37	5577	5751
32768	12768	12178.23	11944	12226
65536	27700	25988.45	25378	26053
131072	56113	55006.55	53606	55017
262144	117805	116668	114853	117256
524288	250671	247659.6	243352	248308
1048576	596622	523000.3	515171	522277
2097152	1139245	1099205	1089087	1100985

Results: Heap Sort

The heap sort is a non-recursive, divide and conquer algorithm. This algorithm creates a heap out of the unsorted list of elements. As the list is "heapified," the larger elements trickle down to the bottom of the heap. This sorting algorithm is ideal when dealing with very large data sets because all of the sorting is performed in-place and this algorithm doesn't require recursion to execute. As a divide and conquer algorithm, the heap sort has a running time of $O(n \log n)$.

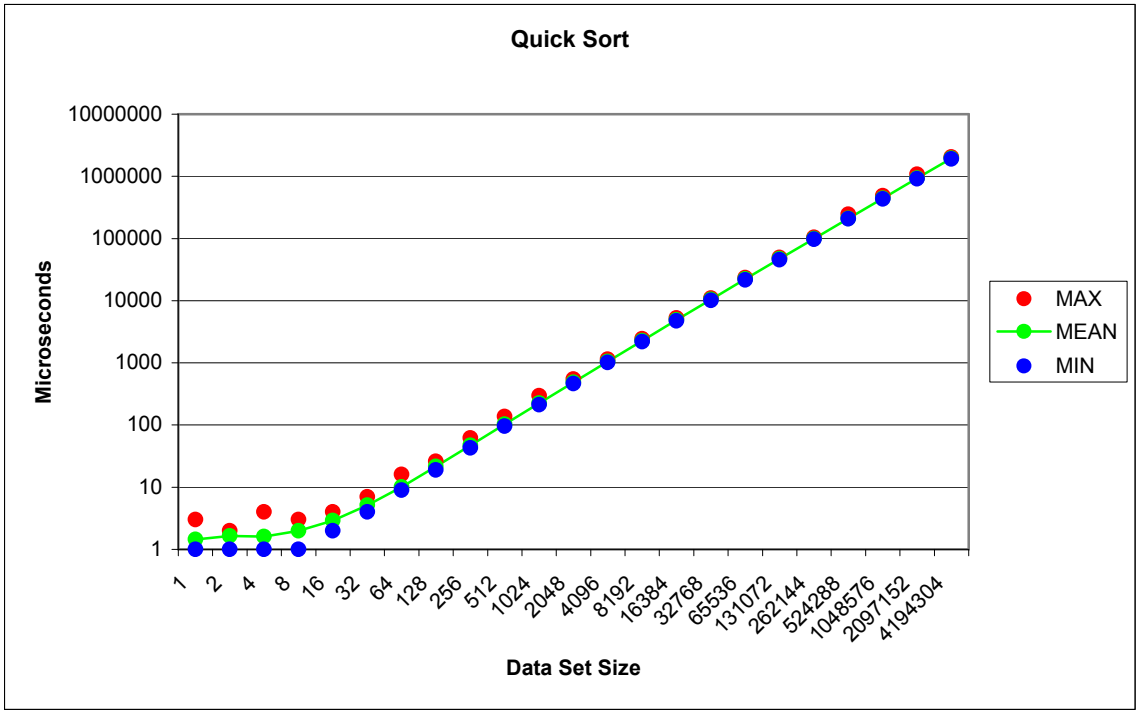


SIZE	HIGH	AVG	LOW	SAMPLE
1	2	1.49	0	1
2	2	1.70	1	2
4	3	1.90	1	2
8	6	2.44	0	3
16	5	3.69	3	4
32	11	6.91	6	7
64	18	13.97	13	14
128	44	30.58	29	34
256	89	67.71	65	67
512	201	150.71	147	148
1024	485	334.16	325	327

SIZE	HIGH	AVG	LOW	SAMPLE
2048	965	733.31	721	730
4096	2143	1608.10	1587	1606
8192	3568	3488.89	3457	3495
16384	7658	7556.40	7505	7586
32768	16838	16464.34	16138	16563
65536	36170	35592.18	34821	35809
131072	77893	76881.37	75305	75805
262144	181284	168369.8	165340	168584
524288	387282	383624.7	377881	384126
1048576	1043095	891644.6	877730	897943
2097152	2087064	2041417	2018684	2027844

Results: Quick Sort

The quick sort is a recursive, divide and conquer algorithm. What makes the quick sort unique is how it utilizes a pivot element to split the unsorted data set into two sub sets, elements smaller and elements larger than the pivot. Recursively, the sub sets are split again using another pivot element, until only one element remains. The quick sort algorithm has an average case running time of O(*n* log *n*), worst case O(*n*^2), sorts in-place, and is very recursive.

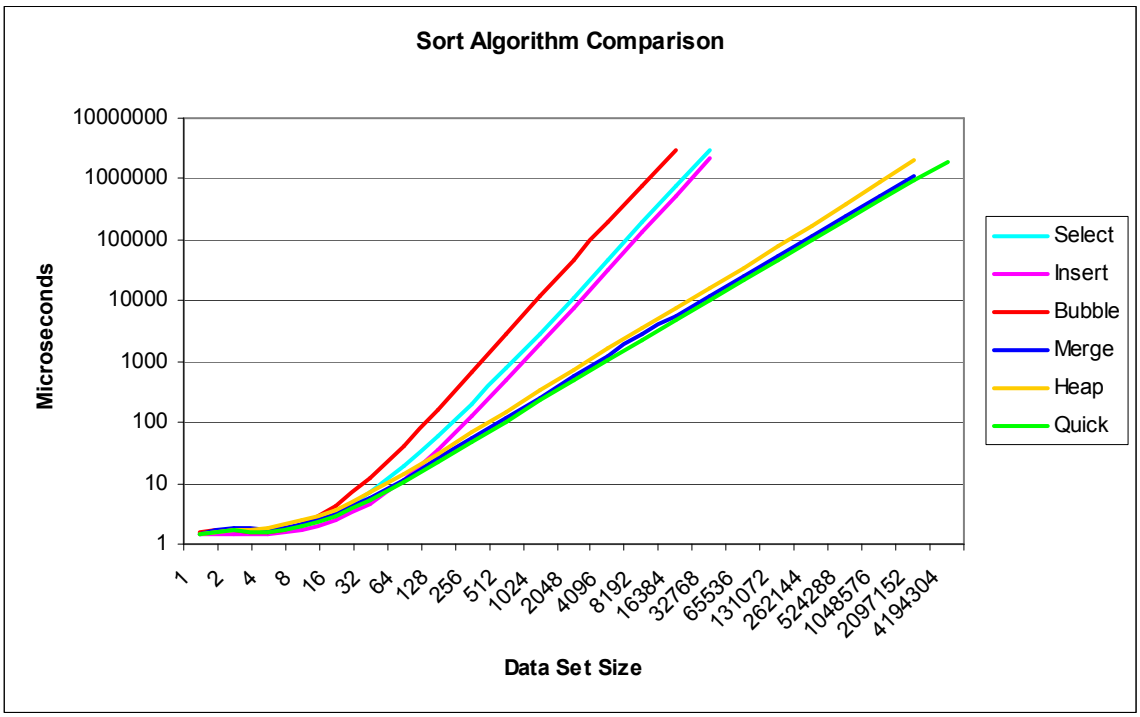


SIZE	HIGH	AVG	LOW	SAMPLE
1	3	1.44	1	2
2	2	1.64	1	2
4	4	1.61	1	2
8	3	2.00	1	2
16	4	2.92	2	3
32	7	5.12	4	5
64	16	10.12	9	10
128	26	21.51	19	22
256	62	47.07	43	47
512	136	102.42	96	99
1024	296	223.88	211	218
2048	548	488.46	464	486

SIZE	HIGH	AVG	LOW	SAMPLE
4096	1147	1051.04	1010	1080
8192	2448	2271.86	2183	2305
16384	5285	4903.11	4717	4766
32768	11006	10466.40	10139	10258
65536	23454	22267.21	21567	21938
131072	49724	47185.31	45717	48266
262144	103765	99999.12	97554	99938
524288	245517	211117.75	206291	218584
1048576	487217	443976.31	434074	459659
2097152	1079728	935841.85	909084	917116
4194304	2054194	1951066	1906597	1934614

Conclusion

By overlapping the mean running time of all six sorting algorithms, it is possible to compare their performances respectively.



As the above graph illustrates, the quick sort algorithm proves to be the fastest but only while the data set to be sorted is greater then 64 elements.

The Merge sort algorithm comes in second place for swiftness, but because of its demand in memory, the heap sort algorithm would be a better choice for larger data sets and is only a little bit slower in speed.

With data sets fewer than 64 elements, the insertion sorting algorithm is the ideal choice but with any data set larger would lower the performance substantially. The same is true with the selection sorting algorithm, which slower in performance than the insertion sort.

The least efficient of the compared sorting algorithms, which also happens to be the easiest to implement, is the bubble sorting algorithm.

Appendix

Typical data used for sorting, created with the rand() function and using seed(time(NULL)).

1804289383	1967513926	596516649	1726956429
846930886	1365180540	1189641421	336465782
1681692777	1540383426	1025202362	861021530
1714636915	304089172	1350490027	278722862
1957747793	1303455736	783368690	233665123
424238335	35005211	1102520059	2145174067
719885386	521595368	2044897763	468703135
1649760492	294702567	1801979802	1101513929

Sample speeds in microseconds collected when running the insertion sort algorithm with a data set size of 32 numbers.

5	4	4	4	4	4	5	5
4	5	4	4	5	4	7	5
5	5	4	4	5	4	4	4
5	7	4	5	4	4	4	5

Sample speeds in microseconds collected from running the quick sort algorithm with a data set size of 262,144 numbers.

100296	100073	98614	99139
101496	100348	99977	99858
100191	102091	100166	99999
99980	99548	97815	97554
99951	102365	101051	99826
99922	100521	99861	99818
99977	98975	98791	98842
99417	100646	100849	101568