Shawn Stern – 0921475

ALGORITHM 1	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
16	0	0	0
1000	1	1	1
10000	47	46	47
25000	292	291	290
40000	747	745	744
100000	4655	4647	4651

ALGORITHM 2	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
16	0	0	0
1000	2	0	0
10000	157	62	93
25000	977	389	582
40000	2409	995	1491
100000	15409	6225	9327

ALGORITHM 3	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
16	0	0	0
1000	0	0	1
10000	23	0	46
25000	145	0	291
40000	372	0	745
100000	2328	0	4652

ALGORITHM 4	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
16	0	0	0
1000	0	0	0
10000	1	0	1
25000	2	1	1
100000	11	6	6
1000000	129	73	66
10000000	1463	798	740

ALGORITHM 5	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
10	30	0	1241
11	671	0	8367
12	6862	0	54219
100	x	0	x
1000	x	0	x
10000	x	0	x
Large tests avoided for random and descending because they took ridiculous amounts of time with no hints of ending… Also, the averages are misleading because some are the average of times like 3ms + 12000ms + 80ms / 3 which shows the random nature of the algorithm.			

ALGORITHM 6	Average runtime over three runs(ms)		
Input Size	Random	Ascending	Descending
16	0	0	0
1000	0	1	0
2500	0	2	3
3000	0	4	3
4000	0	5	6
10000	1	overflow	overflow
25000	1	overflow	overflow
100000	6	overflow	overflow
1000000	72	overflow	overflow
Tests at 2500-4000added because of stack overflow past ~4700 elements for ascending and descending.			

Big-Oh Estimate					
1	2	3	4	5	6
$O(n^2)$ Selection Sort	$O(n^2)$ Bubble Sort	$O(n^2)$ Insertion Sort	O(n*logn) Merge Sort	O(???) (Indeterminate Runtime) Bogo Sort	O(n*logn) Quicksort

Rationale behind choices:

Algorithm 1 – Selection Sort

I believe that the first algorithm is an example of selection sort being used. The runtimes are nearly identical for each number of elements, regardless of the order of elements, which matches with the behavior of selection sort. The algorithm displays $O(n^2)$ growth regardless of the array's sorting (random, pre-sorted, reversed), which matches selection sort. It's pretty fast for small arrays, but loses effectiveness as the orders of magnitude continue to increase.

Algorithm 2 – Bubble Sort

I think the second algorithm is an example of bubble sort. The runtimes are very poor in every category, and the function is outperformed by every function other than Bogo sort. The function demonstrates O(n^2) performance in each category (matching the un-optimized performance of our version of bubble sort) as well. Comparison to other methods was the main driver behind deciding the identity of this algorithm.

Algorithm 3 – Insertion Sort

This algorithm clearly stood out as Insertion sort after the tests on ascending arrays when the runtime was 0ms for every array that was pre-sorted (at least for the array sizes tested). The runtimes for the list in reverse order are roughly double those of the randomly sorted arrays of similar sizing, with both displaying a rough growth rate of $O(n^2)$. This is to be expected with insertion sort, since the worst possible case for the algorithm is an array in perfectly reversed order. This by itself is a huge indication that this algorithm is insertion sort since, out of the algorithms that behave nicely, it is the only one where descending is the worst case.

Algorithm 4 – Merge Sort

This is the one algorithm where I got to go up to 10 billion elements before I got bored with how fast the thing sorts arrays compared to the other methods. This algorithm displays and incredibly low growth rate, comparable to O(nlogn) when graphed and with little difference based on if the arrays are random, pre-sorted, or reversed. Its ability to so quickly sort arrays of ridiculous sizes (regardless of setup) and low growth screams out merge sort.

Algorithm 5 – Bogo Sort

This algorithm stuck out immediately after the first few tests. With a 10 element array, it took anywhere from 0ms to 15 seconds (and more, randomly of course, no way I was willing to wait in some cases) to produce any results. This behavior clearly exhibits the “mix it randomly and check if it worked” methodology behind Bogo sort. It also has atrocious runtimes no matter what size the array is, which lends itself to the O(infinity) runtime that we mentioned in class (again, didn’t feel like testing anything too large).

Algorithm 6 – Quicksort Sort

This is the only sort that exhibited stack overflow errors, which on its own is indicative of the algorithm’s recursive nature. This encourages me to say quick sort right away (since the position of the pivot makes the algorithm prone to overflows, and causes worst case performance even on a sorted array!). It manages to outperform merge sort in the random cases, with runtimes comparable and actually better than O(n*logn), but with the pitfall of being unstable in the ascending and descending cases.