Analysis on Speed of Sorting Algorithms Involving Generated Sets of Arrays and ArrayLists

By Justin He

Hypothesis

For arrays and ArrayLists that are populated with random integer values, Selection Sort will take longer than Insertion Sort to complete. While both sorting algorithms have quadratic running times (both with an average case of O(n^2)), they way Insertion Sort works inherently makes it possible for it to run faster than Selection Sort, because Insertion Sort has the opportunity to save time if the element it is checking has no elements behind its spot that are greater than the element it is looking at. More details later.

For arrays and ArrayLists that are populated with integer values that are organized in reverse order, Selection Sort will take about the same time as Insertion Sort to complete. This case is the worst case scenario for Insertion Sort. In this case, in every interaction of the inner loop of Insertion Sort will scan and shift the entire sorted subsection of the array/ArrayList before inserting the next element (or the element it is "looking" at). Due to this factor, I expect the time of these algorithms to be very similar in this particular case.

For array and ArrayLists that are populated with integer values that are organized in correct ascending order, Selection Sort will take far longer than Insertion Sort to complete. This case is the best case scenario for Insertion Sort, because there is no need for it to shift the element it is "looking" at up the array ladder because there should be no elements preceding it that are higher than the element it is "looking" at. Due to this factor, in this particular case, I expect the time of the Insertion Sort Algorithm to be significantly faster than the time of the Selection Sort Algorithm.

Procedure

In order to carry out this experiment, I started out by importing PrintWriter, Random, and ArrayList. I used PrintWriter to draft CSV files, or Comma Separate Value files, in order to store data. I used the Random import to help populate arrays/ArrayLists. I imported ArrayList in order to involve ArrayLists (using wrapper class Integer) in my experiments.

Essentially, in order to test Sorting Algorithms, I generated an Array/ArrayList and populated it with random values, ascending values, or reverse ascending values depending on the situation. I created multiple methods for Insertion and Selection Sorts, so that Arrays and ArrayLists can be accepted as parameters. I used the currentTimeMillis() method of the System

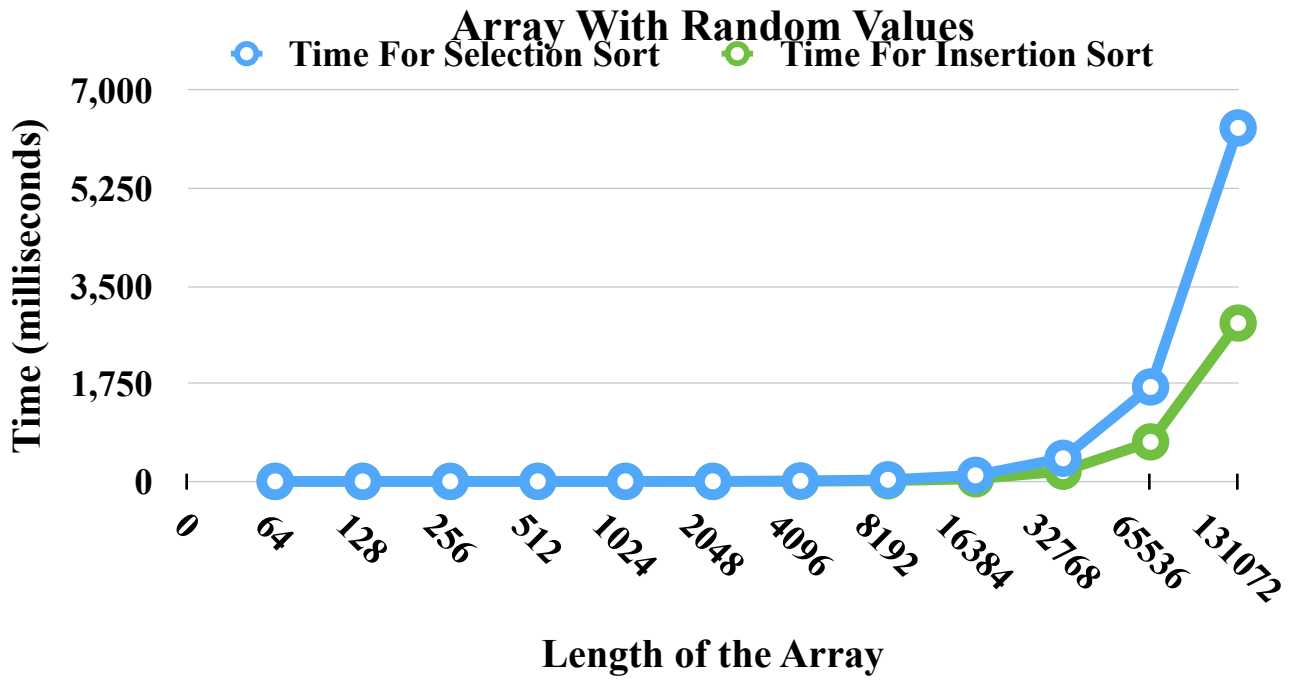
in order to time my tests. Furthermore, in order to prevent irregularities in my code, I ran 10 trials of each situation and then calculated an average in order to get a good measure of times.

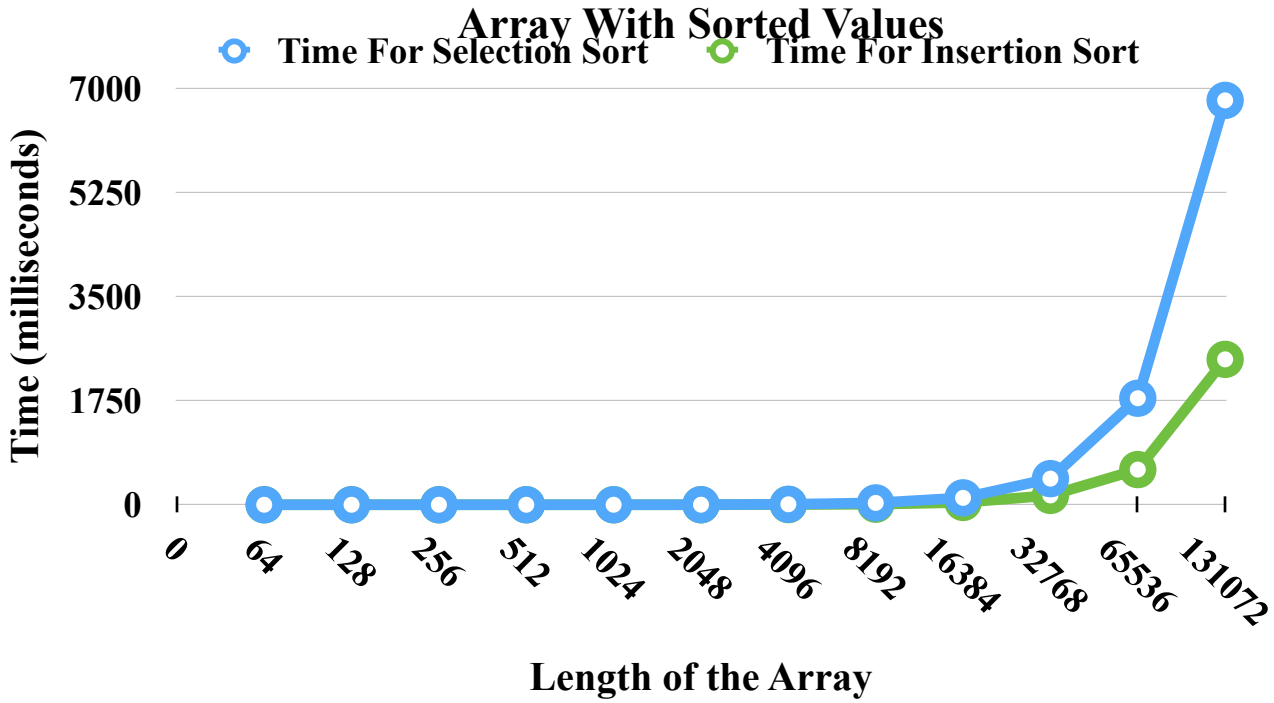
More specifically, in order to generate the random values in Arrays/ArrayLists, I used the nextInt() method of the Random class, which chooses a random number from the rangers Integer.MIN_VALUE and Integer.MAX_VALUE. I also used Arrays.sort() and the .sort() method of the ArrayList class in order to form the 'in-order' populations. For the 'reverse' populations, I simply used Arrays.sort() and the .sort() method of the ArrayList class in an opposite order, starting from the end of an array/ArrayList, and working backwards.

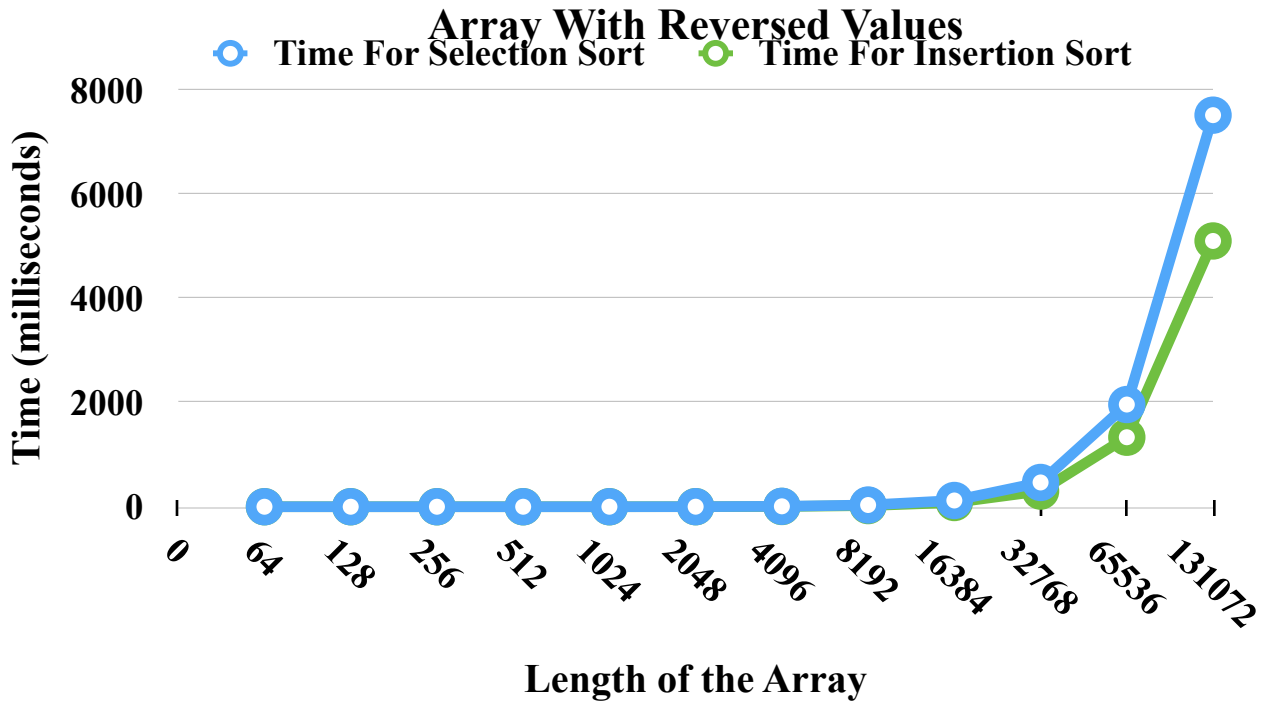
Below is an example of populating, sorting, and timing an array. As clarification, ARV stands for Array Random Value, as this section of code was used to generate, populate, and sort an Array With Random Values. Also, this code here also times the process and writes the information to a CSVFile.

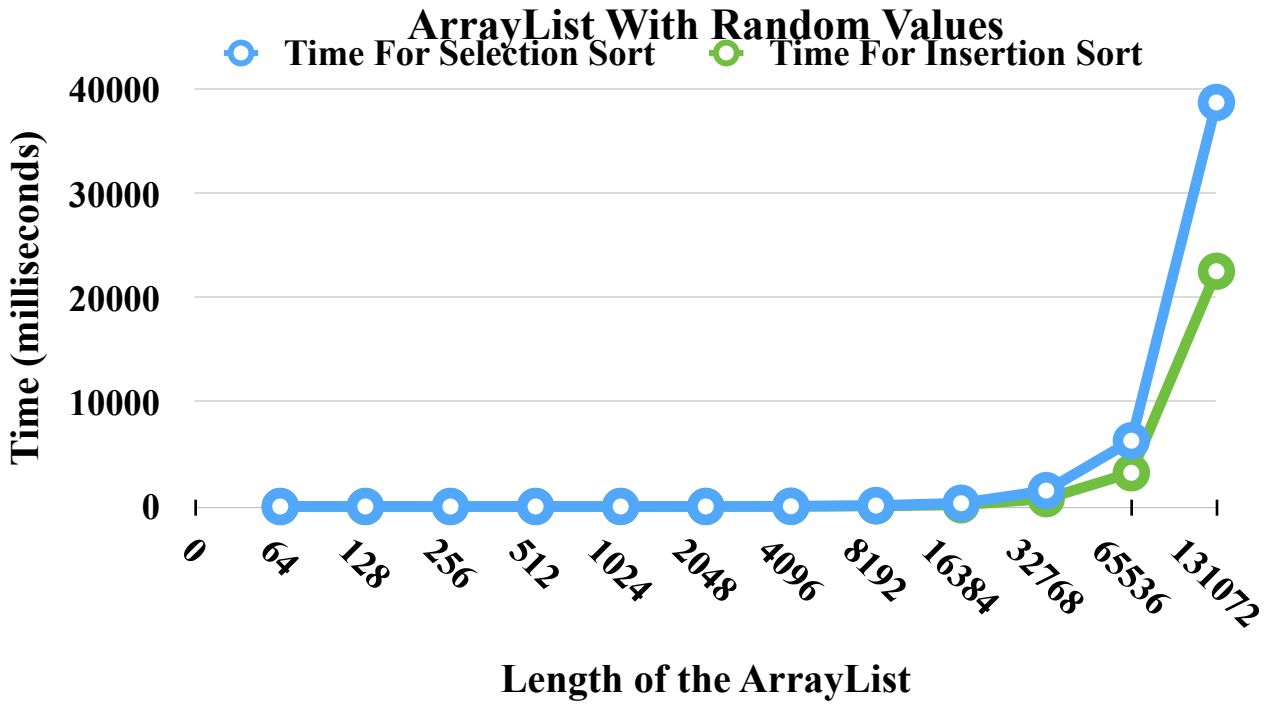
```
CSVFile.println("Array Random Values");
for (int x = 0; x < 12; x++){
     long TotalTime = 0;
     ARVLength = len;
     for (int r = 0; r < 10; r++){
          int[] ArrayStep = new int[len];
          for (int j = 0; j < ArrayStep.length; j++){
               ArrayStep[j] = random.nextInt();
          }
          long starttime1 = System.currentTimeMillis();
          ArrayStep = Selection(ArrayStep);
          long endtime1 = System.currentTimeMillis();
          TotalTime += (endtime1 - starttime1);
     }
     ARVSelection = TotalTime/10;
     TotalTime = 0;
     for (int r = 0; r < 10; r++){
          int[] ArrayStep = new int[len];
          for (int j = 0; j < ArrayStep.length; j++){
               ArrayStep[j] = random.nextInt();
          }
          long starttime1 = System.currentTimeMillis();
          ArrayStep = Insertion(ArrayStep);
          long endtime1 = System.currentTimeMillis();
          TotalTime += (endtime1 - starttime1);
     }
     ARVInsertion=TotalTime/10;
     CSVFile.println(ARVLength + "," + ARVSelection + "," +
ARVInsertion);
     len=len*2;
}
```

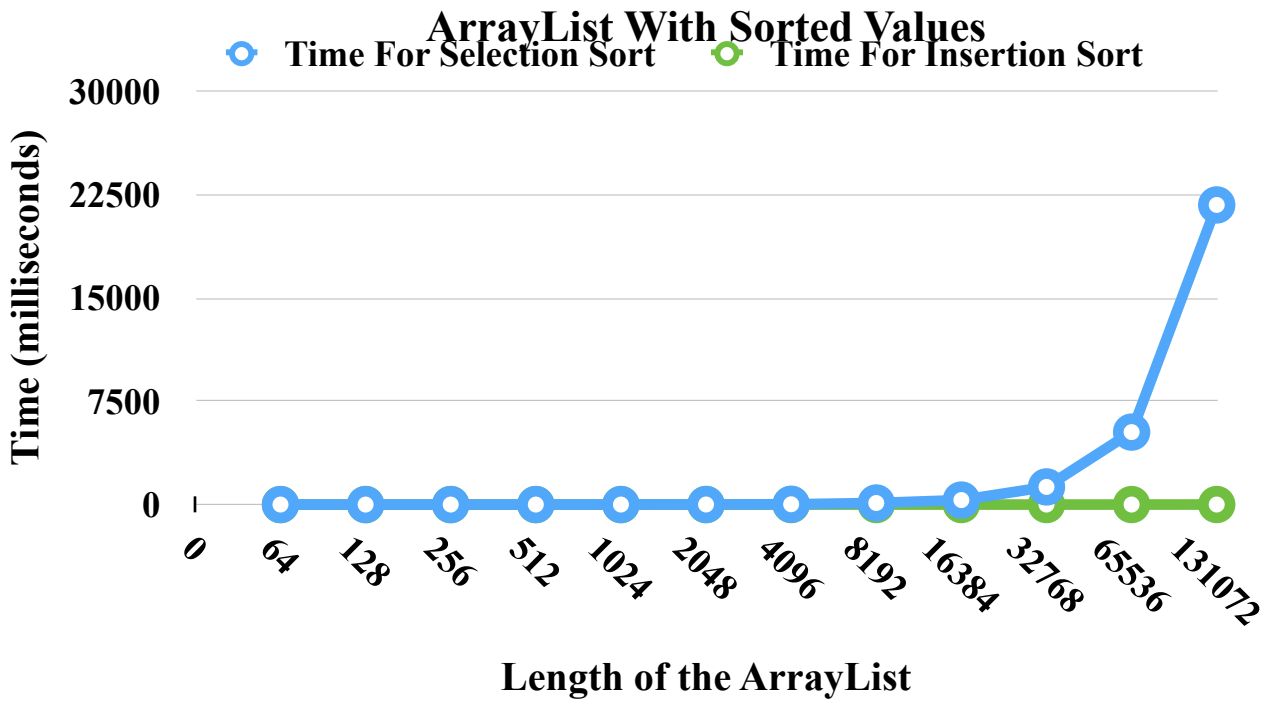
Data *warning: times may be slower from other graphs because I am running this on a (Mac) laptop processor

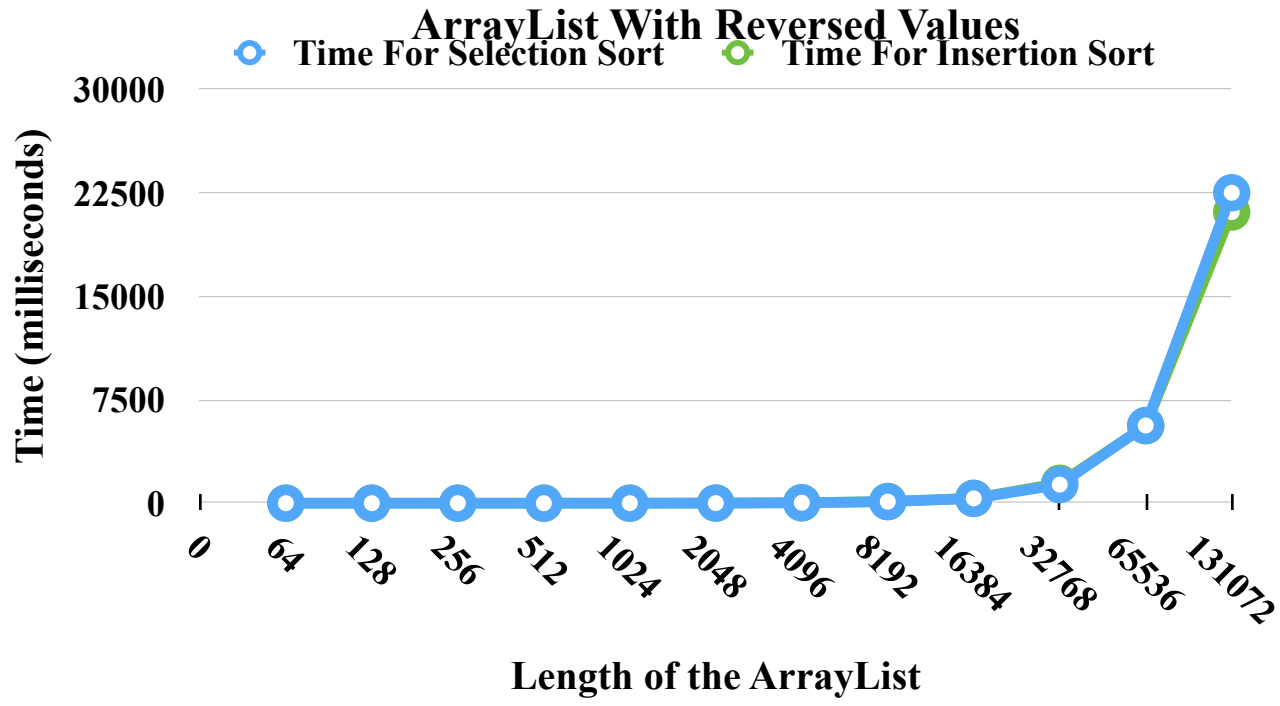





Array With Reversed Values
Time For Selection Sort
Time For Insertion Sort
Time (milliseconds)
8000
6000
4000
2000
0
0
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
Length of the Array


ArrayList With Random Values
Time For Selection Sort
Time For Insertion Sort
Time (milliseconds)
40000
30000
20000
10000
0
0
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
Length of the ArrayList


ArrayList With Sorted Values
Time For Selection Sort
Time For Insertion Sort
Time (milliseconds)
30000
22500
15000
7500
0
0
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
Length of the ArrayList


ArrayList With Reversed Values
Time For Selection Sort
Time For Insertion Sort
Time (milliseconds)
30000
22500
15000
7500
0
0
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
Length of the ArrayList

Evaluation

In essence, it seems as if the results indicate that my hypothesis is true. Let's look at each graph specifically.

In the Array with Random Values Graph, the time for selection sort to complete is longer than the time for insertion sort to complete, as expected. This discrepancy is exacerbated over time, as we can see that when the Array gets significantly longer (at 131072 spots), the difference between the timings of each sorting algorithm is the largest. The data table for this graph is Figure 1 in the Appendix.

In the Array with Sorted Values Graph, the time for selection sort to complete is significantly longer than the time for insertion sort to complete, as expected. This discrepancy is even more than the Random Values Graph discrepancy, because of the fact that this the is the best case scenario for Insertion Sort Algorithm. The data table for this graph is Figure 2 in the Appendix.

In the Array with Reverse Values Graph, the time for selection sort is nearly the same as the time for insertion sort to complete, as expected. This discrepancy is smaller than the other examples. This is best explained by the fact that this is Insertion Sort's worst case scenario, as explained before in the hypothesis. The data table for this graph is Figure 3 in the Appendix.

In the ArrayList with Random Values Graph, the time for selection sort to complete is longer than the time for insertion sort to complete, as expected. This discrepancy is exacerbated over time, as we can see that when the Array gets significantly longer (at 131072 spots), the difference between the timings of each sorting algorithm is the largest. The data table for this graph is Figure 4 in the Appendix.

In the ArrayList with Sorted Values Graph, the time for selection sort to complete is significantly longer than the time for insertion sort to complete, as expected. This discrepancy is even more than the Random Values Graph discrepancy, because of the fact that this the is the best case scenario for Insertion Sort Algorithm. The data table for this graph is Figure 5 in the Appendix.

In the ArrayList with Reverse Values Graph, the time for selection sort is nearly the same as the time for insertion sort to complete, as expected. This discrepancy is smaller than the other examples. This is best explained by the fact that this is Insertion Sort's worst case scenario, as explained before in the hypothesis. The data table for this graph is Figure 6 n the Appendix.

Conclusion

In summation, my hypothesis was mostly correct. All of the data gathered support my hypothesis. The only irregularity I can possibly point out is that the graph for Array With Sorted Values yields a relatively small difference in the time of selection sort and insertion sort, which is not as exaggerated as I expected. Other than that, most of the data clearly supports my hypothesis.

Some problems that could have affected my results might be irregular spikes — I tried my best to control for this by holding 10 trials per length per Array per type of population, however, irregular spikes, especially drastic ones, may have pulled my data in either direction. Furthermore, my computer's processor doesn't have the sole task of testing Sorting Algorithms in the .java file I used to run it; my computer is also working on background tasks. While I tried my best to control for this by not actively using my computer during the testing process, there are always background activities that can distort timing.

APPENDIX

Figure 1. Array Length	Time for Selection Sort (milliseconds)	Time for Insertion Sort (milliseconds)
64	0	0
128	0	0
256	0	0
512	0	0
1024	0	0
2048	1	0
4096	9	2
8192	29	11
16384	109	46
32768	414	183
65536	1691	708
131072	6331	2836
Figure 2		
64	0	0
128	0	0
256	0	0
512	0	0
1024	0	0
2048	1	0
4096	8	3
8192	31	9
16384	116	39
32768	439	157
65536	1790	591
131072	6795	2443
Figure 3		
64	0	0
128	0	0
256	0	0
512	0	0
1024	0	0
2048	1	1
4096	8	5
8192	33	22
16384	120	81
32768	466	300
65536	1962	1334
131072	7517	5102

Figure 4		
64	0	0
128	2	1
256	0	1
512	1	1
1024	2	0
2048	5	3
4096	21	12
8192	94	54
16384	315	160
32768	1527	746
65536	6312	3235
131072	38782	22577
Figure 5		
64	0	0
128	0	0
256	0	0
512	0	0
1024	1	0
2048	5	0
4096	21	0
8192	114	0
16384	327	0
32768	1281	0
65536	5258	1
131072	21742	3
Figure 6		
64	0	0
128	0	0
256	0	0
512	0	0
1024	1	1
2048	7	9
4096	31	29
8192	109	125
16384	348	352
32768	1346	1447
65536	5636	5614
131072	22525	21125