Empirical Analysis of Algorithms

Selection Sort, Insertion Sort, and Bubble Sort

By:

Emily De Lorenzo

Sara Lopez

Eric Sauer

Abstract

Sorting is an everyday function and, depending on the input size, it can be a very quick task or it can take hours to days to complete. When you run sorting algorithms the main concerns are their memory footprint, CPU usage, algorithm correctness and run time. So we are testing the running time complexities (best-case, worst-case, average-case) of selection sort, insertion sort, and bubble sort coded in Java using an empirical analysis to get qualitative data to compare the different running times and complexities of the algorithms. We believe that our data from running these test will lie somewhere around the average-case to best-case for each analysis given that we are taking the best time found for each sorting Algorithm. Our goal is to match run times to their big-O notation and improve algorithmic implementation.

Methodology

Our sorting algorithms are all coded in Java since it was the most known and understood language throughout our group. Although we do not know whether or not this has an effect on our timings or analysis. We decided to increment our array size by powers of 2 starting at 2^10 and going up by 1 every time, eventually getting to 2^20, but when we finally arrived at an array size of 2^20 ut took hours to run, so we decided to kill it and not include that size in our analysis, although 2^19 took a little over 2 hours to run as well. Initially we had also decided to run all of our tests with 64, 128, 256, and 512 trials, the first 2 sets would usually finish in a timely manner and we'd only get results hundredths of seconds better than the previous, but once we got into the higher array sizes the timings were taking a ridiculous amount of time as well, so in the end we decided to just finish out our analysis with using only 10 trials, but we included the other times that we previously got from the other trial sizes we'd ran.

Selection Sort

The selection sorting algorithm has a best-case, worst-case, and average-case of $O(n^2)$,where N is the number of elements, which is assumed due to the nested loop that it passes through every iteration. We chose to make an iterative version of this sorting algorithm based upon the fact that we are timing these trials and if it would have been done recursively then the time would have been exponentially slower.

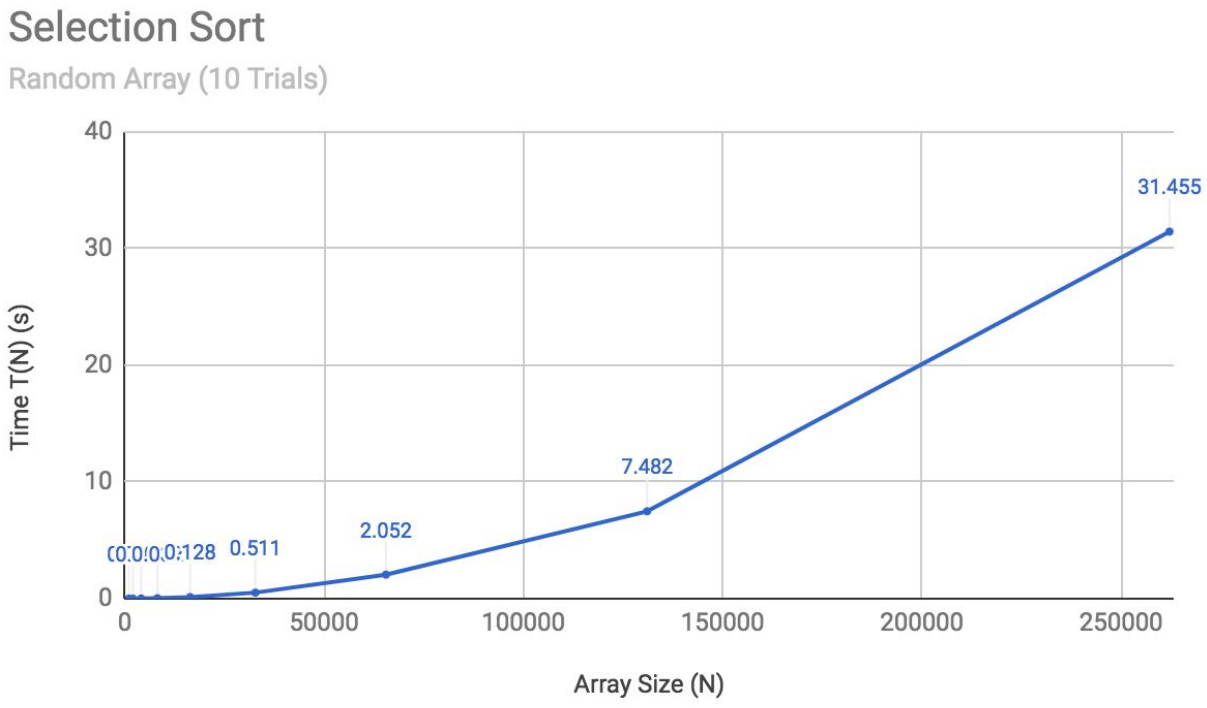
The algorithm itself is a searching and sorting algorithm; during each pass the unsorted element with the smallest/largest value is moved to its proper position in the array. The algorithm executes N - 1 times in order to cycle through the array and put it in order. In the selection sort, the inner loop finds the next smallest/largest value and the outer loop places that value into its proper location.

Testing Environment

Given that each sorting algorithm was used with a different machine, our results all varied from one to the next. The selection sort in particular was ran on a MacBook Pro with an Intel Core i5 processor with a speed of 2.4 GHz. The hardware has 2 cores, 2 L2 caches at 256 KB each, an L3 cache with 3 MB, and 4 GB of built in memory. This algorithm was coded and sorted using java version 8. The algorithm was accompanied by insertion, bubble sort, a randomizer, and a main method. By putting everything in one class it probably made the algorithm's run time longer and less efficient, but was easier to do in order to have everything in one.

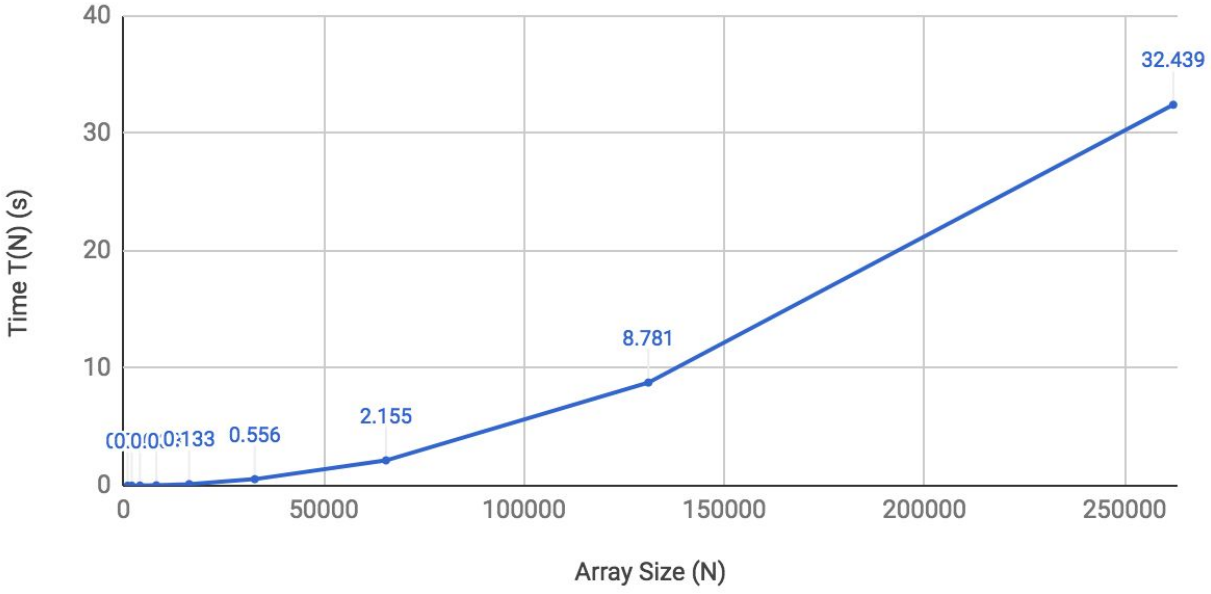
Selection Sort Results

There wasn't much surprise when we began to see the results. They hit around the expected $O(n^2)$ times, but were on the higher side, which could have probably been due to the processor that the trials were run on. And although speed is an important factor in this analysis, it isn't the only factor that is taken into account. The other factors, such as memory usage, code reusability, CPU usage, the algorithm's code, etc. are all very important factors that can increase the time that each trial took and given that the first tests didn't use up much of anything there is a chance that the first few have a better chance of being closer to their $O(n^2)$ time than the others. Something else to take into account is that the analysis was run on purely randomly generated arrays and it also accounts for the time to create all of the randomly generated arrays for every trial. Overall we believe that our results are a close representation of our hypothesis.



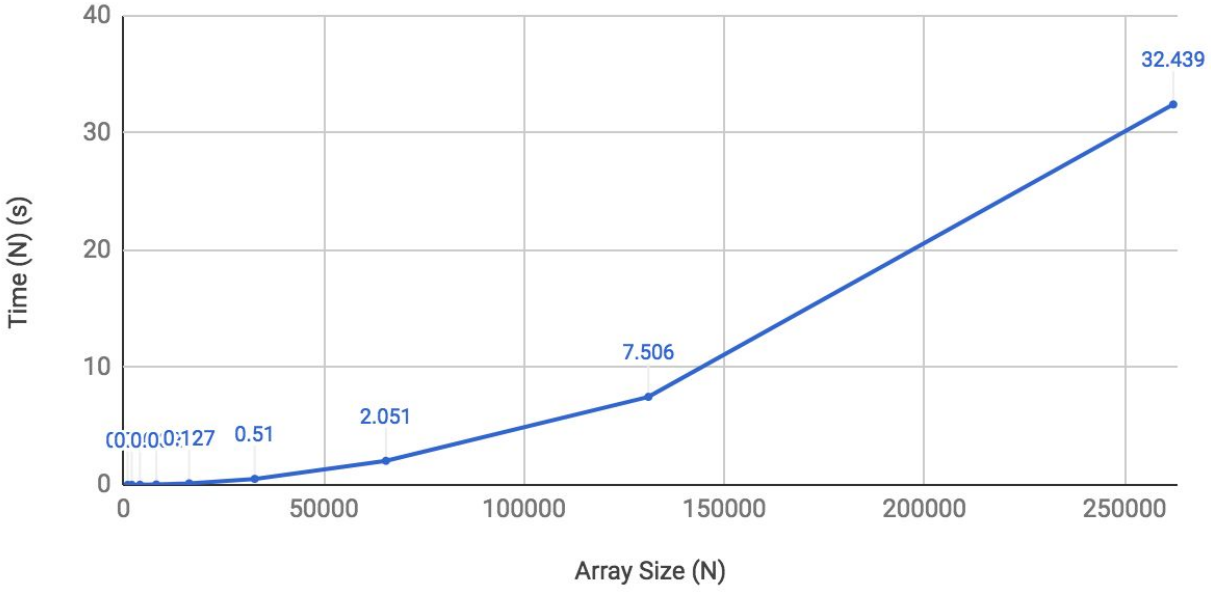
Selection Sort

Descending Array (10 Trials)



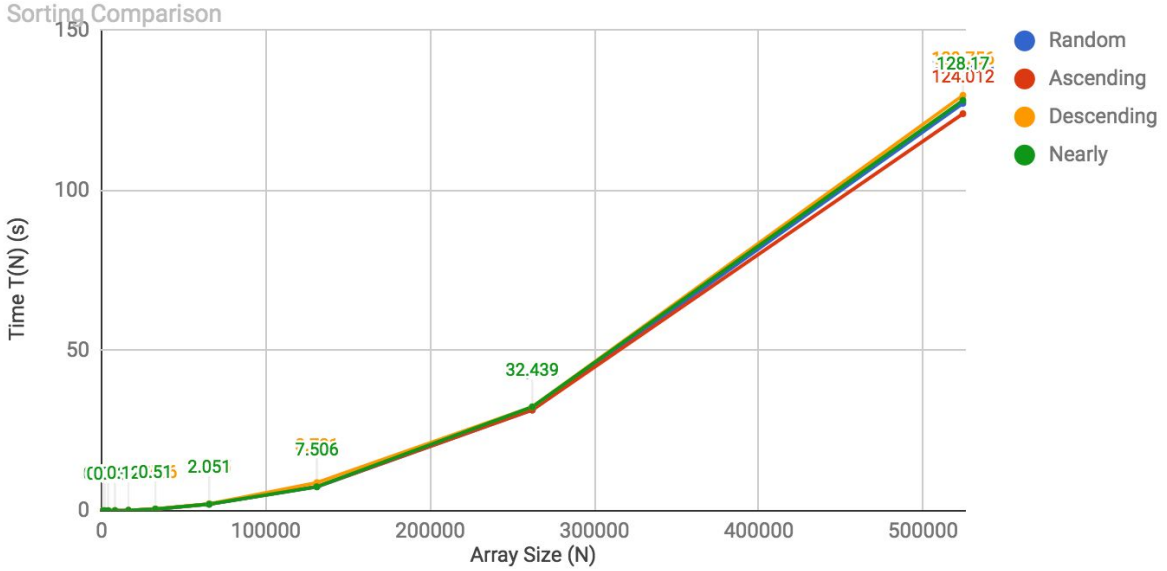
Selection Sort

Nearly Sorted Array (10 Trials)



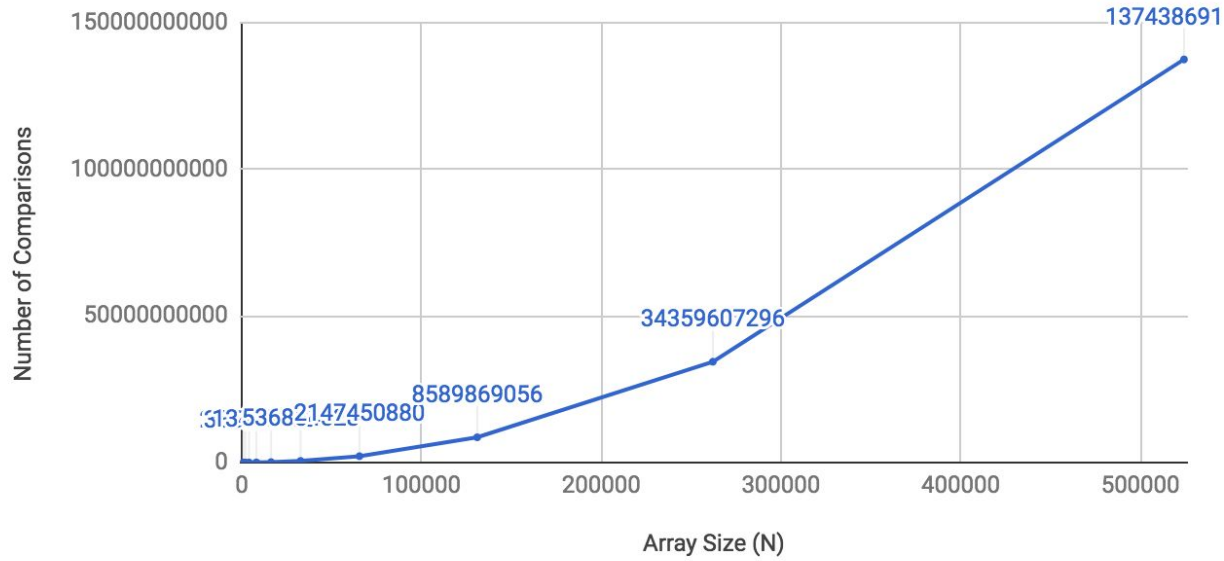
Selection Sort

Sorting Comparison



Selection Sort

Comparisons



Insertion Sort

The insertion sort algorithm has a best case of O(n), an average case(on a random array) of $O(n^2)$, an "almost sorted" case in O(n), and a worst case of $O(n^2)$. Variable N represents the number of elements in the array.

Insertion sort is an in-place, stable sorting algorithm that constructs the final sorted array one item at a time. While it is an efficient way to sort small lists, insertion sort becomes much less effective when dealing with large sets of numbers. As this algorithm iterates through the input array, an element is removed one at a time. Insertion sort finds the location that element belongs in within the sorted section of the list, and inserts it at that spot. This algorithm has been compared the way a card player sorts the cards in their hand. They scan the cards from left to right, looking for the first card that is not in the right spot.

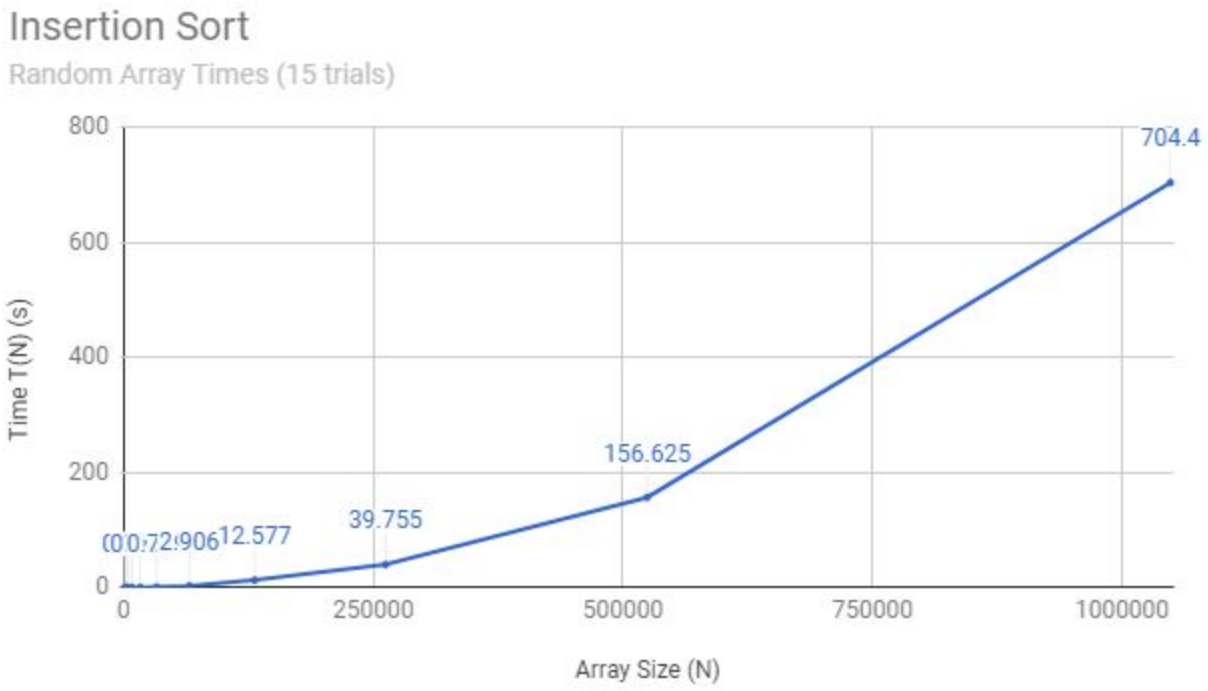
Testing Environment

For this project, our insertion sort was run on an HP Envy m6 Notebook. The operating system on this machine is Microsoft Windows 10 Home. It has an Intel Core i5 processor with a speed of 2.20 GHz. The hardware has 2 cores, 4 logical processors, 2 3MB L2 caches, and 6 GB of installed physical memory. We are also using Java version 8 to run these tests.

Insertion Sort Results

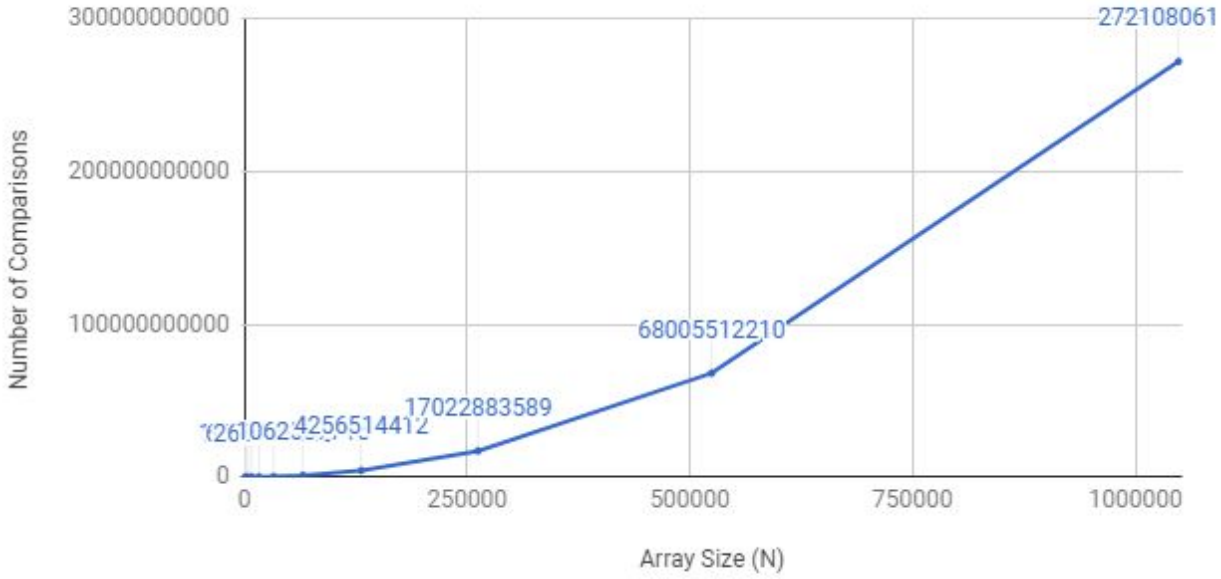
Overall, we weren't very surprised about the results of the insertion sort. It handled the small array sizes very well, and sorted them in a quite small amount of time. However, as the array sizes grew, the sort slowed down a considerable amount and began taking longer to sort the lists. When the array sizes reached sizes of over 1,000,000, the sort took almost double

(triple in some cases) the amount of time. This is what we expected to happen because as mentioned above, insertion sort is good for small data sets but isn't efficient when trying to sort large sets of data.



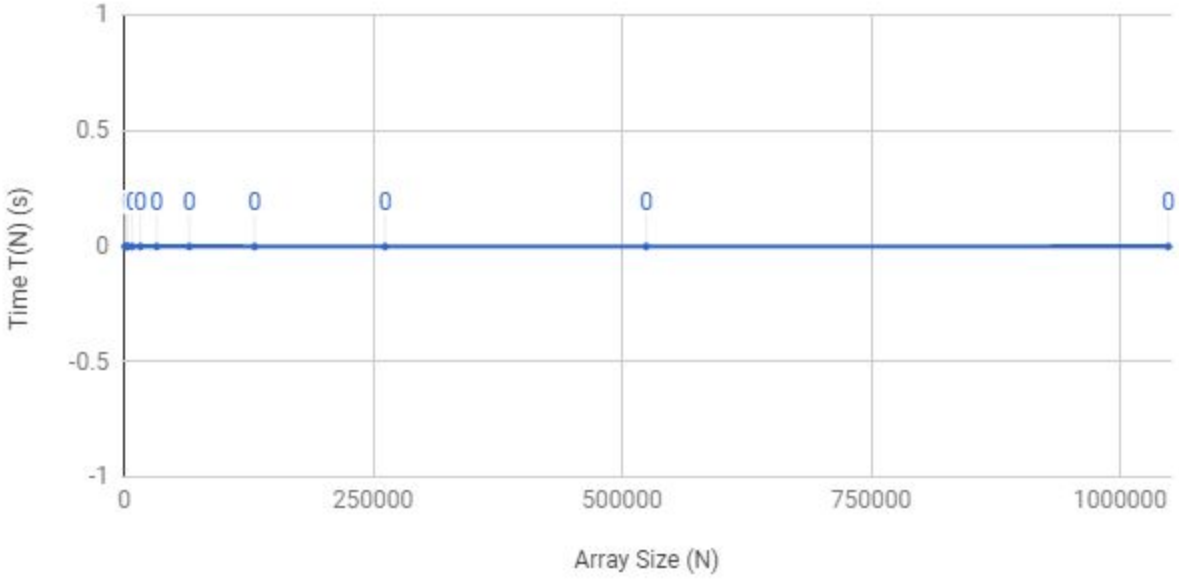
Insertion Sort

Random Array Comparisons (15 Trials)



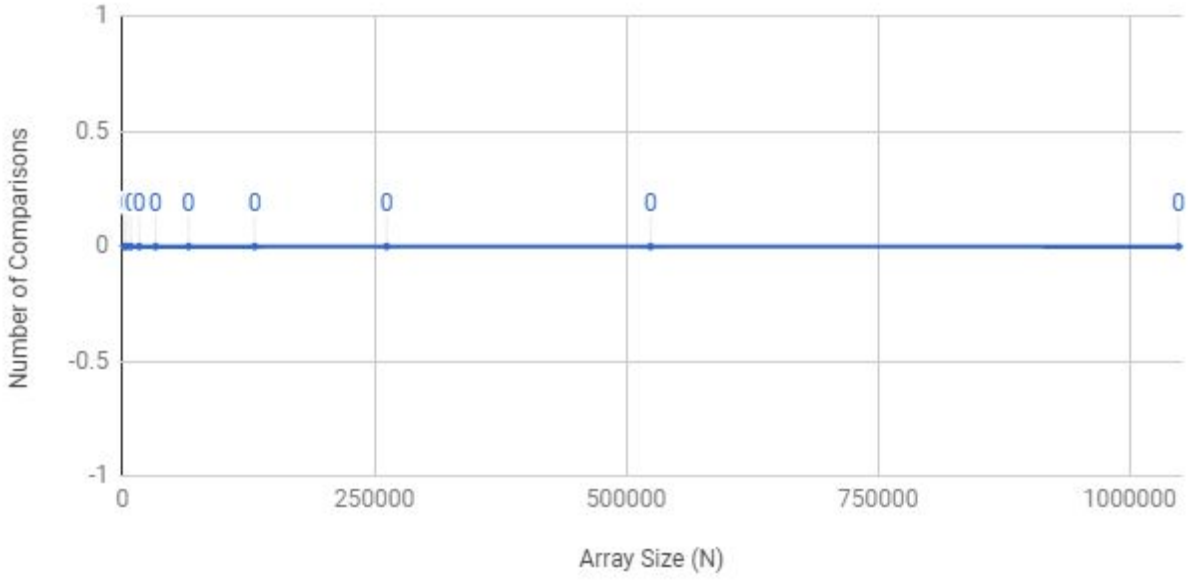
Insertion Sort

Ascending Array Times (15 trials)



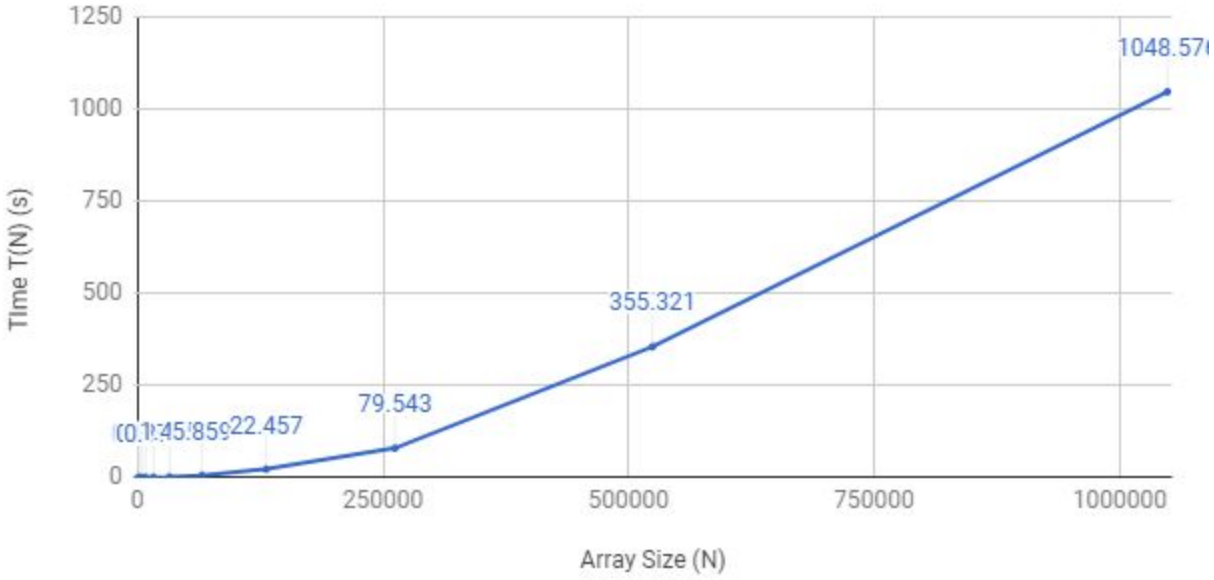
Insertion Sort

Ascending Array Comparisons (15 Trials)



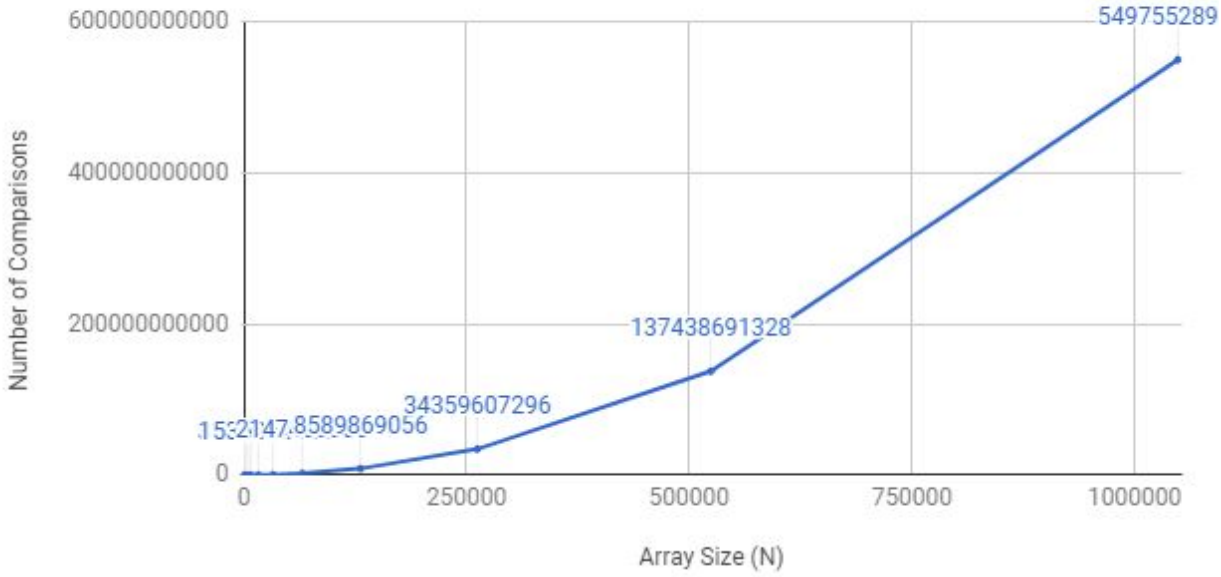
Insertion Sort

Descending Array Times (15 trials)



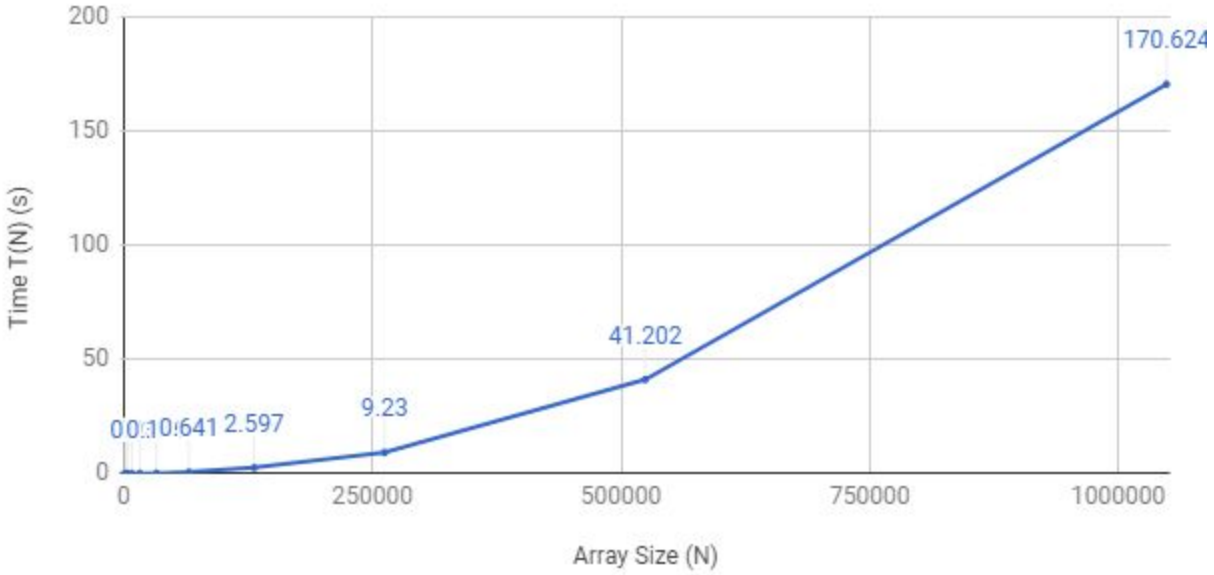
Insertion Sort

Descending Array Comparisons (15 Trials)



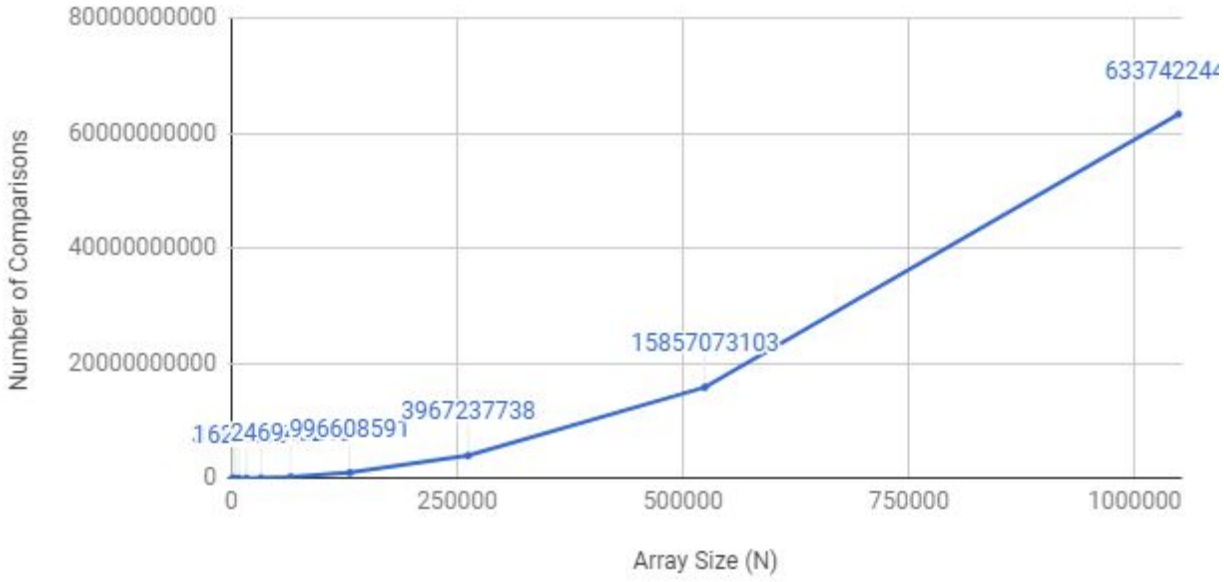
Insertion Sort

Nearly Sorted Array Times(15 trials)



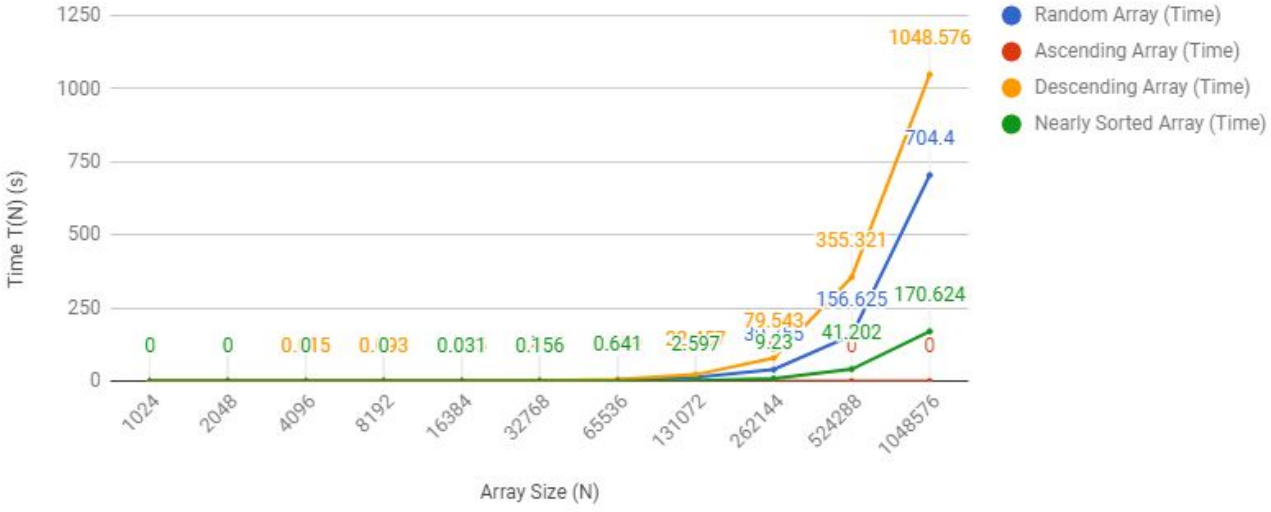
Insertion Sort

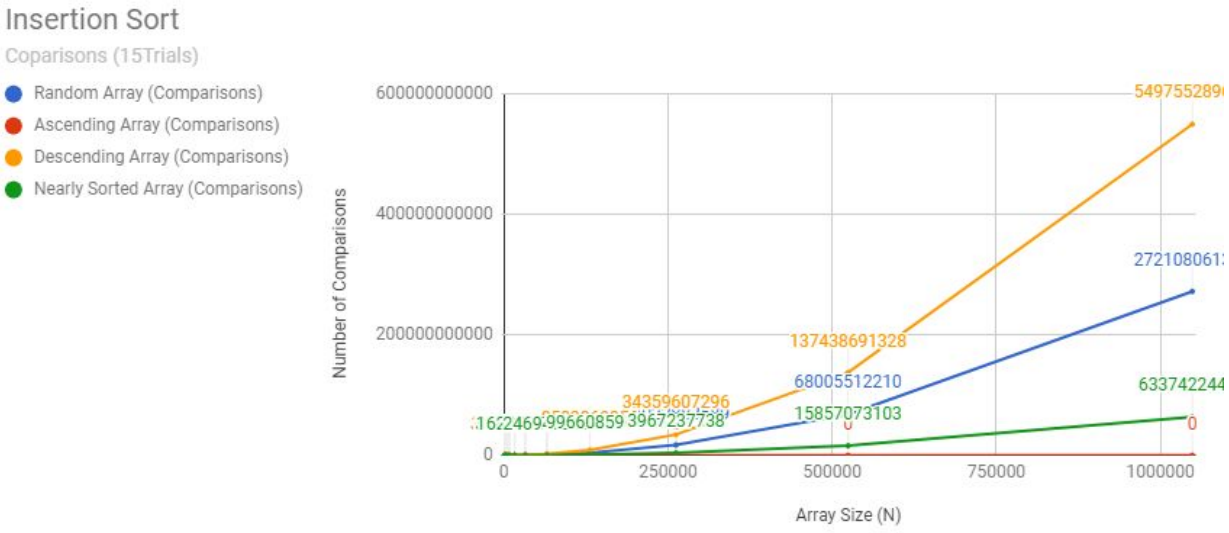
Nearly Sorted Array Comparisons (15 Trials)



Insertion Sort

Times (15 Trials)




Insertion Sort
Coparisons (15Trials)
Random Array (Comparisons)
Ascending Array (Comparisons)
Descending Array (Comparisons)
Nearly Sorted Array (Comparisons)
600000000000
400000000000
200000000000
0
Number of Comparisons
549755289
272108061
137438691328
68005512210
34359607296
15857073103
633742244
0
0
250000
500000
750000
1000000
Array Size (N)

Bubble Sort

Bubble Sort is a simple sorting algorithm that compares each item with its adjacent item. Bubble Sort in the worst case is O(n^2) as it must continue looping through an array until all items are in the correct order. The bubble sort we use also has an average case and best case of O(n^2) as it must check every value until it has gone through the entire array using a nested loop.

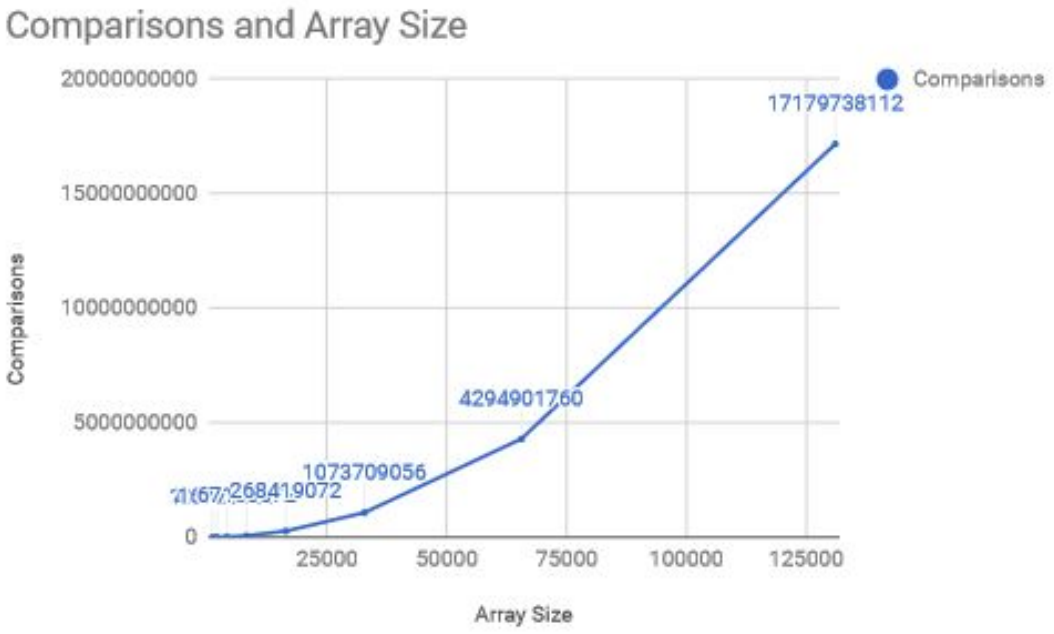
There are implementations of Bubble Sort where if the algorithm has not made any swaps at the end of its current loop, then it will stop to make the best case O(n) but that is not the case for this version of Bubble Sort.

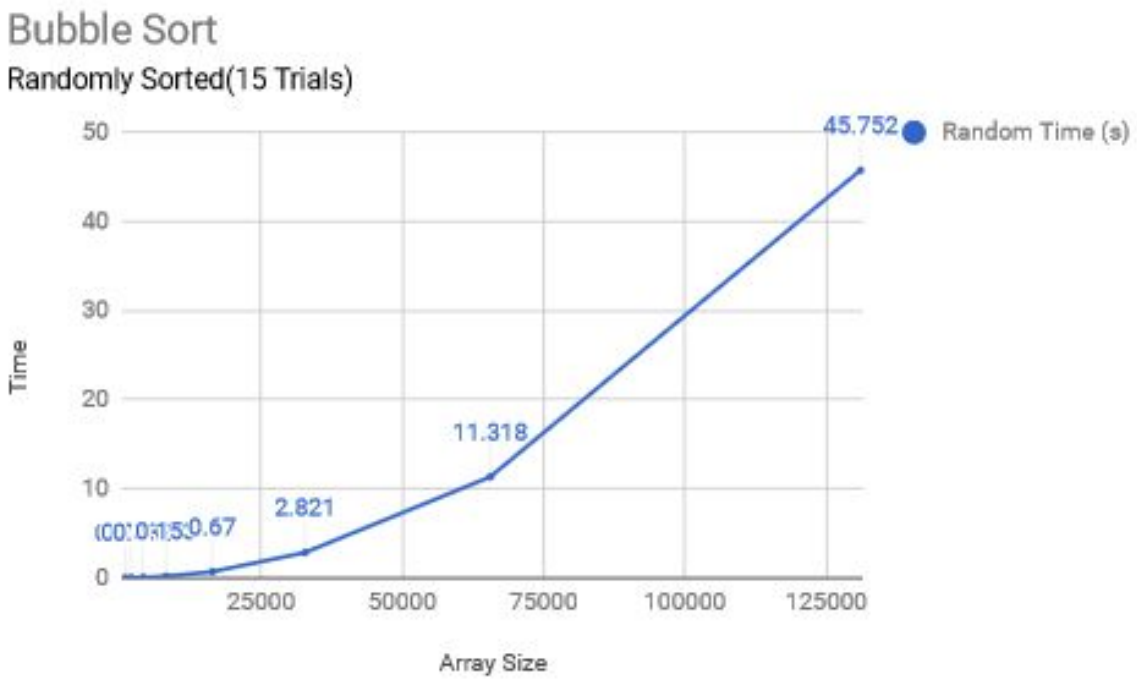
Testing environment

For these trials a Razer Blade was used which has 16 GB of RAM and 15.9 GB of usable RAM. It uses an Intel i7-700HQ processor with a clock speed of 2.80 GHz and a 6 MB cache. The version of Java being ran is Java 8 for these tests.

Bubble Sort Results

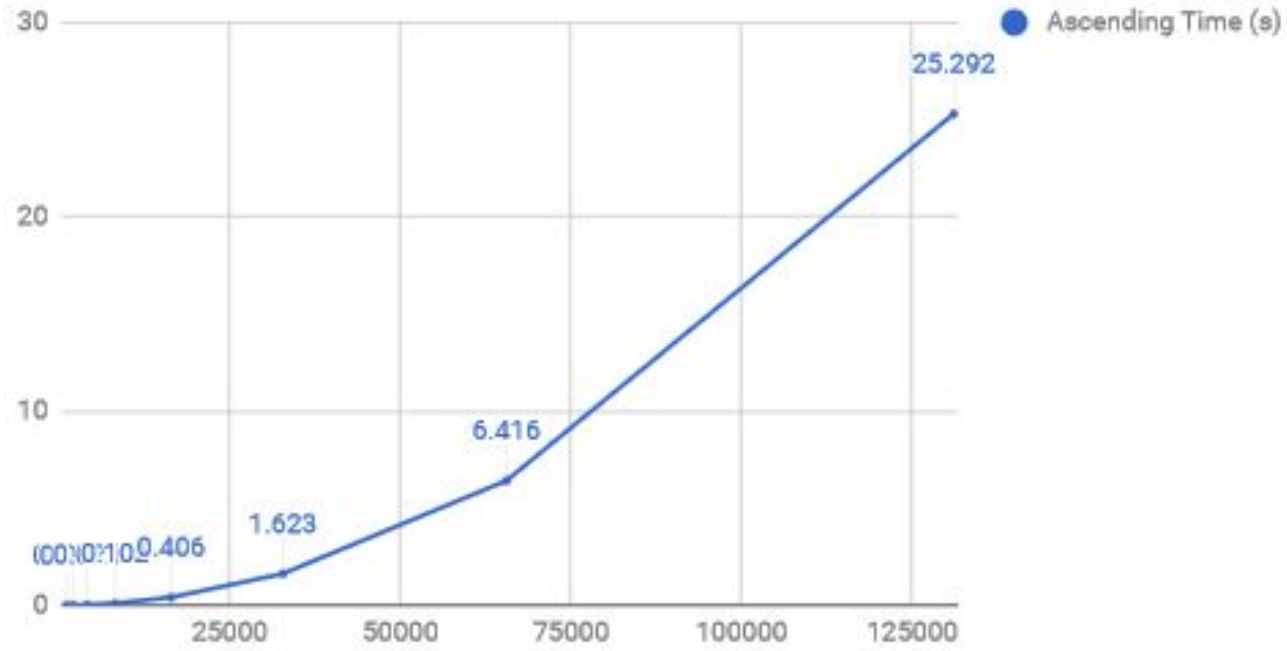
The results of Bubble Sort were slightly surprising just because of how quickly it grew when the array size increased. The highest array size that we were able to use was 131072. When that was doubled and then doubled again, the time to run the tests was over 14 hours. Overall the times are all very consistent even at larger array sizes.




Bubble Sort
Randomly Sorted(15 Trials)
50
40
30
20
10
0
Time
45.752
Random Time (s)
11.318
2.821
0.67
25000
50000
75000
100000
125000
Array Size

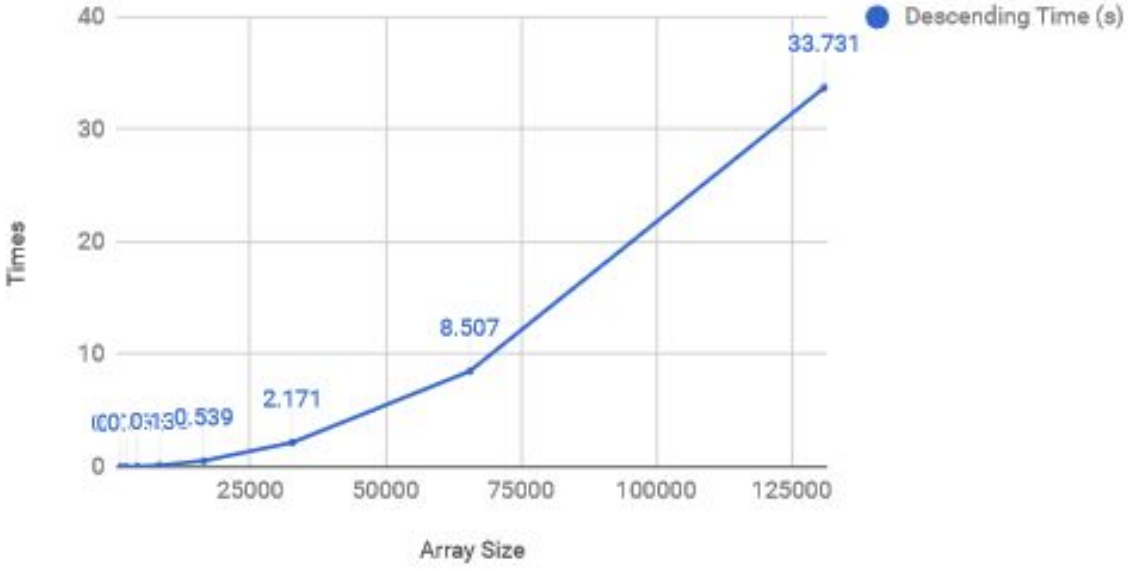
Bubble Sort

Ascending Array(15 Trials)



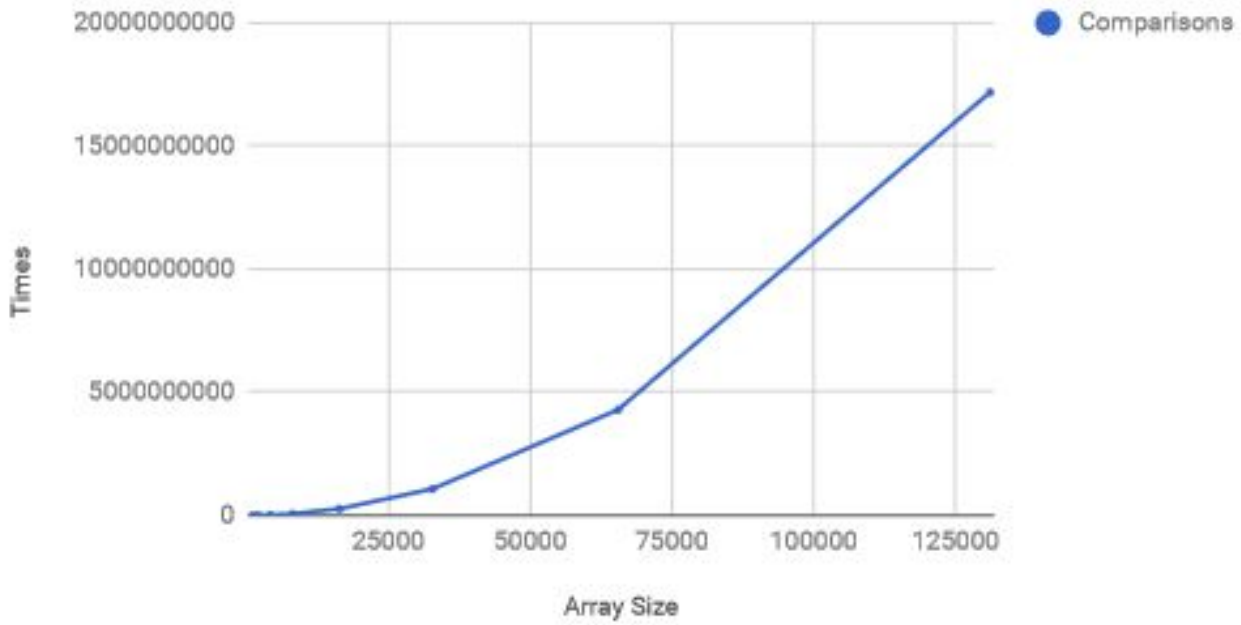
Bubble Sort

Descending Array(15 Trails)



Times and Array Size

Nearly Sorted(15 Trials)



Bubble Sort

For 15 Trials