Tim Pugh

Professor Chris Gilmore

CS 350

March 08 2017

Algorithms Final Project

For my algorithms project I chose to do the path of the searching algorithms. The searching algorithms encompassed binary search, sequential search, and a variant of binary search called interpolation search. Each algorithm I coded and tested takes a sorted array of length 1,000,000 integer values. In certain tests the array increases by a single value, another test has the array values increased by adding 7, and another test is having the array randomly populated but still sorted, that way the array increases in a randomly increasing manner which can be more realistic in a real world environment.

Binary Search is a decrease and conquer search algorithm, and more specifically is a decrease by a constant factor algorithm which cuts the search in half during each check of the sought after value. The way binary search works very quick for a sorted array although does not work on linked lists. Binary search begins by finding the midpoint of an array of size n. It does this by the following math formula:

midpoint = (lowestIndexValue + highestIndex) / 2

After finding the midpoint it will compare the value we're trying to find in the array to the midpoint (we'll say the value were looking for is called "theSearchedForValue", and the

array is called “intArray”). If it’s a match, we return the index where the match was found, else if it’s not a match, it will check if theSearchedForValue < intArray[midpoint]). If it is, we set the lowestIndexValue value to midpoint - 1 and cut the size of the array left to check in half, and conduct the same process to cut the problem in half until it finds theSearchedForValue. If theSearchedForValue > intArray[midpoint] it will conduct the same process but differs in that we set highestIndex = midpoint + 1 which still cuts the problem in half.

Sequential search is a method that feels very like how a person would conduct a search. You are given a sorted array and a value you’re looking for (this value may or may not exist in the array). You start at the beginning of the array, at index 0, and see if the index holds a match for the value you’re trying to find. If not, we proceed to the next index and compare again. We keep going in this fashion until we find our value in the array and return the index where the value was found, or until we’ve gone through the entire array and exhausted all possibility of finding the value which results in the value not being found.

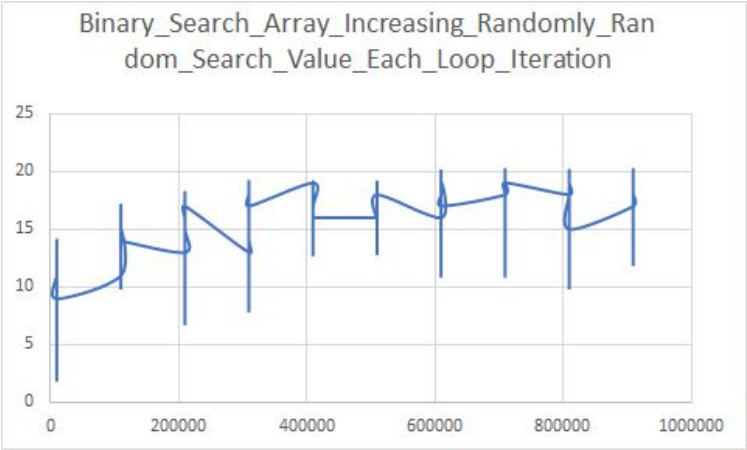
The last algorithm implemented is a variant of binary search called interpolation search. In this algorithm, it’s also necessary to have a sorted array and doesn’t work on linked lists. The way interpolation search works is similar to binary search where it reduces the problem, and is a decrease and conquer algorithm, but instead of cutting the problem in half each time it varies in how much it reduces the problem. Interpolation

search attempts to probe the array for a spot it believes the value we're searching for will exist based on the value. A quick example of interpolation search is using a phone book to look for someone with the last name "Franklin". We know that F comes before M, so it doesn't make sense to split the phonebook right down the middle, but split it somewhere around ¼ of the beginning phonebook of the phone book (since F is the 6th letter in the alphabet and comes roughly a quarter of the way into the alphabet when reciting it). We make a guess on some prior knowledge, and this saves us a bit of time at the expense of some extra calculations. The formula for probing the array and checking for a match (or the point where to split) is:

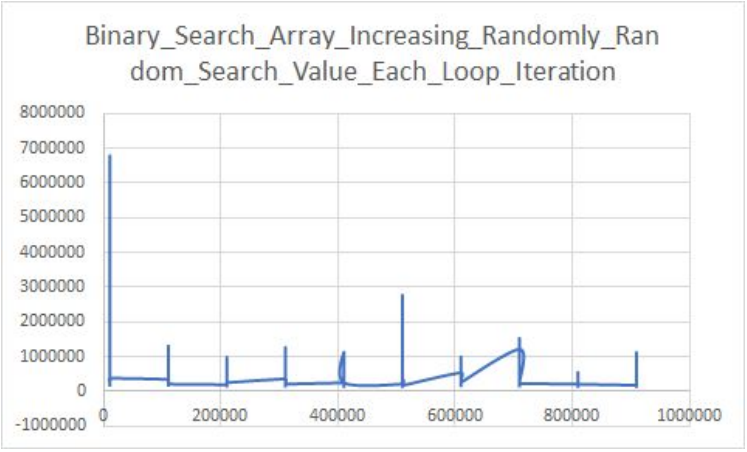
```
mid =lowIndex + (searchedForValue - intArray[lowIndex]) * ((highIndex - lowIndex) / (intArray[highIndex] - intArray[lowIndex])
```

Our testing strategy took into tests that a user may need to analyze in the real world: having an array increasing by 1, an array increasing by a prime number 7 which may add some extra operations guessing midpoints, and an array where the values are increased randomly within a specific range. With those types of arrays, we also would conduct searches on each of the 3 different arrays one of two ways. We would search for the same random value 1000 times on the same array, and the second way was we would keep generating new random values through each of the 1000 loop iterations. We also chose to do our algorithms after conducting research online that indicated that there are performance gains by not choosing to due recursion since it begins to eat more memory. This gave us 3(array types)x3(algorithm types)x2 (methods of choosing a value) = 18 different testing environments, and we increase our array 10 separate

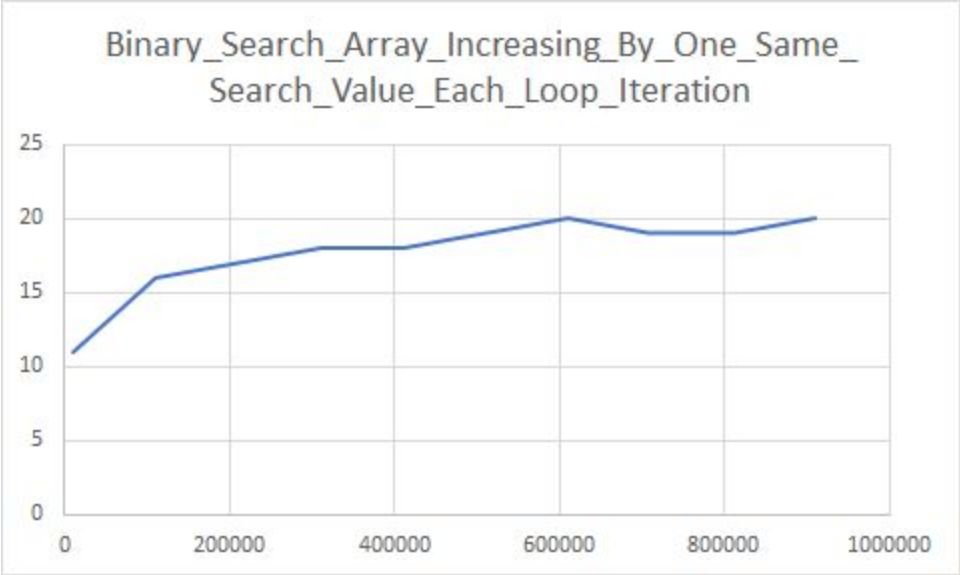
times from 10,000 to 1,000,000 which gives us 10 diffrent array sizes increasing by 10,000 each type. Below we will discuss our results from each graph.



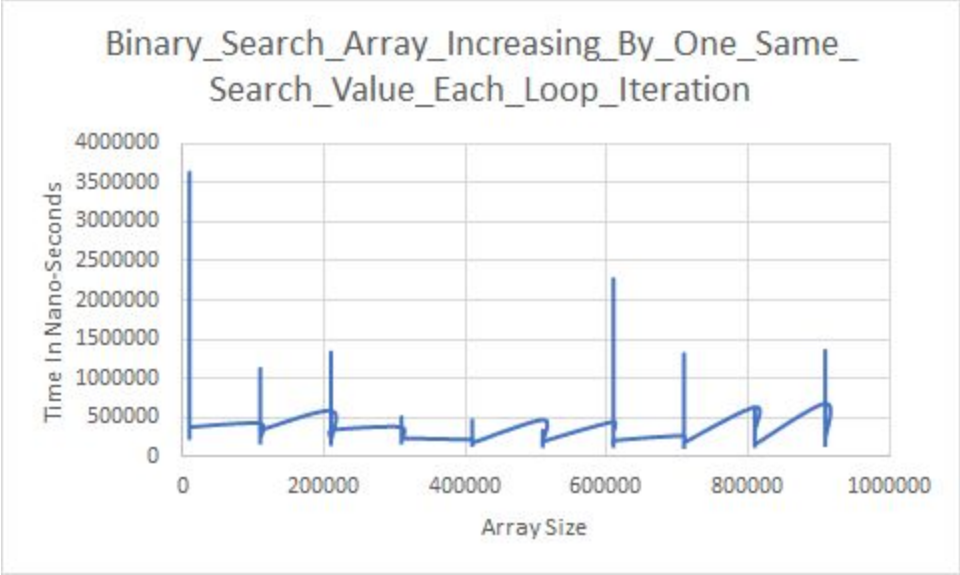
Via binary search where the array increased randomly and we searched for a new value each time, our results were hard to see the number of operations showing a O(log n) search time when looking at the graph. The Number of operations on the left side and the array on the bottom, it's hard to see that performance scales nicely via this testing strategy. Below is our timing in nano seconds (please view the attached excel for our data points!).



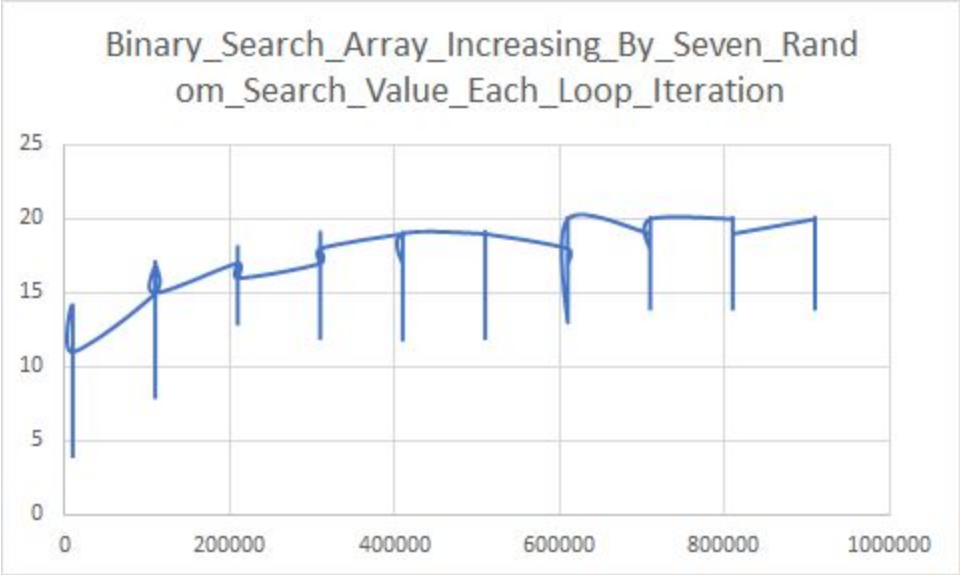
Timings were consistent but due to the quickness of our computer, most searches completed in nanoseconds. Our average time was 297 microseconds via this test method.



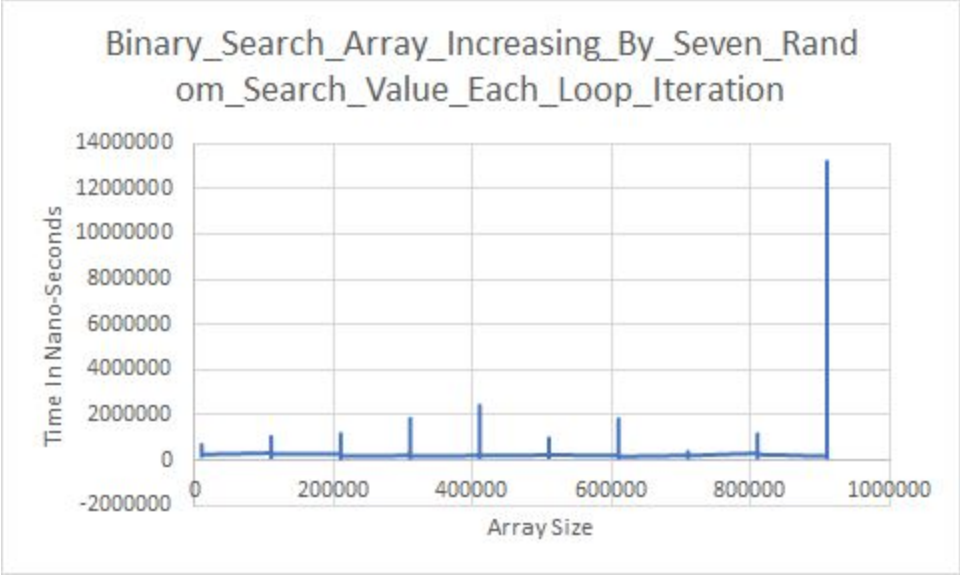
Here we are seeing better results and showing much more consistent figures. This method of testing (view graph label) proved to give us a much more logarithmic chart! Below is our timings.



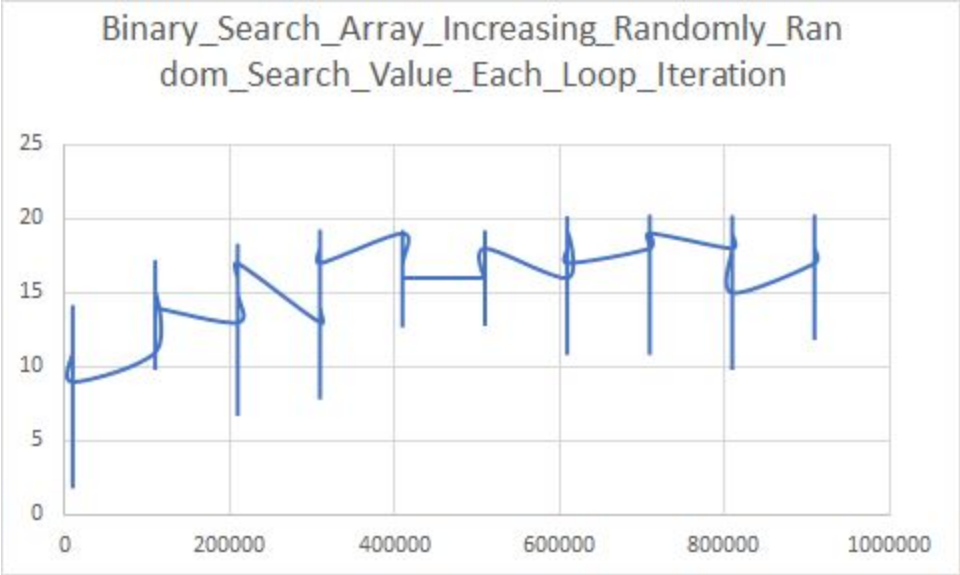
Average on this test scenario was 274 microseconds.

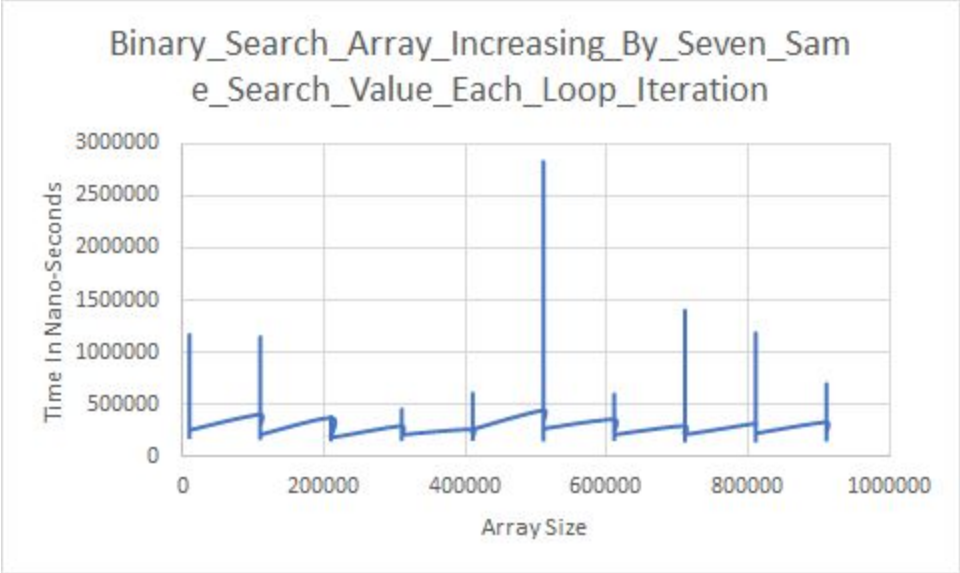


More logarithmic looking results via binary search as our array size increases! Timings below.

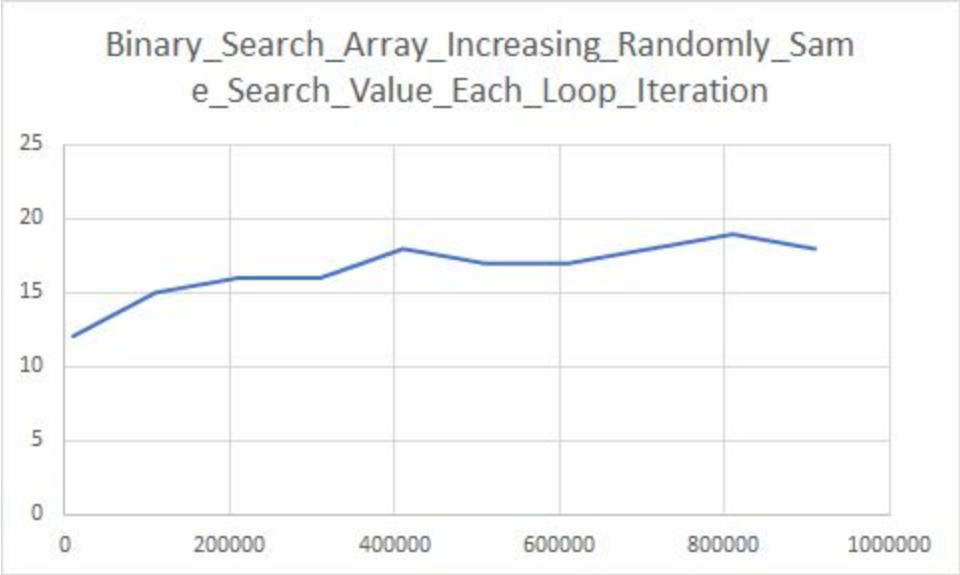


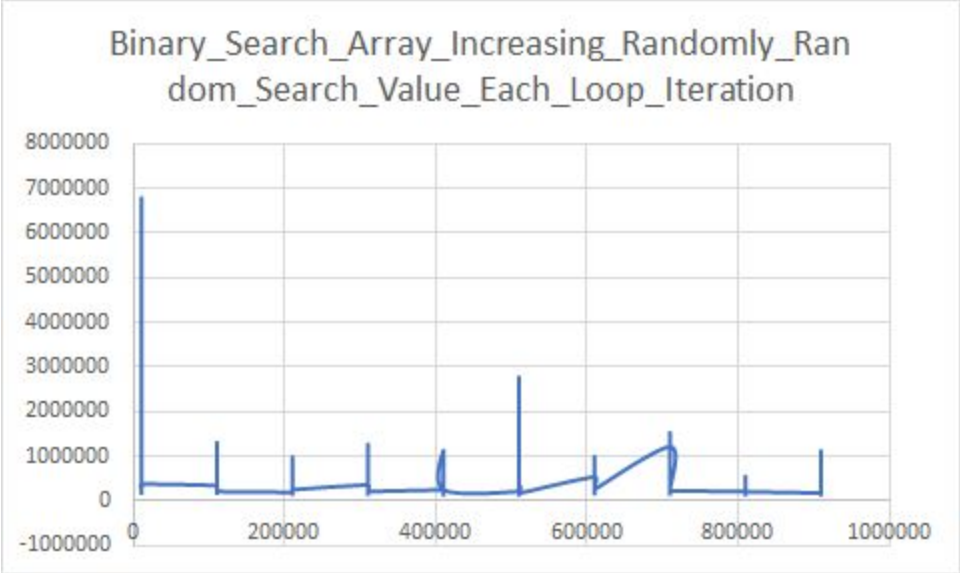
Average in this case 257.5 microseconds.





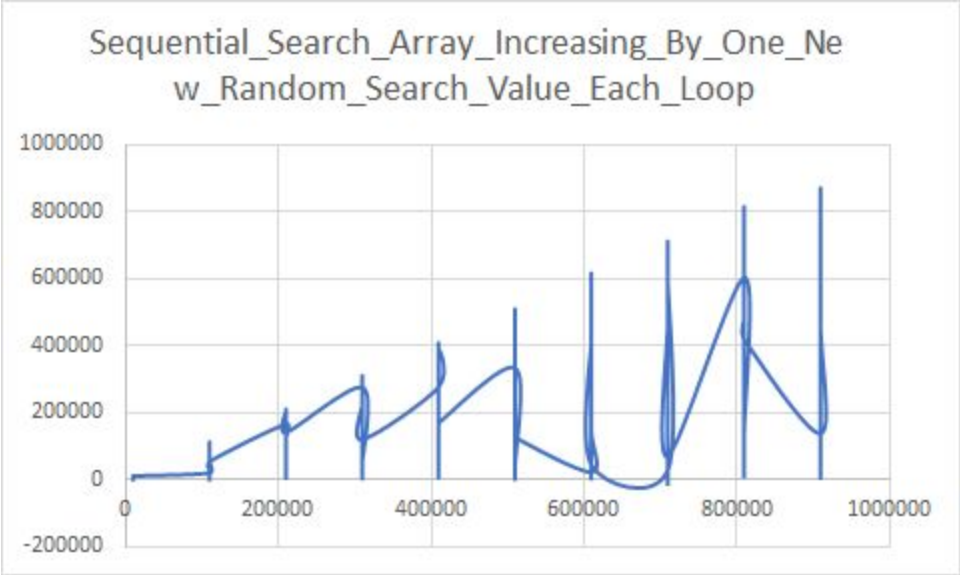
Average in this case was 265.8 microseconds.

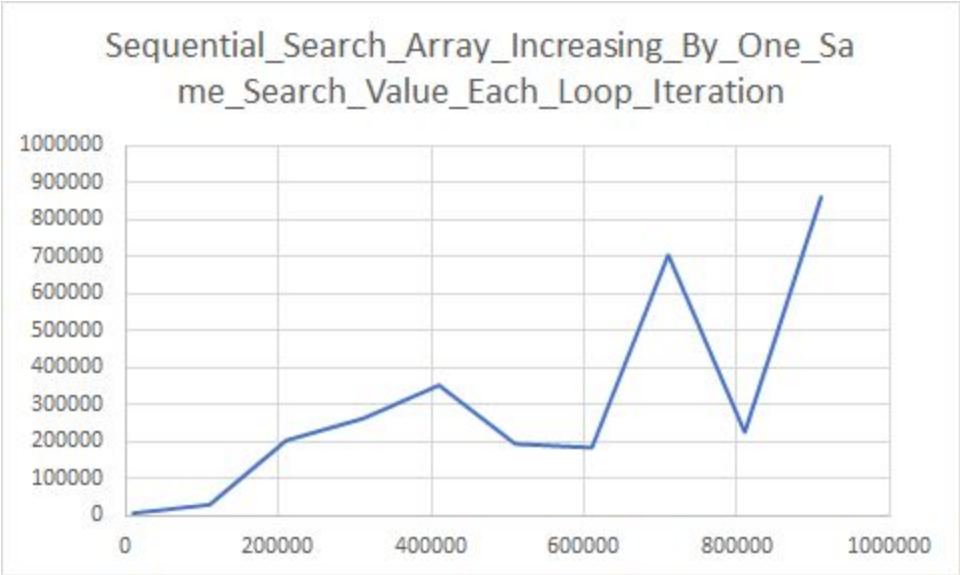


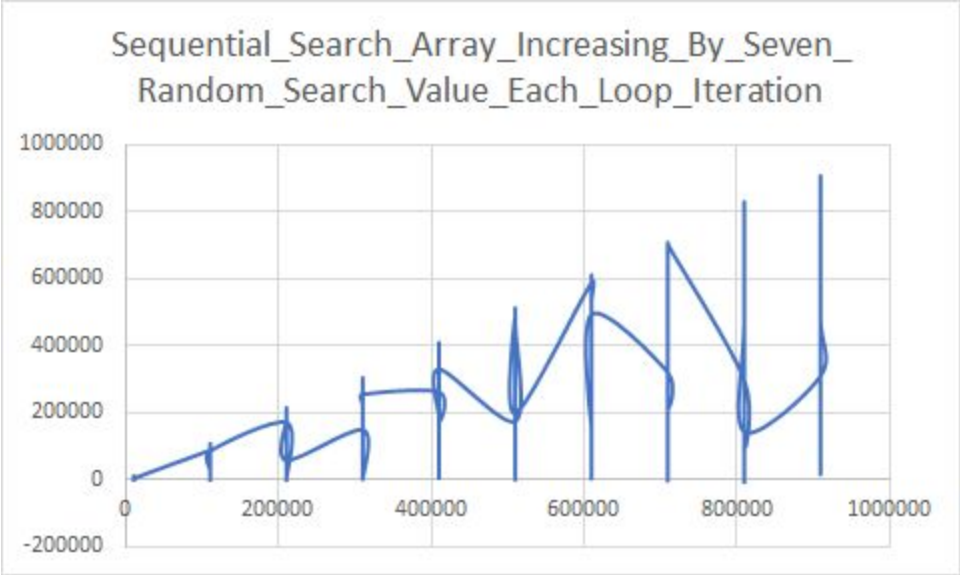


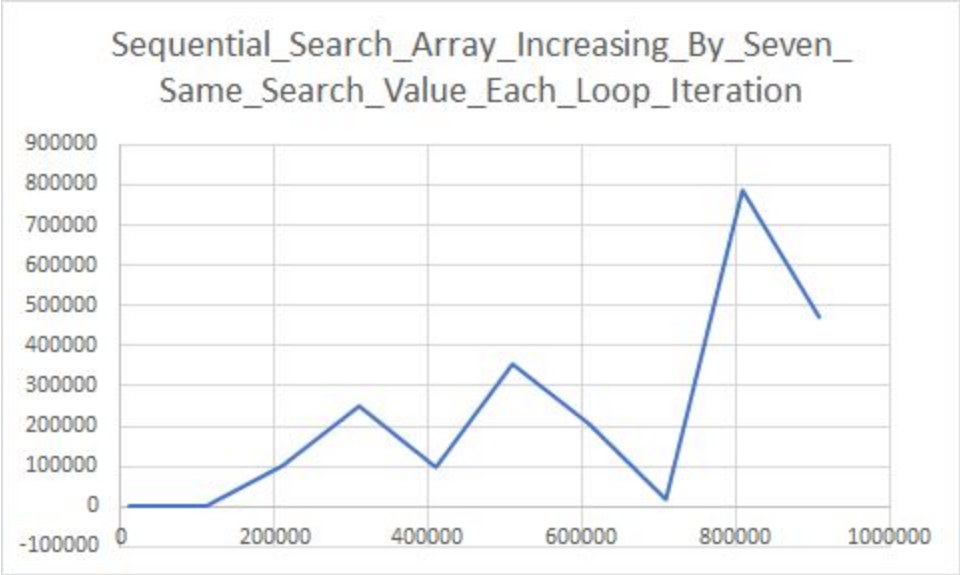
297.6 in microseconds was the average in this case. Taking a look at the final graph we can see the curvature of our line acting very logarithmic as the array size increases which is exactly what we were looking for and expecting.

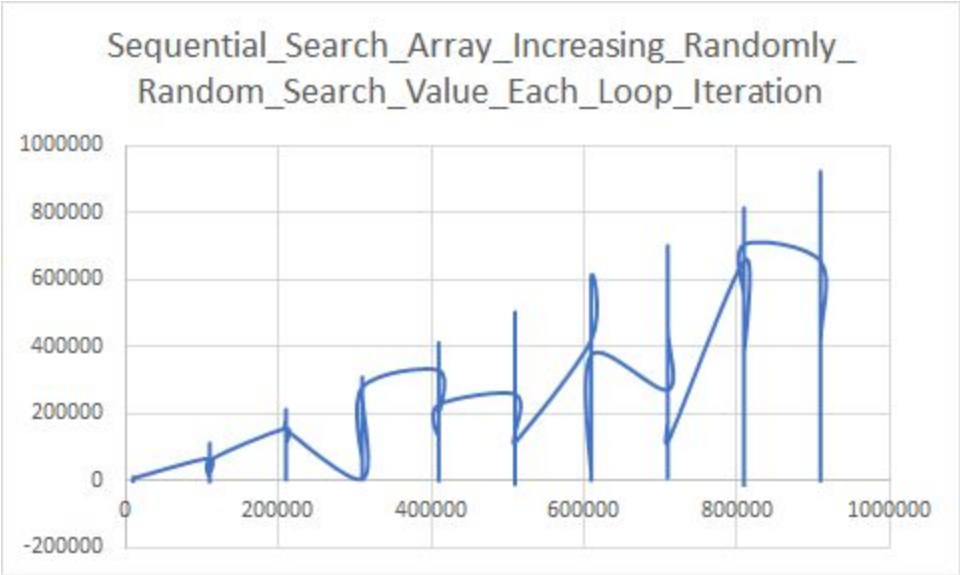
Sequential timings showed constant growth as the array size increased. Below (grouped together) is the graphs from each our test conditions which show the number of operations on the left, and array size on the bottom.


Sequential_Search_Array_Increasing_By_One_New_Random_Search_Value_Each_Loop
1000000
800000
600000
400000
200000
0
-200000
0
200000
400000
600000
800000
1000000


Sequential_Search_Array_Increasing_By_One_Same_Search_Value_Each_Loop_Iteration
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1000000

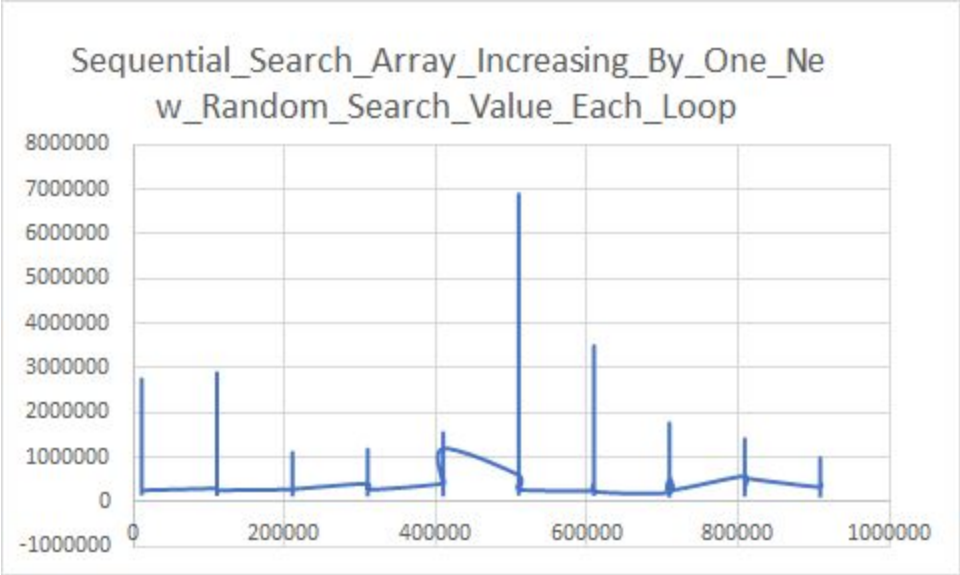

Sequential_Search_Array_Increasing_By_Seven_
Random_Search_Value_Each_Loop_Iteration
1000000
800000
600000
400000
200000
0
-200000
0
200000
400000
600000
800000
1000000

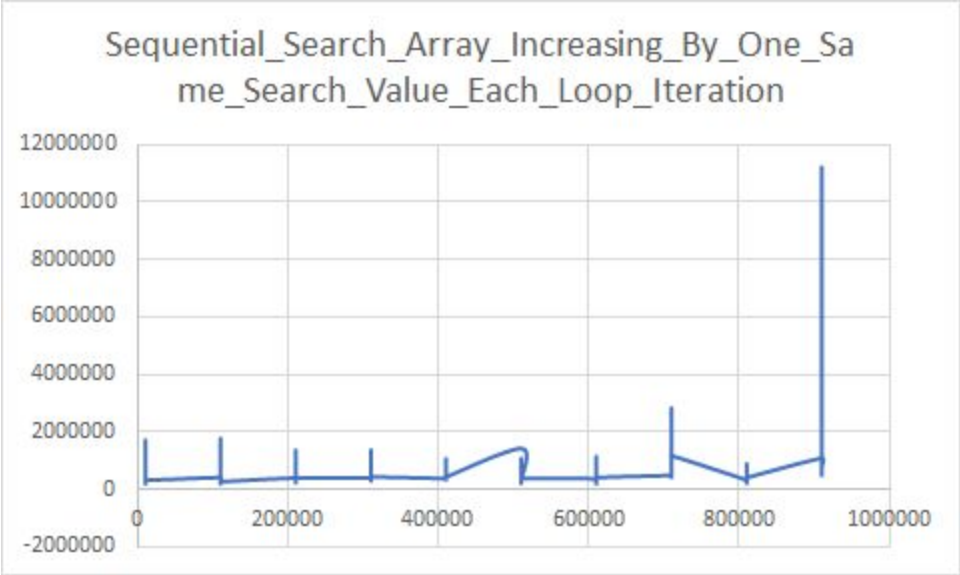


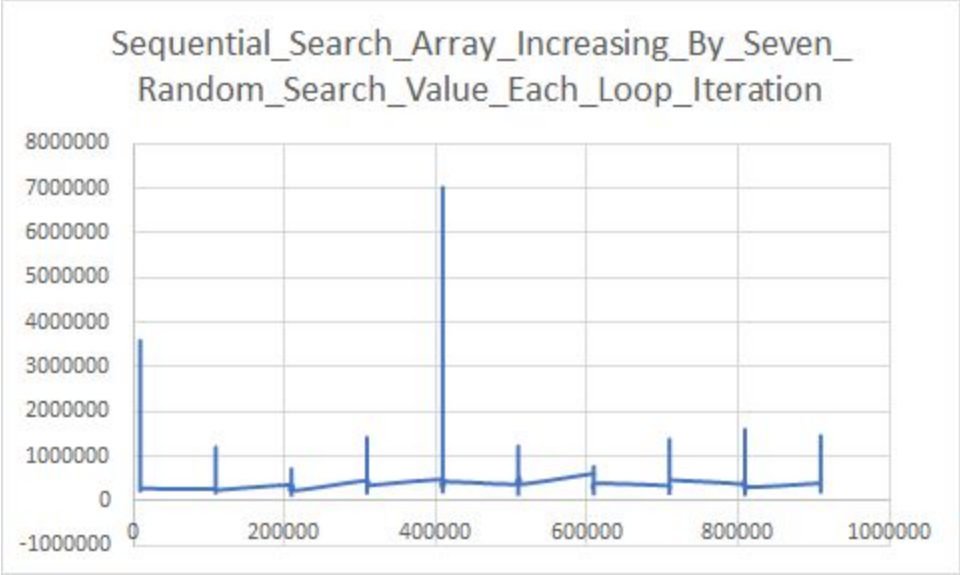


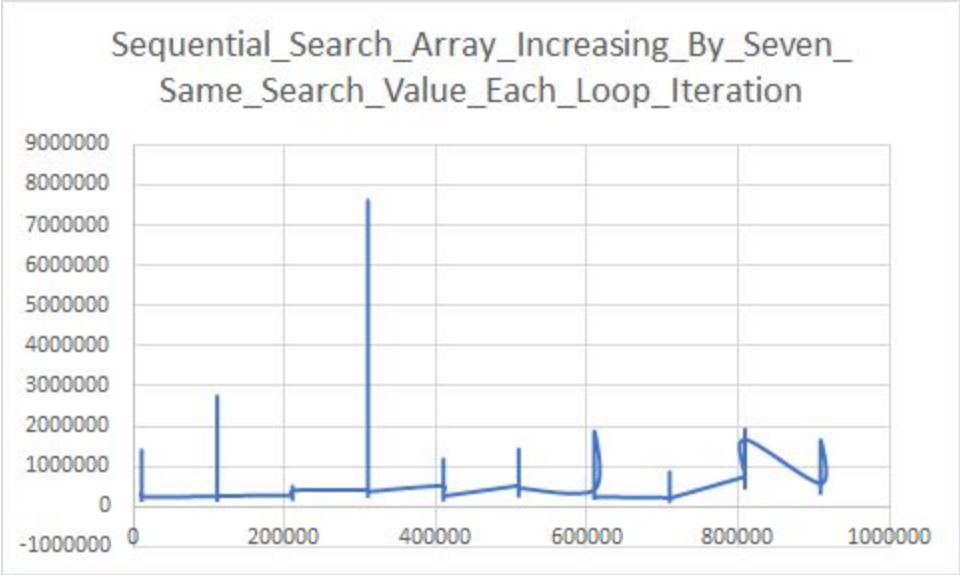
The graphs operation count as the array size increases is showing a constant growth size when compared to binary search curve. This tells us that sequential search is acting as we expected and having a search time in accordance to a O(n) time complexity. Below is grouped our timings for this algorithm in the same order of testing as posted above with the average times. The averages will indicate the long search

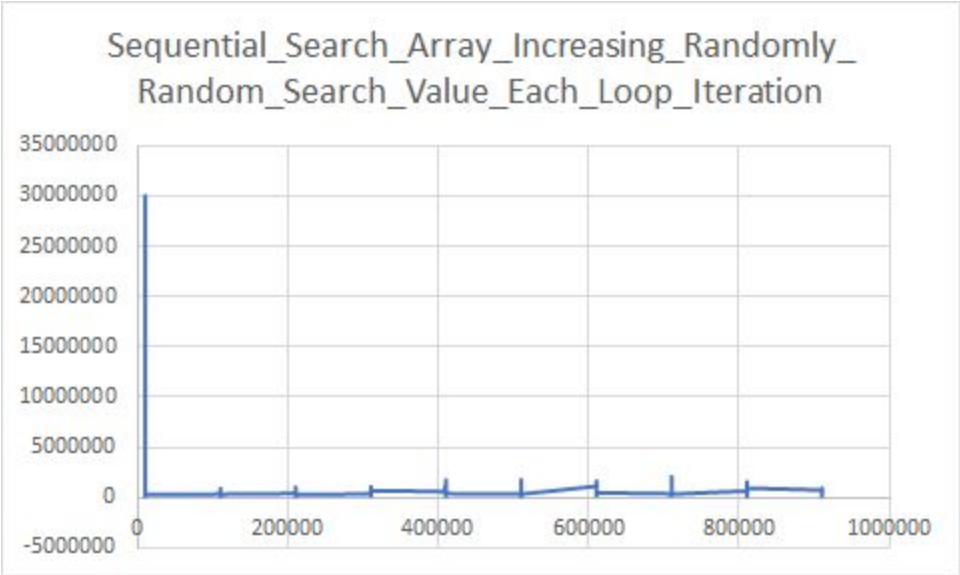
times sequential had to find values we sought.




Sequential_Search_Array_Increasing_By_One_Same_Search_Value_Each_Loop_Iteration
12000000
10000000
8000000
6000000
4000000
2000000
0
-2000000
0
200000
400000
600000
800000
1000000


Sequential_Search_Array_Increasing_By_Seven_Random_Search_Value_Each_Loop_Iteration
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
-1000000
0
200000
400000
600000
800000
1000000

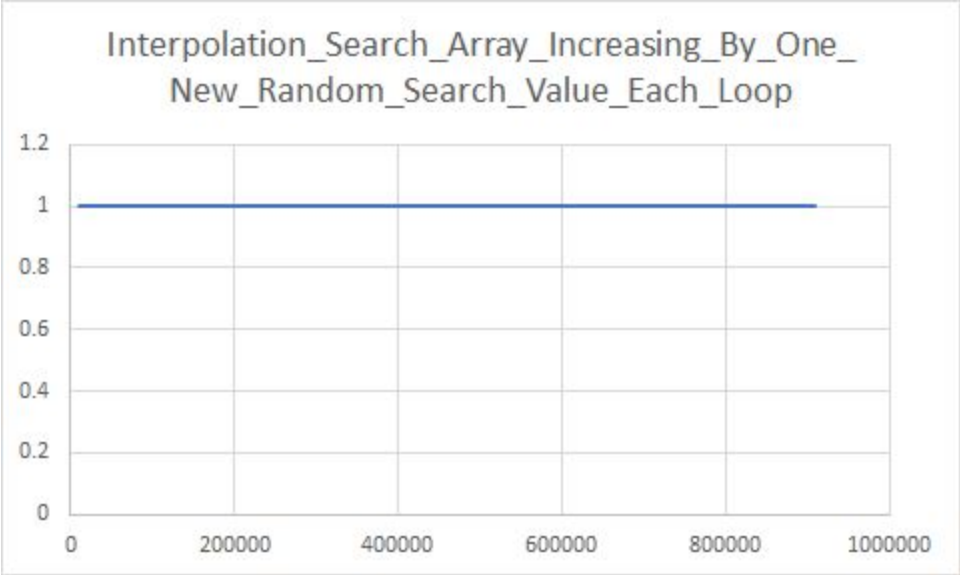


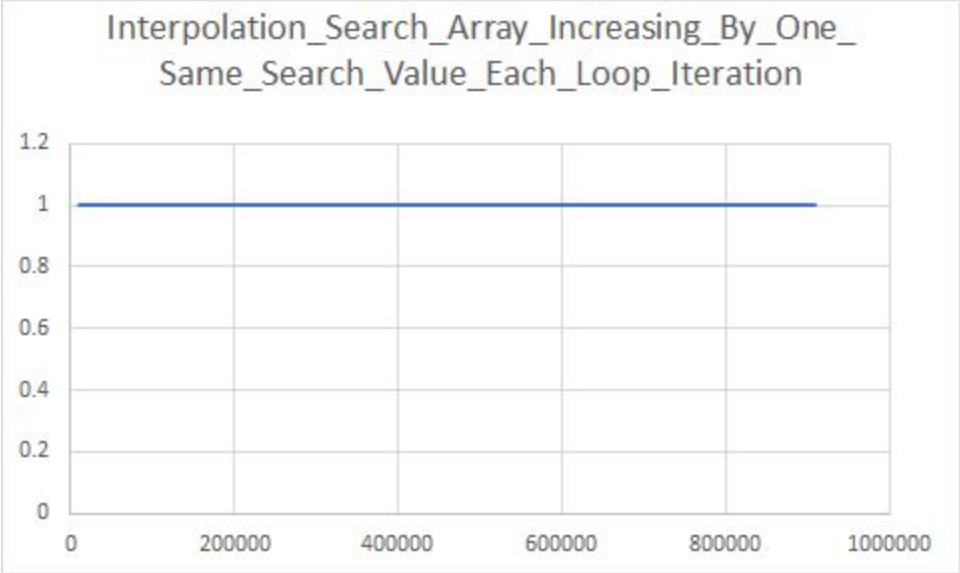


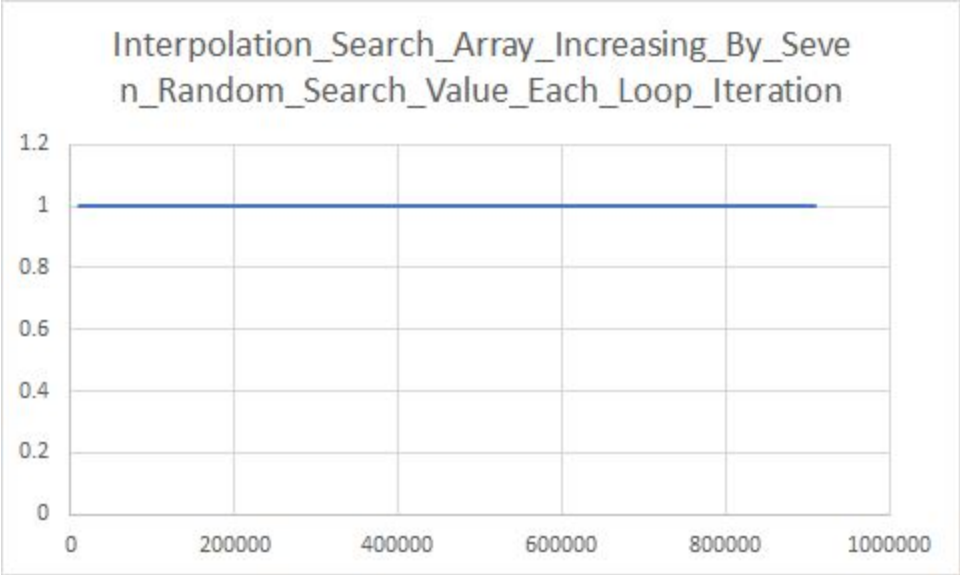
Since sequential is consistently slow and varies in scenarios, the average time is a better indicator when you group the average of each of the 6 cases above. This gave us a average across the six in microseconds (448.4+496.9+402.7+435.6+469.0 + 470.4)/6 = 453.83 microseconds.

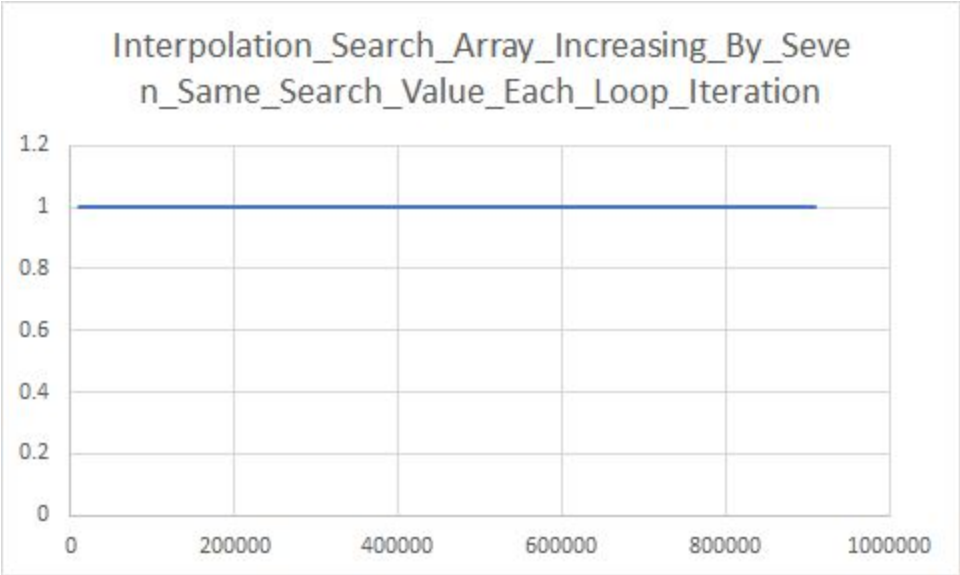
The clear winner of our search algorithms is interpolation search. In many cases it was able to find our searched values in log(log(n)) in the best case scenario. Our array

sizes were likely much too small, and scaling the array size would tax our memory way too much to create a array to properly test it!

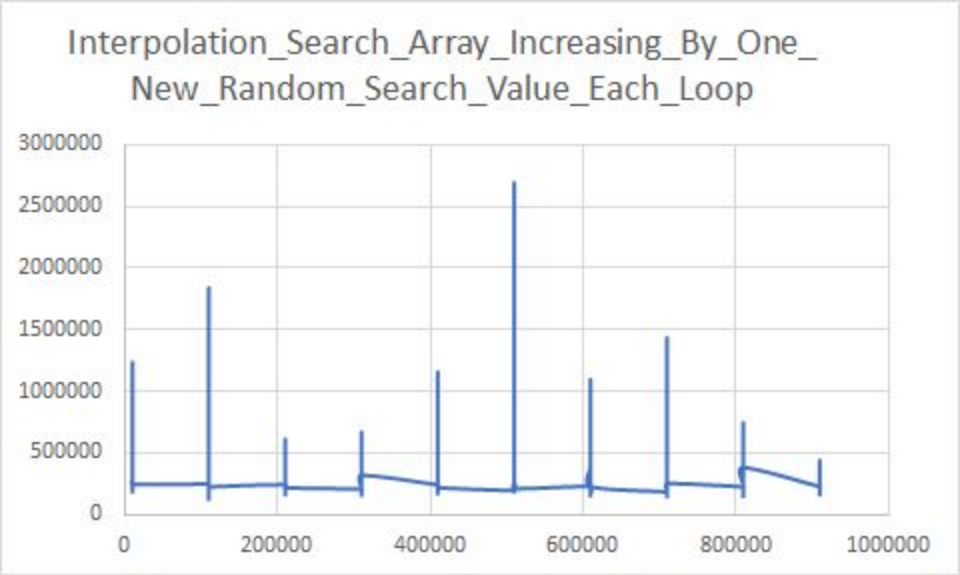


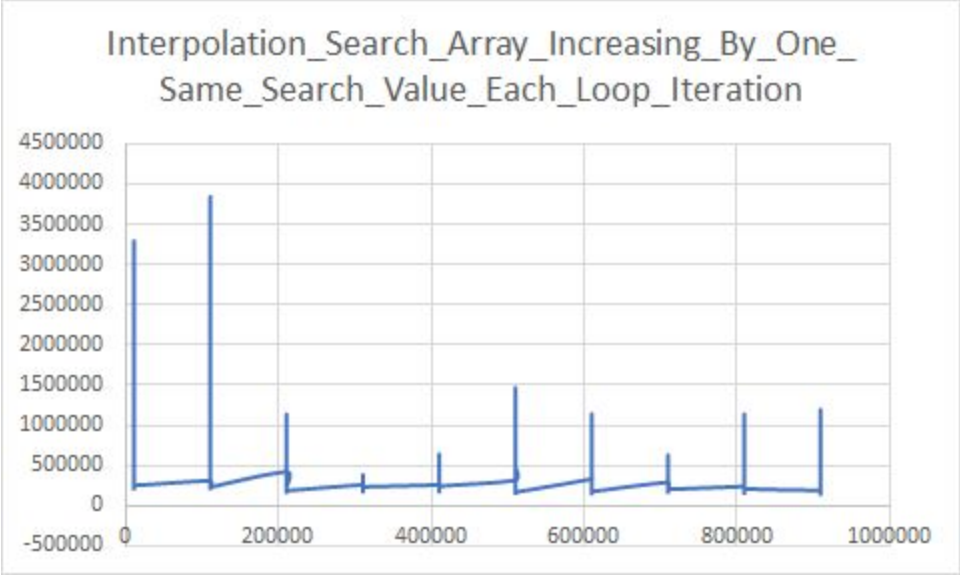


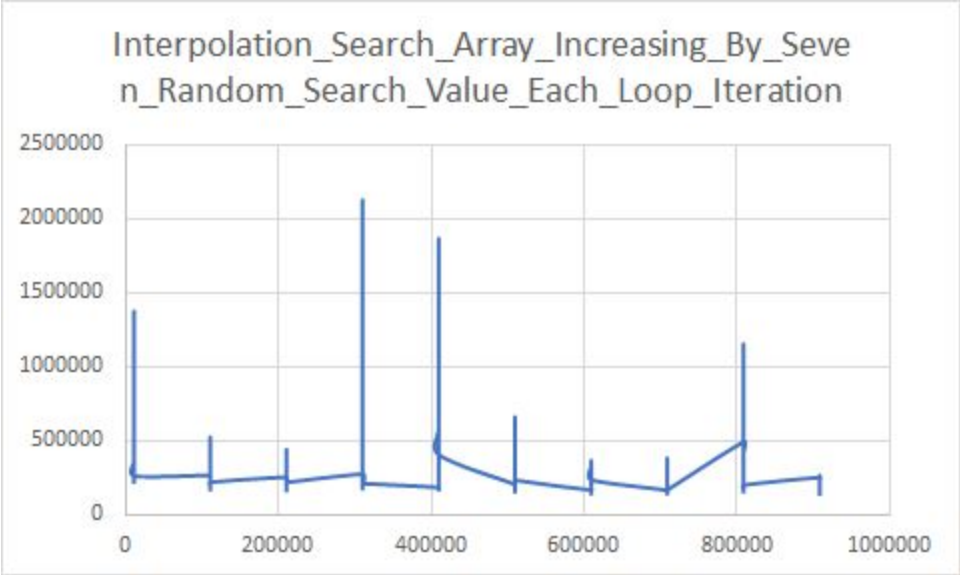


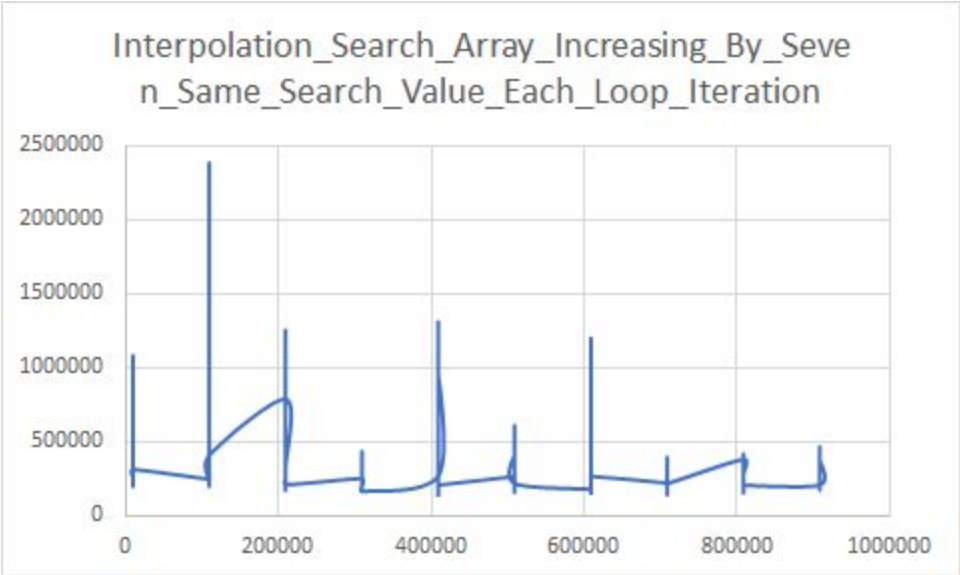


What is interesting for these results are the time the algorithm took to run in between these 4 results.

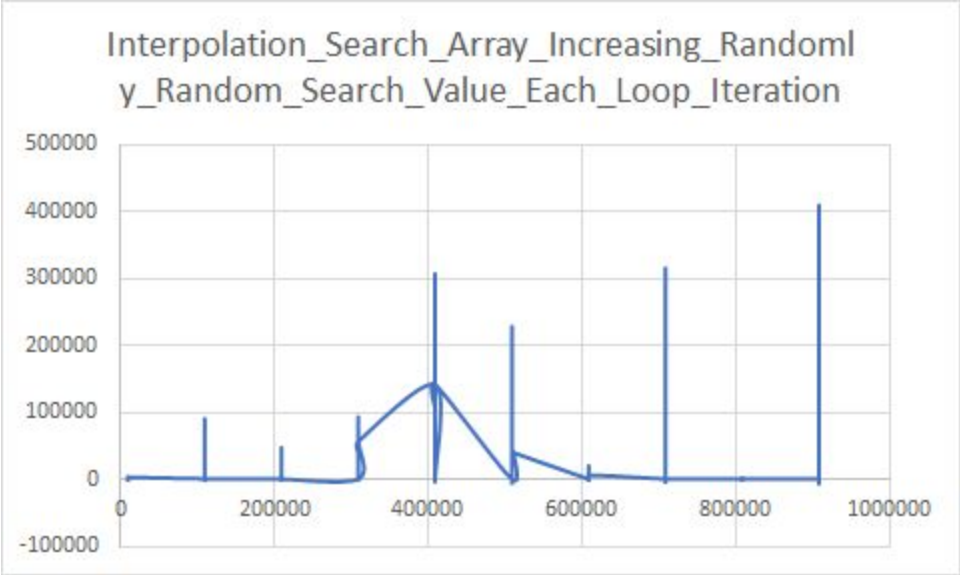

Interpolation_Search_Array_Increasing_By_One_
New_Random_Search_Value_Each_Loop
3000000
2500000
2000000
1500000
1000000
500000
0
0
200000
400000
600000
800000
1000000

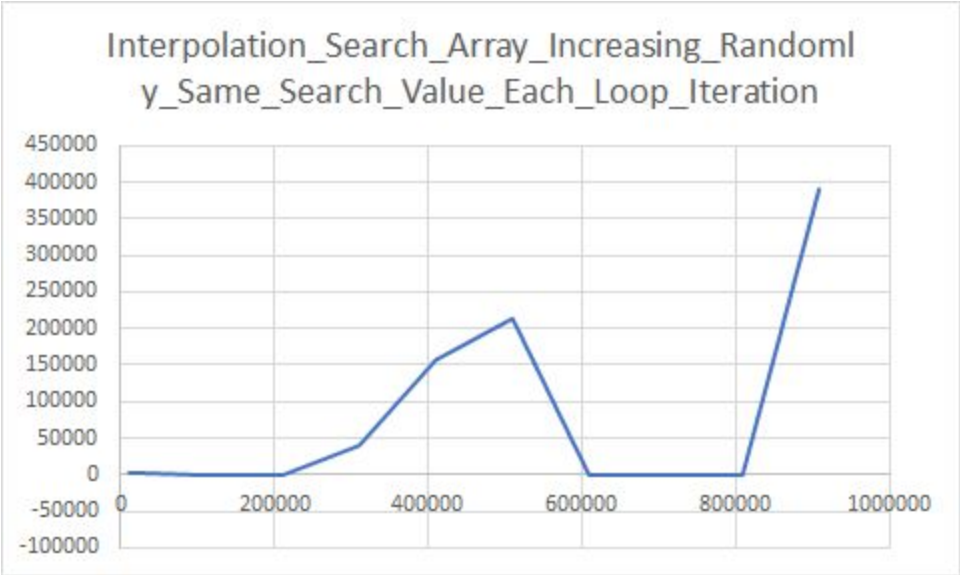

Interpolation_Search_Array_Increasing_By_One_
Same_Search_Value_Each_Loop_Iteration
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
-500000
0
200000
400000
600000
800000
1000000



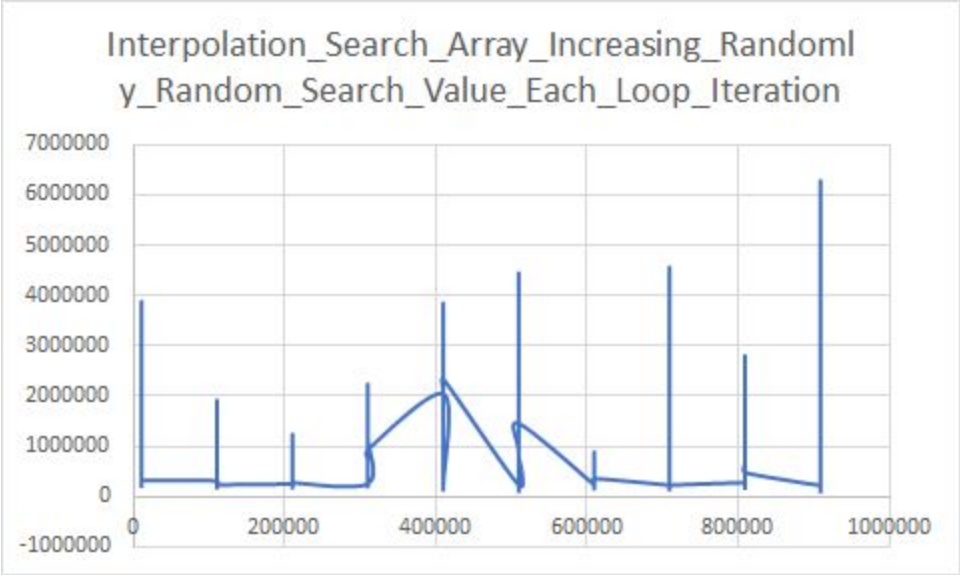


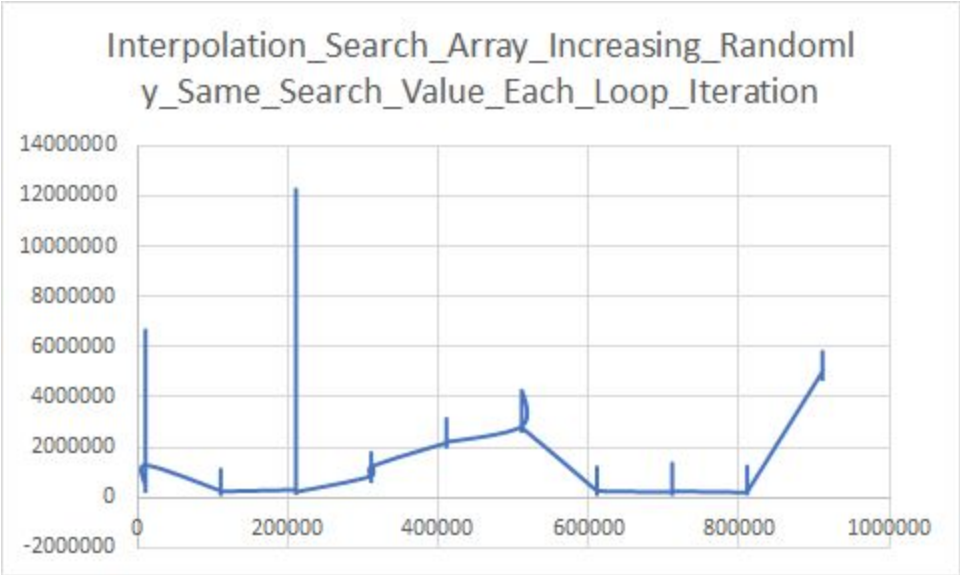
In microseconds the average time across these four was (264.1+262.2+ 255.8+256.7)/4 = 259.7 microseconds! Excellent performance as our array sizes increase. This algorithm scales very well in our test scenarios, but the last two results were quite different.





Here we were able to find a weak spot in interpolation search, even if it did perform better than sequential and binary search still. What this tells us is that with random integer values in a array, and thus a array that isn't distributed uniformly, this array can have issues with performance. Let's look at the timings for this case below.





The average in microseconds between these two graphs is (883.34+1277.98)/2 = 1080.66 microseconds! Even sequential was able to perform better. The reason being this likely performed could be specifically randomly generated array, and the amount of computation the algorithm had to perform to do it's probing. As the algorithm scales it's likely to outperform sequential in the long run, but binary search cleary is more consistent and in real world tests the random integer values is what is likely to be what

is seen. This tells me that binary is truly the most versatile algorithm amongst the three and performs consistently the best, although interpolation search is king when the array is distributed uniformly and beats all others by almost double on average.

I did run into several issues while doing my algorithms.Some of the things I ran into while testing my algorithms was first using linked lists instead of sorted arrays. This degraded the performance considerably in my first tests with sequential search since for each loop we conduct, it has to begin again at the beginning of the linked list. Since I had 1,000,000 objects in the linked list and was counting each node it visited and displaying it, I noticed it would go quickly through the first 100,000 and get slower and slower. I then realized that it was pretty obvious going to be terrible due to the way linked lists work, and arrays are much quicker regardless since they have direct access to array indexes. It was a good learning moment regardless as it gave me a clue of when it’s appropriate to use linked lists and even on, by today's standards and processor speeds, 1,000,000 is rather small, it still took a rather long time to completely finish a sequential search. On my computer it took nearly 12 hours from start to finish to compare every node for our searched value.

The other problem I ran into was I initially created a binary search tree to conduct our binary search algorithm when I meant to use arrays. This again was more than what was necessary but it was nice to re-build the BST class in Java and have it work correctly before scraping it. I did learn that Java doesn’t have a built in library for binary search tree that I could find, but does appear to have other B-Tree data structures. This is useful knowledge to know if in the future I ever need to construct a binary tree in Java

I know I'll need to construct my own which isn't terribly difficult to do. The biggest issue was, interesting enough, interpreting the data. Excel has it's own limitations on the amount of cell's a user can use as data points which limited some of our testing results slightly. This became a headache as getting our charts to correctly display what we expected to become a lesson in using excel and having our data correctly show what we need it too.

Everything else came together nicely for the project. Sequential search and binary search were rather easy algorithms to re-create and interpolation search wasn't to bad after a bit of debugging and resolving an issue. The performance against each algorithm is what was expected as sequential search was clearly the slowest algorithm performance wise and failed to even belong on the same size scale as binary search or interpolation, interpolation search gave very fast results all across the board on each test method, and binary search gave very predictable results. The best graph that shows a clear indication of how the algorithm can perform in real world tests is one of our final tests where we fill the array with random values, have it sorted, and repeatedly search for the same value over 1000 times.

In conclusion I think my algorithms performed as they were expected, and it was good to learn the strengths from each algorithm. Sequential will get the job done but it will surely take its time, but no matter what, brute force will find a way to find to discover our searched for value if it exists. Binary is consistent in it's performance and scales nicely, even with randomly generated values and uniform distribution of values in a sorted array. Interpolation search wins if the array is uniformly distributed, but if the

array is populated with random integers in sorted order, it is not worth choosing as a algorithm. Knowing ahead of time your data will surely point most people in the correct direction of what algorithm to use. For more results on our timing, please view the attached excel spreadsheet to see the recording of our times.