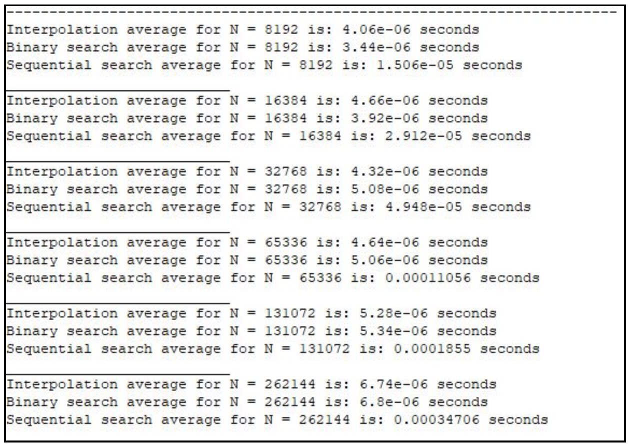
Sunanth Sakthivel
CS 350
Path: Searching

Term Project

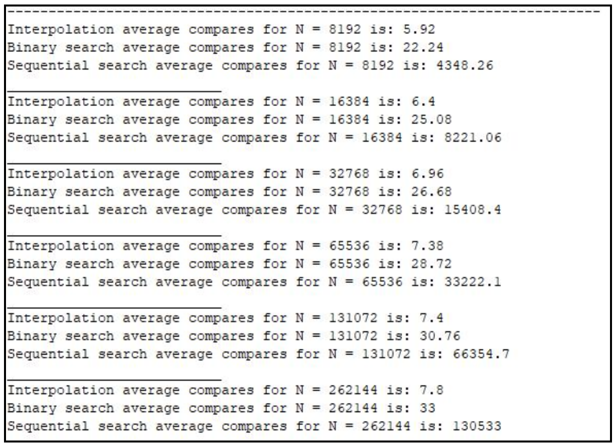
This project will explore three common search algorithms: sequential, binary and interpolation search. The most basic type of search algorithm is called sequential or linear search; this algorithm will search for a given key in an array of n elements by checking each element one by one until found or the list is exhausted. This type of search has a worst case where the key is never found and the entire array is searched which is n comparisons, best case would be only a single comparison where the very first element in the array happens to be the key. The average run time however is approximately (n+1)/2 comparisons which reduces down to a O(n) time complexity. Binary search, is considered a decrease-by-constant-factor algorithm; this search involves comparing the key with the array's middle element, if there is a match then return the index otherwise the same procedure is repeated recursively for either the first half or second half of the array depending on the value of the key. The worst case involves reducing the size of the array by a half each iteration until the final size is 1 which models a O(logn) relationship. Moreover, the average case would be only slightly smaller than the worst case which would also be roughly O(logn). Lastly, interpolation search is similar to binary search but is considered a variable-size-decrease algorithm; binary search always compares the key with the middle value of the array, whereas in interpolation search, the value of the key is taken into account when searching and dividing the array. In other words, interpolation search almost mimics the way humans search for key values based on the value rather than simply dividing the problem in half. Worst case scenarios involve data that is not uniformly distributed and can result in O(n) comparisons, however on the average case when data is uniformly random and distributed the complexity can be measured as O(log(logn)) comparisons.

Implementations of sequential, binary and interpolation searches are done through the following: Sequential search was straightforward and involves setting a flag to default -1 value. Next a for loop is traversed through the entire array until the key matches the array index. If match is found then flag is set to the index value and the for loop is exited and the flag is returned. If the key is not found then the entire array is exhausted and the flag is returned with the default -1 value. On every iteration of the for loop, a comparison counter is incremented by one to indicate that a comparison was made. Binary search was implemented recursively; first its checked to see if the lowest index is less than or equal to the highest index. Next the middle array index is found by taking the average of the high and low index values. Then if the array at the middle index equals the key then we simply return. However, if the value at the middle is greater than the key then we recursively call the binary search function but this time the high index value passed is changed to middle - 1. If the value at the middle is less than the key then we recursively call the binary search function but this time the low index value passed is changed to middle + 1. This recursive splitting in half continues until the key is found or if not found return -1. When array[mid] was checked to see if it is a match with the key, the comparison counter was incremented. Additionally, if array[mid] was checked to see if below or above the key the comparison counter was incremented again. Lastly, interpolation search was implemented similarly to binary search; First it is checked to ensure that the high index is greater than or equal to the low index, and that the key value falls between array[low] and array[high]. Next, unlike binary search which finds the middle of the array, we must use interpolation to find the area to search. The formula used is the following: pos = low + ((key-array[low])*(high-low)/(array[high]-array[low])). One caveat with this formula is the possibility of overflow so the formula was redistributed as: pos = low + ((double)(high-low)/(array[high]-array[low]))*(key-array[low])). The latter formula is distributed in a way that is less likely to result in an overflow and if an overflow were to happen, the former formula is used to calculate the position. Another check to consider is if low equals high, in which cause a floating point exception will result if either of the two formulas are used and therefore it is first checked and if so, position will equal 0. Next it is checked if array[pos] is equal to the key and if so return the position index. However, if the value at the interpolated position is less than the key then we recursively call the interpolation search function but this time the low index value passed is changed to pos+1. If the value at the interpolated position is greater than the key then we recursively call the function again but this time the high index value passed is changed to pos-1. This variable recursive splitting continues until the key is found or if not found return -1. Similarly to binary search, When array[pos] was checked to see if it is a match with the key, the comparison counter was incremented. Additionally, if array[pos] was checked to see if below or above the key the comparison counter was incremented again.

The general testing strategy incorporated in this project involves running multiple trial runs on various array sizes n for the various searching algorithms in order to compare the runtime and comparisons amongst the respective advertised time complexities. The array sizes tested started at 2^13 or 8192 elements and grew at an exponential rate of 2 for 6 runs; in other words, N = 8192, 16384, 32768, 65336, 131072, and 262144. The reason for picking an N that would grow exponentially was to allow for easy correlation between linear and logarithmic growth; algorithms which show exponential growth with an exponential growth in N would be considered linear in growth, whereas algorithms which grew linear with an exponential growth of N would be considered logarithmic in growth. In order to ensure random data, arrays were generated by dynamically allocating arrays of size N and inserting random numbers into the array using the rand() function, the arrays were then sorted. For each array size of N, a total of 50 trial runs were conducted per N size in order to provide an accurate average and account for initial startup, memory allocation, JIT and etc. Each trial run involves first allocating a new random array of N size and choosing a random key using rand(). Next, the three search algorithms: interpolation, binary and sequential search are individually performed in order to search for the random key. The runtime is recorded as well as the number of comparisons for each search algorithm. This process was continued for a total of 50 trials for each array size of N. Lastly, the recorded runtimes and comparisons for respective sizes of N were averaged (for the 50 trials) and displayed.



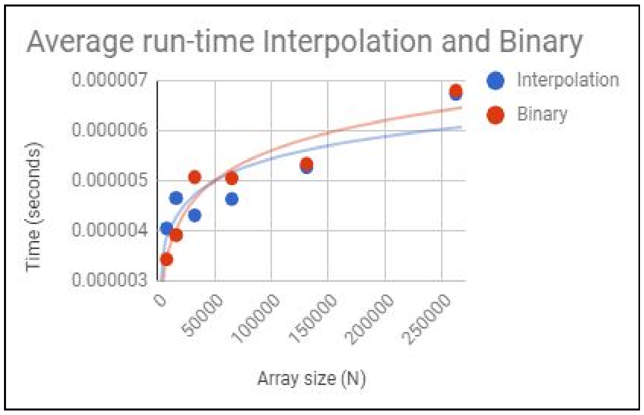
```
--------------------------------------------------------------------------
Interpolation average for N = 8192 is: 4.06e-06 seconds
Binary search average for N = 8192 is: 3.44e-06 seconds
Sequential search average for N = 8192 is: 1.506e-05 seconds

Interpolation average for N = 16384 is: 4.66e-06 seconds
Binary search average for N = 16384 is: 3.92e-06 seconds
Sequential search average for N = 16384 is: 2.912e-05 seconds

Interpolation average for N = 32768 is: 4.32e-06 seconds
Binary search average for N = 32768 is: 5.08e-06 seconds
Sequential search average for N = 32768 is: 4.948e-05 seconds

Interpolation average for N = 65336 is: 4.64e-06 seconds
Binary search average for N = 65336 is: 5.06e-06 seconds
Sequential search average for N = 65336 is: 0.00011056 seconds

Interpolation average for N = 131072 is: 5.28e-06 seconds
Binary search average for N = 131072 is: 5.34e-06 seconds
Sequential search average for N = 131072 is: 0.0001855 seconds

Interpolation average for N = 262144 is: 6.74e-06 seconds
Binary search average for N = 262144 is: 6.8e-06 seconds
Sequential search average for N = 262144 is: 0.00034706 seconds
```

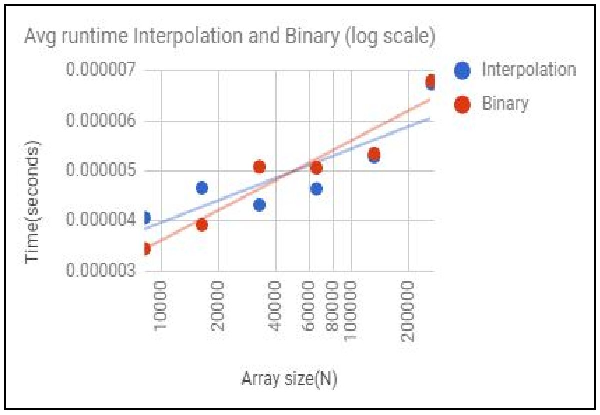


```
--------------------------------------------------------------------------
Interpolation average compares for N = 8192 is: 5.92
Binary search average compares for N = 8192 is: 22.24
Sequential search average compares for N = 8192 is: 4348.26

Interpolation average compares for N = 16384 is: 6.4
Binary search average compares for N = 16384 is: 25.08
Sequential search average compares for N = 16384 is: 8221.06

Interpolation average compares for N = 32768 is: 6.96
Binary search average compares for N = 32768 is: 26.68
Sequential search average compares for N = 32768 is: 15408.4

Interpolation average compares for N = 65536 is: 7.38
Binary search average compares for N = 65536 is: 28.72
Sequential search average compares for N = 65536 is: 33222.1

Interpolation average compares for N = 131072 is: 7.4
Binary search average compares for N = 131072 is: 30.76
Sequential search average compares for N = 131072 is: 66354.7

Interpolation average compares for N = 262144 is: 7.8
Binary search average compares for N = 262144 is: 33
Sequential search average compares for N = 262144 is: 130533
```

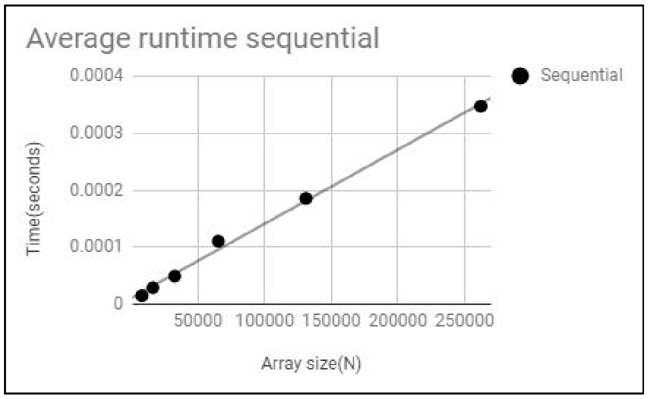
a). **b)**.

Figure 1. **a).** This is the display of the raw runtime averages for array sizes: 8192, 16384, 32768, 65336, 131072, and 262144. Runtimes were recorded in seconds. Averages were conducted through a trial of 50 runs. **b).** This is the display of the raw comparisons averages for array sizes: 8192, 16384, 32768, 65336, 131072, and 262144. Comparisons were recorded based on number of array comparisons with key. Averages were conducted through a trial of 50 runs.



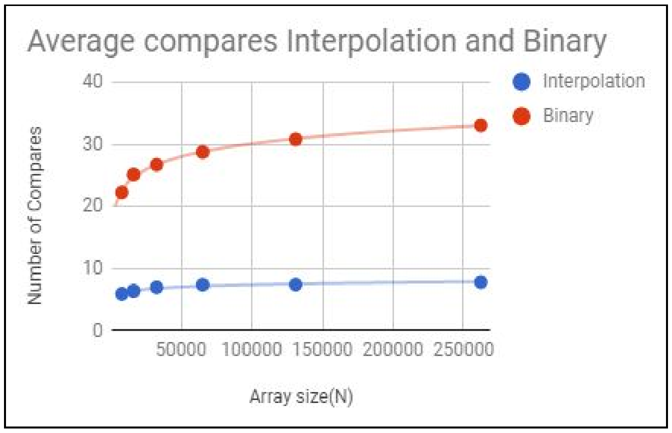


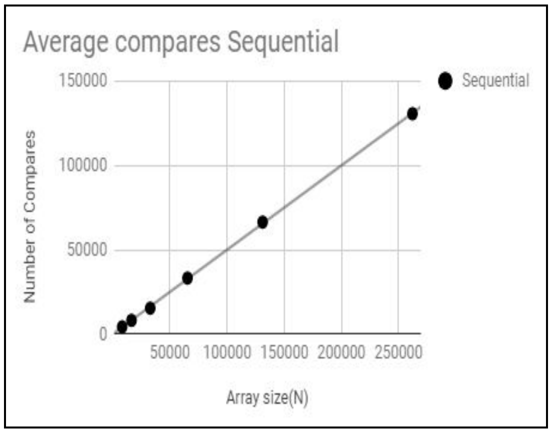
a). **b).**



c).

Figure 2. **a)**. Shows the average run-time of interpolation and binary search in seconds for various array sizes in linear scale. **b).** Shows the same data from figure 3a but in a logarithmic scale. **c).** Shows the average run-time of sequential search for various array sizes. Averages are conducted from 50 trial runs. Logarithmic trend line equation for interpolation: -1.95E-06 + 6.42E-07lnx. Logarithmic trend line for binary: -4.38E-06 + 8.67E-07lnx. Linear trend line for sequential: 1.3E-09x + 1.1E-05





a). **b).**

Figure 3. a). Shows the average number of comparisons for interpolation and binary search for various array sizes. **b).** Shows the average number of comparisons for sequential search for various array sizes. Averages conducted from 50 trial runs. Logarithmic trend line for interpolation: 1.3 + 0.528lnx. Logarithmic trend line for binary: -4.53 + 3lnx. Linear trend line for sequential: 0.499x + 94.6.

Looking at figures 2 and 3 give a general sense of how the runtimes and comparisons trend over an exponential increase in array size (from 8192 to 262144). The obvious observations are the fact that interpolation and binary search outperformed sequential search by a very large margin in both run time and comparison analysis. Figure 2a and 2b suggest that binary outperforms interpolation on smaller sizes of N whereas interpolation outperforms binary at larger sizes of N. In terms of growth relationships, notice that figure 2c clearly shows an linear relationship between the runtime and array size for sequential searches--the linear trend line matches very precisely to the plotted averages. Sequential search linear relationship is also clearly seen in figure 1a; as the size of the array doubles, the run-time in seconds also seems to double. Figure 3b further validates the linear relationship of sequential search by once again displaying a linear relationship between the number of comparisons made and array size. Moreover, based on the data sequential search can be estimated to run at O(n) time efficiency. Figures 2a show that the average run time of interpolation and binary search matches closely to a logarithmic relationship--figure 2b helps validate this observation by showing linear growth in runtime as array size is increasing at an exponential rate unlike sequential search. Runtime analysis would suggest interpolation and binary searches run at O(logn) time complexity. However, when observing the comparison averages in figure 3a, notice that while binary search shows clear logarithmic or O(logn) performance, interpolation search shows growth more closely resembling O(log(logn))--the function grows so slowly that it appears almost constant or growing at a very small constant despite exponential growth in size. The major cause of discrepancy between interpolation runtime (O(logn) and comparisons (O(log(logn)) could be based on the fact that interpolation search involves an expensive math calculation on every iteration as well as floating point conversions that slow the runtime of interpolation search.

Some issues that occured while implementing and testing the various search algorithms is the following: Interpolation search was initially implemented without recursion while binary search was implemented with recursion--clearly this type of implementation would favor interpolation over binary so in order to provide runtime fairness, both interpolation and binary were implemented using recursion. Another problem was the fact that rand%(some number) wouldn't provide random enough numbers because it relies on just using the lowest-order bit when generating numbers, instead (number*(rand()/RAND_MAX)) was used to generate more random numbers using the high-order bits. Additionally, arrays were initially pseudo randomized by inserting random numbers that were guaranteed to be larger than the previous element--this however isn't truly random. Instead, random numbers were first inserted into the array and then the array was sorted. Lastly, as mentioned before, running interpolation search with large array sizes and key values, increased the likelihood of overflow and floating point exceptions--the only way to fix these exceptions was to handle each case at the cost of some runtime performance.

Aspects of the project that went well include the ease of replication for various types of tests due to test automation. In other words, variable definitions can be easily set to different values to perform a variety of different tests such as different sizes of N or different number of trials. Additionally the overall collection and display of data was very organized which easily allows for data analysis and graphical presentations--as mentioned before, a lot of the test automation can be adjusted to receive custom data as desired. The overall implementation of the various search algorithms was also very straightforward (aside from dealing with overflow faults in interpolation search). Adjusting algorithms from non-recursive to recursive counterparts wasn't very difficult as the implementations were virtually the same.

To conclude, based on both run-time and comparison performances, it is very clear that sequential search performs at a O(n) time complexity and is by far the worst search method in comparison to interpolation and binary search. The real question then becomes which search is better, Interpolation or Binary search? Ideally, if time complexity was measured strictly with the number of comparisons (w/ other extraneous factors being ignored or controlled) then it is clear that Interpolation search beats binary search (O(log(logn)) vs O(log(n) respectively) on average. Runtime analysis however suggests that both interpolation and binary search both grow at a similar logarithmic relationship and show similar performances--the big distinction between binary and interpolation search is that interpolation search has a costly mathematical calculation and float conversions that potentially hinder its true potentials. Perhaps if custom implementations of interpolation search were to minimize the mathematical conversions based on the data at hand, the runtime performance of interpolation search would truly shine over binary search. Nevertheless, based on the data collected, Interpolation search is preferred over binary search when the size of the array tends to be very large, when the data is known to be somewhat uniformly distributed (intervals between elements is approximately constant), and when comparisons are very costly (collected data clearly shows interpolation search has far fewer comparisons than binary on average). Binary search on the other hand has one very powerful advantage over interpolation search and that is consistency and reliability; conducting a binary search will always result in a O(log(n)) performance regardless of whether the data is uniformly distributed or not and is thus not bound to the data set.