Matthew Knowlton (80745066)

Dr. Monika Akbar

Elementary Data Structures and Algorithms

26 October 2022

Comparing Linear and Binary Search Algorithms using Hardware Execution Timing

1. Runtime Environment

The computer used to conduct the experiment was an HP 15-inch OMEN gaming laptop. It has an Intel i7-7700HQ as a processor with 4 physical cores and 8 threads clocked at 2.8 Gigahertz (GHz). It contains an 8 Gigabyte (GB) stick of Double Data Rate 4 (DDR4) small outline dual in-line memory module (SODIMM) memory clocked at 2133 Megahertz (MHz). The operating system choose for testing was Windows 10 Home Edition, version 21H2. Amazon Corretto version 17.0.4.9 was chosen as the Java Development Kit (JDK) version for the experiment (*Table 1*).

Computer	OMEN by HP 15-ax256nr Laptop PC
CPU	Intel® Core™ i7-7700HQ (2.8 GHz, up to 3.8 GHz with Intel® Turbo Boost Technology, 6 MB cache, 4 cores)
GPU	NVIDIA® GeForce® GTX 1050 (2 GB GDDR5 dedicated)
RAM	8 GB DDR4-2133 SDRAM (1 x 8 GB)
HD	1 TB 7200 RPM SATA
OS	Windows 10 Home 21H2 (Build: 19044.2130)
JAVA	Amazon Corretto 17.0.4 9 (x64)

Table 1

2. Experiment

The purpose of this experiment was to gather data and show the correlation of the predetermined O notation with the real-world time measurements to complete each search algorithm using a range of elements as input on hardware. The first step was to create an array which contains 10000 elements and assign each element a value of a random number between 0 and 9999. Then choose a random index from the array and use its value as an input into the search algorithms. Run and record elapsed time for each algorithm using the same array and search term. In order to eliminate run-to-run variance run this step at least twenty times and average the resulting times. Repeat these steps increasing the number of array elements by 10000 each time. Finally plot the results on a chart using the array length as the x axis and the time it took to complete the search on the y axis.

In order to make it easier to gather results, I made a few tools to allow me to gather the data faster. First, I created a method to return an array with a given number of elements each assigned a random value from a given minimum to a given maximum. This allowed me to increase and decrease the size of the array as well as the range of values within the array at will. I then created a method that takes an integer for the size of the array and an integer of the number of times it should run. The method then creates an array of the given number of elements using the previous method and runs the array through the search algorithms using a random index value from the array. It times each run and averages the run times together and the returns a string with the number of elements, the average linear search time, the average binary search time, the average recursive binary search time, the time it took to create the array, and the time it took to sort the array in a comma separated format. I also created a method that takes a start value, end value, period value, and the number of runs in order to run the previous method through a range of array sizes from the start value to the end value for every period value and then gathers the resulting strings into an array list. The final method I wrote took the array list from the previous method and writes the data to a comma separated value (csv) file to make it easier to import into Microsoft Excel and generate the graphs.

Finally, I created an Excel spreadsheet which contains a power query that imports the csv file from the java program into an excel spreadsheet. It then generates a graph to compare the times of the three search algorithms, a graph to chart the time to sort the arrays, and a graph to chart the time to create the arrays. On each refresh of the input csv file, these charts regenerate with the newly updated values.

3. Results

Using the tools I built, I generated a graph for all the values from 1,000 to 1,000,000 using 1,000 size increments. The results for the linear search algorithm follow the expected slope for a O(n) function. The time required to execute a linear search grows with the increase in array size. (*Figure 1*) The binary and recursive binary search algorithms both show a general O(log n) slope using the original data. This is because binary (and recursive binary) search only requires a log n number of comparisons to find the given element. However, since these values are so low compared to the linear search, it does not show on the graph in Figure 1. So, I created a second graph with just the binary and recursive binary algorithms. (*Figure 2*) The standard java array sorting method which is necessary for the binary and recursive binary algorithms to function shows a O(n log n) slope. (*Figure 3*)

These results show a strong correlation between the estimated o notation and the real-world time it takes to execute them. However, there are some caveats to consider. When running the experiment any background task running may take up CPU time and thus would affect the run time of each algorithm. The cooling of your CPU may also affect the results as it may downclock in order to prevent overheating thus skewing your results. Also, if you are running using a laptop, your CPU may downclock in order to save battery further skewing your results. Furthermore, if you were to run this experiment with a different JDK version, you may end up with different results as different JDK versions may behave slightly differently.

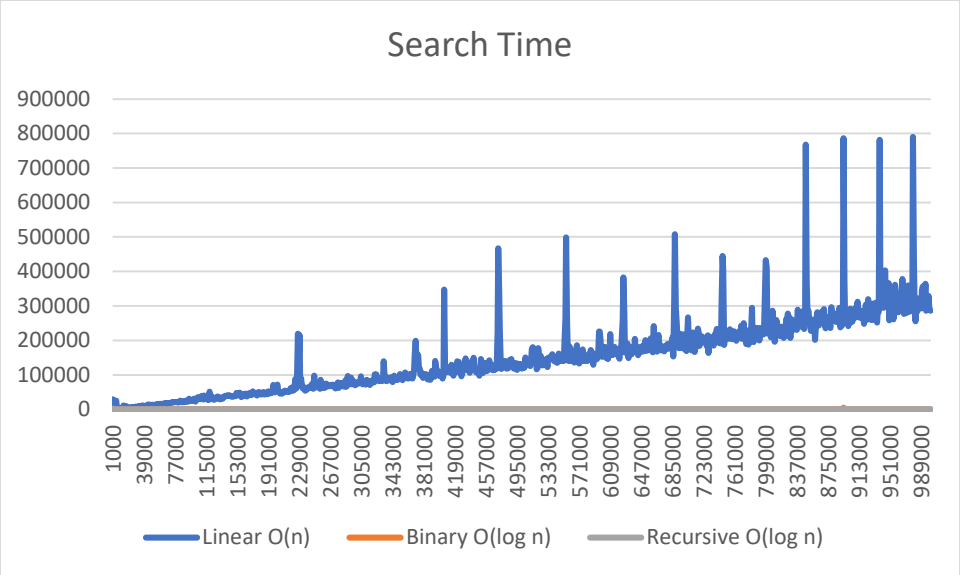

Figure 1

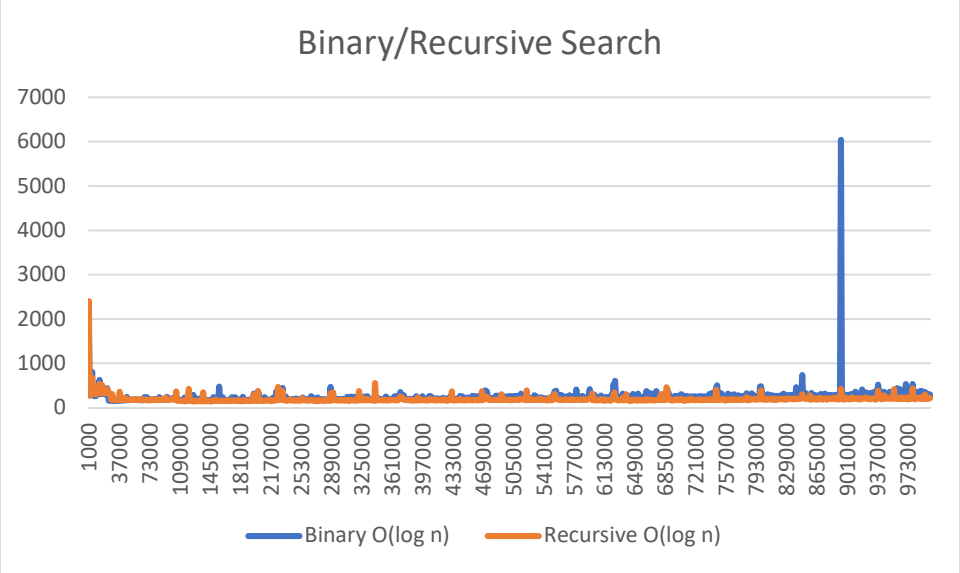

Figure 2

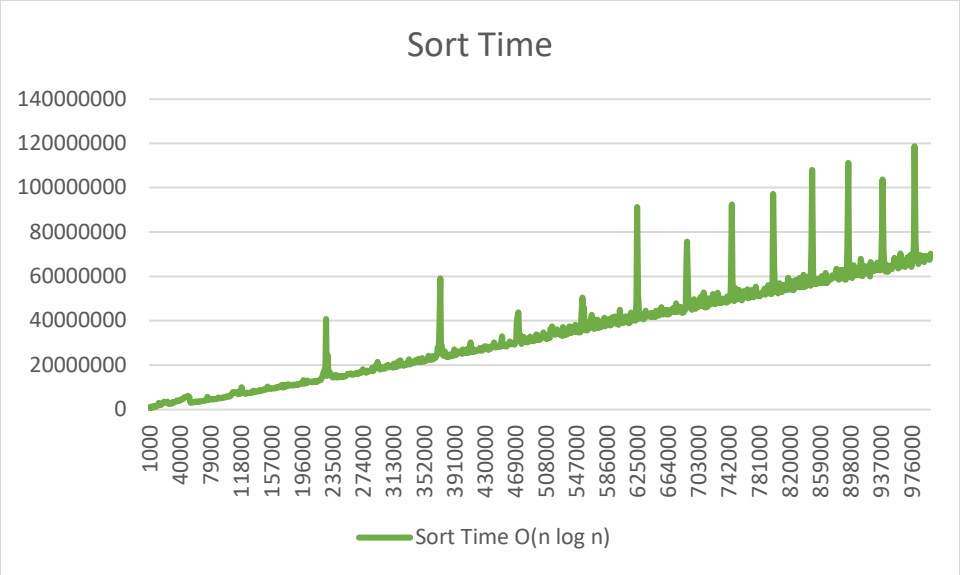
Sort Time
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
40000
79000
118000
157000
196000
235000
274000
313000
352000
391000
430000
469000
508000
547000
586000
625000
664000
703000
742000
781000
820000
859000
898000
937000
976000
Sort Time O(n log n)

Figure 3