Joaquin Hidalgo
CS 2401
Lab 07
MW @3:00-4:20pm

RUN-TIME analysis

Steps to gather data:

1. Setup method to generate a random array of length size ‘X’ (variable: X).
   Now fill-up array length of X, with doubles from 0 to 100.
   Lastly, order the array from smallest to largest.
2. Setup method to generate a random position in the array which can be anywhere between from position 0 to (X-1). This will be called our key position.
3. Have two methods which will later will be our comparison for this lab.
   Method 1 is going to linearly search through the array until we find that, “random position”. It will return the index of the array that matches the key.
   Method 2 is going to be our binary search which will return the index of the matching key.
   Method 1 has a run-time of O(n).
   Method 2 has a run-time of O(log(n)).
4. Time stamp the algorithm right before and after the search algorithm has been called. Subtract the end time from the start time and save value array of time-stamps. Now we have time stamps for linear search and binary search.
5. Gather 30 time-stamps for the variable ‘X’ in linear and binary each. Average the time-stamps over these 30 tries and display the average run-time.
6. Now try changing the variable ‘X’ to different sizes, to measure different array lengths. Change ‘X’ to 10000, 40000, 160000, 640000 and 1280000.
7. Done.

Discussion:

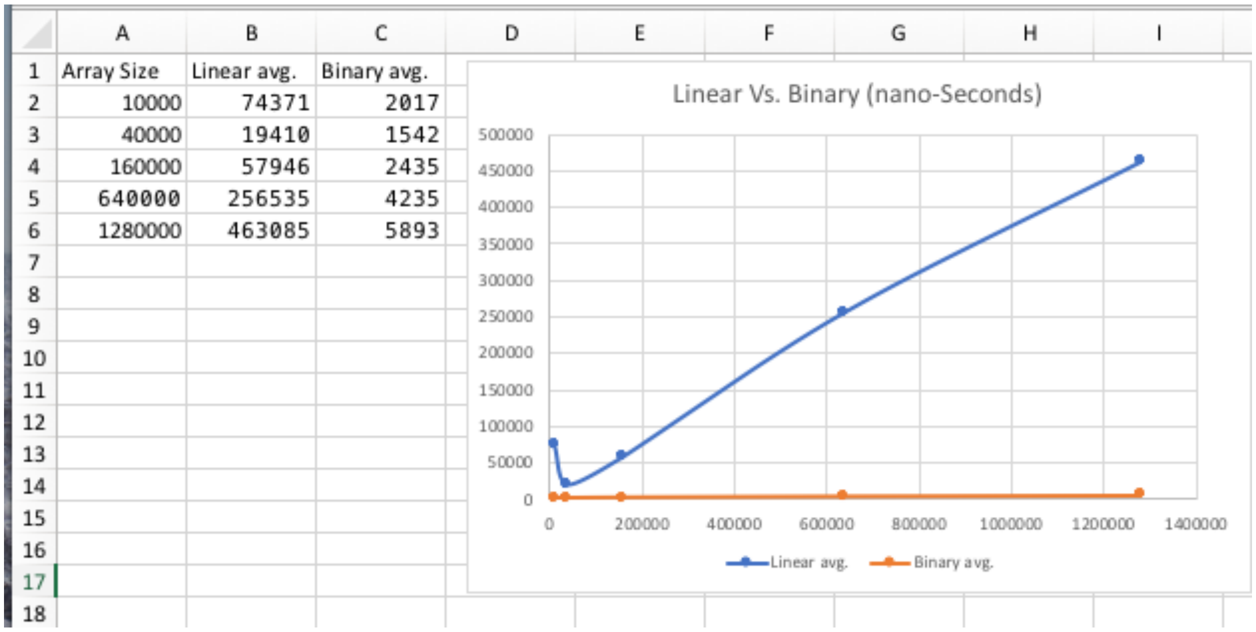
One way to make sure, we have a more reliable data sets is to repeat the time-trial for each algorithm 30 times then compute the average. Many factors can alter how we record our data sets for example, when we look for our ‘Key’ in the array, that position is generated randomly. In the linear search, if for the 30 times we generated a random position and all those times we got, Zero, then the algorithm will only check the first position, notice it equals the key and return that index. Although that sounds great, it would not check the other thousand positions and then our results will show that our linear search is faster than it should be. Same thing with our binary search, if our random generated position is exactly in the middle of the array for the 30 times repeated, it would make our binary algorithm seem faster than it is.

Lastly, System.nanoTime() gives a more precise measurement of time (10^-9) than using System.currentTimeMillis() (10^-3). Linear O(n) & Binary O(log (n)) on the graph below shows that over time, as the array size increases, the binary search algorithm grows as a log(x) graph would and the linear algorithm grows at a rate of O(n) so as the size gets bigger, the run time get way bigger too, unlike binary.

Although, I did notice something unusual when the linear and binary averages are returned, the array of 10,000 is slower than the array of 40,000 and sometimes 160,000. I hope to learn more in my career of Computer science to find out why that is. :)

Results:

```
Array size , linear average(nanoSeconds), binary average(nanoSeconds)
10000,        74371,                        2017
40000,        19410,                        1542
160000,       57946,                        2435
640000,       256535,                       4235
1280000,      463085,                       5893
```



Array Size	Linear avg.	Binary avg.
10000	74371	2017
40000	19410	1542
160000	57946	2435
640000	256535	4235
1280000	463085	5893

Running Environment:

Computer: MacBook Pro (mid-2012)
Processor: 2.5 GHz intel core i5
Memory: 8 GB (1600MHz DDR3)
Operating System: macOS High Sierra v. 10.13.6
Java version: JDK 9.0.4