

CS3310

First Assignment (a0)

Report

Brandon Rodriguez

09-27-17

Preface

First off, a working, compiled jar of the assignment can be found in the root directory, called “a0.jar”. To run this, open up a terminal, navigate to the directory, and type “java -jar a0.jar”. The rest should be self-explanatory.

To find the source code of the assignment, navigate from the assignment’s root folder to “root/a0/src/com/CS3310/a0”. A majority of the code is located within the “Controller.java” class.

Used libraries are included in the “root/a0/lib/” folder.

JavaDocs can be found at “root/Documents/JavaDocs/index.html”.

Runtime output logs can be located at “root/Documents/RuntimeOutput/” and are organized by type of input provided. The value “n” denotes the size of both array dimensions, which capped at 8000 because my laptop started having memory heap errors for larger values. The value “m” denotes the number of times a random character was changed.

Program was created using JetBrains’ “IntelliJ” IDE.

If not familiar with IntelliJ:

Should the jar file not execute, you should be able to load it into IntelliJ and get the same results. When first opening the project, use “Import Project” through IntelliJ’s splash screen. Then select “Create project from existing sources”.

If IntelliJ doesn’t automatically do it for you, make sure to set the “src” directory as the “Sources Root”, and the “tests” directory as the “Test Sources Root”. You may also need to load in JUnit, which should be provided within the lib folder.

You may also need to “Edit Configurations” and set Main as an application to launch from. But after that, it should work. Just run the program with the green arrow button and it should launch.

Problem Statement

In an attempt to examine various data structures and efficiency of various algorithms, this assignment was created. Specifically, it focuses on comparing binary and linear searches within standard arrays.

This assignment uses unsorted arrays for linear searching and sorted arrays for binary searching. It tracks time spent by milliseconds (although, with small values of n and m , the time is so trivial that it could not be read) and prints out for user display.

Program Description

The program starts by prompting the user for an array dimension, which is used to create a $(n \times n)$ 2d array. It then populates every spot in this array with a random lowercase ascii letter. Next, the program creates a comparable array of sorted values (starting with 'a', ending with 'z') and displays time spent for both.

The program also asks the user for a "name", which honestly can be any arbitrary string of characters. This value is used for the rest of the program as the string which searches are ran against.

The program feeds this name value in character by character, first to the linear search, then to the binary search, outputting the index of each character when found. If the character is not found, then an index of $[-1, -1]$ is given instead.

The program also keeps track of which indexes have already been "found", skipping over those to account for potential repeating characters.

For the last part, the program prompts the user for an integer. The program then replaces a random letter in the name with a new random character, and repeats this equal to the integer that was provided. For every replacement, the name is once again ran through the linear and binary searched. For this section, individual "found indexes" are not printed out.

However, at the end of the character replacement loop, various stats are printed out, including things like total and average times for each search. Again, for low values of n and m , these values are may be so trivial that they can't even be recorded in milliseconds, and thus print out as all 0's. For more meaningful data, use larger values of n and m .

Algorithm Descriptions

The used algorithms are mostly just standard linear and binary searches, with one modification: They both had to be sensitive to previously-located values. As such, only the modified sections will be described here.

Commonality

First off, I will say for both searches that they used a similar means of storing what values have already been found. For each, there was an array of equivalent size and dimensions. This array also stored characters in every index. However, instead of random characters, only two characters were stored here.

The first character was an 'o', which was used to denote that the given index had not been "found" yet. The second character was a 'x', which was used to denote that the given index had been "found".

Upon a search match in either search, a check was done on the secondary array at the equivalent index. If the char located there was an o, then the search was deemed valid. If it was an x, then the linear/binary search ignored the "found" value and continued on as normal. Upon any matches, the location at the secondary array was updated from 'o' to 'x'.

This arrangement meant that all "valid or not" comparisons took exactly 1 time, as the index to check was already known, and there is no need to look anywhere else. However, it also meant that the program took about twice as much memory due to twice the storage needed for the additional arrays.

Psuedo-Code:

```
on search match,
    if secondaryArray[foundIndex] equals 'o'
        then
            secondaryArray[foundIndex] now equals 'x'
            exit loop and return foundIndex
        else
            ignore search match and continue search
end search match
```

Linear Search

The linear search really didn't change too much from the above modifications. It just sometimes ignored a given match and continued to the next value. (See above for psuedo-code.)

Binary Search

The binary search actually had to be decently modified to accommodate the changes. It ended up getting two additional functions created for this very reason.

Essentially, upon a binary search match with a larger dataset, it becomes inefficient to "just ignore the current value and continue the search as normal".

Instead, once a match was found, the search would essentially stop. If the "found" value was valid, then it just returned that. Otherwise, if the value had already been used, the search would enter two new recursive loops- first to check the values directly left, then (if still no match) the values directly right. These two recursive checks would continue until:

1. **A valid match.**
2. **An invalid index was given**, which meant that it was searching past the start/end of the array, and thus a valid value would not be found in that direction.
3. **A different character was found.** Due to the array being pre-sorted, this also meant that a valid value would not be found in that direction.

If, even after these new recursive functions, a match was still not found, then an index of [-1,-1] was returned.

Psuedo-Code:

```
on search match:
    if secondaryArray[foundIndex] equals 'o'
        then
            secondaryArray[foundIndex] now equals 'x'
            exit loop and return foundIndex
    else
        stop binary search
        check values to left where char still matches search
            if found and secondaryArray[thatNewIndex] equals 'o'
                secondaryArray[thatNewIndex] now equals 'x'
                exit loop and return thatNewIndex
            else
                Try the same but with values to the right, instead
                    if match and secondaryArray[otherNewIndex] equals 'o'
```

```
secondaryArray[otherNewIndex] now equals 'x'  
exit loop and return otherNewIndex  
else  
    no matches found  
    return -1 index  
end try  
end then  
end search match
```

Tables of Observed Time Complexities

Note: Format is of (Minute : Second : Millisecond)

Any values which read as 0 are marked as “=” for readability.

Linear (Unsorted) Search

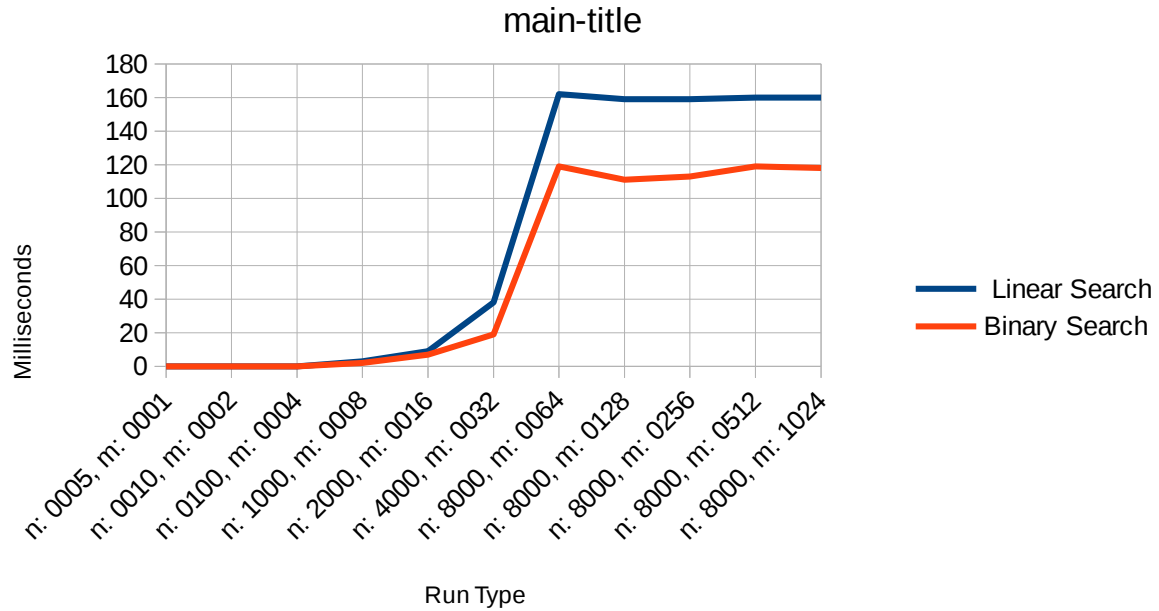
	Average Time	Total Time	Shortest Iteration	Longest Iteration
n: 0005, m: 0001	= : == : ==	= : == : ==	= : == : ==	= : == : ==
n: 0010, m: 0002	= : == : ==	= : == : ==	= : == : ==	= : == : ==
n: 0100, m: 0004	= : == : ==	= : == : 003	= : == : ==	= : == : 001
n: 1000, m: 0008	= : == : 003	= : == : 027	= : == : 001	= : == : 008
n: 2000, m: 0016	= : == : 009	= : 00 : 157	= : == : 003	= : == : 043
n: 4000, m: 0032	= : == : 038	= : 01 : 217	= : == : 012	= : == : 159
n: 8000, m: 0064	= : == : 162	= : 10 : 397	= : == : 137	= : == : 275
n: 8000, m: 0128	= : == : 159	= : 20 : 471	= : == : 137	= : == : 268
n: 8000, m: 0256	= : == : 159	= : 40 : 178	= : == : 135	= : == : 277
n: 8000, m: 0512	= : == : 160	1 : 22 : 283	= : == : 135	= : == : 261
n: 8000, m: 1024	= : == : 160	2 : 44 : 178	= : == : 136	= : == : 207

Binary (Sorted) Search

	Average Time	Total Time	Shortest Iteration	Longest Iteration
n: 0005, m: 0001	= : == : ==	= : == : ==	= : == : ==	= : == : ==
n: 0010, m: 0002	= : == : ==	= : == : ==	= : == : ==	= : == : ==
n: 0100, m: 0004	= : == : ==	= : == : 002	= : == : ==	= : == : 001
n: 1000, m: 0008	= : == : 002	= : == : 023	= : == : ==	= : == : 016
n: 2000, m: 0016	= : == : 007	= : == : 123	= : == : 003	= : == : 024
n: 4000, m: 0032	= : == : 019	= : == : 628	= : == : 012	= : == : 059
n: 8000, m: 0064	= : == : 119	= : 07 : 638	= : == : 050	= : == : 776
n: 8000, m: 0128	= : == : 111	= : 14 : 217	= : == : 050	= : == : 197
n: 8000, m: 0256	= : == : 113	= : 29 : 178	= : == : 050	= : == : 187
n: 8000, m: 0512	= : == : 119	1 : 01 : 424	= : == : 050	= : == : 207
n: 8000, m: 1024	= : == : 118	2 : 01 : 797	= : == : 050	= : == : 207

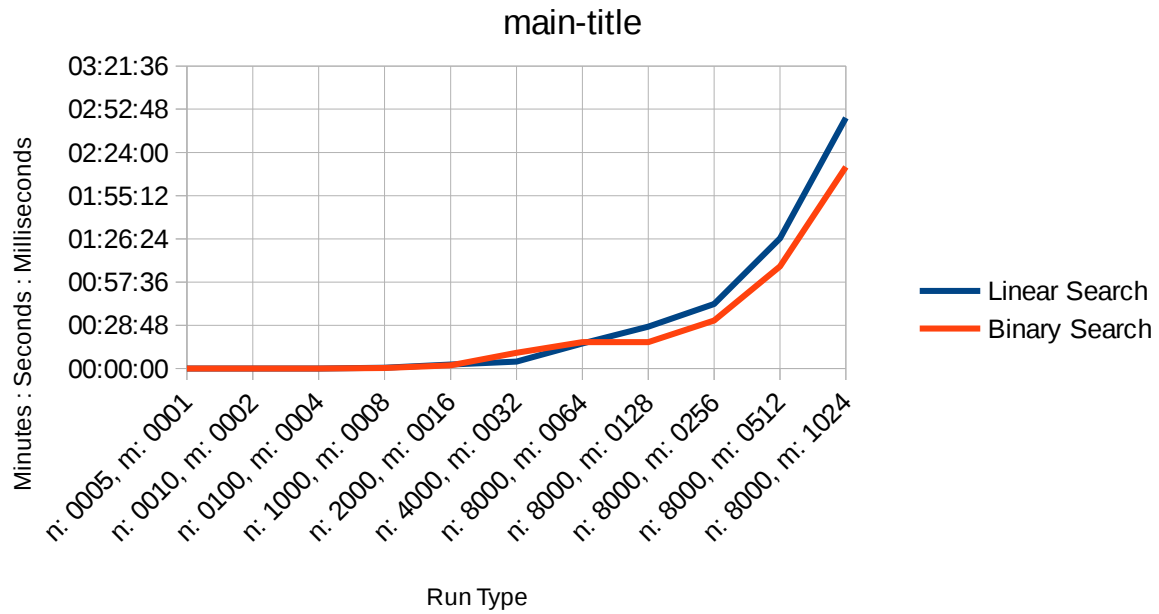
Plots of Observations

Average Time



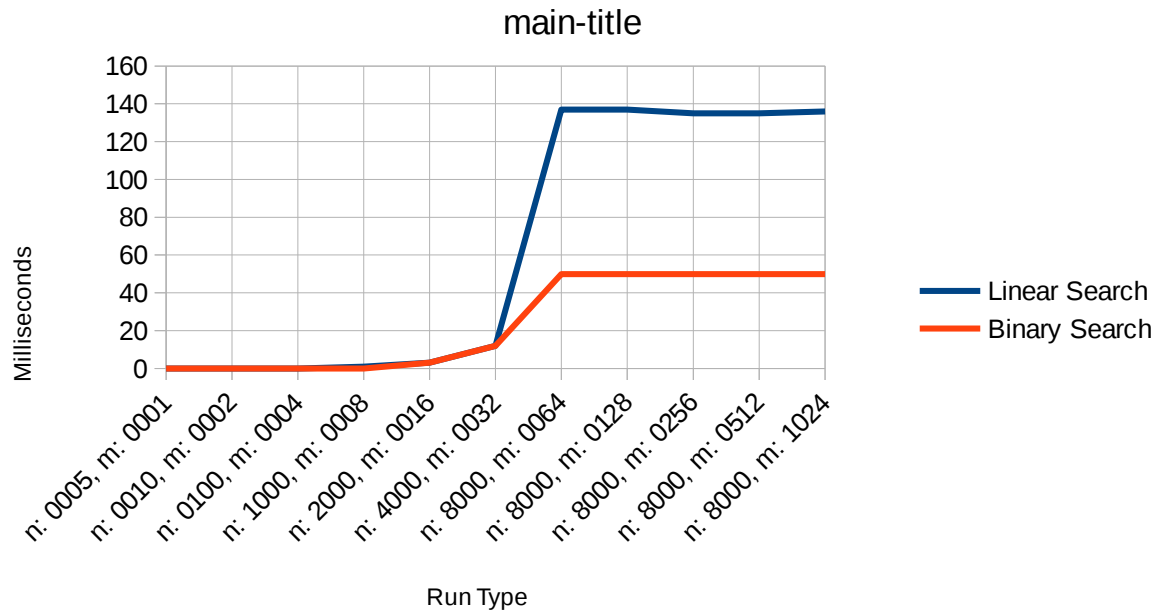
	Linear Search	Binary Search
n: 0005, m: 0001	000	000
n: 0010, m: 0002	000	000
n: 0100, m: 0004	000	000
n: 1000, m: 0008	003	002
n: 2000, m: 0016	009	007
n: 4000, m: 0032	038	019
n: 8000, m: 0064	162	119
n: 8000, m: 0128	159	111
n: 8000, m: 0256	159	113
n: 8000, m: 0512	160	119
n: 8000, m: 1024	160	118

Total Time



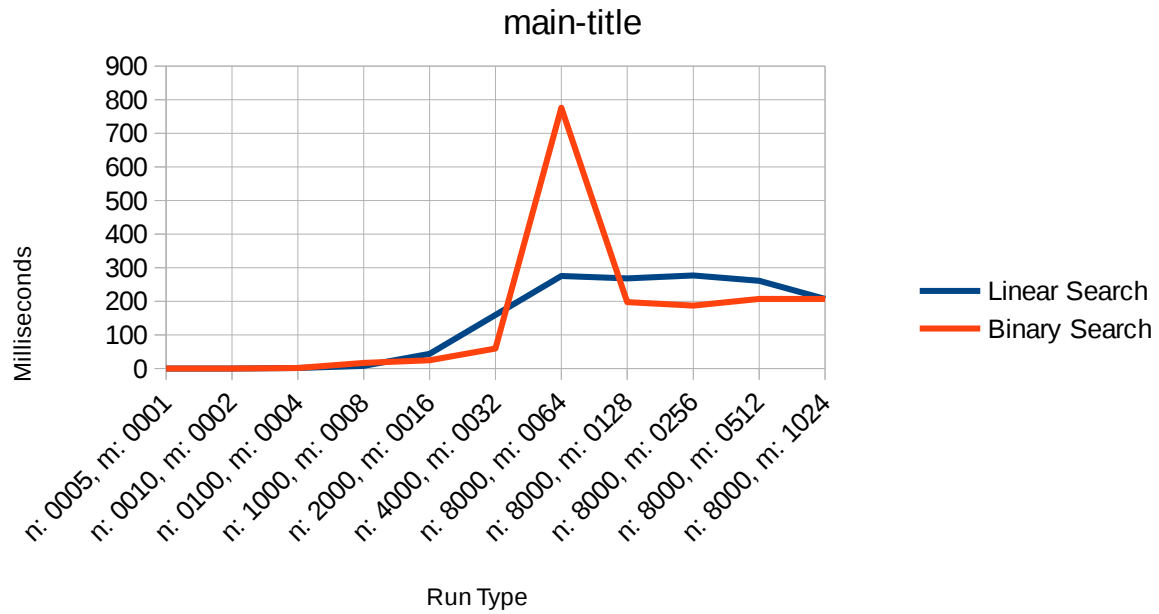
	Linear Search	Binary Search
n: 0005, m: 0001	0 : 00 : 000	0 : 00 : 000
n: 0010, m: 0002	0 : 00 : 000	0 : 00 : 000
n: 0100, m: 0004	0 : 00 : 003	0 : 00 : 002
n: 1000, m: 0008	0 : 00 : 027	0 : 00 : 023
n: 2000, m: 0016	0 : 00 : 157	0 : 00 : 123
n: 4000, m: 0032	0 : 01 : 217	0 : 00 : 628
n: 8000, m: 0064	0 : 10 : 397	0 : 07 : 638
n: 8000, m: 0128	0 : 20 : 471	0 : 14 : 217
n: 8000, m: 0256	0 : 40 : 178	0 : 29 : 178
n: 8000, m: 0512	1 : 22 : 283	1 : 01 : 424
n: 8000, m: 1024	2 : 44 : 178	2 : 01 : 797

Shortest Time



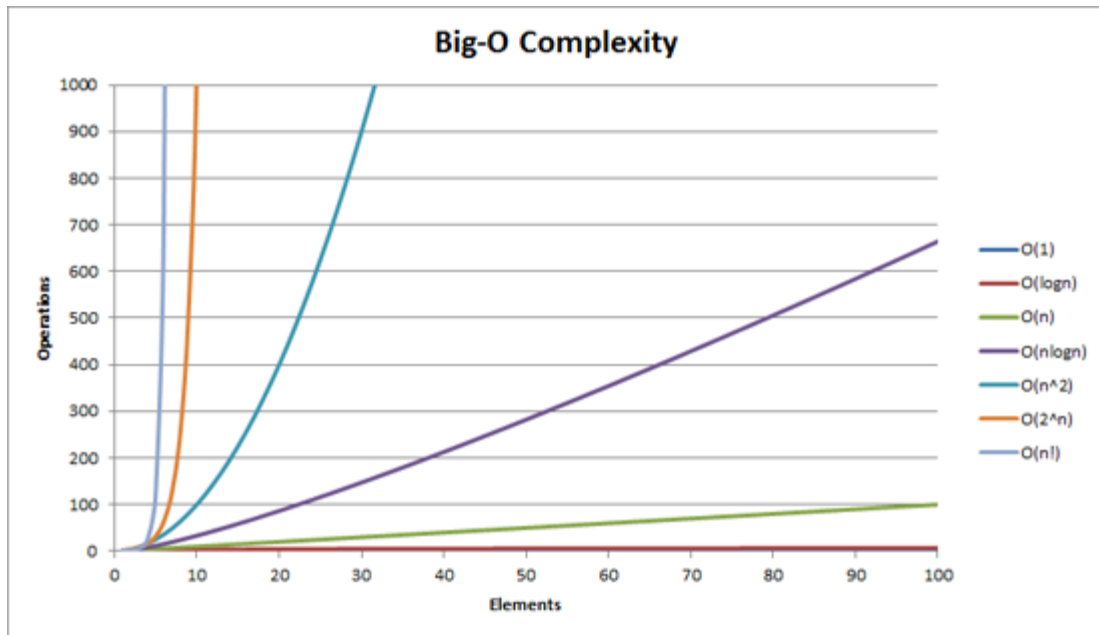
	Linear Search	Binary Search
n: 0005, m: 0001	000	000
n: 0010, m: 0002	000	000
n: 0100, m: 0004	000	000
n: 1000, m: 0008	001	000
n: 2000, m: 0016	003	003
n: 4000, m: 0032	012	012
n: 8000, m: 0064	137	050
n: 8000, m: 0128	137	050
n: 8000, m: 0256	135	050
n: 8000, m: 0512	135	050
n: 8000, m: 1024	136	050

Longest Time



	Linear Search	Binary Search
n: 0005, m: 0001	000	000
n: 0010, m: 0002	000	000
n: 0100, m: 0004	001	001
n: 1000, m: 0008	008	016
n: 2000, m: 0016	043	024
n: 4000, m: 0032	159	059
n: 8000, m: 0064	275	776
n: 8000, m: 0128	268	197
n: 8000, m: 0256	277	187
n: 8000, m: 0512	261	207
n: 8000, m: 1024	207	207

Theoretical Complexity Graph



Concluding Analysis

Total Time Analysis

The expected theoretical times of various equations can be seen above, in the “Big-O” graph.

Linear search is expected to follow the line of $O(n)$. Meanwhile, because the array is pre-sorted, binary search is expected to follow the line of $O(\log n)$ [otherwise, it would follow $O(n \log n)$]. To get the best feel for if these are accurate, we would probably want to look at the total times.

The total time is the only value that changed significantly based on the number of searches to run (IE: the number of characters replaced, or m). As expected, there was very little difference between the two at lower values, with linear search even having a lower time at points. However, as m grows arbitrarily large, the two begin to deviate, with binary having the generally smaller values.

The current data does not show them deviating particularly far, but the gap would likely increase as m grew. The current empirical data does indeed match up with expected theoretical. Unfortunately, due to time limitations, data could not be gathered for even larger values of m .

Sidenote: It can be stated that, from the current data, the linear search does not look 100% linear. At least from the current data, it looks to be more of a curve that slowly grows with larger m values.

First off, this perception may be entirely due to the given dataset. A normal line graph likely should have values that change in consistent increments. However, the given values are more multiplicative. This results in the data to the right being skewed as it “jumps” to far greater values across the x -axis. Adjusting the input values would likely account for this and give a more “standard” linear graph.

Should the linear search still look skewed even after this, then it’s possible that the “modified” part of the linear search is also significantly increasing the time spent. For, as it replaces more and more values, it will have to travel further and further to get a non-repeating value. The worst case will stay the same in these instances, but the best case will, by necessity, be closer and closer to the full array. By extension, this means that the average will also skew significantly which in turn, will increase the total time.

This may be trivial in smaller datasets but may have a large impact of exceedingly large ones.

This likely would not affect binary as much because, once binary has a match, any further values will be nearby, due to sorting.

Analysis of Other Values

An interesting note is that, based on current data, the average, shortest, and longest times seem to directly correlate to the size of the array to search through. Once the array size stopped growing, they all appear to have plateaued and returned consistent, mostly-static values.

The only value at this point that changed to any significant degree was binary search on “longest time”, which was due to a single data point and could be attributed to random error, such as computer freezing up, a very abnormal dataset that was literally worst-case scenario, or some other unlikely (but potential) outside factor.

It’s difficult to judge without more data. As stated above, both more linear of input values for the x-axis (as opposed to them generally being doubled), as well as simply more tests with larger n and m values, would significantly help determine how well the emperical data matches the theoretical.

However, given the current data set and the expected behavior, I am going to tentatively declare that it does indeed line up. Linear search is equal or better, at lower values. And as the values increase, linear begins to fall off until there is a stark difference and binary becomes the clear winner.