Comparison of Linear and Binary Search Algorithms and Their Associated Run Times
Lab Assignment #1

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
August 30, 2017

1. **Problem Specification**

The goal of this lab was to implement, test, and compare the binary and linear search algorithms as well as their associated physical running times as the searches occurred in one's execution of the program itself. The program first reads in a series of 1,000 possible random integers from a provided file, from which the program runs tests for each search key within. The list in which the program runs a linear search followed by a binary search is generated randomly as well using 2^N inputs where 4 <= N <= 25 as the number of entries in each randomized list. Each of the random values could be any randomly generated number from 1 to 2^N. The associated time elapsed over the course of each search operation as well as the index in the array where each element is found is printed if the element was found in the array. Otherwise, "Not here!" is printed before the next, larger array is generated and repeated within the range of N previously mentioned.

2. **Program Design**

The design of this program was fairly simple, requiring no additional object classes and most of the functionality relied on the binarySearch and linearSearch methods themselves. Additionally there were two other methods added for the sake of reading in the input file and generating the randomized lists. All other calculations, inputs, and outputs were generated in the main() method.

The steps taken to develop the program were:

a) Develop and test working versions of the linear and binary searches, which were both written and tested vigorously during the in-class portion of the lab. While the binary search also has an iterative implementation, the binary search algorithm implemented in this program is recursive.

b) Develop the initializeList method which takes a number "num" as a number to initialize the size of a generated list as well as the upper bound of values which can fill each index in the resultant array. A fully randomized array of size "num" is then returned.

c) Develop the readInNums class, which accepts a Scanner object to be used in order to read in t he integers from the "input_1000.txt" file, scanning one integer at a time into a returned array. This array is later used when choosing previously randomly generated values to be searched for in the also randomly generated lists.

d) Create a loop in main to span the values of N from 4 to 25 so that for each randomized value picked from the list of searchKeys, all values of 2^N could be sent to the initializeList method, storing the resultant generated list each time in a "nums" integer array, which is then used along with the associated searchKey in linearSearch followed by the binary search. Note : the same list is used for each sort, however, the list is first sorted using a built-in Java function before the binary sort can be called.

e) Many intermediate print statements were originally included in the program for the sake of functionality testing and efficiency. When the values of the nums arrays became too large for the standard eclipse terminal output, these were deleted. The last stage in development of this code was cleaning up the code, adding comments for code readability, and adding nice f ormatting for the values which are printed in the terminal for easy access of data collection.

The following methods were used in this lab:

a) readInNums (Scanner derp) : receives a Scanner object, locally referred to as “derp” within the scope of this method and scans the “input_1000.txt” file provided, returning an array of all the values which are scanned in.

b) initializeList(int num) : initializes an integer array of size “num” and then fills every index of that array with a randomly generated number in the range of 1 to “num” before returning the resultant array “numbers”.

c) linearSearch(int[] list, int v) : conducts a linear search comparing each value of list with v to determine if v is present in list. Returns the value of the index where v is found if it is found and returns -1 otherwise.

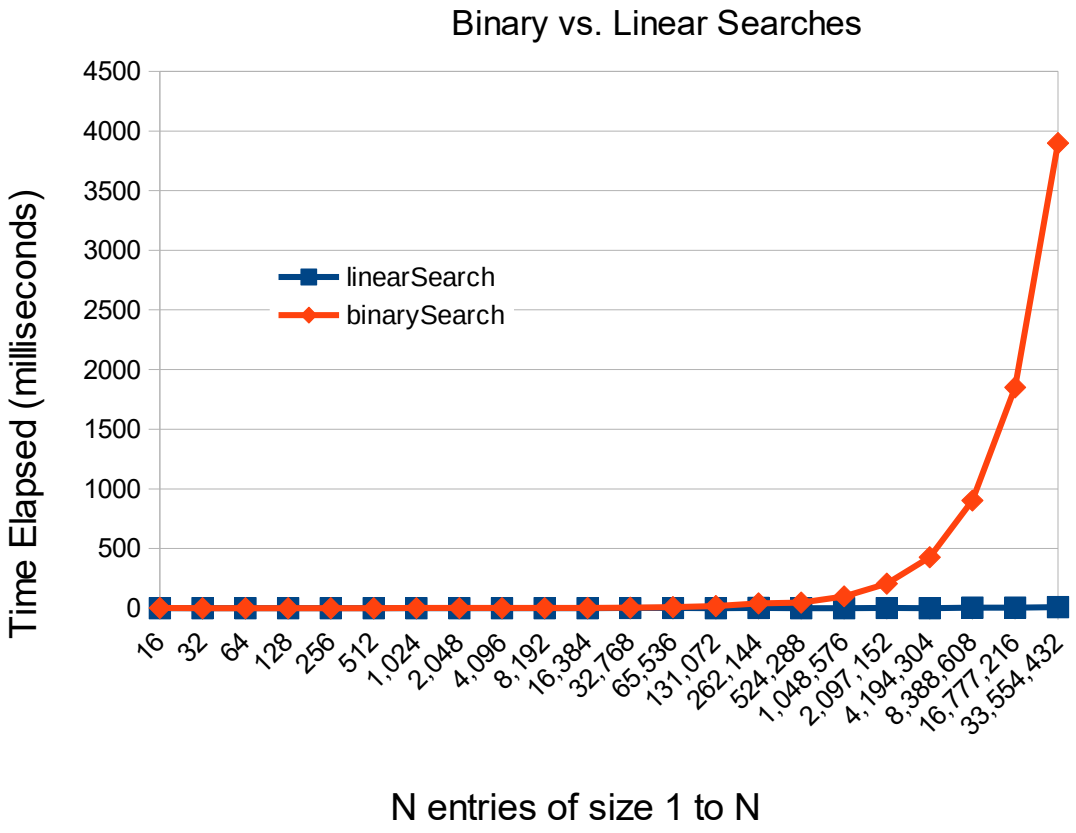
c) binarySearch(int[] list, int v) : conducts a recursive binary search comparing the middle value of list with v to determine if v is present in list. If v does not match the middle element of the list, the method will make a recursive call on either the left or right half of list depending on whether v is greater or smaller than the middle value currently being compared. This process continues until either the index of the found value v is returned or returns -1 otherwise.

3. **Testing Plan**

Testing this code is fairly straightforward. Upon running the program, a random value is picked from the searchKeys array which was generated from the input file and that singular value is then searched for in the randomly generated lists first of size 16 and then 32 and so on. All of the associated information, which can allow us to draw conclusions about differences in these two sorts and their run times is easily displayed in the standard eclipse output window. By comparing the running time of each search with the same randomly generated lists for each search algorithm, we can see the direct growth of time elapsed for each search as it differs over differing sizes of N. What follows is the output of the selected search value 21 from the list of 1,000 possible values mentioned before.

4. **Test Cases**

Search Results and Times for the value 21					
Input Size		**Linear Search**		**Binary Search**	
2^n where n =	**Input Array Size**	**Value found at Index :**	**Time (milliseconds)**	**Value found at Index :**	**Time (milliseconds)**
4	**16**	Not here!	0	Not here!	2
5	**32**	Not here!	0	Not here!	0
6	**64**	13	0	15	0
7	**128**	7	0	15	0
8	**256**	71	0	21	0
9	**512**	Not here!	0	Not here!	0
10	**1,024**	Not here!	0	Not here!	1
11	**2,048**	Not here!	0	Not here!	1
12	**4,096**	2,467	0	17	1
13	**8,192**	Not here!	0	Not here!	1
14	**16,384**	5,705	0	10	3
15	**32,768**	Not here!	1	Not here!	5
16	**65,536**	1,985	1	17	8
17	**131,072**	50,425	0	20	18
18	**262,144**	Not here!	2	Not here!	41
19	**524,288**	239,563	0	23	46
20	**1,048,576**	60,521	0	25	99
21	**2,097,152**	Not here!	1	Not here!	205
22	**4,194,304**	203,495	0	15	426
23	**8,388,608**	8,271,538	4	15	902
24	**16,777,216**	12,146,697	5	29	1,851
25	**33,554,432**	27,933,549	9	19	3,898



5. **Analysis and Conclusions**

Given that on average and worst case, binary search is projected to take O(log n) time and that in those same cases, linear search would be O(n) time, one would expect that linear search would show more growth than binary search in the direct comparison of physical run times. However, in this case, I believe additional time is accounted for in the binary search run time due to requiring the list to be first sorted and including that in the run time of binary search in general. Hence, as is shown in the graph, the binary search plus preliminary sorting shows considerable growth when compared to the negligible growth of time elapsed for larger values of N using the linear search alone.

6. **References**

The majority of code snippet examples I used in my code come from my experience as a programmer (looking back to reference code I've written before). I got the idea to use milliseconds comparison for time analysis from Stack Overflow. I referenced the book for a refesher on the linear and binary searches by definition. Finally, I looked up a more concrete binary search algorithm here: http://www.geeksforgeeks.org/binary-search/ I made my own implementation in Java, but clearly it is pretty similar to theirs.