Computing how Average Run Time Grows with Respect to Input Size Using Linear vs. Binary Search

Homework #1

By
Michael Harden

CS 303 Algorithms and Data Structures
August 26, 2021

1. Problem Specification

This assignment was designed to measure and compare how computation time grows in correlation to the size of an input array for the searching algorithms Linear Search and Binary Search.

2. Program Design

The program contains two classes, a Search class, which provides the methods for searching through an array, and SearchDriver, which implements the search class. The driver class tests the correctness of the Search classes algorithms, and it calculates the run times of each algorithm.

The SearchDriver uses the following steps to calculate the runtime of both algorithms:

a) A while loop is initiated, which runs until an index, starting at 16, is greater than 2^25, taking steps in multiples of 2.
b) An instance of the Search class is initialized with an array that matches the size of the index
c) The array is populated with numbers ranging from 1 to the value of the index.
d) A for loop runs, searching the array for each key in a list of keys using linear search. These keys are read from a file passed through read_keys()
e) Before each search the start time is stored, and after the search is complete the end time is stored. The difference of the times is taken and added to the total time, for that iteration of the while loop.
f) After all keys have been searched the average search time is calculated and stored as the final average time for that array size.
g) For binary search, the array must be sorted. This time is calculated and accounted for in the total binary search time, for that pass of the while loop.
h) Steps e and f are repeated for binary search.
i) After this the index steps, and the next iteration of the while loop runs until the index reaches its limit.

The following methods and constructors were created in the Search class:

a) __init__(data:list[int[])

    Constructs an instance of the search class and sets the data class variable to the array passed in

b) linear(key) -> int

    Searches through the data class variable linearly. If a value in data matches the key, the index of that value is returned, otherwise, -1 is returned. If multiple

instances of the key are in the data, the index of the first instance found is returned.

c) binary(key) -> int
Calls the recursive helper method __binary_assist() and returns its output.

d) __binary_assist(left, right, key) -> int
Left is the left most possible index where the key could be found, Right is the right most possible index where key could be found. The first base cases is where left is greater than right and the array has been searched through with no match found. When this case is met -1 is returned. The second base case is where the key is equal to the value of the index halfway between the left and right pointers. In this case the index of the value is returned. The middle value is calculated by taking the sum of the left and right pointer and dividing by 2. If the value at the middle index is greater than the key, the function __binary_assist() is called recursively. Left is passed in, and middle minus one is passed into right. If the value at the middle index is less than the key, the function __binary_assist() is called recursively. Middle plus one is passed into left, and right is passed in. If multiple instances of the key are in the data, the index of the first instance found is returned.

3. Testing

The SearchDriver class was used to test both searching algorithms. Values stored in the data, duplicate values in data, and values not in data were all tested for. The binary search algorithm was further tested where both left and right pointer are on the same index, and arrays with only one value were tested.

4. Test Cases

Linear Search Tests:

Test Number	Key Input	Array Input	Expected Output
1	12	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	2
2	91	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	9
3	62	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	3
4	15	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	-1
5	4	[4]	0

Binary Search Tests:

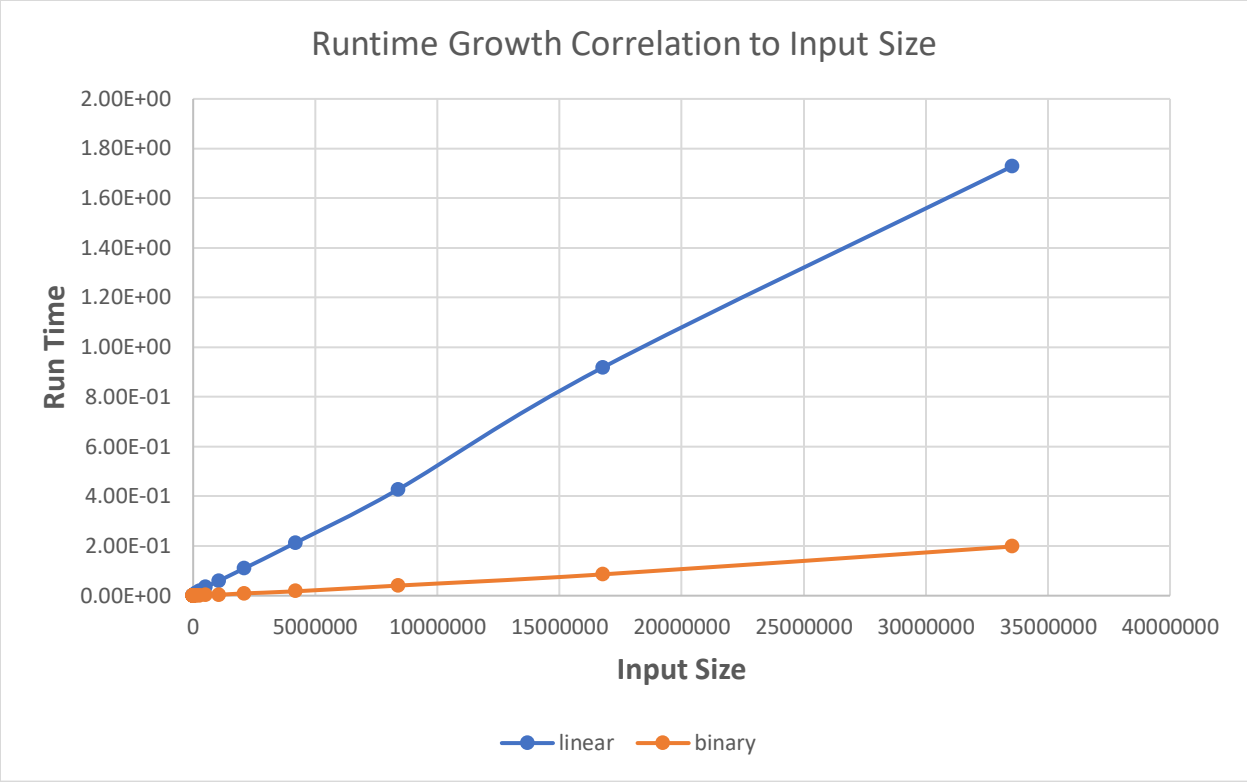
Test Number	Key Value	Array Input	Expected Output
1	12	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	1

2	91	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	9
3	62	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	5
4	15	[5, 22, 12, 62, 50, 35, 62, 72, 85, 91]	-1
5	4	[1, 2, 3, 4, 5, 6, 7, 8, 9]	3
6	9	[1, 2, 3, 4, 5, 6, 7, 8, 9]	8
7	1	[1, 2, 3, 4, 5, 6, 7, 8, 9]	0
8	19	[19]	0
9	11	[19]	-1

5. Analysis and Conclusions

The times recorded from the test shows how binary search is the optimal choice when making many searches. Even with unsorted data, the time to sort is amortized across all binary searched, and binary search has better performance. As shown in the chart below.

A case where this would not stand true is where the data must be sorted for every search. In this case, the time it takes to sort the data would cause a binary search to be sub-optimal, and a linear search would be the winner.



The times recorded for input size are listed below:

Input Size	Linear Search Time (seconds)	Binary Search Time (seconds
16	1.72E-06	2.36E-06
32	2.79E-06	2.58E-06
64	5.01E-06	2.89E-06
128	9.07E-06	3.47E-06
256	1.72E-05	3.74E-06
512	3.81E-05	4.71E-06
1024	8.26E-05	6.16E-06
2048	0.00015363	7.59E-06
2^{12}	0.00029942	1.18E-05
2^{13}	0.0006353	1.55E-05
2^{14}	0.00124869	2.75E-05
2^{15}	0.00249527	5.91E-05
2^{16}	0.00494612	0.00011026
2^{17}	0.00992693	0.0002526
2^{18}	0.01878449	0.00062605
2^{19}	0.03491435	0.00140521
2^{20}	0.05968933	0.00328957
2^{21}	0.11015326	0.00931462
2^{22}	0.21321706	0.01755525

2^{23}	0.4276244	0.04057694
2^{24}	0.91888989	0.08579071
2^{25}	1.72936226	0.19803157

6. References

The format of this report is based on is the format provided to us in the files section of Algorithms and Data Structures CS 303