CAB301 Assignment 1

Average Case Efficiency Analysis of a Binary Search Algorithm

Student name: Thomas Feldman

(Student no. n8306699)

Date submitted: 15 April 2018

# Summary

The following report will analyse the average-case efficiency of a given algorithm and test it experimentally. Firstly, I will explain what a Binary Search algorithm is and how it works, while identifying the basic operation. Doing so I will show the theoretical order of growth, before moving on to experimental testing to determine whether the implementation introduces additions variables not accounted for. By the end of this paper, you should be able to see the trend that occur within the data, and how closely the data matches theory.

# 1. Binary Search Algorithm

The Binary Search algorithm is a search algorithm that finds the key value within a sorted array. It does so by initially comparing the key value with the middle index position of a given array. If the values are not equal, the algorithm will determine which half the value cannot be in and discards it. The algorithm will then reoccur on the remaining half until the key value has been found. If the key value is not found in the remaining section, then the key value is not in the array.

# 2. Theoretical Analysis of the Algorithm

# 2.1 Basic Operation

The binary search algorithm has a very evident choice of basic operation. As the search algorithm initially compares the key value to the middle value of the given array, and will loop around this comparison, this is the most appropriate choice of basic operation. Furthermore, we can clearly see it is the most performed operation in the algorithm, providing further proof that this is the valid choice. This comparison is the determining factor for a successful or failed search, as ultimately this will be the last calculation performed on the final section of the array before either finding; or failing to find the key value.

## 2.2 Average Case Efficiency

Although not specifically needed, it is best to quickly assess the best and worst-case efficiency of the binary search algorithm, as they will prove to be useful in the proof of the theoretical average case efficiency. The best-case efficiency is quite clear when you look at the algorithm itself; as it starts with a simple key comparison to the middle value, it is evident that if these values are equal, then the algorithm stops. Therefore, the best-case efficiency of a binary search algorithm is O (1) comparisons.

In Levitin’s textbook [1, pg.151-152], he shows proof of worst-case efficiency being O(log n). I will also assume this to be true, as this is the expected answer given that the algorithm simply reduces the array by half every iteration. Therefore, the number of iterations needed must be roughly log2n. Levitin also goes on to say that the average case for a number of K comparisons made is only slightly smaller than worst-case, however he does not show proof.

As we are interested in the proof that the average case efficiency is roughly equal to the worst-case efficiency, I will use an example from the data used to implement the algorithm.

Take this 16-index length array (now known as A). We are going to search for the Key Value = 5 (now represented as K).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

In the array above, the middle index (A[m]) is equal to 16/2= 8. Therefore, the middle value = 7 (now known as M). As K = 5 and M = 7, the comparison is not equal, and the rest of the algorithm will continue. Since we have read the algorithms description, we know that the algorithm will remove the half that K cannot be in. Leaving us with:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

(It is also important to note that this is the array after 1 basic operation)

Now the search reoccurs, giving us A[m] = 8/2 = 4. Therefore, M = 3, which is not equal to K.

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 | 6 | 7 |

Repeating the process for finding the middle element for every sub-array

|  |  |
| --- | --- |
| 4 | 5 |

A[m] = 2

|  |
| --- |
| 5 |

A[m] = 1

You can see a pattern that occurs during the algorithm; after every comparison with A[m], our range is divided into half of the current range. So, we can say that to find K in the array of 16 elements, we had to perform the basic operation 4 times.

Therefore, we can say that: 16 \* (1/2)4 = 1. So, for n elements, the equation becomes: n \* (1/2)k = 1.

Expanding this equation: n \* (1 / 2k) = 1;

2k \* (n / 2k) = 2k

n = 2k

Now we look to the definition of a logarithm: “a quantity representing the power to which a fixed number (the base) must be raised to produce a given number.” If we apply this to our equation, we get: log2 n = k. As this is equal to what Levitin suggests the theoretical average case efficiency is, we can safely assume that this proof is correct.

# 3. Methodology, Tools, and Techniques

1. Experiments run on a desktop PC, running an AMD 8 core fx-8350, with Microsoft Windows 10 operating system. All algorithms and experiments were implemented using C# multi-paradigm programming language, and Microsoft Visual Studio Community 2017. Visual Studio Community is a free, fully featured and widely used IDE. One of the biggest advantages of using Visual Studio is its ItelliSense.

2. All data was test using a sequentially generated array and selecting a random key value between 0 and max index range. The test size per array length was 100. Array length started at 1 and incremented in such a way that they were either 1 \* 10x or 5 \* 10x. The list is 1, 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000. This was to generate enough results per array length, and to see the effects of a larger array size.

# 4. Algorithm Implementation

Using System.Linq.Enumerable, I created a sequential array, ranging from start point 0, to a passed integer n (see figure 4 line 80). The method int[] createArray (int n) would take a passed int value for n (see list in section 3) and using Enumerable.Range(0,n).ToArray() would generate and return the sequential array.

To gather necessary data, I used both a sufficient number of tests per array size, and several increasingly larger arrays. In testing, I found that if the program was to be only run once, the time for this would be much higher than if the test was run multiple times in concession. Adding this to the fluctuating results, I opted to test 100 times for an array of length n at one time, by looping the main method (see figure 6) and taking an average. This allowed me to gather much better results and was more convenient to do so as well.

To accurately time these tests, I used System.Diagnostics.Stopwatch to time all 100 tests and divided by 100 to gather the average. In figure 6, it can be seen that I have started the stopwatch immediately prior to the search method call, and stopping immediately after, I was able to gather accurate run times for all tests. Averaging all 100 tests gave me millisecond results to 17 decimal places, giving incredibly accurate results.

# 5. Counting Basic Operations

1. Adding a simple variable to the algorithm allowed me to accurately track the number of basic operations (see figure 5 line 99). Adding the basic operation variable here is correct, as the algorithm must do the basic operation to determine if the key value exists, even if there is only 1 index remaining and it is not the key value. Therefore, we can assume this is correct, as the results of experimentation also prove.
2. Using the same method explained in section 4 to measure execution times, I also simultaneously counted the average basic operations performed by the algorithm. Doing so eliminates the variable of multiple sample key values, thus attributing to the validity of the results.
3. Figure n shows this relationship between array size and average basic operations count. Contributing to the line of the graph are 13 array sizes with their corresponding average basic operations count, however; for each of these 13 data points, 100 tests were conducted to gather a fair average basic operations count. Thus, the true size of each of these data points is really 13\*100 = 1300 tests.
4. As you will be able to see in figure n1, much like the log2 n table in figure 3, when the array size grows, it’s expected that the basic operations count will also rise. With most of the values being within 1-2, which can be attributed to my use of integers and not doubles, causing some values to be rounded down as the program added them, it is safe to assume that the trend grows as intended. The other variances can be boiled down to the random key value; as log2 n is a purely logarithmic relation, it would have been highly unlike for my results to exactly replicate these, as it is expected that average-case efficiency, according to Levitin’s theory [1, p.g.151], Cyesavg(n) ≈ log2 n − 1 and Cnoavg(n) ≈ log2(n + 1) (where yesavg is a successful search and noavg = failed search as both fall within the bound of my testing with random key values). Therefore, I believe this testing adequately shows that the binary search algorithm works, and my testing has been sufficient.

# 6. Average Execution Time

1. Using the same method used in section 4 of this report, I used System.Diagnostics.Stopwatch to time all 100 tests and divided by 100 to gather the average execution time as you will see in figure 6.

2. Using the same method as previously stated, I ran 100 tests on each array size, timing each test, and finding an average. Doing so gave millisecond values to 17 decimal places, thus it was fair to assume these were as accurate as possible.

3. As stated in section 5, Figure 2 shows this relationship between array size and average execution time. Contributing to the line of the graph are 13 array sizes with their corresponding average execution time, however; for each of these 13 data points, 100 tests were conducted to gather a fair average execution time. Thus, the true size of each of these data points is really 13\*100 = 1300 tests.

4. While the results show a somewhat fluctuant growth, the overall trend shows that when the array size grows, the runtime will also show slight growth. As the differences between them are all very small, we can assume that the graphed data does in fact behave as explained in section 2. All variance I did come across could potential be attributed to the age of my pc, and potentially different results may have been seen dependant on operating conditions. However, I stand by my assumptions that small variances in the run time as the array size grows, matches our expected O(log n) growth.

# References

[1] Levitin, Anany. Pearson, 2012. “*Introduction to the design & analysis of algorithms, 3rd ed*.”

[2] Hacker Noon. (2018). “*What does the time complexity O(log n) actually mean?”.* [online] Available at: https://hackernoon.com/what-does-the-time-complexity-o-log-n-actually-mean-45f94bb5bfbf [Accessed 28 May 2017].

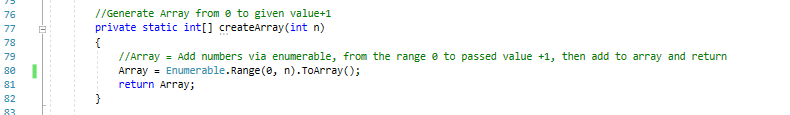
# Figures

(Figure 1: average basic operations vs array size of n length)

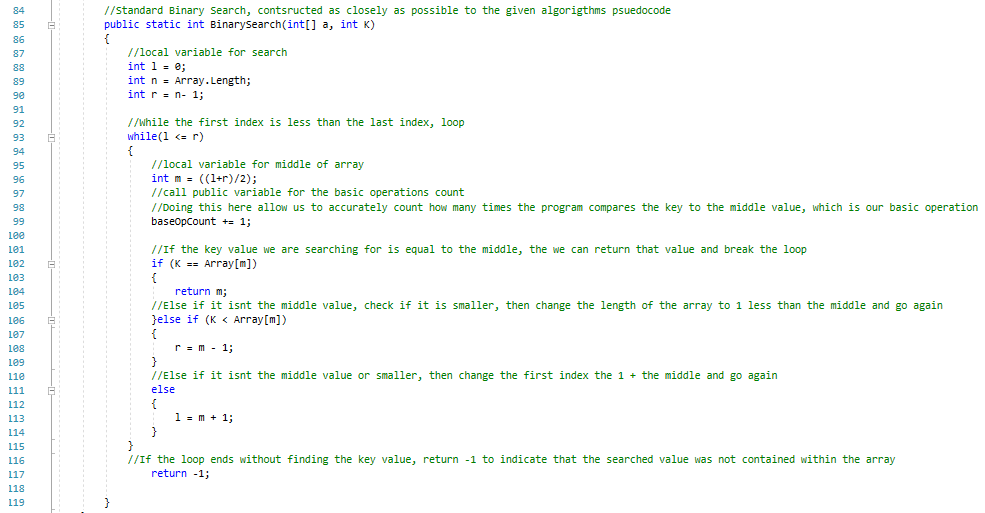
(Figure 2: Average execution time vs array size of n length)

Figure 3: Logarithm Table

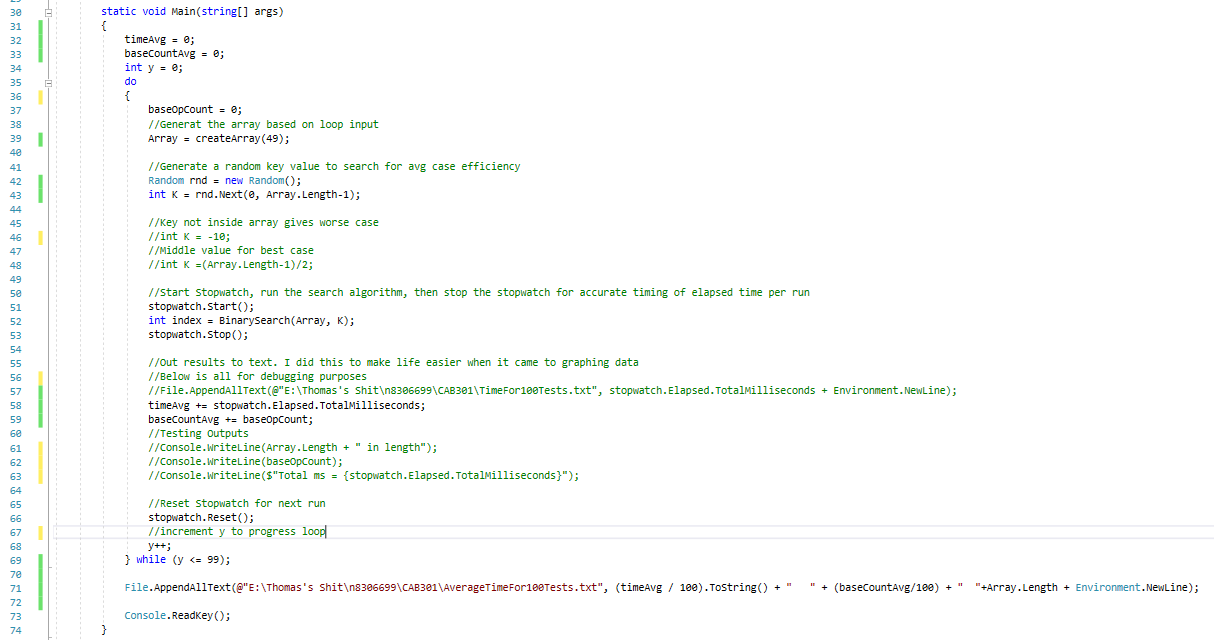
|  |  |
| --- | --- |
| *n* | log2*n* |
| 0 | undefined |
| 1 | 0 |
| 2 | 1 |
| 3 | 1.584963 |
| 4 | 2 |
| 5 | 2.321928 |
| 6 | 2.584963 |
| 7 | 2.807355 |
| 8 | 3 |
| 9 | 3.169925 |
| 10 | 3.321928 |
| 20 | 4.321928 |
| 30 | 4.906891 |
| 40 | 5.321928 |
| 50 | 5.643856 |
| 60 | 5.906991 |
| 70 | 6.129283 |
| 80 | 6.321928 |
| 90 | 6.491853 |
| 100 | 6.643856 |
| 200 | 7.643856 |
| 300 | 8.228819 |
| 400 | 8.643856 |
| 500 | 8.965784 |
| 600 | 9.228819 |
| 700 | 9.451211 |
| 800 | 9.643856 |
| 900 | 9.813781 |
| 1000 | 9.965784 |
| 10000 | 13.287712 |



(Figure 4: Code for generating the array used in testing)



(Figure 5: Binary Search Algorithm, implemented as closely to given pseudocode algorithm)

(Figure 6: Main)



(Figure 7: Given pseudocode algorithm)