CAB301 Algorithms and Complexity

Assignment 1

Empirical Analysis of an Algorithm

Due: 15th April 2018

Student: Matthew Pini

Student Number: n9700498

**Summary**

This report observes the fourth Algorithm, the Binary Search and analyses its average case efficiency in relation to Levitin’s claims. The Binary Search was implemented through the C#...

**Description of the Algorithm**

The Binary Search Algorithm implements a non-recursive search through a sorted array and returns either the index of the first found search or a “-1’ if they key was not found. Due to the type of search this is, the array given is **required** to be sorted (in this case, ascending order) in a way which complements the search method of the given Binary Search. The Binary Search accomplishes its search by first splitting its input array in half

The Binary Search Algorithm implements a non-recursive search through an array which has already been sorted into ascending order and returns either the index of the first found search or a “-1” if the key was not found.

**Theoretical Analysis of the Algorithm**

This section describes the Binary Search Algorithms time efficiency in terms of its average case efficiencies.

**Identifying the Algorithms Basic Operation**

Identifying the basic operation of the given Binary Search is extremely simple as there is only a single operation being performed every time for all end results of the search. The “if” statement where the binary search first compares the given key to the middle position of the array is by the far most important and commonly completed piece of the algorithm. This is due to this section being the starting point and the ‘cross roads’ for the rest of the search. Due to this sections massive impact of the rest of the search, it is appropriate to make this the algorithms basic operation.

**Average-Case Efficiency**

Unlike other Algorithms, the average case efficiency for a non-recursive binary search is slightly different. According to Levitin, the average-case efficiency is split between a successful and unsuccessful result which is also, not much better than its worst case scenario of not finding anything. Due to this, it could be assumed that a non-recursive binary search doesn’t have a singular average-case but rather an average-case (Successful average-case efficiency) and a worst-case (Unsuccessful average-case efficiency).

**Successful Average-Case Efficiency**

A Successful average-case would be in the case of actually finding the key within the given array.

**Unsuccessful Average-Case Efficiency**

An Unsuccessful average-case would be in the situation that the key could not be found within the given array. This could also be considered the worst-case scenario for the binary search.

**Methodology, Tools and Techniques**

This section overviews the computing environment with which the code was written and tests occurred in.

1. The Binary Search was written in C# due to its general and widespread knowledge along with its ease of access and ability to create simple, easy to understand code.
2. Instead of creating a separate array ordering code, the use of C#’s Array.Sort() function was used to sort the random arrays for testing.
3. All Basic Operation experiments where completed on a custom build computer. Its specifications consists of a 64 bit Intel i7-6700k 4.00ghz CPU with 4 cores and 8 logical processors along with 32gb of RAM (installed physical memory) also running an MSI 970 GPU. All non-essential programs where closed for the duration of these tests.
4. All graphs where created using Microsoft Excel. This is due to its ability to keep logged data then converting it into a clean and easy to read graph.

**References**

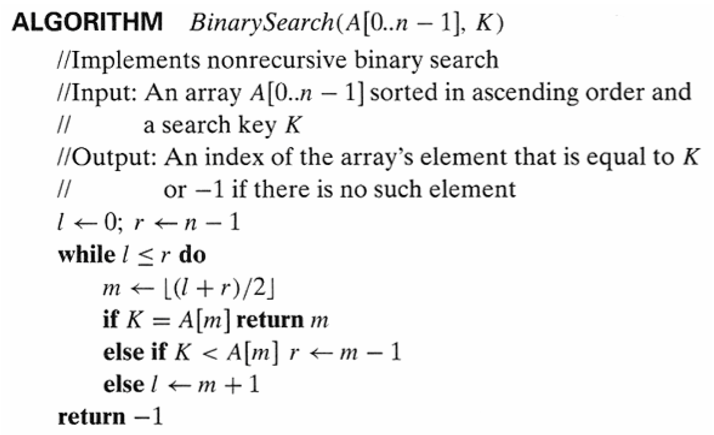


Figure 1: Non-recursive Binary Search. A is the Sorted array to be input (Must be in ascending order). n – 1 is the length of the array. K is the key you wish to search for within the given array. l is the lower point in the array while r is the upper point in the array; these determine which section of the array we are searching in. m is the point in the array we are comparing the key to.

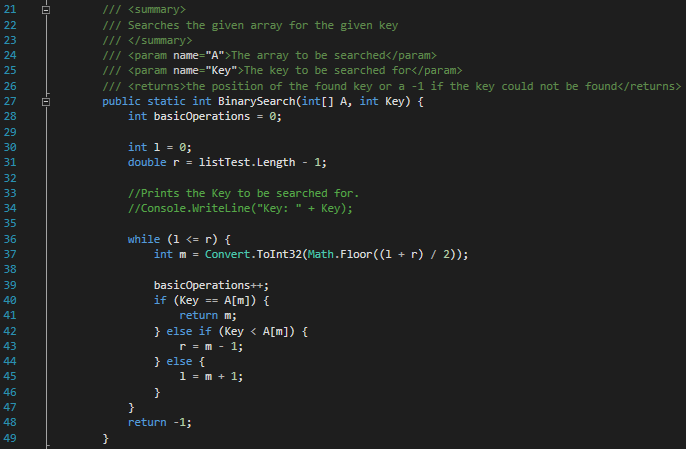


Figure 2: Implemented Non-recursive Binary Search in c#. The only difference in comparison to Figure 1 is that the “Key” input refers to K and this can be changed to output the amount of basic operations instead of if the key has been found or not. There are several notable changes to this code in comparison to the given Algorithm which tries to make it as streamline and as similar to the given code.

The first change is the addition of the ‘basicOperations’ integer and the ‘basicOperations++;’ line which is used to measure the amount of basic operations performed. These lines of code will be blocked out during any testing which does to require said information to allow for as little interference as possible.

Secondly due to the way C# Math.Floor works, it is required to be converted to an integer to be able to interact with the rest of the algorithm without having to change the rest. It was decided that this would be the best way to keep changes to a minimum.

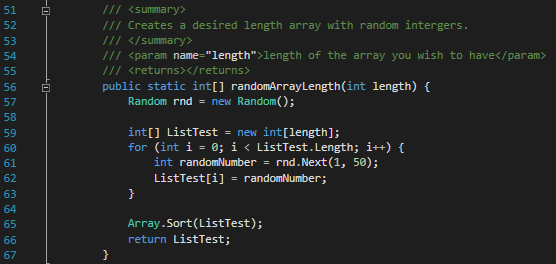


Figure 3: Random array of a specified length generator. This is used during the testing of the Binary Search. It inputs a specified length and outputs a random integer between 1 and 50 for each index in an array and outputs the created array.

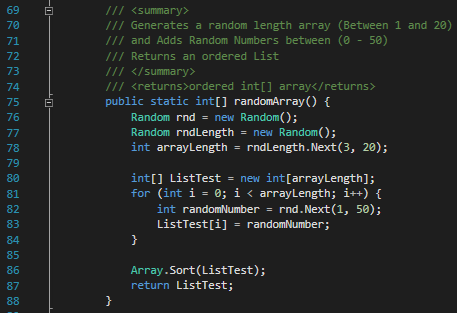


Figure 4: Random length random array generator. Creates an array of random length (Between 3 and 20) and fills with random values between 1 and 50, then outputs the generated array.

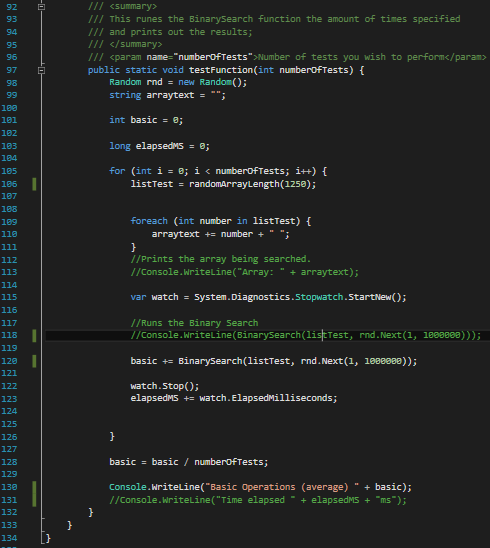


Figure 5: Test function. This function is made to test the binary search array and measure its execution time while ignoring the time taken to randomly create the arrays. This is to allow the test to resemble the given algorithm as closely as possible. Please be aware that due to C#’s use of its Stopwatch function (line 115), there might be small outliers in regards to the test.

This test functions main role is to allow for the test of ONLY the BinarySearch function which is run on line 118 and 120 (Depending on which test is run) to be run separately from the random generation of the arrays (line 106). This TestFunction keeps track of both the basic operations (line 101) and the elapsed time (line 103) so that it can be used to test both the average number of basic operations (figure 6) and the average time (figure 8)

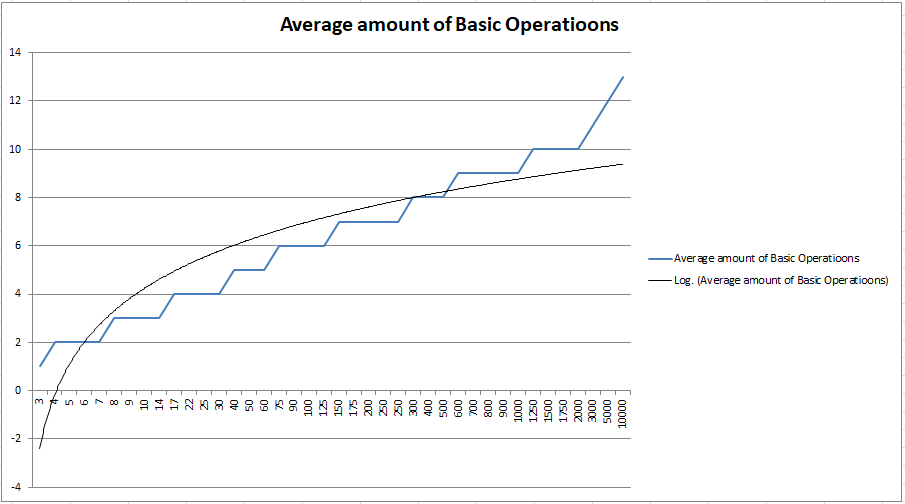


Figure 6: Line graph showing the increase of the Algorithms average Basic operation in relation to the length of the array searched. This graph shows the actual input data shown by the blow line and then its logarithmic trend line shown in black.

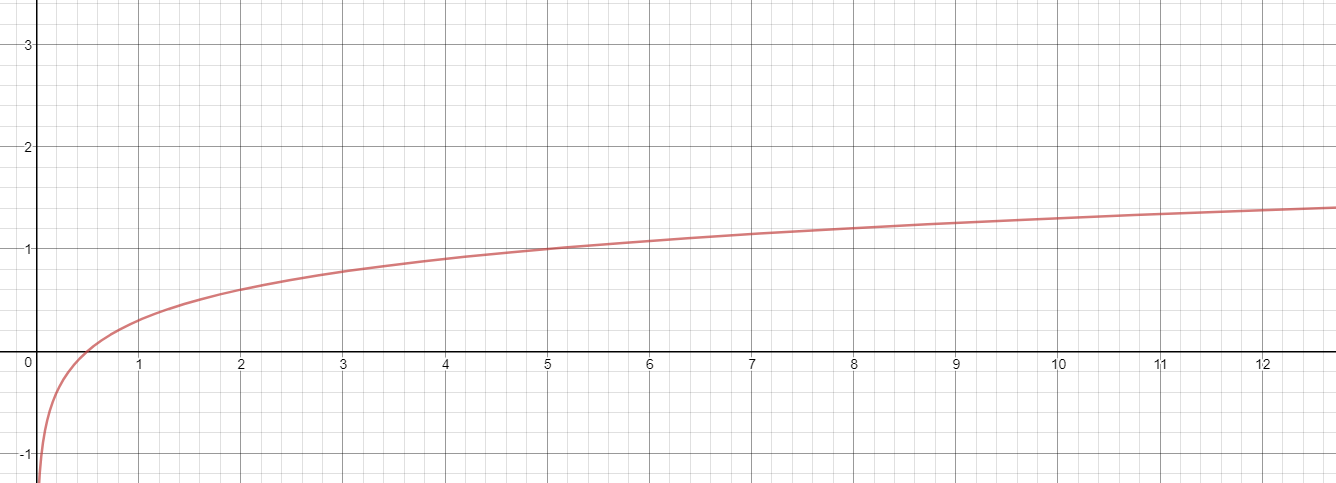


Figure 7: Line graph of Levitin’s approximate average-case for the Binary Search.