Student Number: 10134030 Page 1

Analysing Algorithm Complexity Through Proof Theory and Execution

**Abstract**

*Purpose*: Analyse complexity proofs while comparing the growth complexity between the linear and the binary search algorithms.

*Methods*: Linear and binary search algorithms were implemented in attempt to find a varying number of target integers within different sized lists; nonetheless, half the integers to be searched were purposefully absent to maximise the effects of incurring the worst-case scenario.

*Results*: When the number of integers to search was less than 250, the linear search provided marginal computational benefits. However, the binary search possessed a lower growth rate which resulted in its imminent outperformance of the linear search as k grew.

*Conclusions*: The binary search proves to be a more effective algorithm when approached with a growing data set. Although the linear search provides marginal benefits in speed to begin, the slow performance associated with handling large data sets do not justify its implementation. Word Count: 142

**Introduction**

The objective was to identify the number of target variables to be searched where the binary search algorithm began to outperform the linear search in runtime.

The analysis consisted of two parts: theoretical and application. First, the theoretical section was aimed at solving proofs consisting of Big-O, Big-Ω, and Big-Θ complexity principals. (The theoretical section will be included in another document within the zip file.) The notations are used to classify algorithms based on their performance as the input size becomes larger.[[1]](#endnote-1) Each notation mentioned above bounds the programs complexity differently. Big-O bounds the function from above forming its asymptotic upper bound; therefore, there exists a no in which all the values of some c \* g(n) are greater than f(n).[[2]](#endnote-2) Big- Θ bounds the function from above and below. Finally, Ω-notation provides the asymptotic lower bound contrary to Big-O; therefore, there exists a no where all the values of some c \* g(n) are less than f(n).[[3]](#endnote-3)

The ‘application’ component consisted of implementing the linear search and binary search to analyse the algorithms’ performance when handling a growing dataset. The brute force search is the simplistic searching algorithm to exist. The search entails starting from the first element and looking at every subsequent element until the desired target is found.[[4]](#endnote-4) The binary search requires somewhat more complexity. It requires the list to be sorted prior to being searched. This process will be further unpacked in the methods section; however, the intuition behind this algorithm is to complete additional work early by sorting the list to avoid unnecessary computation in the future when one is required to search the list.

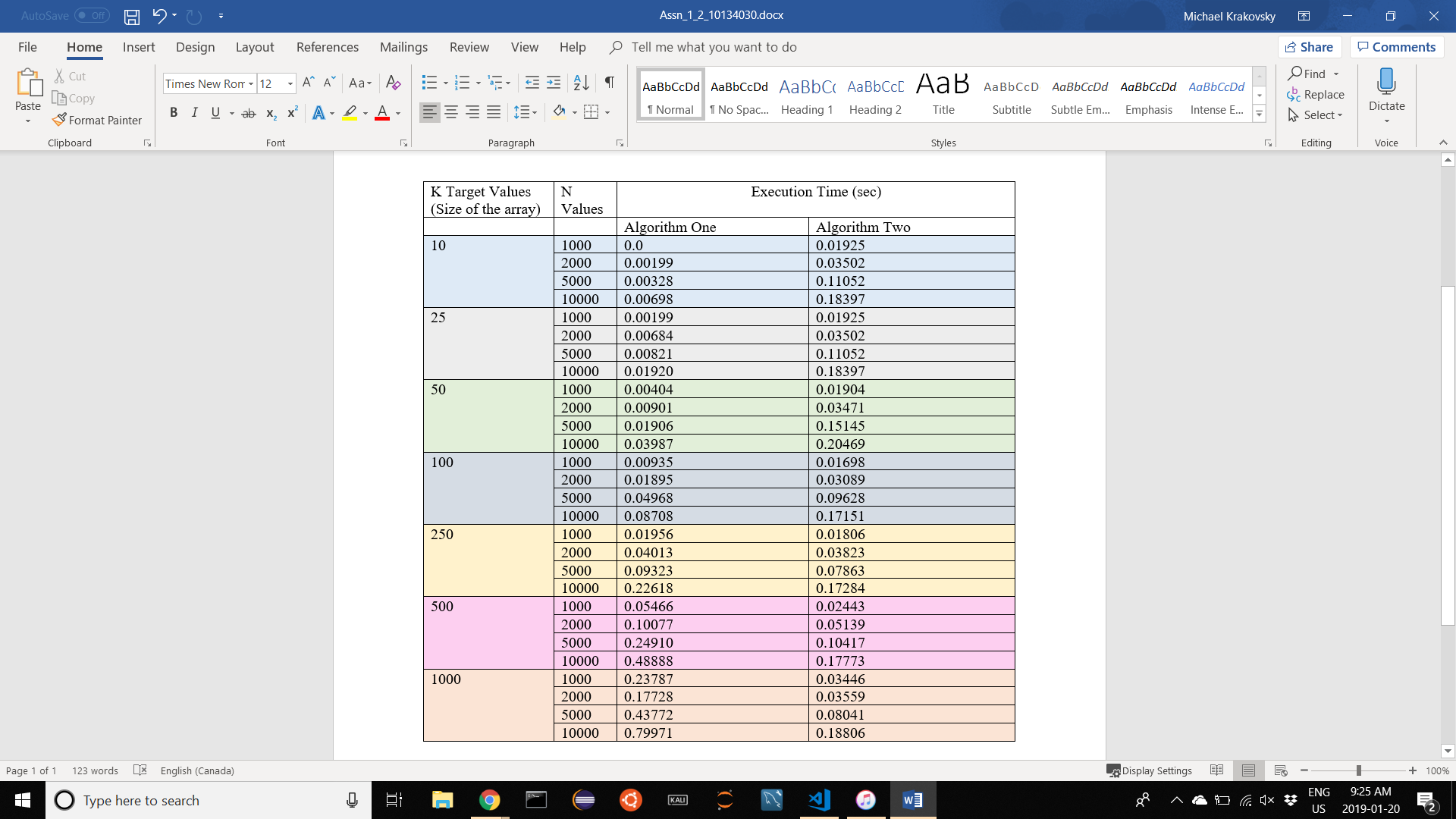
The scientific question was to determine where the number of comparisons in a linear search surpassed that within the binary search. The testable predictions will be applied to seven different data sets which varies the number of target values to search within the list and the number of values that are required to be searched.

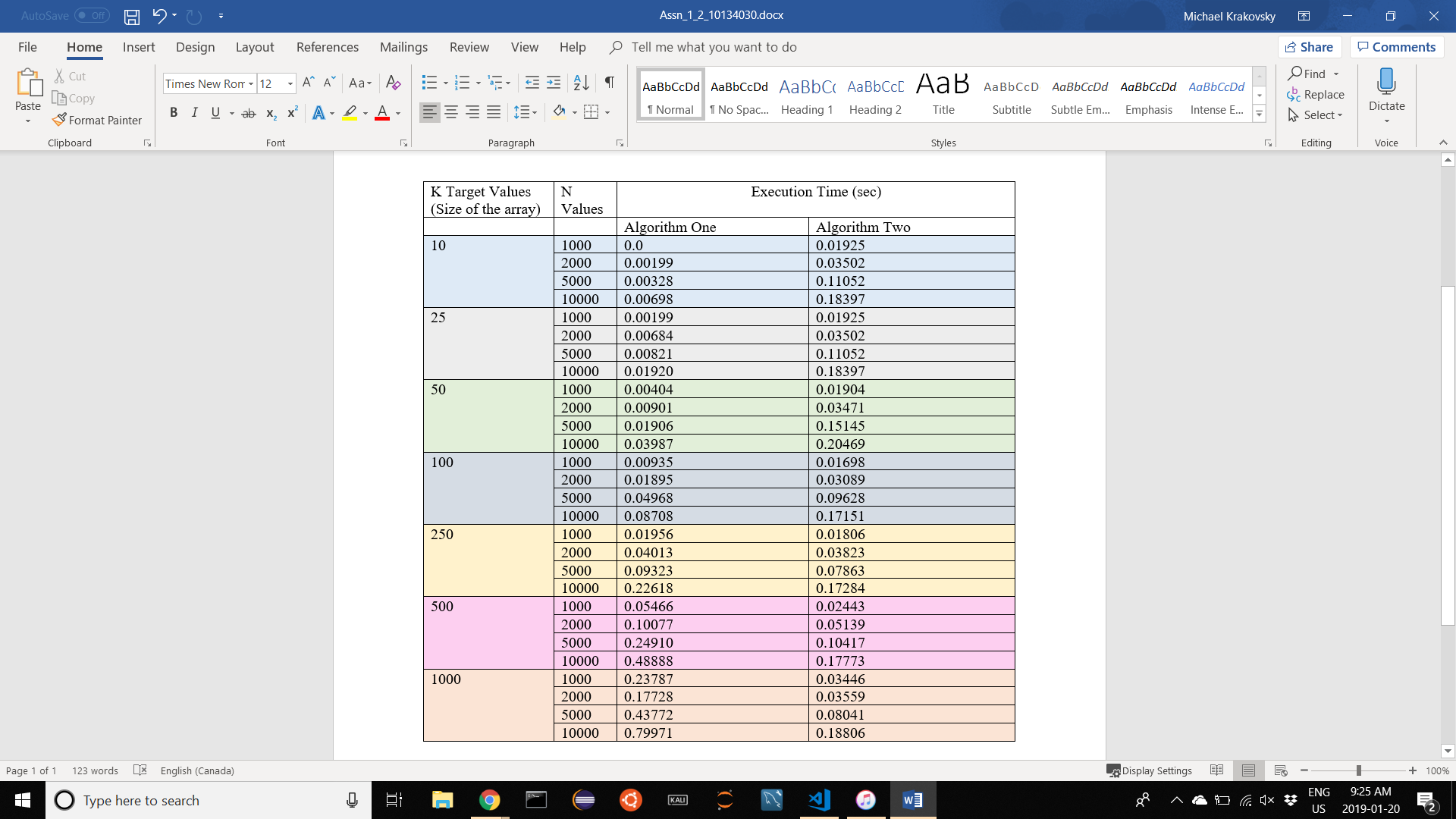
**Methods**

The program was split into three distinguishable sections: environment creation, sorting and searching, and testing. In addition, helper functions were created to enhance the code’s readability. There are two arrays that were pivotal to implementing the algorithm. First, a list with random numbers ranging from 10 to 100’000 was created with a size of either 1000, 2000, 5000 or 10’000. Next, the target list, which contains the values that will be searched within the random value list, was produced. The list was composed of values that existed within list and values absent from the list. To accomplish this task, half the array was picked by choosing a random value within the array while the remaining values were chosen to be outside the random numbers range.

Once the test set was initiated, the algorithms were implemented. The first algorithm implemented was the brute force algorithm. The brute force algorithm was a simple ‘for’ loop that went through the entire list until the element was located. The second algorithm was composed of both the sorting and searching algorithm. The sorting algorithm utilised was the merge sort possessing an O(nlogn) complexity and was implemented recursively. A function was then created to first to sort the list prior to searching for the value. The sorting algorithm was only used with algorithm two since brute force searches do not require prior sorting.

The testing process was condensed into the main function. There are two parameters that the user can manipulate to vary the results. First, the parameter ‘numIters’ controls the number experiment iterations. Since the target values are random, it was necessary to iterate the experiment 500 times to produce consistency. Second, the parameter ‘k’ controls the list size which holds the values to be searched. The size of k began at 10 but was increased to display the growth complexity of both algorithms. After each iteration, the runtime was recorded in a dictionary where the keys indicate the size of the list that was be searched. Once the entire experiment was run, the time averages were recorded in the results section.

**Results**



To reiterate, K represents the number of values that ***will be*** attempted to be found within the list N, while N represents the total amount of values within the list ***to*** ***be*** searched. (i.e. The list where the algorithms search through for the target value.)

**Discussion**

The results from the experiment insinuates that the binary search and sort combination is quicker than the linear search as K grows. In addition, the binary search becomes the preferable algorithm as K approaches 250.

When K equalled 10, the linear searched outperformed the binary search; however, this was predictable since the binary search required each list of size N to be sorted prior to being searched. Therefore, the majority of algorithm’s two run time was caused by the sorting algorithm itself which is dependent on the size of N. Nonetheless, the linear function continued to grow linearly while algorithm two remained relatively constant because the size of N was kept constant. For instance, when K is doubled in the chart above, algorithm one’s execution time is also doubled. There does exist a variance in time which can be attributed to other processes running on the computer during the execution. It was not until when K equalled 250 in which the binary search outperformed the linear search. When we analyse the same section where N is doubled, the binary search possessed similar execution times because its complexity growth reached its asymptotic boundary. The asymptotic boundary was determined by the size of N which, as stated before, remained constant throughout the entire experiment.

The growing difference in run time was exacerbated by the linear search worst case scenario. If the value was not located in the list, the algorithm had to unnecessarily run through the entire loop only to be disappointed by its lack of success; however, the binary search had to perform no more than log2(N) searches for an absent value. For example, searching for an absent value in a list of size 10’000 using brute force would require 10’000 comparisons. Otherwise, if the same list was sorted, implementing binary search would only require 14 comparisons. Since the number of absent values to be searched is directly proportional to half the size of K, the sorting performed at the beginning of algorithm two rendered this algorithm attractive as K grew.

To conclude, algorithm two is the preferred method of choice since its growth complexity is more favourable than the brute search. Although the brute search is quicker to execute when K is small, the algorithm is rendered unfavourable after the realisation that K will grow.

**References**

1. T., Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction To Algorithms*(3rd ed., Vol. 1, p. 47). Massachusetts Institute of Technology. [↑](#endnote-ref-1)
2. T., Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction To Algorithms*(3rd ed., Vol. 1, p. 47). Massachusetts Institute of Technology. [↑](#endnote-ref-2)
3. T., Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction To Algorithms*(3rd ed., Vol. 1, p. 48). Massachusetts Institute of Technology. [↑](#endnote-ref-3)
4. T., Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction To Algorithms*(3rd ed., Vol. 1, p. 543). Massachusetts Institute of Technology. [↑](#endnote-ref-4)