An Optimised Binary Search Algorithm

Amannah, Constance Izuchukwu & Michael, Chief James Philip
Department of Computer Science
Ignatius Ajuru University of Education,
Rivers State,
Nigeria
Email: aftermymsc@yahoo.com

Abstract
The binary search algorithm, which is used in searching a linear collection of sorted items, has a fast runtime of O($\log n$), but incurs the overhead of performing a search even when the search key is in the non-feasible search space (i.e. outside the range of values in the list). This research presents an optimized Binary Search algorithm that ensures that search is performed if and only if the search key is within the feasible search space, thus exhibiting a faster time complexity than the Binary Search algorithm- in particular, when the search key is outside the feasible search space of the list. The waterfall design approach was chosen since this design is simple, and also the problem was well understood. The design was implemented using java programming language by compares Binary Search Algorithm. When the search key is outside the feasible search space i.e. searching a least for an element that is not in the feasible search space, therefore enabling search to be performed with reduced time thus ensuring search efficiency, and achieving the state of research objectives.

***Keywords**: Algorithm, Optimization, Array, Search Space, Feasible Search Space, Non-feasible Search Space, Complexity, Time Complexity.*

1. INTRODUCTION

One very common application for computers is storing and retrieving information. For example, the telephone stores information such as the names, addresses and phone numbers of its customers. When directory assistance needs to get phone number for someone, the operator must look up that particular piece of information from among all data that have been stored. Taken together, all of this information is one form of a database which is organized as a collection of records. As the amount of information to be stored and accessed becomes very large, the computer proves to be a useful tool to assist in this task. Over the years, as computers have been applied to these types of tasks, many techniques and algorithms have been developed to efficiently maintain and process information in databases. The processes of "looking up" a particular data record in a database is called searching, it is also the way to look for something in a list. Looking up a phone number, finding a website via a search engine and checking the definition of a word in a dictionary all involve searching large amounts of data (Deitel and Deitel, 2007).

In Computer Science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the root of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph. In order to do an efficient search in a database, the records (items) should be maintained in some order-mostly referred to as data structure. A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. A data structure provides a means to manage large amount of data efficiently, such as large databases and internet indexing services. Types of data structures include: Array, container, list, set, queue, deque, stack, string, tree, graph, array dictionary, and so on. Depending on the type of data structure employed and the type of application, several searching algorithms exist. These include: linear search, binary search, binary tree search, breath-first search, depth-first search, heuristic search, best-first search, and so on.

In a linear data structure, searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding its location. Two popular search algorithms are the simple linear search and the faster but more complex binary search (Deitel and Deitel, 2007). Linear search is the simplest search algorithm that is applied in finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found (Knuth, 1997). Linear search algorithm is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost if all list elements are equally likely to be searched for. Therefore, they also impose additional requirements. Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list (www.wikipedia.org). Binary (half-interval) search algorithm finds the location or index of an item in a sorted set of data elements by comparing the key to a designated middle element; and if not equivalent, repeatedly constraining the middle element comparison to the smaller relevant half of the set until a match is obtained, or the list is exhausted of which an invalid index will be returned to signify it found no match with the key. The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted. The first iteration of this algorithm tests the middle element in the array, if this matches the search key, the algorithm ends. Assuming the array is sorted in ascending order, and then if the search key is less than the middle element, it cannot match any element in the second half of the array and algorithm continues with only the first half of the array (i.e. the element up to, but not including the middle elements). If the search key is greater than the middle element, it cannot match any element in the first half of the array and the algorithm continues with only the second half of the array (i.e. the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half the remaining elements. The algorithm ends by either finding an element that matches the search key or reducing the sub-array to zero size (Deitel and Deitel 2007). While linear search scans each array element sequentially, a binary search in contrast is a dichotomy divide and conquer search algorithm (Knuth, 1997).

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it does not require the data in the array to be stored in any particular order. Its disadvantage however is its inefficiency. When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the linear search should not be used on large arrays if the speed is important (Gaddis, Walters, and Muganda, 2013). The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted (Deitel and Deitel, 2007). However, according to (Asagba, Osaghae, and Ogheneovo, 2010), existing literatures never consider the storage time incurred in the binary searching.

Statement of the Problem

The binary search exhibits a complexity of O(logn) logarithmic runtime. O(logn) runtime is among the fastest running time for conventional algorithm. In spite the runtime significance of the binary search algorithm, it is constrained with non-essential search space and redundant search space options. This is traced to the decimation approach employed in the binary search algorithm. This paper is therefore designed to optimized conventional binary search algorithm taking into account the space and redundant drawbacks of the existing algorithm. The sequential iterated methodology is an approach that supports the optimization process of the proposed algorithm.

Specific Objectives of the Study

The aim of this study is to develop an optimized binary search algorithm that complements the conventional binary search algorithm. In order to achieve this aim, the following objectives were considered:

- Evaluation of the template of the binary search algorithm
- Evaluation of the space bottleneck of the binary search algorithm
- Evaluation of the redundant search options of the binary search algorithm
- Determination of a model framework that optimizes the redundant options.
- Determination of a model framework that optimizes the space constraint.

Scope of the Study

This research covered searching of data in a linear collection of data elements called “an array”. In particular, the search is performed when the key is outside the range of values in the list. The data type of the element in the list is integer, which are positive whole number that does not have fractional parts e.g. 2, 12, 10 etc. They are divided into two parts such as static and dynamic. Searching algorithm is an algorithm for finding an item with specified properties among a collection of items. The ways of searching an array involves linear and binary search.

2. RELATED LITERATURE

2.1 Theoretical Background

An algorithm is a sequence of unambiguous instructions for solving a problem, that is, for obtaining a required output for solving legitimate input in a finite amount of time (Levitin, 2012). An algorithm can be viewed as a tool for solving a well specified computational problem. The statement of the specific in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relation. An algorithm is said to be correct if, for every input instance, it halts with the correct output. a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some instances, or it might halt with an incorrect answer. (Cormen, Leiserson, Rivest and Stein, 2009). We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, space efficiency, indicating how much extra memory it uses (Levitin, 2012). Analyzing an algorithm has come to mean predicting the resources such as memory, communication bandwidth, or computer hardware are primary concern, but most often it is computational time that we want to measure (Cormen et al, 2009). The analysis of algorithm is the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage location and space complexity – (www.wikipedia.org). Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate; we can often discard several interior algorithms in the process (Cormen et al, 2009).

2.2 Best-Case, Average Case and Worst-Case Complexities

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. This complexity is commonly expressed using Big-O notation which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, that is, as the input size goes to infinity.

The best, worst and average case complexity refers to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

Best-case Complexity: This is the complexity of solving the problem for the best input of size n

Worst-case Complexity: This is the complexity of solving the problem for the worst input of size n.

Average-case Complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the input

The worst-case time complexity indicates the longest running time performed on an algorithm given by input of size n, and this guarantees that the algorithm finishes on time. Moreover, the order of growth of the worst-case complexity is used to compare the efficiency of two algorithms.
In the theory of algorithms, the Big-O notation is typically used for three purposes:

1. To hide constants that might be irrelevant or inconvenient to compute.
2. To express a relatively small "error" term in an expression describing the running time of an algorithm.
3. To bound the worst case (Sedgewick and Flajolet, 2013).

2.3 The Big-O Notation

In mathematics, Big-O notation describes the limiting behaviour of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. It is a member of a larger family of notations that is called Landau notation, Bachmann-Landau notation (after Edmund notation Landau and Paul Bachmann), or asymptotic notation. In Computer Science, Big-O notation is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in input size. Big-O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as order of the function. A description of a function in terms of Big-O notation usually only provides an upper bound on the growth rate of the function.

2.3.1 Formal Definition of Big-O Notation

Let F and g be two functions defined on some subset of the real numbers. One writes:
$F(x) = O(g(x)) as\ X \rightarrow \propto$

if and only if there is a positive constant M such that for all sufficiently large values of X, $f(x)$ is at most M multiplied by the absolute value of $g(x)$. That is, F(X) = $O(g(x))$ if and only if there exists a positive real number M and a real number X_o such that:

$IF\ (x)| \leq M|g(X)|\ for\ all\ X\ \geq\ X_o$

In many contexts, the assumption that we are interested in the growth rate as the variable X goes to infinity is left unstated, and one writes more simply that$f(x) = O(g(x))$. The notation can also be used to describe the behaviour of f near some real number a (often, a = O): we say

$F(x) = O(g(x)) as\ X \rightarrow\ a$
if and only if there exist positive numbers δ and M such that

$$|\ F(x)| \leq M|g(x)|for\ |X - a| < \delta$$

If $g(x)$ is non-zero for values of X sufficiently close to a, both of these definitions can be unified using the limit superior.

$$F(x) = O(g(x)) as\ \varphi \rightarrow a$$

If only if

$$Lim_x \rightarrow a|\frac{f(x)}{g(x)} < \alpha$$

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function F is derived by the following simplification rules:

If $f(x)$ is a sum of several terms, the one with the largest growth rate kept, and all others omitted.

If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on X) are omitted.
It can be illustrated thus:
Let $f(x) = 6X^4 - 2X^3 + 5$, and suppose we wish to simplify this function, using O notation, to describe its growth rate as X approaches infinity. This function is the sum of three terms: $6X^4, -2X^3, and\ 5$; of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of X namely $6X^4$. Now one may apply the second rule: $6X^4$ is a product of 6 and X^4 in which the first factor does not depend on X. Omitting this factor results in the simplified form X^4. Thus, we say that $f(x)$ is a "big-oh" $of\ (X^4)$. Mathematically, we can write $f(x) = O(X^4)$. One may confirm this calculation using the formal definitions: Let $F(x) = 6X^4 - 2X^3 + 5\ and\ g(x) = X^4$. Applying the formal definition from above, the statement that $f(x) = O(X^4)$ is equivalent to its expansion.

$$|f(x)| \leq M|g(x)|$$

For some suitable choice of X_0 and M and for all $X > X_0$. To prove this, let $X_0 = 1\ and\ M = 13. Then, for\ all\ X > X_0$

$$|6X^4 - 2X^3 + 5| \leq 6X^4 + |2X^3| + 5$$

$$\leq 6X^4 + 2X^4 + 5X^4$$
$$\leq 13X^4$$

$$\leq 13|X^4|$$

So,

$$16X^4 - 2X^3 + 5| \leq 13|X^4$$

2.3.2 Determining the Efficiency of an Algorithm

The Big-O notation indicates the worst-case runtime for an algorithm – that is, how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends particularly on the number of data elements. Suppose an algorithm is designed to test whether the first element of an array has 10 elements, this algorithm requires one comparison. If the array has 1000 elements, it still requires one comparison. In fact, the algorithm is completely independent of the number of elements in the array. This algorithm is said to have a constant run time, which is represented in Big-O notation as O(1). An algorithm that is O(1) does not necessarily require only one comparison. O(1) just means that the number of comparisons is constant. It does not grow as the size of the array increases. An algorithm that tests whether the first element of an array is equal to any of the next three elements is still O(1) even though it requires three comparisons. An algorithm that tests whether the first of an array is equal to any of the other elements of the array will require at most n-1 comparisons, where n is the number of elements in the array. If the array has 10 elements, this algorithm requires up to 9 comparisons. As n grows larger, then n part of the expression “dominates”, and subtracting one becomes inconsequential.

Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of n-1 comparisons (such as the one we described earlier) is said to be O(n). An O(n) algorithm is referred to as having a linear run time. O(n) is often pronounced “on the order of n” or more simply “order n”. Now suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array. The first element must be compared with every other element in the array. The second element must be compared with every other element except the first it was already compared to the first). The third element must be compared with every element the first two.

In the end, this algorithm will end up making (n-1) + (n-2) +…+ 2+1 or $n^2/_2 - {}^n/_2$ comparisons. As n increases, the n^2 term dominates and the n term becomes inconsequential. Again, Big-O notation highlights the n^2 term, leaving $n^2/_2$. But as we have seen, constant factors are omitted in Big-O notation.

Big-O is concerned with how an algorithm's run time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons, with four elements, the algorithm requires 16 comparisons; with eight elements, 64 comparisons. With this algorithm, doubling the number of elements quadruples the number the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements the algorithm requires eight comparisons; with eight elements; 32 comparisons. Both of these algorithms grow as the square of n; so Big-O ignores the constant and both algorithms are considered to be $O(n^2)$, which is referred to as quadratic run time and pronounced "on the order of n squared" or more simply "order n-squared".

When n is small, $O(n^2)$ algorithms (running on today's billion – operation – per-second personal computers) will not noticeably affect performance. But as n grows, you will start to notice the performance degradation. An $O(n^2)$ algorithm running on a billion-element array would require a trillion operations" (where each would actually require several machine instructions to execute. This could require several hours to execute. There are algorithms with more favourable Big-O measures. These efficient algorithms often take a bit more cleverness and work to create, but their superior performance can be well worth the extra, especially as n gets large and as algorithms are combined into larger programs (Deitel and Deitel, 2007). Table 2.1 is a list of classes of some functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first; C is some arbitrary constant.

Table 1: Big-O Runtimes

Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O((log n)c)	Polylogarithmic
O(n)	Linear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n)$	Exponential

(Source: www.wikipedia.org).

2.4 The Linear (Sequential) Search Algorithm

The linear search is a very simple algorithm. Sometimes called a sequential search, it uses a loop to sequentially step through array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will unsuccessfully search to the end of the array (Gaddis et al, 2003).

2.4.1 Performance of Linear Search Algorithm

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order to find a value stored in the last element (so the algorithm actually reads an element of the array 20,000 times). In an average case, an item is just as likely to be found near the beginning of the array as near the near. Typically, for an array of N items, the linear search will locate an item in N/2 attempts if an array has 50,000 elements, the linear search will make a comparison with 25,000 of them in a typical case. This is assuming, of course, that the search item is consistently found in the array, (N/2) the average number of comparisons. The maximum number of comparison is always N). When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the linear search should not be used on large arrays if the speed is important (Gaddis et al, 2013).

2.5 The Binary Search Algorithm

The binary search algorithm is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be sorted in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array elements have been eliminated from further searching. If the desired value was not found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm immediately tests its middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until either the value being searched for is found or there are no more elements to test (Gaddis et al, 2013).

2.5.1 Performance of Binary Search Algorithm

With each test that fails to find a match at the probed position, the search is continued with one or other of the two-sub-intervals, each at most half the size. More precisely, if the number of items, N, is odd then both sub-intervals will contain (N-1)/2 elements, if the original number of items is N then after the first iteration there will be at most N/2 items remaining, then at most N/4 items, at most N/8 items, and so on. In the worst-case scenario, searching a sorted array of 1023 elements will take only 10 comparisons when using a binary search. Repeatedly dividing 1023 by 2 (because after each comparison we are able to eliminate half of the array) and rounding down (because we also remove the middle elements) yields the values 51,255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 (210-1) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an array of 1, 048,575 (220-1) elements takes a maximum of 20 comparisons to find the key, and an array of over one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous improvement in performance over the linear sear. For a one billion element array, this is a difference between an average of 500 million comparisons for the linear search and maximum of only comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as log2n. All logarithms grow at roughly the same rate, so in Big-O notation, the base can be omitted. This results in a Big-O of O(logn) for a binary search which is also known as logarithmic run time (Deitel and Deitel, 2007). Table 2.2 gives a summary of the performance of the binary search algorithm.

Table 2: Binary Search Runtimes

Data structure	Array
Worst-case performance	O(logn)
Best-case performance	O(1)
Average-case performance	O(logn)

(Source: www.wikipedia.org)

2.6 Modified Binary Search Algorithm

Ankit, Rishikesh and Tanaya (2014) on their study looked at the modification to the traditional binary search algorithm. They found that binary search uses only the middle element to check whether it matches with the input element. The aim of their study is to suggest changes to the traditional binary search algorithm in a way to optimize them. The method used is the modified method that is the way to suggest changes to the binary search algorithm which optimizes the worst cases of the traditional binary search. They were limited to look at the run-time analysis (order of growth). This run-time analysis is based on algorithm analysis which is focused on the growth rate of the running time as a function of the input size, n. This run-time is also based on a way of analyzing algorithms using mathematical notation. Time factor may be a barrier of not looking at this.

2.7 Analysis of Linear and Binary Search Algorithm

Sapinder, Ritu and Arvinder, (2012) worked on analysis of linear and binary search algorithm. The aim of this study was to compare and contrast the linear and binary search algorithm. Here, linear search is simple to implement in a larger time while searching an item, where as binary search algorithm searches an item with smaller time as compared to linear search. They were limited in discussing about the run-time analysis (order of growth) and Big-O' notation also the best case, average case and worst case complexity. The method used in solving the problem of the work was visual basic to analyze the different metrics of linear and binary search algorithms.

2.8 Parallel search and Binary Search

Digalakis, Marin and Vega (2003) on their study considered parallel search and binary search. Their view is that parallel search is a search algorithm that searches an item in unordered array. They considered that the searching time obtained in parallel search is better than the searching time obtained in binary searched. They viewed binary search as an algorithm for locating the position of an element x in a sorted list. The aim of their study is to look for their efficiency. In the limitation of study, they were limited in discussing the run-time analysis (order of growth) and Big-O' notation. The run-time analysis focuses on the growth rate of the running-time as a function of the input size. This also focuses on analyzing algorithms using mathematical notation for functions. On their work, they looked at the best, worst and average cases. In their result, after comparison on their efficiency, two parameters were used to study the efficiency of parallel systems. The speed-up that indicates the factor by which the execution time for the application change is calculated as:

$$\text{Speed-up} = \frac{\text{Execution time for one processor}}{\text{Execution time for processors}}$$

On comparing, they found out that the efficiency of the system over the binary search increases linearly when the number of item increases. This method that is used is Heapsor.

3. Methodology

The methodology employed in this research is the waterfall model. The waterfall was chosen since it is sequential software development model in which development is seen as flow steadily downwards (like a waterfall) through several phases. The waterfall model maintains that one should move to a phase only when its proceeding phase is completed and perfected. Phase of development in the waterfall model are thus discrete and there is no jumping back and forth or overlap between them. Also, the problem was well understood. This research will be implemented using the java programming language. The running time for optimized binary search will be measured; that of binary search will be measured also and recorded. The running time will then be used for the analysis of their performance.

4. THE PROPOSED METHOD

The optimized Binary search algorithm assumes that the list to be searched is sorted. Its aim is to determine if the key searched fir is within the feasible search space that is within the range of the values in the list. If the key is outside the range that is in the non-feasible search space; then the key cannot exist in the list, therefore, needless to perform the search. If it is within range, the search can then be performed; therefore, search is performed if and only if the key is within the feasible search space, otherwise the algorithm terminates. Thus the time complexity performance will improve from O(logn)-logarithmic runtime, to O(1)-constant runtime. The former represents a vast improvement on the later.
The optimized Binary search works as follows:

Determine lower and upper index.
If key is less than elements in lower index, or if key is greater than element in upper index, therefore, if the key is outside the feasible search space and cannot be contained in the list. Hence return-1 and terminate otherwise the key is within the feasible search space.

Determine mid index
If key equals the element, return the mid index (key has been found), else if key is less than the mid element then set upper index to mid-1, else 1, else the key is greater than the mid-element, hence set lower index to mid +1. Repeat until is found.
Return-1 if search is exhausted and key is not found. It is to be noted that the optimized binary search algorithm focuses on improving the time complexity in particular when the key is out of range of the value in the lists. It however, exhibits the same time complexity as the Binary search space that is the key is within range. Hence, this research does not consider the performance when the key is within the range of values in the sorted list.

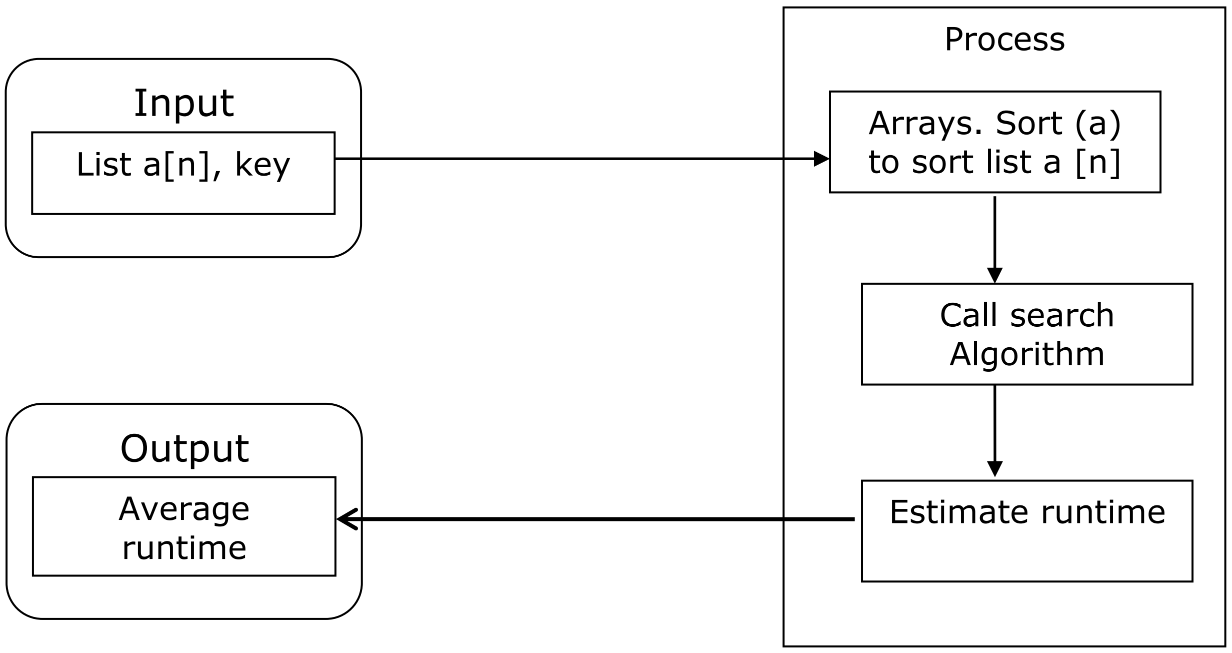


Fig. 1: Architecture of the Proposed Method

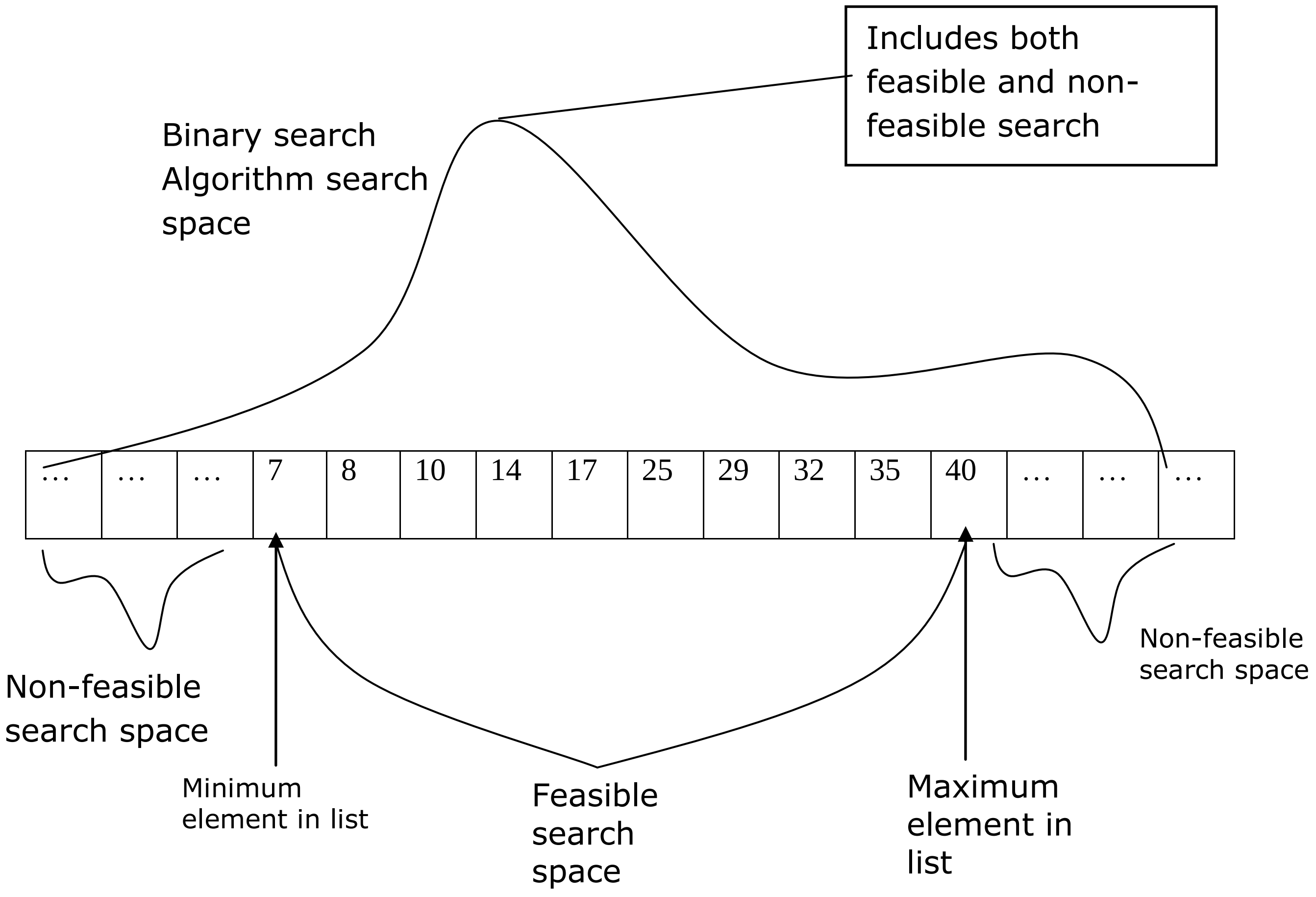

Includes both feasible and non-feasible search
Binary search Algorithm search space
…
…
…
7
8
10
14
17
25
29
32
35
40
…
…
…
Non-feasible search space
Minimum element in list
Feasible search space
Maximum element in list
Non-feasible search space

Fig. 2: Binary algorithm search Space

From figure 2 the search space of the Binary Search algorithm includes both feasible and non-feasible search spaces. This means that search will be performed even when the elements is in the non-feasible search space i.e. is outside the range of values, thus never present in the list.

Let key =50 (which is in the non-feasible search space)
In each iteration of the algorithm, the bottom half of the array is discarded until there is only one element.

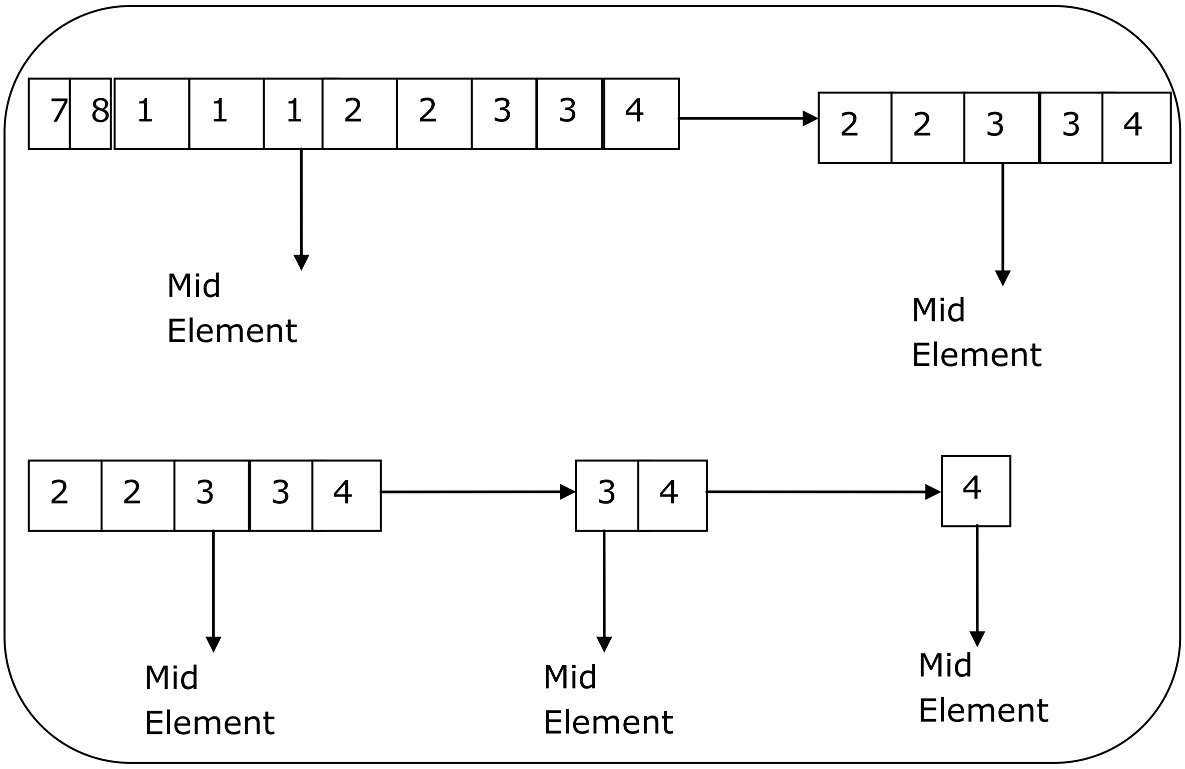


Fig. 3: Binary Search Illustration

After 3 iterations, the algorithm is unsuccessful i.e. does not find a match (40 ≠ key). It returns-1 (indicating the key was not found) and terminates.

Optimized Binary Search-Illustration
Let us assume a sorted list of 10 elements i.e. B [0-9] as shown in figure 4

7	8	10	14	17	25	29	25	35	40

Fig. 4 sorted list B [0-9]

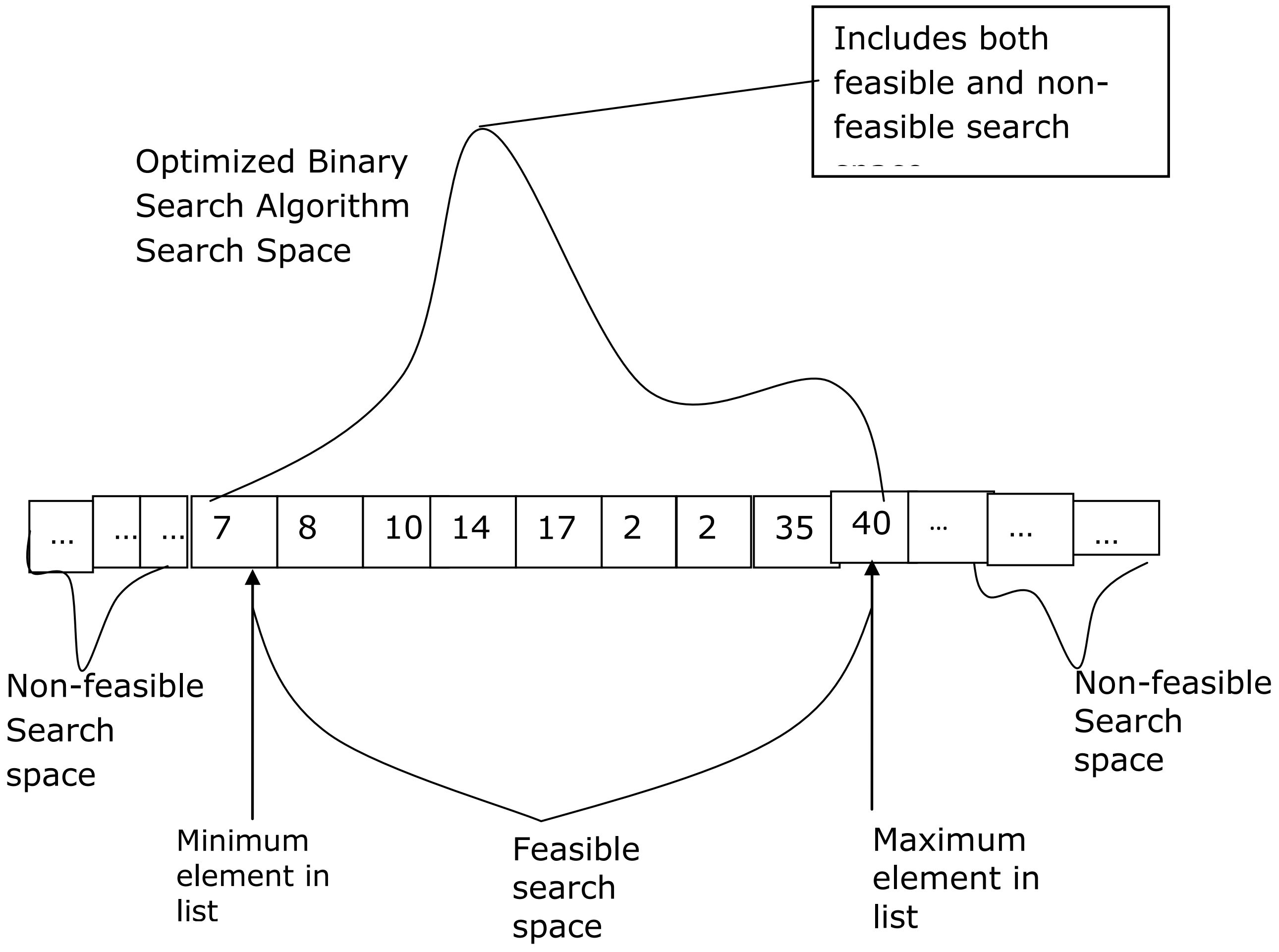

Includes both feasible and non-feasible search space
Optimized Binary Search Algorithm Search Space
...
...
...
7
8
10
14
17
2
2
35
40
...
...
...
Non-feasible Search space
Non-feasible Search space
Minimum element in list
Feasible search space
Maximum element in list

Fig. 5: Optimized Binary Search Space

From figure 5, the search space of the optimized binary search algorithm includes only the feasible search space; hence elements in the non-feasible search space are never searched.

Let key =50 (which is in the non-feasible search space). The algorithm tests the key with the minim and maximum elements in the list to determine if the key is with the feasible search space or not.

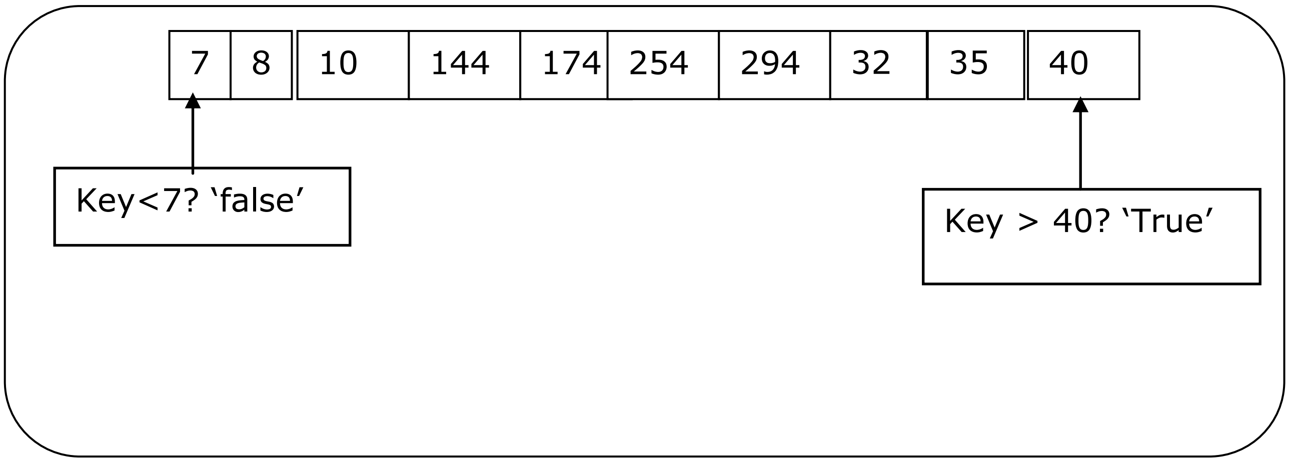


Fig 6: Optimized binary Search illustration

Since key >40 is true, the algorithm includes that the key is out of range that is within the non-feasible search space, therefore the algorithm performs no search. It returns-1 (indicating the key was not found) and terminates.

5. RESULTS

Table 3: Runtime Values (milliseconds)

SEARCH VALUE (Integers)	OPTIMISED RUNTIME (Milliseconds)	ORDINARY RUNTIME (Milliseconds)
10	15	234
20	70	422
30	31	639
40	62	843
50	47	1045
60	31	1248
70	62	1560
80	78	1748
90	47	1950
100	63	2168

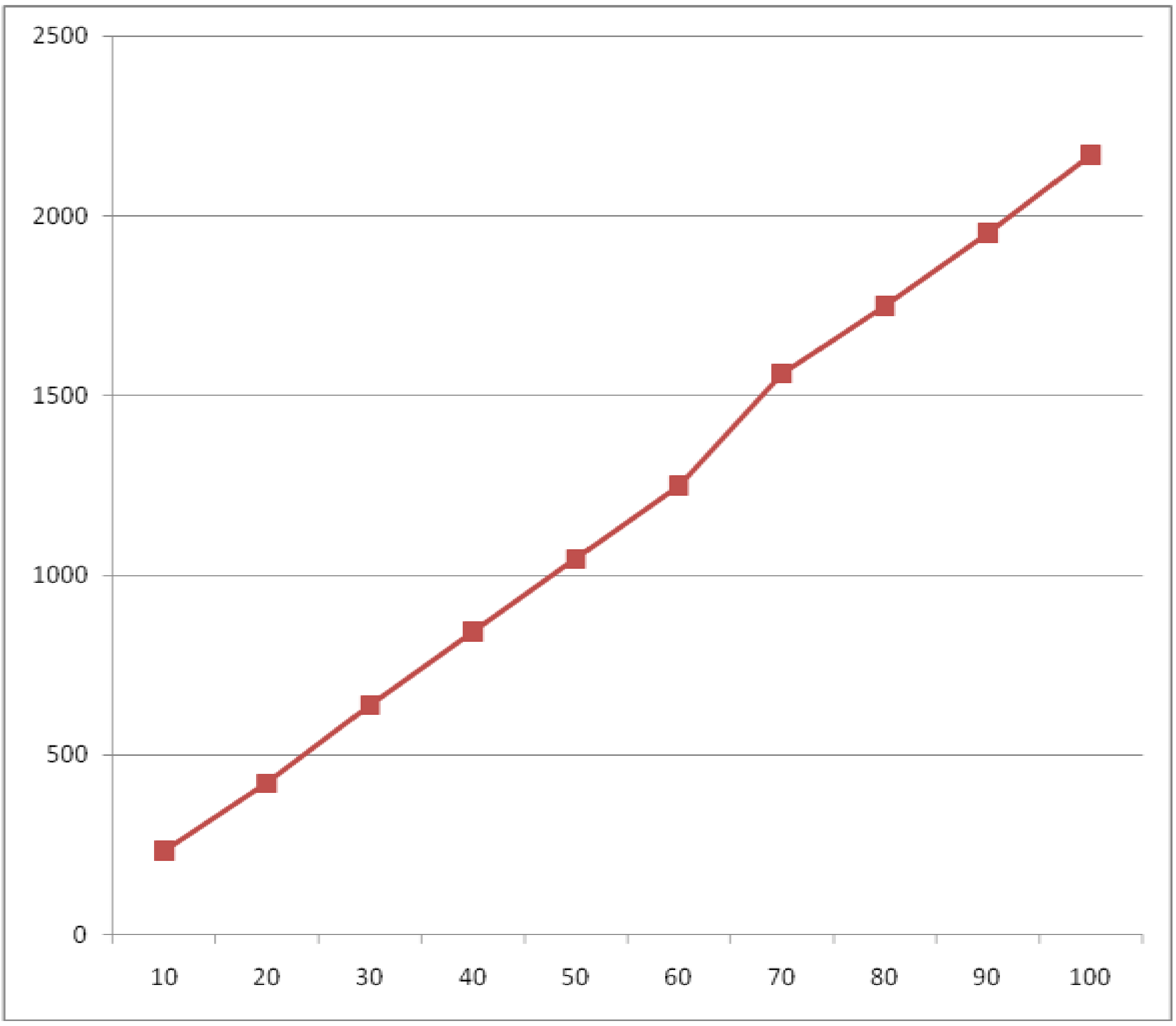


Fig.7: Ordinary Binary Search Runtime

From table 3 (demonstrated in figure 7), it can be seen that the average runtimes of Binary search increases roughly, as the input size increases i.e. from 234 ms when n=10 to 2168ms when n=100.

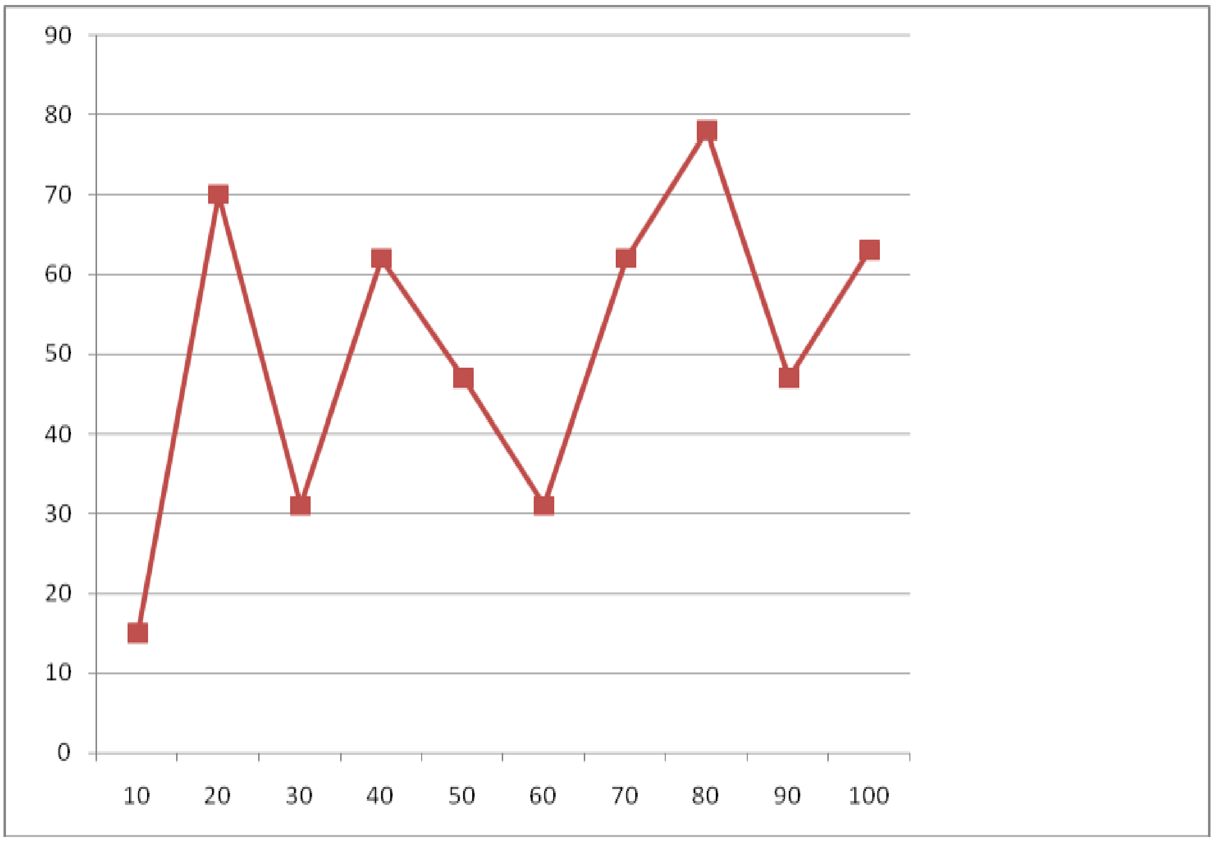


Fig. 8: Optimized Binary Search Runtime

Also, in table 3 (demonstrated in figure 8) shows the average runtimes of the optimized binary search which appears to be in an interval, ranging from15ms to 78ms as the input size is varied from n=10 to n=100, the average runtimes decreases.

6. SUMMARY AND CONCLUSION

The binary search algorithm is a clever algorithm that is more efficient than the linear search algorithm. It is a dichotomic divides and conquer search algorithm with a runtime of O (logn), but requires that the data to be searched is sorted. The binary search algorithm however, exhibits the same runtimes of O (logn) even when the element searched for is not in list-in particular, when the element is outside the range of values, thereby incurring extra overhead which could be avoided.

The optimised binary search algorithm presented in this research is an optimization of the Binary search algorithm, eliminating the need for a search if the key (element) is outside the range of values in the list i.e. outside the feasible search space. Average runtimes (in milliseconds) of the optimized Binary search and Binary search algorithms was measured and the data used to validate the efficiency of the optimized Binary search over the Binary search, thereby satisfying the objectives of this research.

5.2 Conclusion

This research has presented an optimized binary search algorithm which exhibits a runtime of O (1)- constant runtime-when search is performed on a linear collection of element in which the key is out of the range of values in the list i.e in the non-feasible search space. This is a vast improvement on the Binary Search algorithm which has a runtime of O (Logn)- Logarithmic runtime-even when key is in the non-feasible search space.
Analysis of the runtimes of the optimized Binary Search and the runtimes of Binary algorithm shorted that the average runtimes of Binary Search increases roughly, as the input size, n, increases.

The optimized Binary Search will thus enable search to be performed with faster time, thereby ensuring search efficiency; however, it will exhibit the same runtime as the Binary Search algorithm if the vale searched for is within the feasible search space (within the range of values in the list).

REFERENCES

Ankit, R.C., Rishikesh, M., and Tanaya, M. (2014): Modified Binary Search Algorithm. *International Journal of applied Information Systems* (ISAIS): 224-0868 Foundation Computer Sciences, New York, USA, 7 (25).

Asabga, P.O. Osaghae, E.O. and Ogheneovo, E.E, (2010): "*Is Binary Search Techniques faster than linear Search techniques?*", "University of Port Harcourt. Rivers State.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2009): *Introduction to algorithms*, 3rd edition. Massachusetts Institute of Technology. USA

Deittel, P.J. and Deitel, H.M. (2007):" *Java How to Program*, 7th edition Upper saddle River, New Jersey: Prentice Hall.

Digalakis, J., Marvin, M.J., Yega (2003): Parallel Evolutionary algorithms on message Passing clusters. http://www.it.uom.gr/people/digalakis/digamarg 2003pdf.Retrieved 20/4/2012.

Gaddis, T., Walters, J., and Muganda, G. (2013): *Starting out with C++- Early Objects*, 8th edition. Reading, Massachusetts: Addision-Wesley

http://en.wikipedia.org/wiki/\Binary-search- algorithm-Binary Search. Retrieved:24/05/2014.

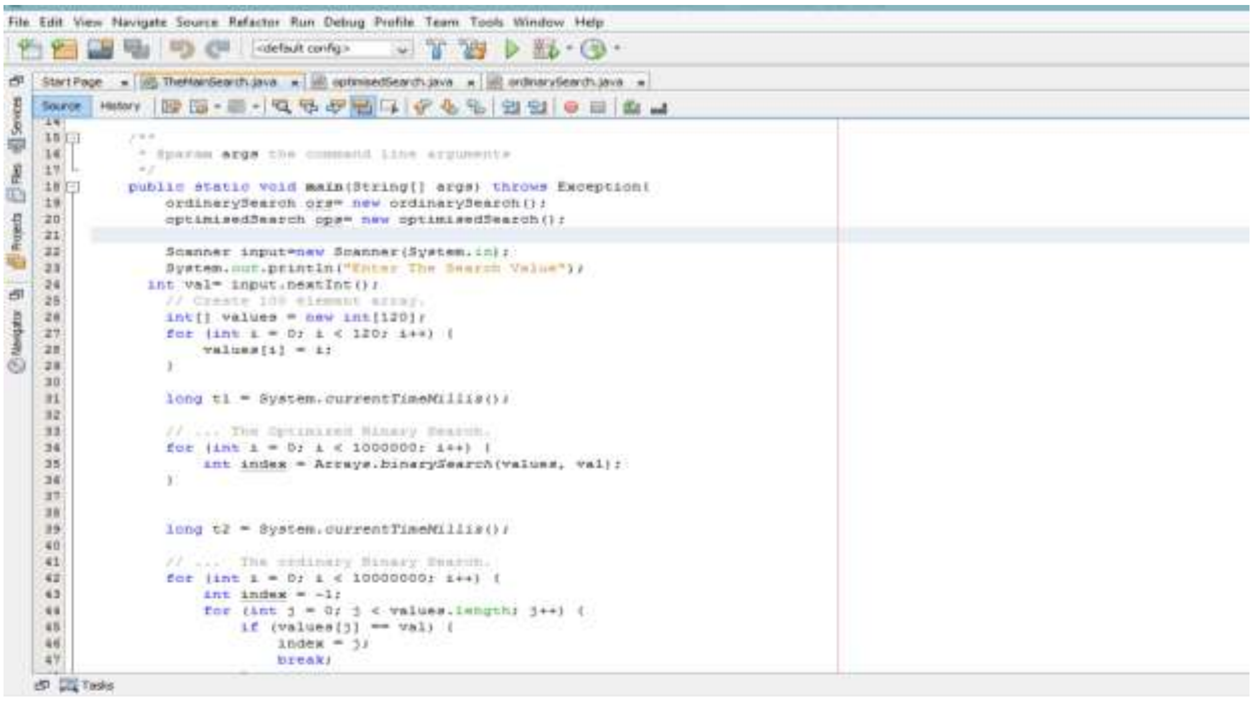
Knuth, D. (1997): *The Art of Computer Programming*, 3rd Eddition. Reading, Massachusetts: Addison-Welsley.

Levitin, A (2012): *Introduction to the design and analysis of algorithm*, 3rd edition. Upper Saddle River, New Jersey: Pearson Education Inc.

Safinder, Rity, and Arvinder, S. (2012): Analysis of Linear and Binary search Algorithm. *International Journal of Computers and Distributed Systems*, 1 (20).

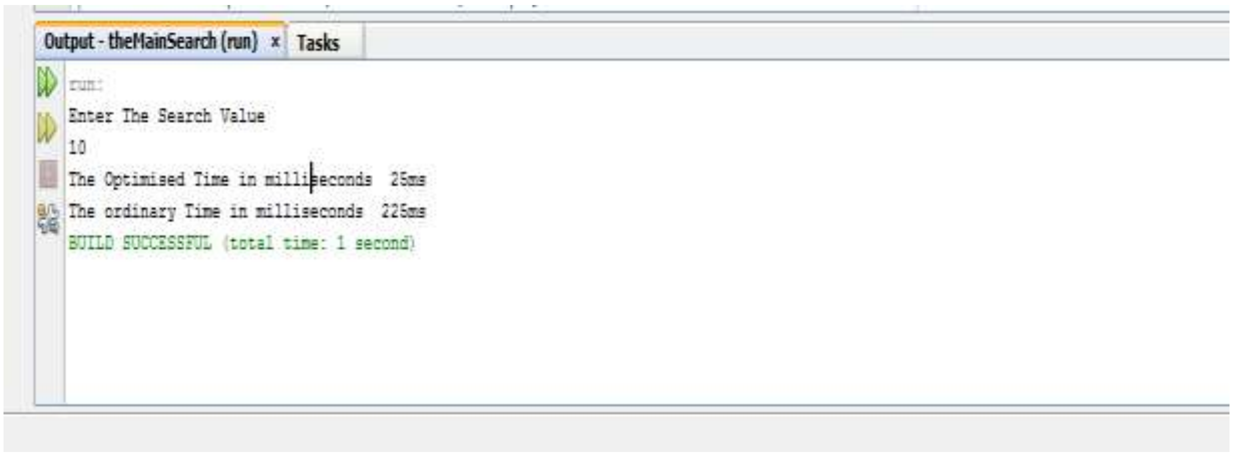
Sedgewick, R. and Flajolet, P. (2013): *An Inrtroduction to the Analysis of Algorithms*, 2nd edition.. Upper Saddle River, New Jersey: Pearson Education Inc.

APPENDIX A
SAMPLE OUPUT
SAMPLE OUTPUT FROM IDE ENVIRONMENT



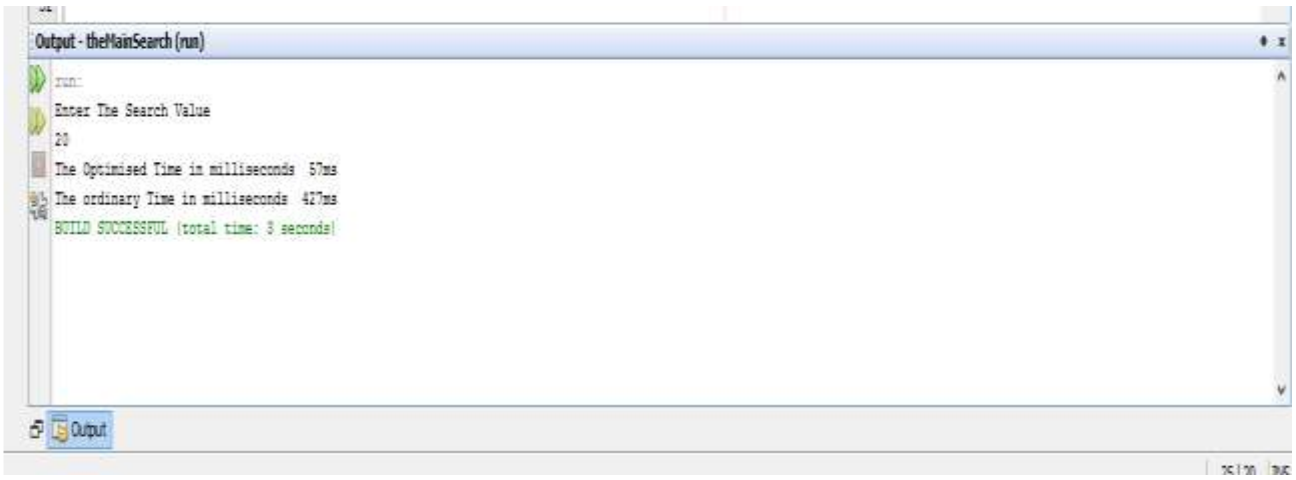
```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception{
        ordinarySearch ors= new ordinarySearch();
        optimisedSearch ops= new optimisedSearch();

        Scanner input=new Scanner(System.in);
        System.out.println("Enter The Search Value");
      int val= input.nextInt();
        // Create 100 element array.
        int[] values = new int[120];
        for (int i = 0; i < 120; i++) {
            values[i] = i;
        }

        long t1 = System.currentTimeMillis();

        // ... The Optimized Binary Search.
        for (int i = 0; i < 1000000; i++) {
            int index = Arrays.binarySearch(values, val);
        }


        long t2 = System.currentTimeMillis();

        // ...  The ordinary Binary Search.
        for (int i = 0; i < 10000000; i++) {
            int index = -1;
            for (int j = 0; j < values.length; j++) {
                if (values[j] == val) {
                    index = j;
                    break;
```

INPUT SIZE N=10



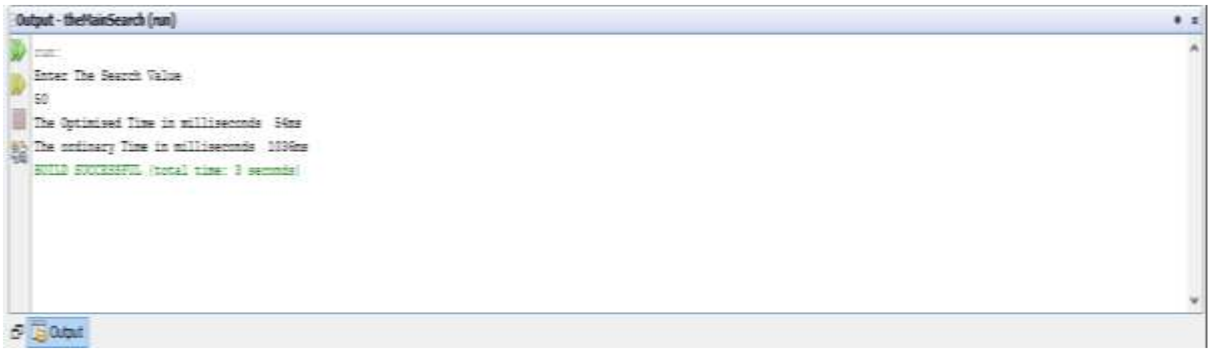
```
run:
Enter The Search Value
10
The Optimised Time in milliseconds  25ms
The ordinary Time in milliseconds  225ms
BUILD SUCCESSFUL (total time: 1 second)
```

INPUT SIZE N=20



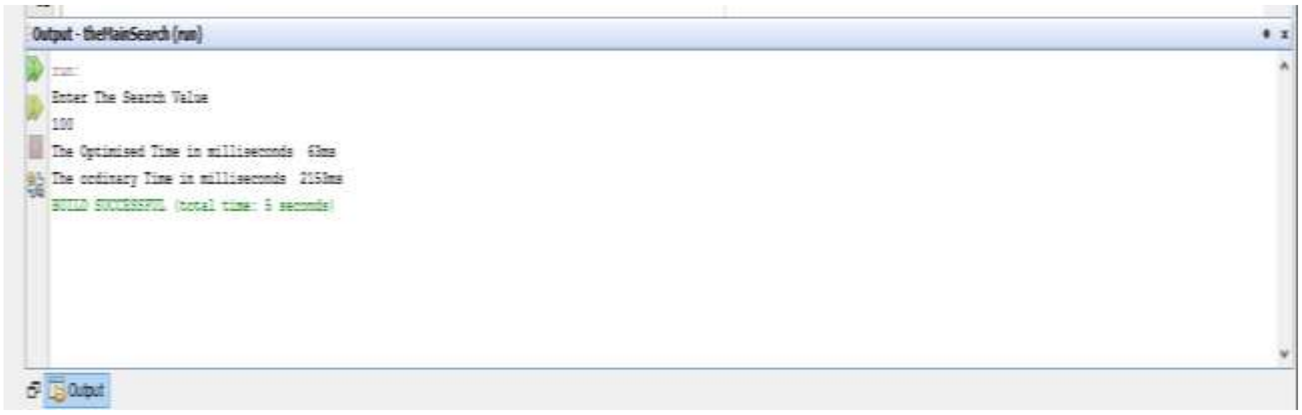
```
run:
Enter The Search Value
20
The Optimised Time in milliseconds  57ms
The ordinary Time in milliseconds  427ms
BUILD SUCCESSFUL (total time: 3 seconds)
```

INPUT SIZE N=50



```
run:
Enter The Search Value
50
The Optimised Time in milliseconds  54ms
The ordinary Time in milliseconds  1036ms
BUILD SUCCESSFUL (total time: 3 seconds)
```

INPUT SIZE N=100



```
run:
Enter The Search Value
100
The Optimised Time in milliseconds  63ms
The ordinary Time in milliseconds  2153ms
BUILD SUCCESSFUL (total time: 5 seconds)
```

APPENDIX B
PROGRAM LISTING
PROGRAM 1: Optimised Binarysearch.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package themainsearch;
/**
 *
 * @author PRECISION
 */
public class optimisedSearch {
    int searchElementoptimized(int numList[], int toSearch) {
  int startIndex = 0;
  int endIndex = numList.length - 1;
  int midindex = (startIndex + endIndex) / 2;
  int midElement = numList[midindex];
  int foundIndex = 0;
    long startTime = System.nanoTime();
  while (startIndex <= endIndex) {
   if (midElement < toSearch) {
    startIndex = midindex + 1;
    midindex = (startIndex + endIndex) / 2;
    midElement = numList[midindex];
   } else if (midElement > toSearch) {
    endIndex = midindex - 1;
    midindex = (startIndex + endIndex) / 2;
    midElement = numList[midindex];
   } else {
    foundIndex = midindex;
    break;
   }
  }        long endTime = System.nanoTime();
 long optimizedTime=endTime-startTime;
 System.out.println(optimizedTime);
  return foundIndex;   }
}
```

PROGRAM 2: BINARY SEARCH. JAVA

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package themainsearch;
/**
 *
 * @author PRECISION
 */
public class ordinarySearch
```

```
int searchElement1(int numList[], int toSearch) {
  int foundIndex = 0;
  long startTime = System.nanoTime();
  for (int i = 0; i < numList.length; i++) {
   if (numList[i] == toSearch) {
    foundIndex = i;
   }
  }
  long endTime = System.nanoTime();
 long ordinaryTime=endTime-startTime;
 System.out.println(ordinaryTime);
  return foundIndex;
     }
```

APPENDIX C
MAIN SEARCH CODE (OPTIMIZED AND BINARY SEARCH ALGORITHM)

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package themainsearch;
        import java.util.Arrays;
import java.util.Scanner;
/**
 *
 * @author PRECISION
 */
public class TheMainSearch {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception{
        ordinarySearch ors= new ordinarySearch();
        optimisedSearch ops= new optimisedSearch();
                Scanner input=new Scanner(System.in);
        System.out.println("Enter The Search Value");
      int val= input.nextInt();
            // Create 100 element array.
            int[] values = new int[120];
            for (int i = 0; i < 120; i++) {
                values[i] = i;
            }
            long t1 = System.currentTimeMillis();
            // ... The Optimized Binary Search.
            for (int i = 0; i < 1000000; i++) {
                int index = Arrays.binarySearch(values, val);
            }
            long t2 = System.currentTimeMillis();
            // ...  The ordinary Binary Search.
            for (int i = 0; i < 10000000; i++) {
                int index = -1;
                for (int j = 0; j < values.length; j++) {
                        if (values[j] == val) {
                            index = j;
                            break;
                        }
                }
//
            }

            long t3 = System.currentTimeMillis();

            // ...  Calculating and Displaying the Time.
       long Optimized=t2 - t1;
            System.out.println("The Optimised Time in milliseconds "+" "+Optimized+"ms");
        long ordinary=t3 - t2;
            System.out.println("The ordinary Time in milliseconds "+" "+ordinary+"ms");
    }
}
```