Computer Science Extended Essay

Investigating the time complexities of various Recursive and Iterative algorithms

Research Question:

To what extent do the **problem-solving approaches**, **Recursion** and **Iteration**, compare in terms of their **runtime performance** upon **insertion** of **randomized sorted** values?

Word Count: 3,938

Session: May 2020

Contents

1 Introduction

This essay will focus on two problem solving approaches, **recursion** and **iteration** specifically, for several algorithms and how their time complexities vary on input of different set sizes in order to find the most time-efficient approach. On further exploration of these approaches, algorithms such as Binary Searching in an array, Insertion Sort, and searching in Binary Search Trees will be examined as they can be written using both iterative and recursive statements. For a varied set of values, the runtime of both programs will be investigated and recorded. The expected graphs will be compared to the graph obtained and will be analysed in detail. Hence, the question:

To what extent do the problem-solving approaches, Recursion and Iteration, compare in terms of their runtime performance upon insertion of randomized sorted values?

Recursion and ***Iteration***, are both ways of unravelling a problem. However, programmers prefer using recursive methods rather than iterative just because it uses fewer lines of code, but the major disadvantage of recursion is that it cannot be used for all algorithms, and it also causes overheads. This is because all algorithms that can be written using recursive statements can be written using iterative statements, but only a limited number of algorithms written in iterative statements can be written using recursive statements. Thus, the most time-efficient approach for specific algorithms for an input of large data will be determined in this essay.

2 Theory

2.1 Iteration

Iteration is the process of calculating a desired result by the means of repeated cycle of operations. This process is convergent, indicating that as the number of iterations increase, the process comes closer to the desired result.

Loop is an important term in order to explain the concept of iteration. It is a programming structure which iterates a statement until a certain task is completed, meaning that the program will repeat until the given condition isn't satisfied. Iterative statements usually make use of loops such as for loops and while loops.

2.2 Recursion

Recursion occurs when a method calls itself until some certain terminating condition is met. This is accomplished without any loops. Recursion is a good alternative to iteration and it follows one of the most basic problem-solving techniques, which is to break down the problem further into sub-problems. Hence, recursion is the idea of taking a problem and reducing it into a smaller version of the same problem. The goal is to *not* have infinite recursion.

With this essay, a conclusion will be reached indicating whether an iterative or recursive approach is more time-efficient, helping programmers around the world to choose the approach more efficient for solving complex recursive problems with large data input.

2.3 Time Complexity

Time complexity can basically be referred to as the running time of a program. Performing an accurate calculation of a program's operation time is a labour-intensive process as it depends on the compiler and the type of computer or speed of processor.

Therefore, an accurate measurement will not be made in this essay; just an estimated value of runtime will be measured.

The time complexity is represented by the Big-O notation, O(n) where n is array size. For example: in a nested for loop, the Big-O notation will be $O(n^2)$. This is because the nested for loop program will be executed n*n times. The worst case for time complexity is where there are maximum program executions, and the best case is when there are minimum program executions.

2.4 Complex Algorithms

The java code for the following algorithms which have been retrieved from online and later edited by me have been added in the appendix. To gain an understanding of how each program works, comments have been added to the code.

```
for (a = 0; a < noOfElements; a ++)
{
    arr[a] =  rand.nextInt(1000000); // insertion of randomized values
    System.out.println(arr[a]);
}
```

Figure 2.4.1: Insertion of random values in an array

```
Arrays.sort(arr); // in-built function to sort the array
```

Figure 2.4.2: Using the in-built function to sort arrays

The piece of code in figures 2.4.1 and 2.4.2 are identical for all programs as they are used to insert random values in an array, and the other function is to sort arrays for Binary Search and Binary Search Trees.

2.4.1 Binary Search in an Array

Binary Search is a very intuitive algorithm. It can be deduced from the name itself that it is a searching algorithm, which means that this algorithm will search for an element in an array of numbers. Also, the term “binary” refers to something involving of two things, where in this case, for comparisons, array is divided into 2 parts.

However, this will only work for sorted arrays as the middle element is assumed to be the median value of the array. Without sorting, the median value can be anywhere, and if the array is divided in half, the number to be searched for can be removed along with that.

2.4.1.1 Procedure

1. Inserting randomized values in an array
2. Sorting the array
3. Comparing the value searched with array’s middle element
4. If value searched = middle element, the value searched for is the middle element
5. If value searched < middle element, the right half of the array is ignored
    i. Continues to divide the left half of the array until the middle element of the array is equal to the value searched
    ii. If not equal, the element searched for is not present in the array
6. If value searched > middle element, the left half of the array is ignored
    i. Continues to divide the right half of the array until the middle element of the array is equal to the value searched
    ii. If not equal, the element searched for is not present in the array

2.4.1.2 <u>Conducting Binary Search in a Sorted Array</u>

A list of sorted elements for binary search, specifically from 1 to 20, has been shown in Figure 2.4.1.1

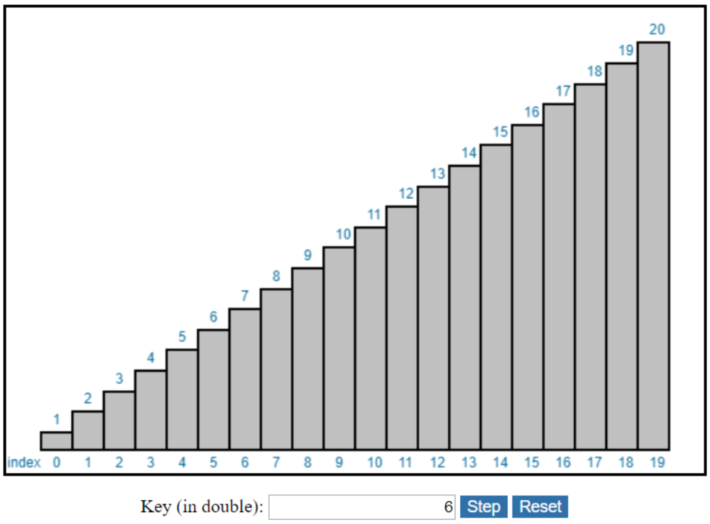

Figure 2.4.1.1: List of elements to be sorted

Following images are illustrations for the procedure of Binary Search if the element '7' is to be searched:

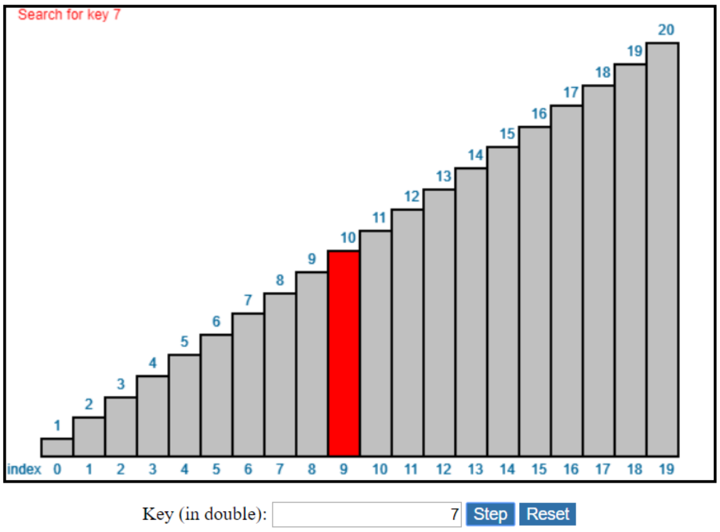

Figure 2.4.1.2: Middle element of the initial array highlighted in red

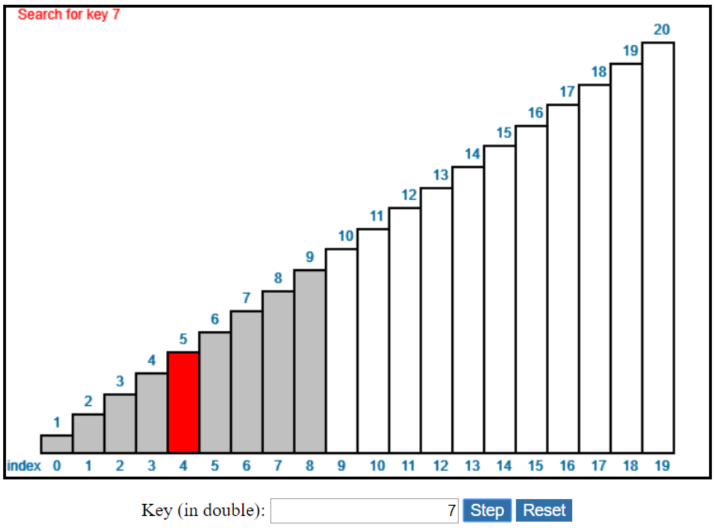

Figure 2.4.1.3: Right side of the array ignored; middle element of the left array highlighted

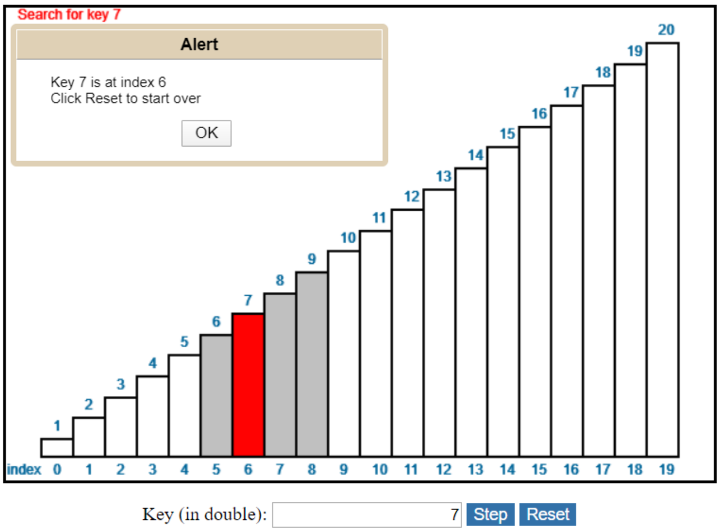

Figure 2.4.1.4: Left side of the left array ignored; middle element of the right array highlighted; value found

In this case, only 3 comparisons are made instead of 4, which is the expected value. This is because in the 3rd comparison itself, the value was found. The expected value is determined using the Big O Notation, explained in Section 2.3, which is $O\ (\log_2(n))$. For instance, if the set size is 32, the number of comparisons will b: $(\log_2(32)$, which is 5. Hence, for 32 numbers, there will be a total of 5 comparisons.

2.4.2 Insertion Sort

Insertion sort, one of the several sorting algorithms, is an intuitive sorting technique and is used for sorting an array or a list of numbers. It is not the best sorting algorithm in terms of performance, but it is more efficient than sorting algorithms such as Bubble sort and Selection sort. However, the comparison of performance of sorting algorithms will not be looked into as it is out of scope of this essay.

Insertion sort compares every element with all the elements in the array and only after doing this, the element is inserted in the correct position. There will always be a portion of the array which is sorted.

2.4.2.1 Procedure

1. Insertion of randomized values in an array
2. The first array element is compared to the second.
3. If first element is greater, it will swap places with the second element and the next element will be investigated.
4. If first element is smaller, there will be no swapping of elements, and then the 3rd element will be compared with the 1st one. If the 3rd one
5. This process is repeated until the first element has been compared with all elements in the array.
6. This process is repeated for all elements in the array. Each element in the array is compared with all elements.
7. If there is an element smaller than the first element, it will be placed before the first element. Hence, for ‘n’ number of elements, there are $n * n$ comparisons. As a total of n^2 comparisons will be made, the time complexity becomes $O(n^2)$

2.4.2.2 <u>Conducting Insertion sort on random values</u>

The following images illustrate the logic behind insertion sort:

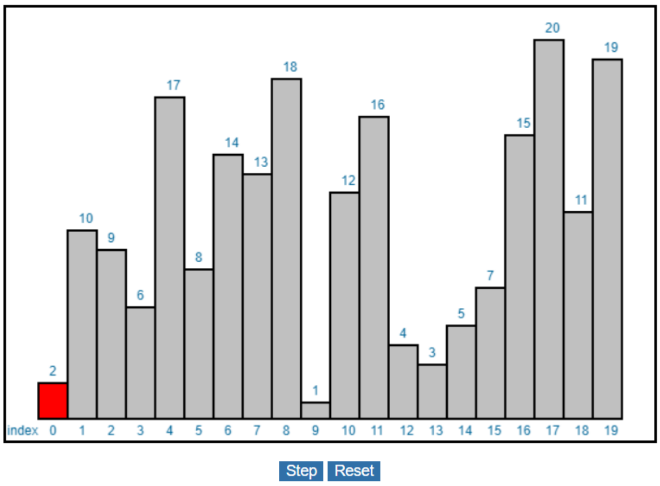

Figure 1.4.2.1: List of elements to be sorted

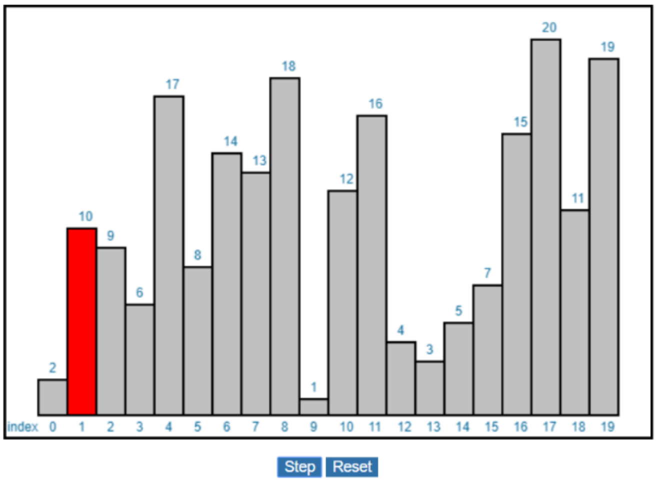

Figure 2.4.2.2: Checks the next element

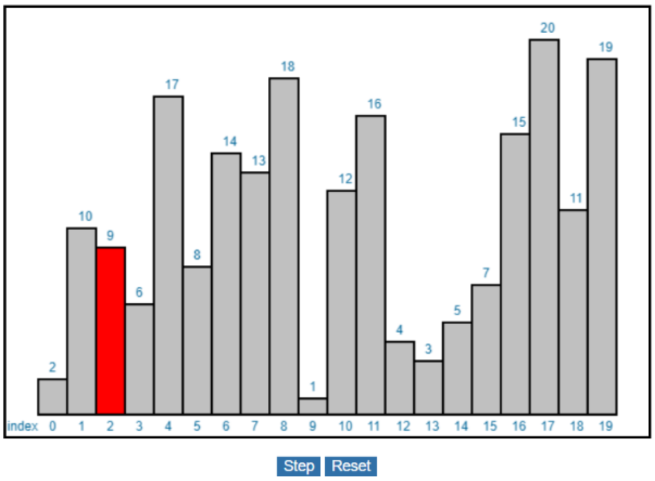

Figure 2.4.2.3: Checks the next element as the previous elements are sorted. 9 placed before 10.

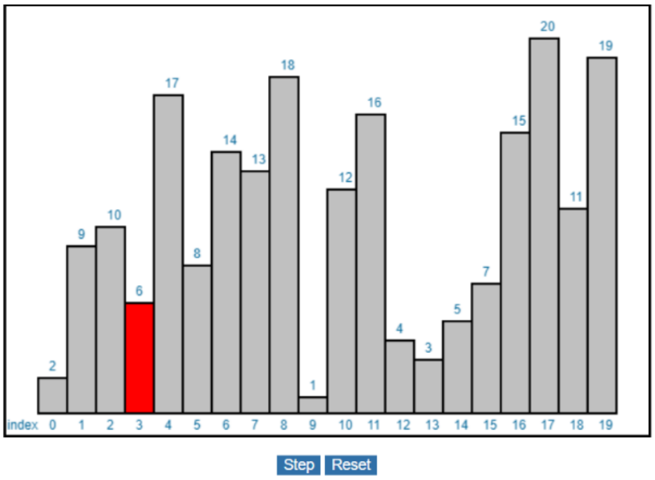

Figure 2.4.2.4: Moving on to the next element. 6 is placed before 9; 6 is compared to both 9 and 10

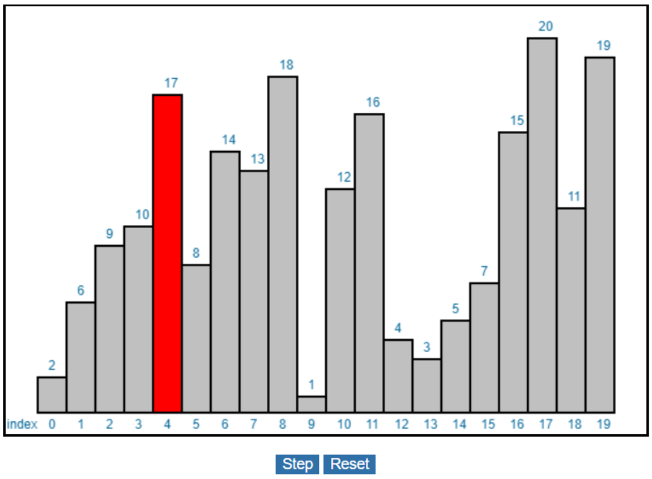

Figure 2.4.2.5: No changes for the element 17; left side is still sorted

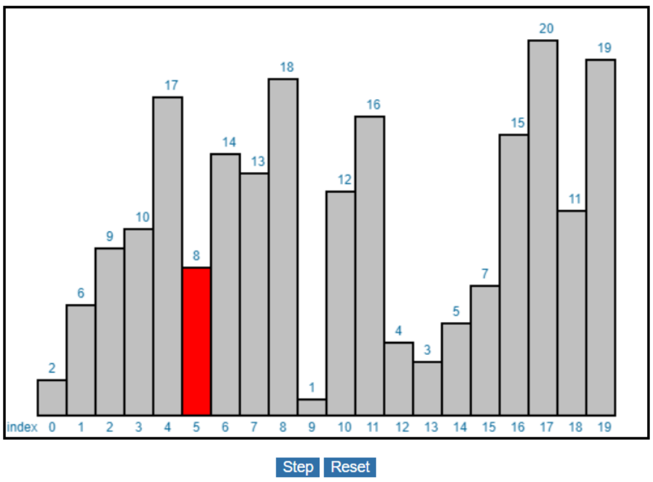

Figure 2.4.2.6: 8 is shifted to the left side, between 6 and 9

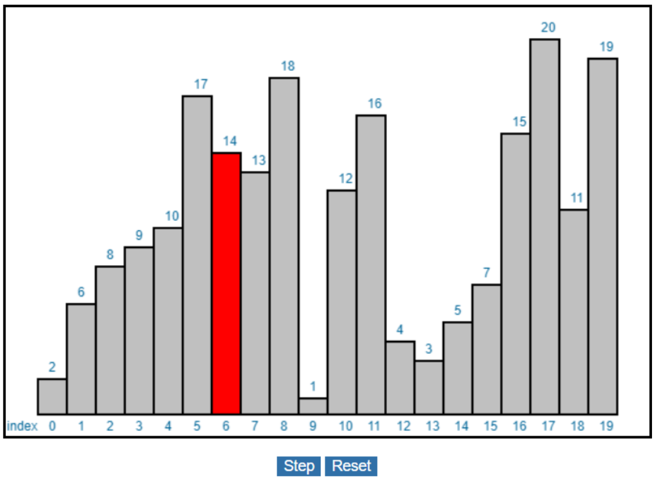

Figure 2.4.2.7: 14 is placed before 17

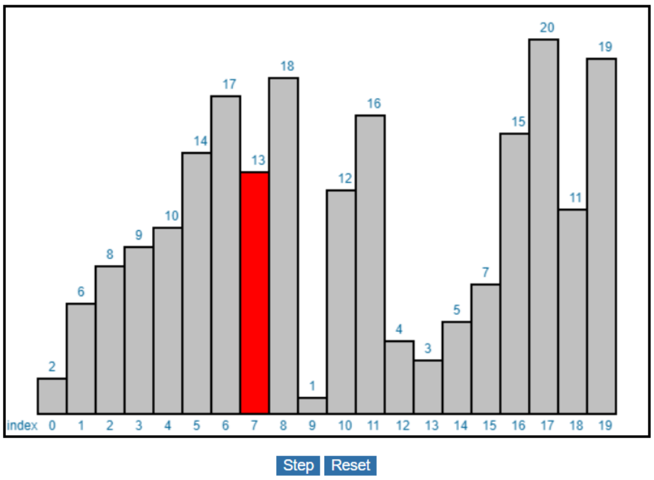

Figure 2.4.2.8: 13 is placed before 14

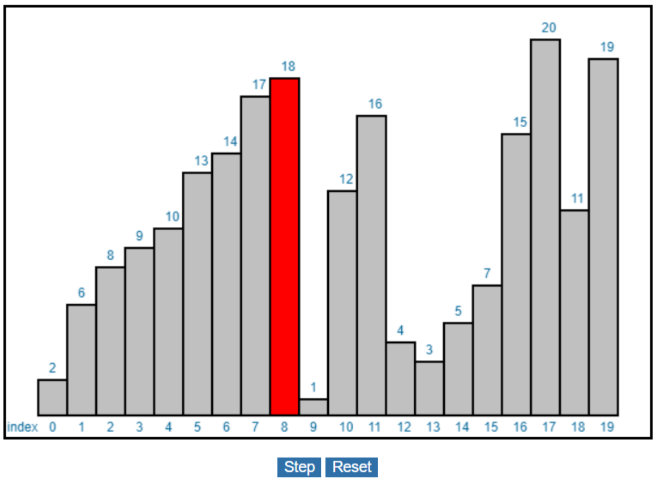

Figure 2.4.2.9: 18 stays at the same place

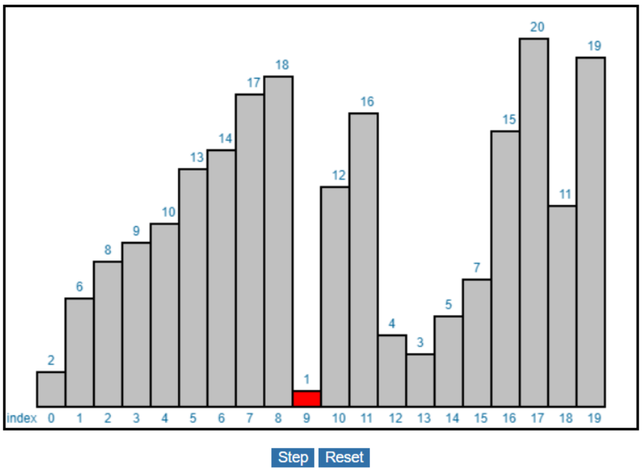

Figure 2.4.2.10: 1 is placed right at the beginning of the list

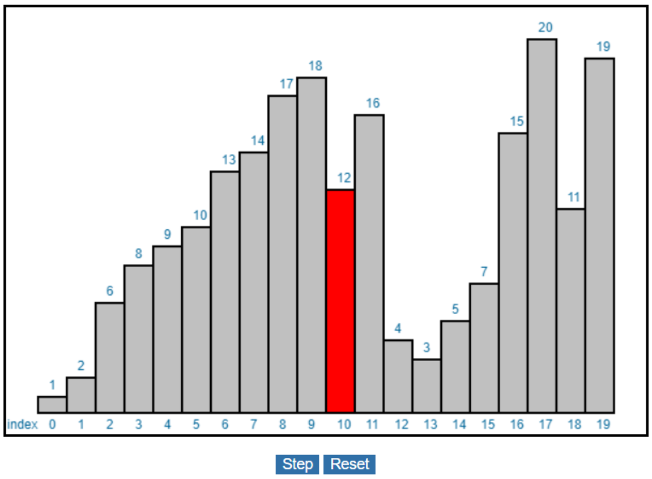

Figure 2.4.2.11: 12 is placed before 13

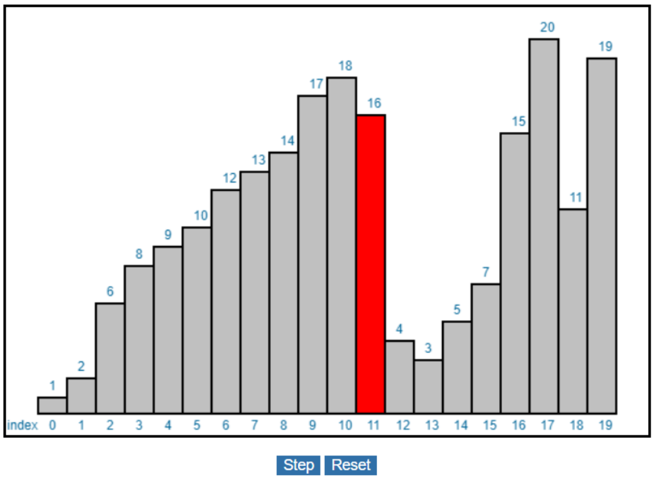

Figure 2.4.2.12: 16 is placed before 17

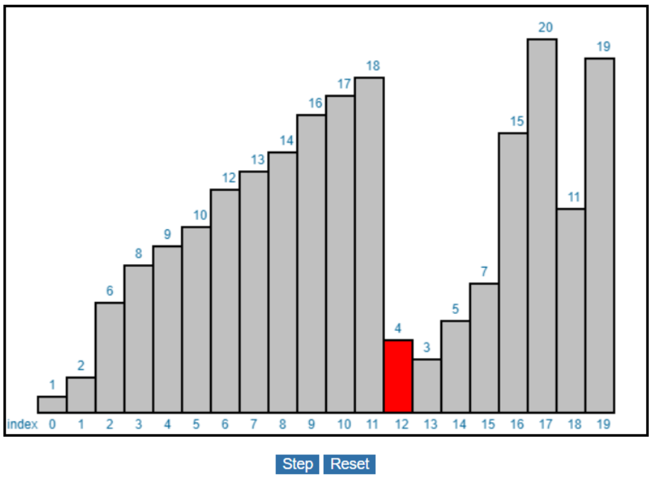

Figure 2.4.2.13: 4 is placed before 6

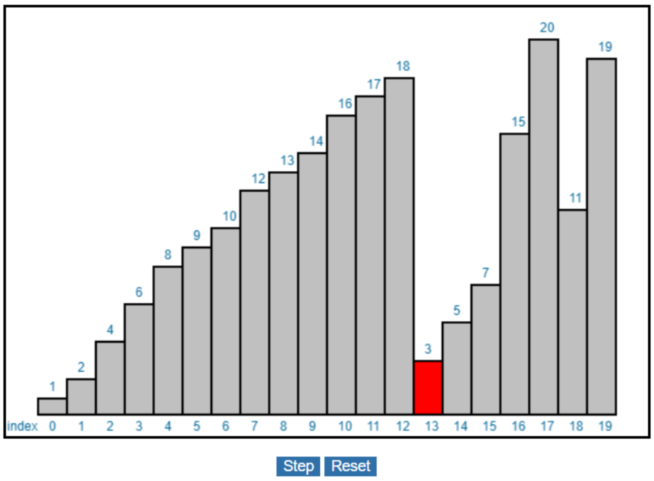

Figure 2.4.2.14: 3 is placed before 4

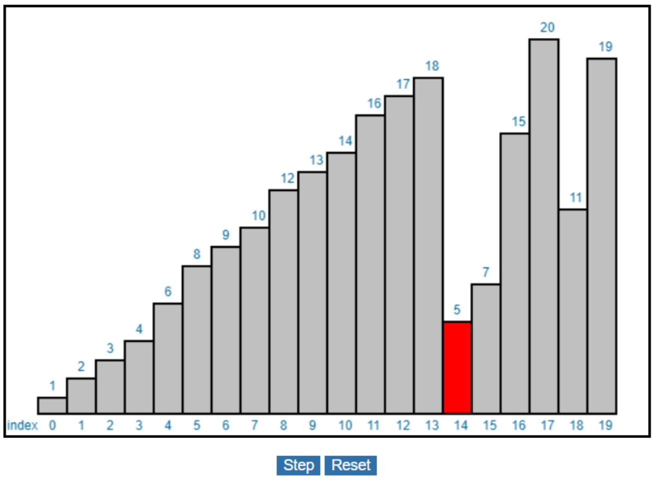

Figure 2.4.2.15: 5 is placed before 6

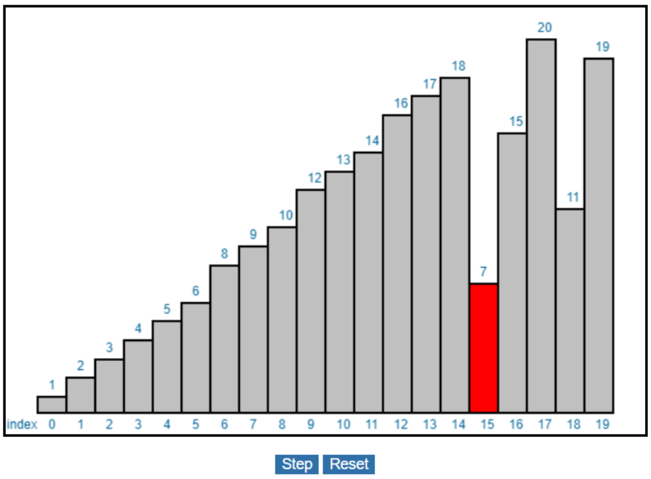

Figure 2.4.2.16: 7 is placed before 8

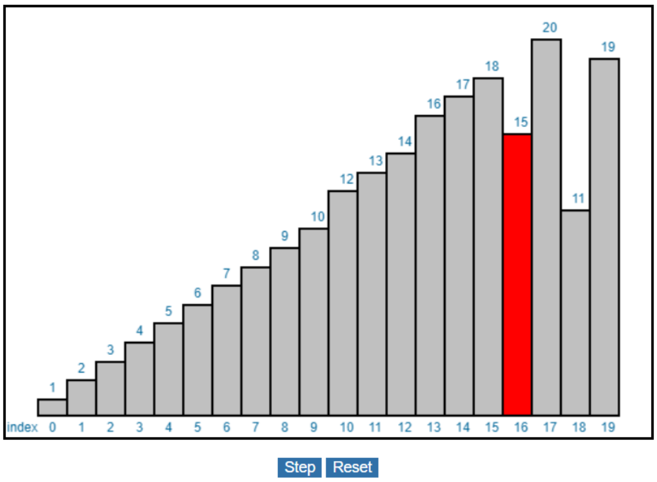

Figure 2.4.2.17: 15 is placed before 16

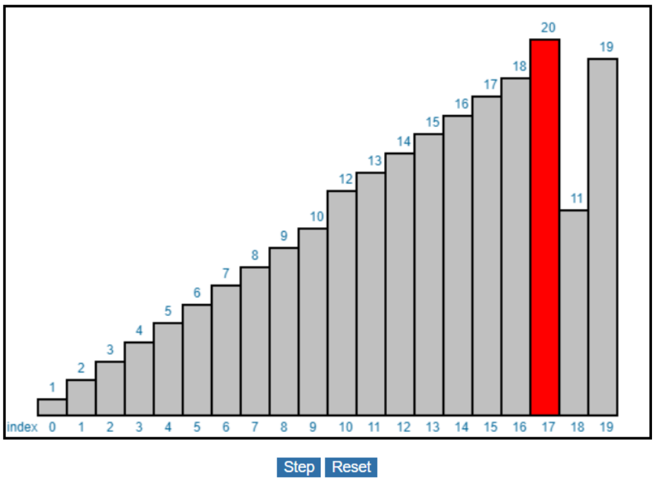

Figure 2.4.2.18: 20's position doesn't change

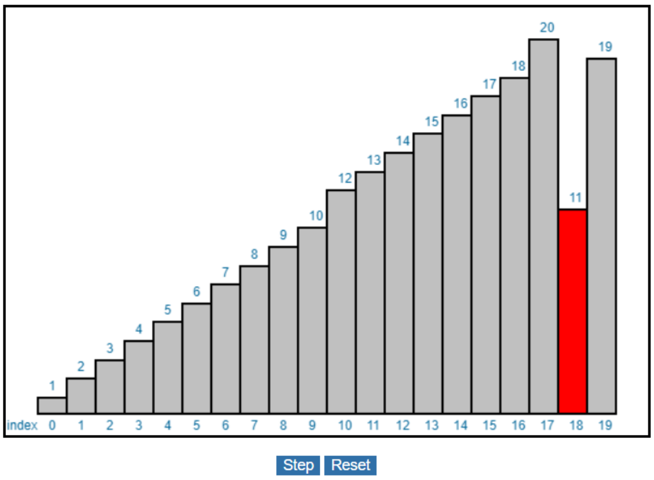

Figure 2.4.2.19: 11 is before 12

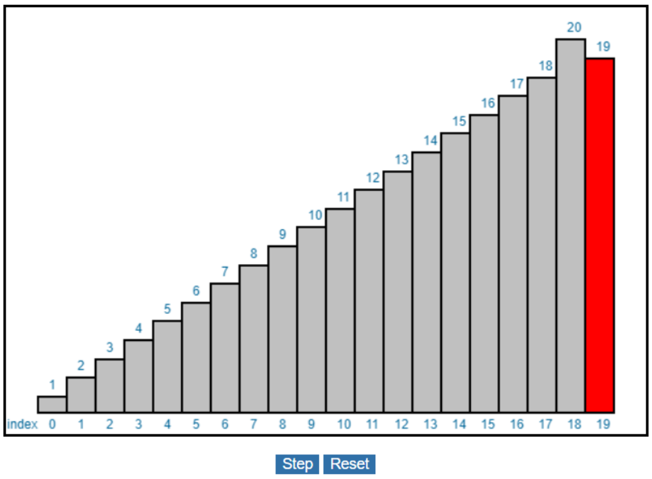

Figure 2.4.2.20: 19 is placed before 20

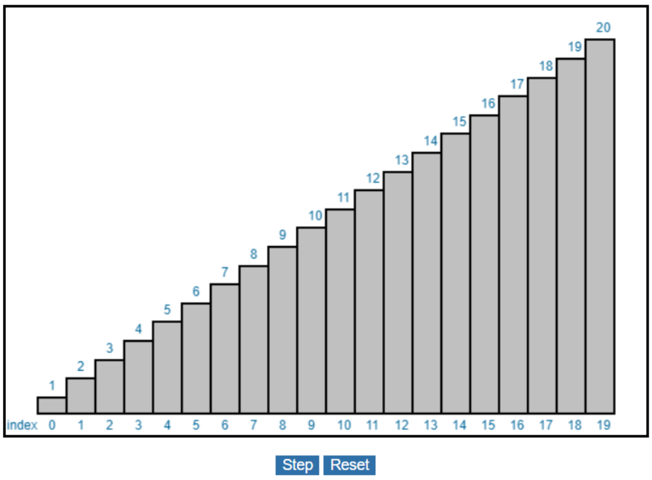

Figure 2.4.2.21: Elements sorted

2.4.3 Binary Search Trees

A Binary Search Tree is a special kind of tree which usually minimizes the cost of operation. For each node in a tree, the values of all left nodes are smaller than values of all right nodes. It is also known as the ordered or sorted binary tree. As it is a binary tree, a binary search tree obtains all properties of a binary tree, but it also has its own properties such as:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

As these algorithms are quite complex and can be written using both iterative and recursive statements, they are a perfect fit for this investigation.

2.4.3.1 Procedure

1. Insertion of randomized values in an array
2. Sorting the array in ascending order
3. Converting the sorted array to a balanced binary search tree using a user-defined function
4. Searching for the element in the binary search tree using the user-defined function

2.4.3.2 <u>Inserting nodes in a Binary Search Tree</u>

The following images demonstrate the logic behind insertion of nodes in a binary search tree:

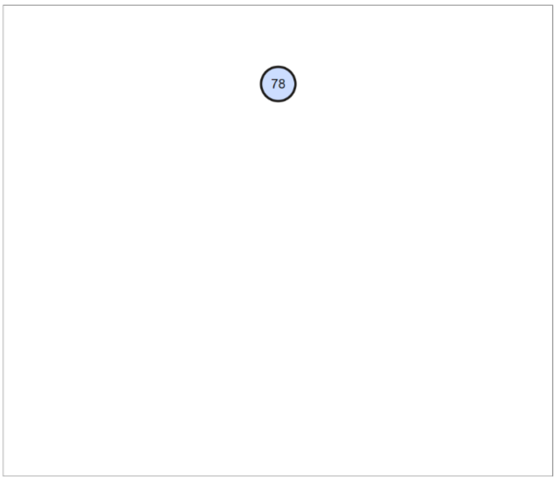

Figure 2.4.3.1: Root node, which is 78

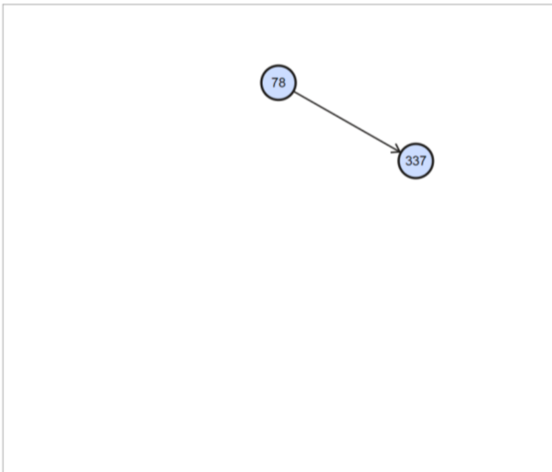

Figure 2.4.3.2: 337 placed on the right sub-tree of root node

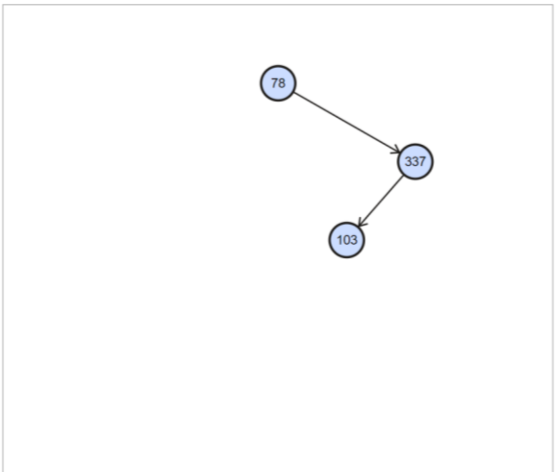

Figure 2.4.3.3: 103 placed on the left sub-tree of 337

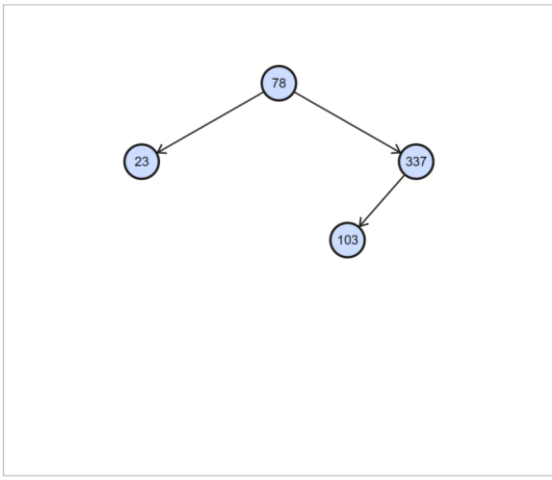

Figure 2.4.3.4: 23 is placed on the left sub-tree of 78

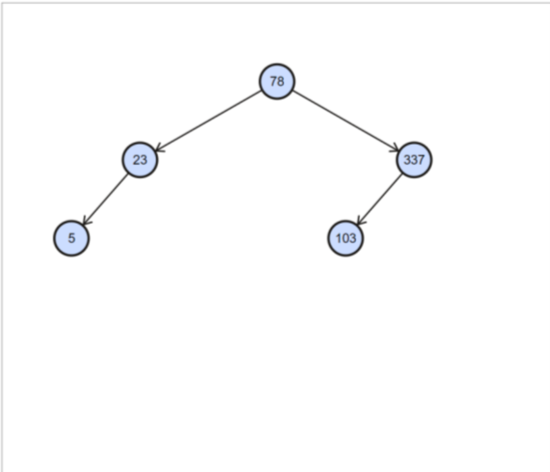

Figure 2.4.3.5: 5 is placed on the left sub-tree of 23

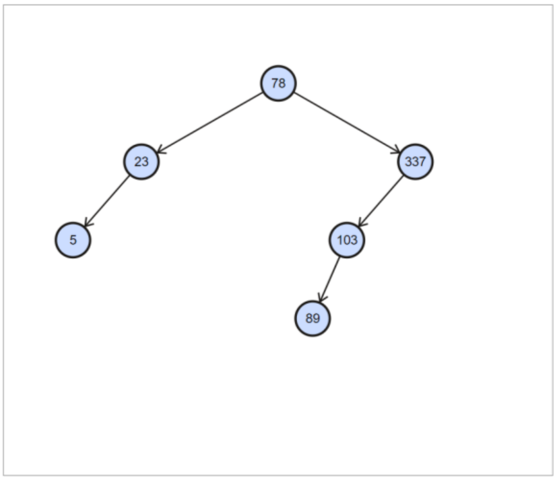

Figure 2.4.3.6: 89 is inserted on the left sub-tree of 103

2.4.3.3 <u>Searching for a node in a Binary Search Tree</u>

The following images demonstrate the logic behind searching for a node in a Binary Search Tree:

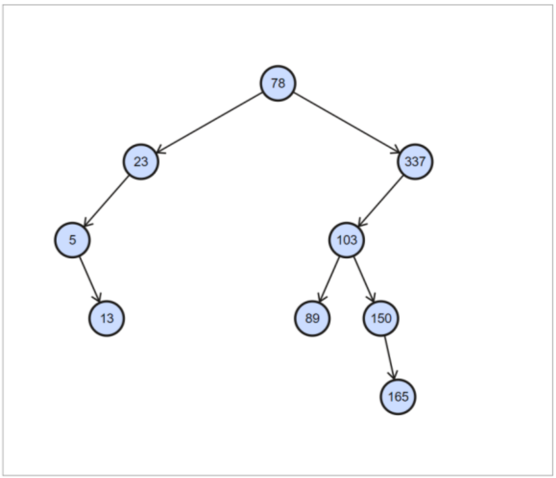

Figure 2.4.3.7: 89 is compared with 78

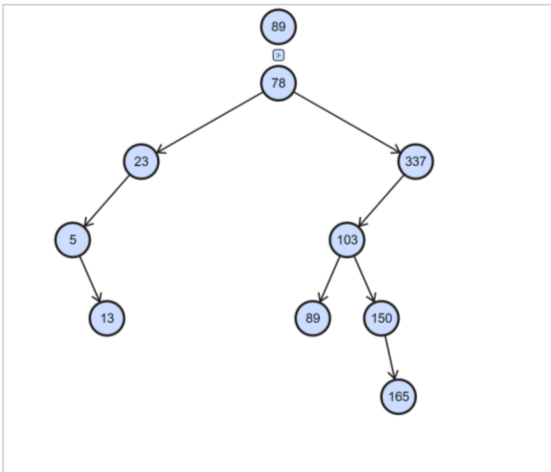

Figure 2.4.3.8: 89 > 78, right sub-tree of 78 is checked

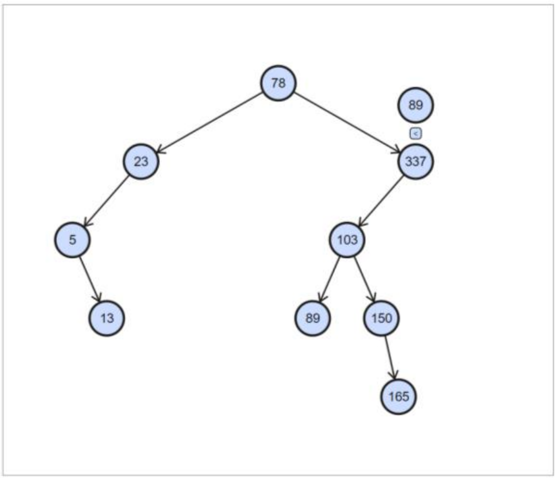

Figure 2.4.3.9: 89 < 337, left sub-tree of 337 is checked

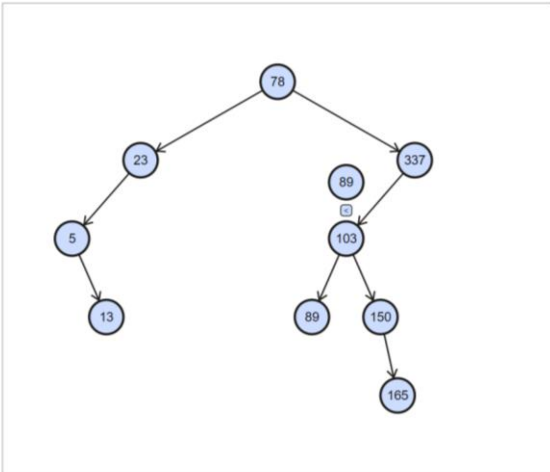

Figure 2.4.3.10: 89 < 103, left sub-tree of 103 checked

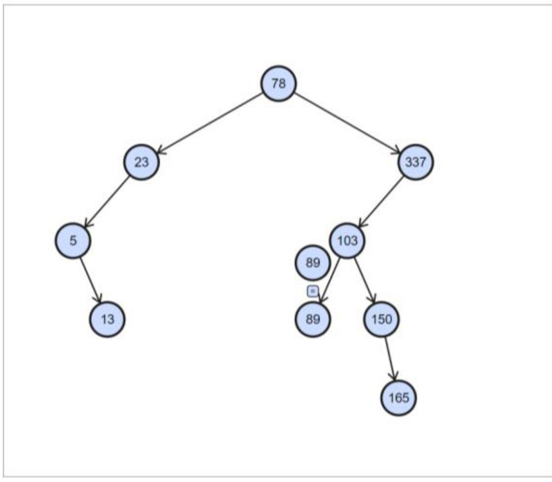

Figure 2.4.3.11: 89 = 89, node found

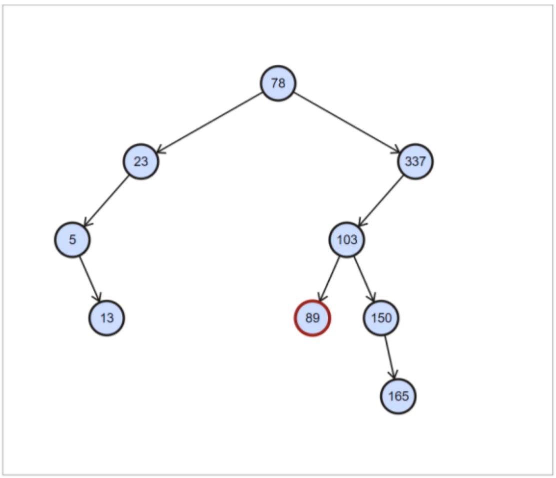

Figure 2.4.3.12: Value searched for is found, highlighted in a red border

3 Hypothesis

The theory of algorithms that will be used for this experiment have been discussed and explained in great detail in Section 2. Along with this, it is also important to identify the approach which is less time consuming by analysing the graphs obtained.

This experiment will determine the relationship between the **runtime**, y-axis, and the varied **set sizes**, x-axis. The set sizes for each algorithm will be varied to determine the relationship between the independent and dependent variables.

Recursion "overhead" is a concept that must be explained in order to form a strong hypothesis. The number of function calls are directly proportional to the execution times of the recursive programs. For each function call, there is a certain amount of "overhead" which takes up memory and resources. Each function call takes a small amount of time to be set up, implying that as the set size increases, the number of function calls increase. The time taken for function calls to be set up will increase, in turn, increasing the overall execution time of the recursive program.

I hypothesize that for Binary Search in an array, there will be a **logarithmic relationship** for both, the iterative and recursive approach. For sorting elements in an array using insertion sort, there will be a **polynomial relationship**. Finally, for searching in a Binary Search Tree, there will be a **logarithmic relationship**. I have postulated these relationships on basis of the time complexities of the respective algorithms, which have been explained in sections: Time Complexity – Binary Search, Time Complexity – Insertion Sort, Time Complexity – Binary Search Trees

I also believe that programs written using recursive statements will take **more** time to execute than those written using iterative statements. This is because recursive methods cause overheads of repeated function calls, leading to higher execution time, unlike iteration, which has no overhead of repeated function calls and a lower execution time.

4 Methodology

The independent, dependent, and controlled variables will be explained in this section.

4.1 Independent Variables

In this investigation, the independent variables are the differing sizes of the array for computing the algorithms.

For executing binary search, a range of numbers from 10000 to 100000 with an interval of 10000 between each set will be chosen.

For insertion sort, the amount of numbers to be sorted will be in a range from 3000 to 24000, with an interval of 3000 between each set of data. This is because of java's stack overflow error, which doesn't allow more than 26000 numbers to be sorted recursively as the allocated stack memory exceeds.

For binary search tree, the number of nodes that will be added to the tree will range from 100 to 1000 with an interval of 100 between each set of data.

These set sizes will ensure that the data recorded can be plotted on a graph and the relationship between the iterative and recursive can clearly be observed. Also, for binary search and binary search tree, the time taken to sort the numbers before conducting the search has been considered and hasn't been added to the actual runtime, hence, yielding accurate results.

4.2 Dependent Variables

For this experiment, runtime was the dependent variable. The time for each execution will be calculated by storing the system time in nanoseconds before and after the execution of the programs, and then the difference of these two will be the runtime. These calculations are quite precise and accurate as 10 trials will be taken for each data set. Also, the time calculated will be measured in nanoseconds, hence, giving a precise runtime value.

4.3 Controlled Variables

The variables in this experiment which will be kept constant are:

Variable	Description
Computer and the operating system used	Acer Nitro 5 – Windows 10 OS Processor – 2.3 GHz Intel core i5-8300h Memory (RAM) – 8.00 GB System type – 64 bit
IDE used	Eclipse IDE was used to perform all experiments and record all trials.
Runtime calculation	For each program, the runtime will be calculated using the same in-built classes and in-built methods

Table 4.3.1: Variables controlled for this experiment

5 Binary Search in an Array

5.1 Time Complexity – Binary Search

For this experiment, the array will be sorted first, and then Binary Search will be conducted. For any sorted array, the time complexity of Binary Search will be $O\log_2(n)$ where n is the number of elements. If the array is unsorted, the time taken to sort the array first will also have to be taken into consideration. For this, the time complexity changes to $O\,n\log_2(n)$, where n is the number of elements. However, for this experiment, the array will be sorted first, and subsequently, the time will be measured.

5.2 Processed Tables – Binary Search

Below shows the processed table for the sets that have been recorded. For the raw data, refer to Binary Search Time Trials.

Binary Search in an Array		
Set Size	**Iterative**	**Recursive**
10000	148710	201320
20000	130250	170640
30000	136560	174200
40000	133670	162780
50000	131900	158480
60000	99940	121180
70000	96560	115040
80000	94110	92270
90000	90350	88000
100000	84500	88510

Table 5.1: Iterative and Recursive runtime Trials – Binary Search in an Array

5.3 Expected Shape of the Graph

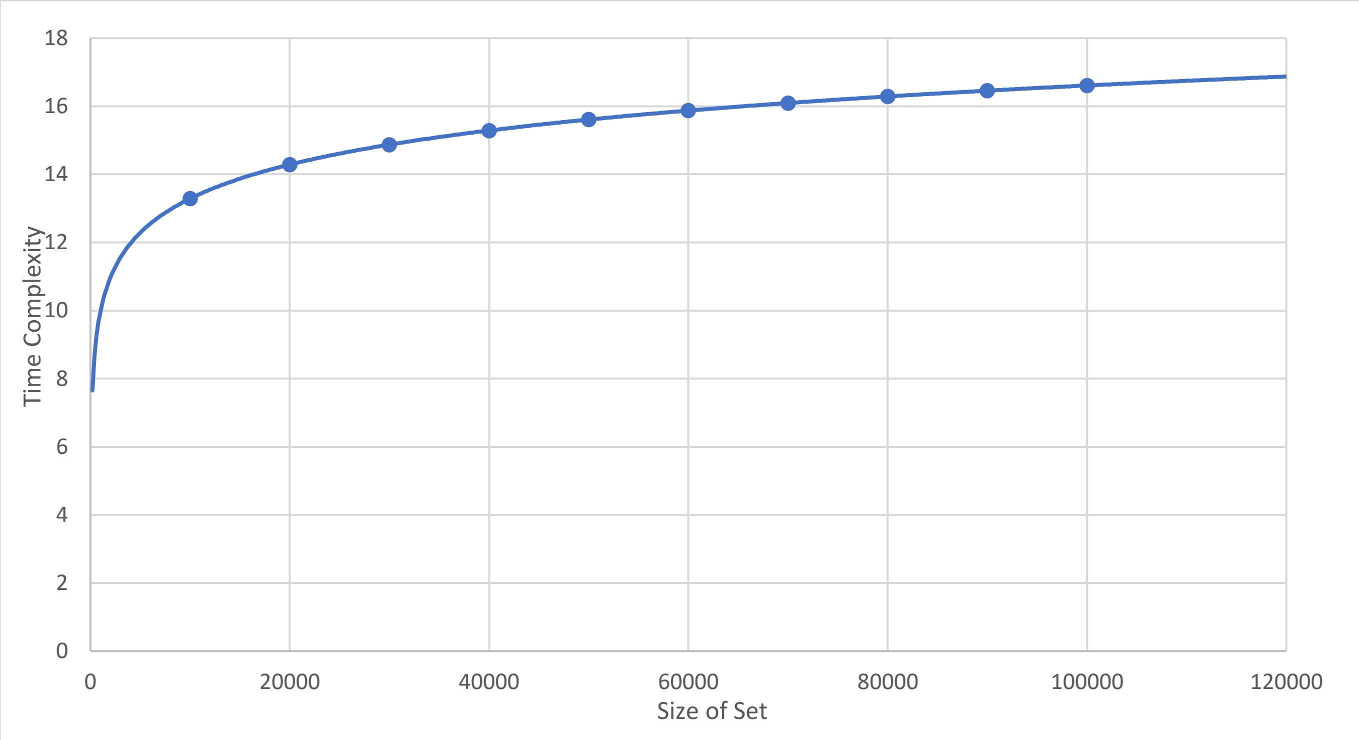

Figure 5.3.1: Expected shape of the logarithmic graph – Binary Search in an Array

5.4 Graph of Runtime against Set Size

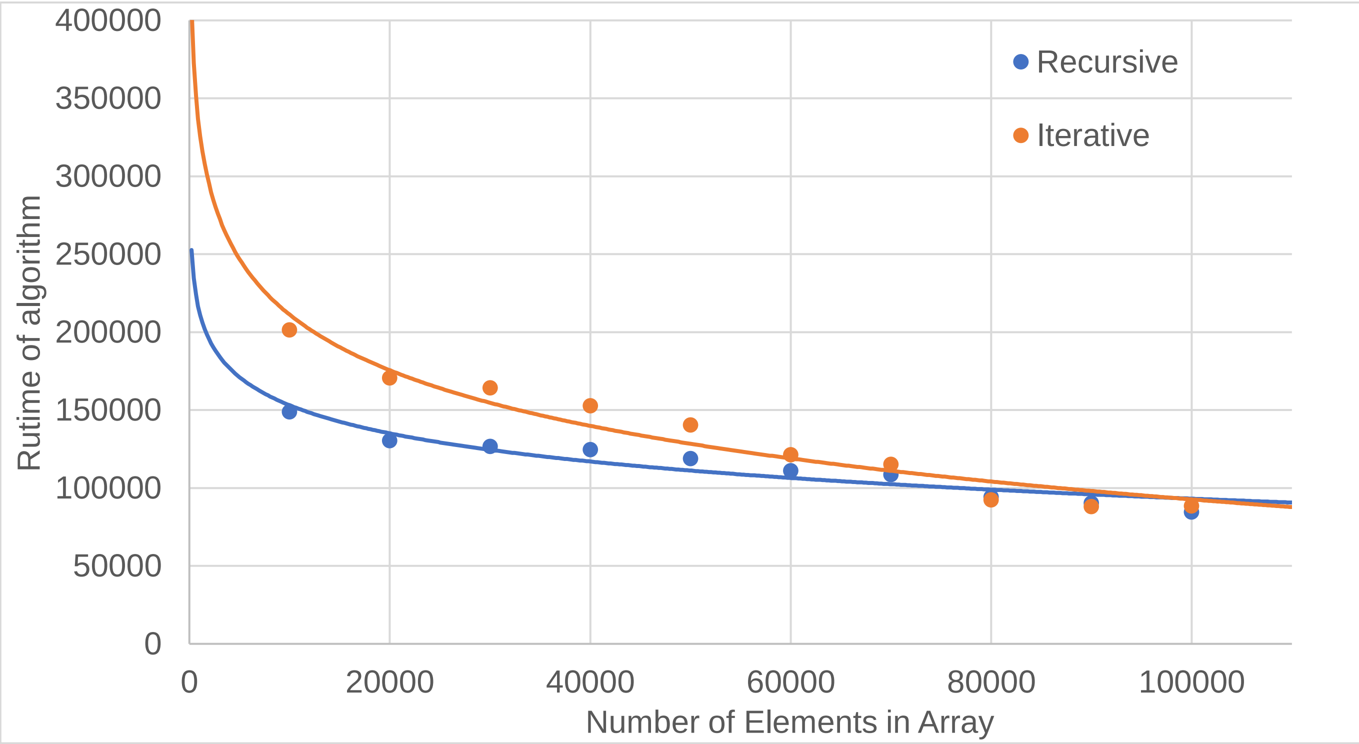

Figure 5.4.1: Iterative vs Recursive best-fit logarithmic graph – Binary Search in an Array

5.5 Discussion of Results Obtained

My first hypothesis of binary search, a logarithmic relationship between the **runtime** and **set size** for both approaches**,** has been shown to be untrue for all the set values. However, my second hypothesis of recursive programs having a greater program runtime than iterative programs has been shown to be partially correct.

The first hypothesis has been shown to be incorrect due to the dissimilarity in shape between the graph obtained and the expected graph. The expected graph is of $\log_2 n$, which is the average time complexity of binary search and the equations of the graphs obtained are:

Iterative	Recursive
$f(x) = -18042(log_2 x) + 392816$	$f(x) = -35688(log_2 x) + 685430$

Table 5.5.1: Equations of the Iterative and Recursive graphs obtained – Binary Search in an Array

Another notable feature of the graphs as well as the equations are their consistently decreasing gradient, indicating that increasing the set size will decrease the runtime. It was also observed that these graphs cross the x-axis for extremely large x-values, leading to negative runtime values, which is practically impossible as instead of the runtime decreasing with increasing set size, it should increase since increasing set size will lead to an increase in the number of comparisons that will be made using binary search, leading to an increase in runtime.

Differentiating between the recursive and iterative graph, my hypothesis has partly been supported by the results as the recursive graph seems to have a lesser runtime than the iterative graph before the set size of 98124, which is the intersection of the 2 curves, or set size where execution time is same, after which the iterative graph seems to have a lesser runtime than the recursive approach, supporting my hypothesis.

6 Insertion Sort

6.1 Time Complexity – Insertion Sort

Insertion Sort has a time complexity of $O(n^2)$ for both average and worst cases. Due to its time complexity, insertion sort is not recommended for sorting, however, the aspect of the favoured sorting algorithm will not be covered as it is out of scope of this essay.

6.2 Processed Tables – Insertion Sort

Below shows the processed table for the sets that have been recorded. For the raw data, refer to Insertion Sort Time Trials.

Insertion Sort		
Set Size	**Iterative**	**Recursive**
3000	5703200	12200110
6000	12617500	23038310
9000	24424140	29782290
12000	39315710	38824600
15000	57307050	49096000
18000	77990970	63004950
21000	104962760	77379190
24000	136523450	94362430

Table 6.2.1: Iterative and Recursive runtime Trials – Insertion Sort

6.3 Expected Graph – for iterative and recursive

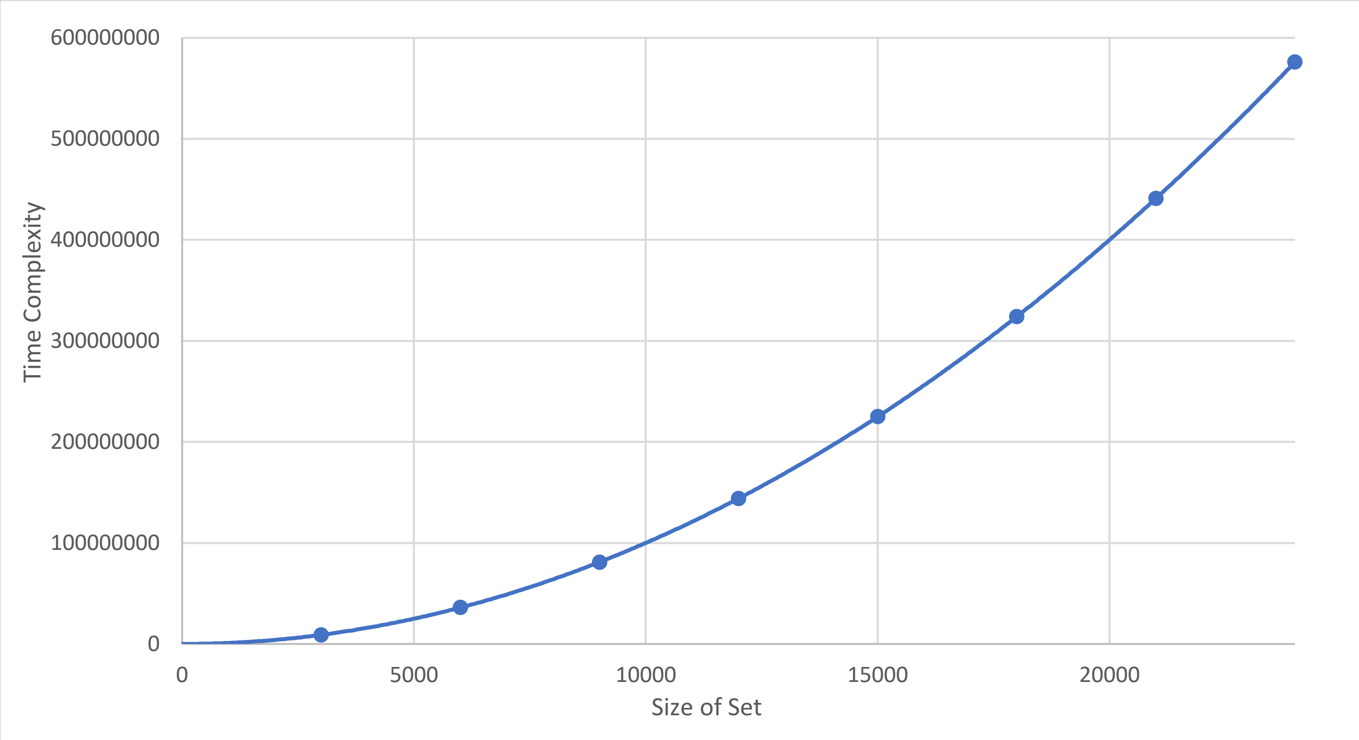

Figure 6.3.1: Expected shape of the polynomial graph – Insertion Sort

6.4 Graph of Runtime against Set Size

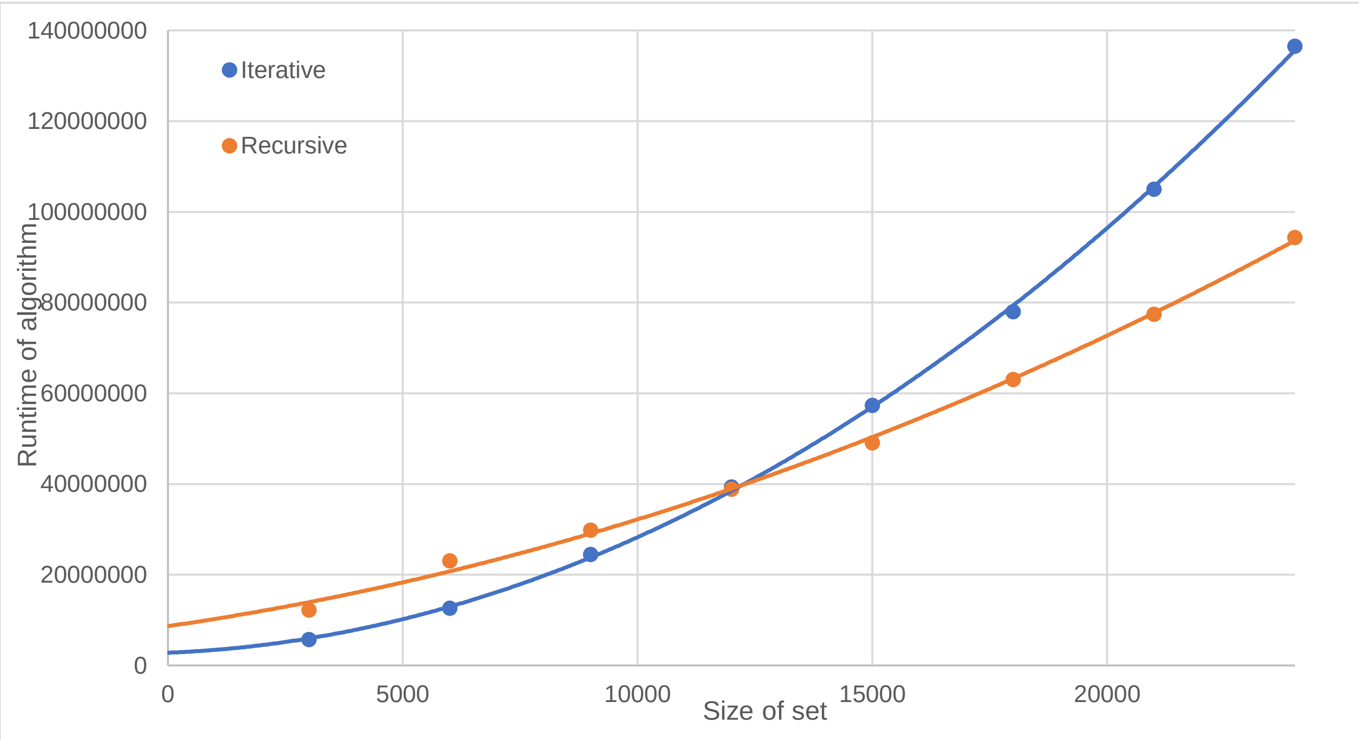

Figure 6.4.1: Iterative vs Recursive best-fit polynomial graph – Insertion Sort

6.5 Discussion of Results Obtained

My first hypothesis of insertion sort, a polynomial relationship between the runtime and set size for both approaches, has been shown to be true for all the set values as it is clearly supported by the results. My second hypothesis, however, of recursive programs having a greater runtime than the iterative approach is true, but to an extent.

The first hypothesis has been shown to be correct due to the shapes of iterative and recursive graph obtained being quite alike to the expected shape of graph. The polynomial relationship was apparent also due to equations of the best fit curve obtained for both approaches:

Iterative	Recursive
$f(x) = 0.2135x^2 + 410.81x + 3000000$	$f(x) = 0.0849x^2 + 1504.1x + 9000000$

Table 6.5.1: Equations of the Iterative and Recursive graphs obtained – Insertion Sort

As expected, the runtime will continually increase as the set size is increased. For large data, the runtime will be significantly large as well. However, as the two curves have the same runtime when the set size is 12295, this signifies that iterative is faster than the recursive approach, but only if the set size is less than 12295. A set size greater than this would result in the recursive approach having a lower runtime than the iterative approach, disproving a part of my second hypothesis as I hypothesized that the iterative approach will take a shorter amount of time to execute than the recursive approach.

Hence, it can be deduced through this analysis that for large data, using the recursive approach has a shorter runtime than the iterative approach.

7 Binary Search Trees

7.1 Time Complexity – Binary Search Trees

To search for a node in a Binary Search Tree, the average case of time complexity for searching in Binary Search Trees will be taken into consideration, which is $O\log_2(n)$, where '*n*' is the number of elements, same as binary search. The time complexity for Binary Search and Binary Search Trees is the same. However, the worst case for the time complexity is when the tree is completely unbalanced. In this case, the time complexity for Binary Search Tree becomes $O(n)$ as the node will be searched only once. For 'n' searches, the time complexity would become $O(n^2)$.

7.2 Processed Tables – Binary Search Trees

Below shows the processed table for the sets that have been recorded. For the raw data, refer to Searching in Binary Search Trees Time Trials.

Searching in a Binary Search Tree		
Trials	**Iterative**	**Recursive**
100	68400	51860
200	76300	72640
300	99740	98810
400	118690	120440
500	130390	130830
600	140490	139670
700	153710	160920
800	163080	166890
900	167560	173370
1000	177500	182250

Table 7.2.1: Iterative and Recursive runtime Trials – Searching in a Binary Search Tree

7.3 Expected Graph – for iterative and recursive

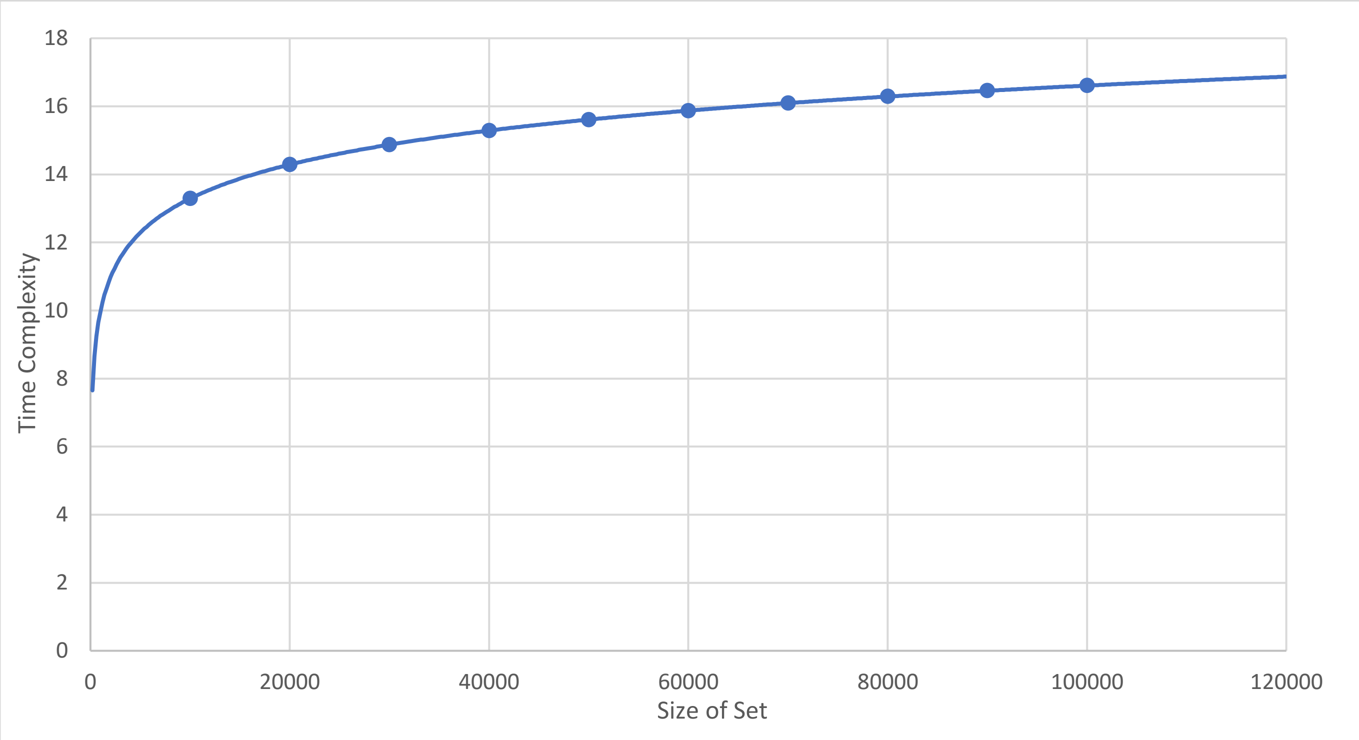

Figure 7.3.1: Expected shape of the polynomial graph – Searching in a Binary Search Tree

7.4 Best-fit curves obtained of Runtime against Set Size

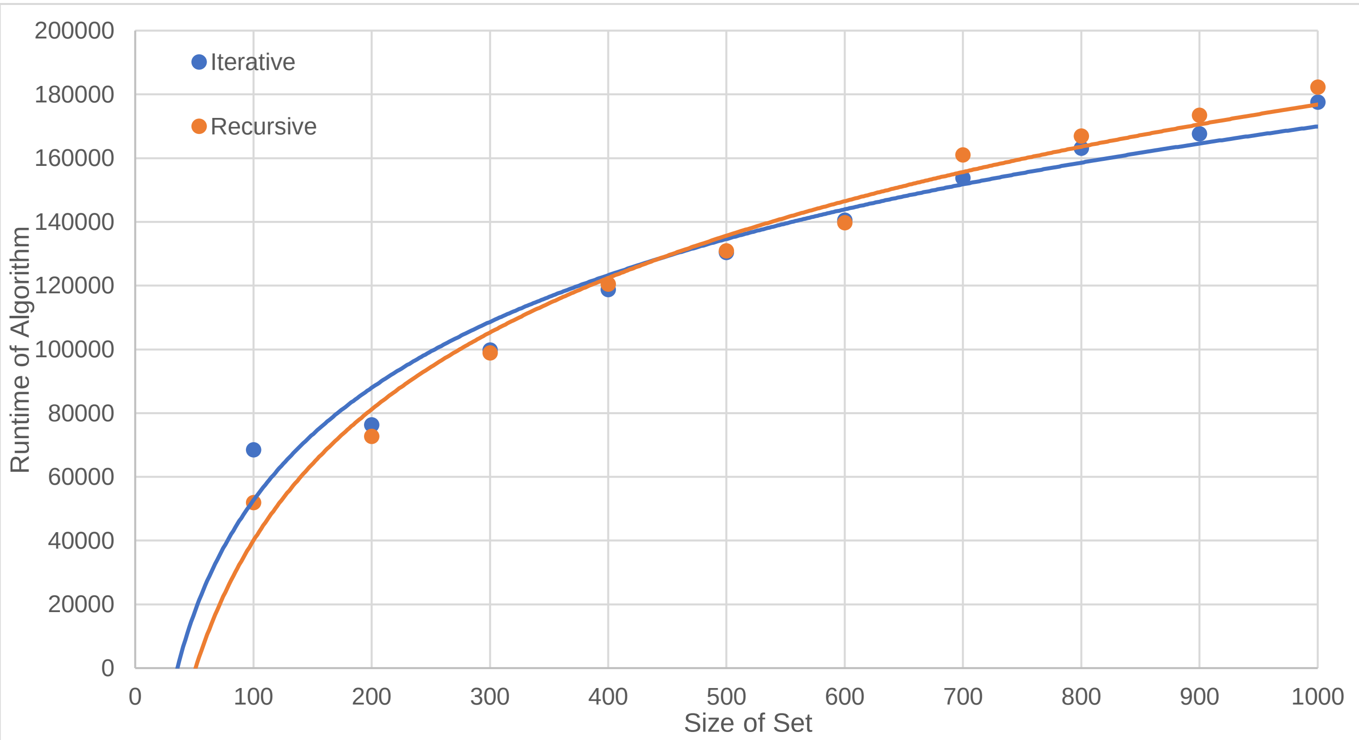

Figure 7.4.1: Iterative vs Recursive best-fit polynomial graph – Searching in a Binary Search Tree

7.5 Discussion of Results Obtained

My hypothesis that searching in a binary search tree will have a relationship such both approaches will have a logarithmic relationship has been proven to be true for all the set values as it is supported by the results obtained. My second hypothesis, however, of recursive programs having a greater runtime than the iterative approach is true, but to an extent.

The first hypothesis has been shown to be correct due to the shapes of iterative and recursive graph obtained being quite alike to the expected shape of graph. The polynomial relationship was apparent also due to equations of the best fit curve obtained for both approaches:

Iterative	Recursive
$f(x) = 35311(log_2 x) - 181963$	$f(x) = 41190(log_2 x) - 233648$

Table 7.5.1: Equations of the Iterative and Recursive graphs obtained – Searching in a Binary Search Tree

As expected, the runtime will constantly increase as the set size is increased. For large data, the runtime will be significantly large as well. However, as the two curves have the same runtime when the set size is 493, this signifies that recursive is less time-consuming than the iterative approach, but only if the set size is less than 493. A set size greater than this would result in the recursive method having a greater runtime than the iterative method.

It can be deduced through this analysis that for large data, using the iterative method has a shorter runtime than the recursive method.

8 Conclusion

The aim of this experiment was to apply the theory explained in section _ and write complex java programs such that the runtime of differing set sizes is recorded. Taking it further, using the data recorded, relationship between runtime and various set sizes of recursive and iterative algorithms were observed by plotting best fit curves according to their respective time complexities. Taking it further, this investigation also aimed at how the time-set size relationship differed for each recursive and iterative algorithm.

Conclusions reached for each algorithm from the results obtained have been summarized in Table 8.1:

Algorithm	**Conclusion**
Binary Search in an Array	Recursive method is more time-efficient than the iterative method for a set size less than 98124. For values greater than this, the iterative method is more time-efficient.
Insertion Sort	Iterative method is more time-efficient than the recursive method for a set size less than 12295. For values greater than this, the recursive method is more time-efficient.
Search in Binary Search Tree	Recursive method is more time-efficient than the iterative method for a set size less than 493. For values greater than this, the iterative method is more time-efficient.

Table 8.1: Final conclusions reached for the runtime after a thorough analysis of each algorithm

To answer the research question of my essay, my answer would be that the runtime performance of an iterative or recursive is dependent on the set sizes, and also the algorithm being investigated. How the values are inserted were taken care of by first storing them in an array, sorting them, and then measuring the runtime. With the final conclusions reached in Table 8.1, the recursive approach for insertion sort proved to be more time-efficient with a larger set size when compared to the iterative approach, which is more time-efficient with a smaller set size. However, the iterative approach for binary search and searching in binary search trees proved to be more time-efficient with a larger set size when compared to the recursive approach, which is more efficient with a smaller set size, hence proving my hypothesis to be partially right.

To conclude, iterative and recursive algorithms compare in terms of their runtime to a great extent, as evident in Table 8.1, but the algorithm being experimented on is also a great factor as it was found that the recursive method for insertion sort is more time-efficient, but the iterative method for binary search and searching in binary search trees is more time-efficient.

9 Limitations

While taking trials, there was a possibility for the runtime to be affected due to applications running in the background as this could cause the runtime to increase. Thus, this was taken care of by taking the trials again and closing all applications in the background.

Also, due to randomized values being added to the binary search trees, I realized that initially, the binary search trees formed were unbalanced, which can lead to erroneous data collection.

To prevent this, I first stored the randomized values in an array, and then this array was sorted. I added a function which would convert a sorted array to a balanced binary search tree.

Moreover, the equations of the best fit curves for Binary Search and Binary Search Trees were generated in terms of natural logarithm, and they had to be in terms of log base 2. Hence, they converted to log with base 2 by changing the base.

Another problem I faced was that I couldn't use a set size greater than around 26000 for recursive insertion sort due to the StackOverFlow error. This is because of the limited stack memory allocated by the java virtual machine which had been exceeded. Figure 9.1 shows the output received on an input of 26000:

```
How many numbers do you want to sort?
26000
Exception in thread "main" java.lang.StackOverflowError
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
	at InsertionSortRec.insertionSort(InsertionSortRec.java:19)
```

Figure 9.1: StackOverFlowError in Java; allocated stack memory exceeded

10 Bibliography

Bolaji. *Iteration vs Recursion*. 2018. September 2019. https://medium.com/backticks-tildes/iteration-vs-recursion-c2017a483890

codility. *Time Complexity*. 2015. Codility Limited. 24 February 2020. https://codility.com/media/train/1-TimeComplexity.pdf

CS, Cornell. *Recursion*. 11 May 1998. Cornell University - CS. 4 March 2020. http://www.cs.cornell.edu/info/courses/spring-98/cs211/lecturenotes/07-recursion.pdf

Differences, Tech. *Difference between Recursion and Iteration*. 30 May 2016. Tech Differences. 22 February 2020. https://techdifferences.com/difference-between-recursion-and-iteration-2.html

Elbou, Mohamed Ould. *Overhead of Recursion*. 2008. '. 21 January 2020. http://www.cs.iit.edu/~cs561/cs331/recursion/recursionoverhead.html

Gabriel Alves, Karleigh Moore, Adonis Ampongan, and 3 others contributed. *Insertion Sort*. 2020. Brilliant.org. 2 March 2020. https://brilliant.org/wiki/insertion/

GeeksforGeeks. *Binary Search Tree Data Structure*. 2019. 12 October 2019. < https://www.geeksforgeeks.org/binary-search-tree-data-structure/ >.

—. *Iterative Searching Binary Search Tree*. 2019. 12 October 2019. < https://www.geeksforgeeks.org/iterative-searching-binary-search-tree/>.

—. *Java Program for Binary Search Recursive and Iterative*. 2019. 17 September 2019. < https://www.geeksforgeeks.org/java-program-for-binary-search-recursive-and-iterative/ >.

Liang, Y. Daniel. *Binary Search Animation using JavaScript and Processing.js*. n.d. Armstrong CS. 3 March 2020. http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

—. *Insertion Sort Animation using JavaScript and Processing.js*. n.d. Armstrong CS. 2 March 2020. http://cs.armstrong.edu/liang/animation/web/InsertionSort.html

Melezinek, Jakub. *Binary Tree Visualizer*. n.d. CTU FIT Web and multimedia. 3 March 2020. http://btv.melezinek.cz/binary-search-tree.html

Techie Delight. *Insertion Sort Algorithm*. 2018. 18 September 2019. < https://www.techiedelight.com/insertion-sort-iterative-recursive/ >.

What is Recursion. SparkNotes. 2019. Barnes & Noble. 3 February 2020. https://www.sparknotes.com/cs/recursion/whatisrecursion/section1/page/3/

11 Appendix

11.1 Binary Search Time Trials

11.1.1 Iterative

Iterative Binary Search										
Set Size	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{Avg}**
10000	145000	133600	208200	148600	216900	123200	146300	125900	117000	122400	148710
20000	107000	127700	109600	163400	108500	125000	165600	149500	128300	117900	130250
30000	112500	124500	125200	139200	118900	132800	180800	169100	151700	110900	136560
40000	135700	127300	144900	166700	129900	130200	136100	125000	116000	124900	133670
50000	143300	158800	121300	125300	119900	128300	147400	121300	134700	118700	131900
60000	150400	102300	111800	93900	89900	73500	93800	99100	90200	94500	99940
70000	97400	108200	102100	109300	100200	92500	92400	88500	84500	90500	96560
80000	94300	81500	83200	93400	106000	106800	110900	115800	72000	77200	94110
90000	84000	86500	86000	89000	102000	79500	83500	87600	103400	102000	90350
100000	92100	70500	89400	80600	108200	101000	71000	79200	81200	71800	84500

11.1.2 Recursive

Recursive Binary Search											
Set Size	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{Avg}**
10000	167800	263000	176000	191700	165600	179000	211300	290500	193500	174800	201320
20000	174900	152500	161200	160700	212100	198700	158400	152800	167300	167800	170640
30000	253600	170100	155600	161400	152000	163100	157500	184600	177200	166900	174200
40000	156800	164900	173000	159600	155600	158900	181500	155100	164500	157900	162780
50000	158000	187400	158100	163100	155700	182100	157100	111000	161500	150800	158480
60000	116100	129900	141300	169500	103000	116000	113000	96000	112100	114900	121180
70000	110100	108400	119200	107600	109200	135100	108600	106700	112100	133400	115040
80000	103000	82100	92200	116300	89400	85600	90400	90900	92800	94900	92270
90000	92700	82400	87800	100100	81800	96800	80800	84300	83300	92200	88000
100000	97900	80200	79900	79200	102100	75600	90800	88500	90900	92800	88510

11.2 Insertion Sort Time Trials

11.2.1 Iterative

Iterative Insertion Sort											
Set Size	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{Avg}**
3000	5571700	5447600	5665500	6044400	5757500	5757500	5447700	6045000	5436300	5858800	5703200
6000	11575800	13833000	13833000	12102800	13990900	11581800	11658400	11440800	11843500	14315000	12617500
9000	21747900	24014900	24387000	25841800	25841800	24440400	24720400	24081800	24307100	24858300	24424140
12000	39360900	40355200	39516800	40328700	38105800	39662800	38210000	40441000	37674100	39501800	39315710
15000	58175000	56086600	61133800	57078200	60767400	53886700	56640900	57826800	57323900	54151200	57307050
18000	82927100	76111200	75508700	75565900	76402700	84785500	77409300	76856400	78588600	75754300	77990970
21000	114401400	106567900	101083700	101451300	103749700	101740000	104452800	106904600	103935700	105340500	104962760
24000	136984900	131234000	142141900	142683000	134931800	132206000	135651900	133879700	142985300	132536000	136523450

11.2.2 Recursive

Recursive Insertion Sort											
Set Size	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{Avg}**
3000	12339200	12679300	11393100	12361600	12313300	11986900	12821300	12220600	12053600	11832200	12200110
6000	21855800	24689600	22978500	25043600	23035100	22439700	22873600	22394500	22429800	22642900	23038310
9000	29435100	30289100	27729200	33855600	30124800	29482300	29466300	30008600	28217700	29214200	29782290
12000	37457900	46288500	37815400	37267600	37192200	42843700	37841200	37886700	37002300	36650500	38824600
15000	49182600	49068100	51009700	47589200	48143000	48128500	48622100	48826200	48834300	51556300	49096000
18000	66255200	64227700	60327000	61600600	61196100	63015900	60541700	68815300	62473000	61597000	63004950
21000	77266700	76983200	78279300	76245800	77808900	76925300	76661300	77643200	78203200	77775000	77379190
24000	94556800	104556800	94553700	98956800	94516200	99256800	90556800	89556800	86556800	90556800	94362430

11.3 Searching in Binary Search Trees Time Trials

11.3.1 Iterative

Recursive Binary Search Tree										
Set Size	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{Avg}**
100	58900	46200	47300	57300	50700	48400	51200	52100	57400	49100	51860
200	116400	86500	62700	81000	58800	59600	58200	76900	60300	66000	72640
300	95900	89300	107100	63900	88200	125300	112400	145800	72500	87700	98810
400	93400	168100	152000	102100	92700	97700	155500	123500	125300	94100	120440
500	144800	85900	196300	118300	111500	150200	135700	72800	140600	152200	130830
600	168000	99200	189200	132600	127500	181100	130300	127400	115800	125600	139670
700	142500	210200	180500	112200	129800	203200	200100	173400	94300	163000	160920
800	222500	143900	182200	115400	136700	191600	193600	177400	139800	165800	166890
900	169300	203900	138000	195900	247000	123800	199500	138600	173500	144200	173370
1000	175200	182500	195700	207900	203300	144800	214400	115500	199900	183300	182250

11.3.2 Recursive

Iterative Binary Search Tree											
Set Size	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**TAvg**
100	65300	131400	122300	66300	67600	59000	82900	58800	58300	82100	68400
200	79900	59800	63700	67800	83500	62800	65600	62900	63400	63600	76300
300	74900	126400	90500	117500	91500	62600	100500	110200	64300	69000	99740
400	69000	111900	97500	92800	89700	88300	89100	179000	118900	70700	118690
500	119900	86700	157600	116200	79300	127800	108900	86300	84200	127000	130390
600	151800	130800	118700	114300	100100	195700	184900	124800	113300	170500	140490
700	142000	145100	200200	121100	186400	181800	135400	142100	148000	195000	153710
800	233900	128800	187800	101100	131100	173400	134100	138100	204400	198100	163080
900	144600	180500	197700	153100	195600	150500	140700	140700	159200	143000	167560
1000	163800	166600	301100	198800	151500	262700	140500	199500	183000	167500	177500

11.4 Binary Search Java Code

11.4.1 Iterative

```
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
public class BinarySearchIte{

	public static void BSIte(int array[], int first, int last, int
middle, int searchedElement) {
		while (first <= last) // continues to run until the first
		{
			if (array[middle] < searchedElement) // if inputValue >
middleElement --> left half of array to be ignored
				first = middle + 1;
			else if (array[middle] == searchedElement) // if
inputValue = middleElement --> element found in the middle
			{
				System.out.println(searchedElement + " found at
location " + (middle + 1));
				break;
			}
			else // if inputValue < middleElement --> right half of
array to be ignored
				last = middle - 1;
				middle = (first + last)/2;
		}
		if (first > last)
			System.out.println(searchedElement + " isn't present in
the list\n");
	}

	public static void main(String[] args){
		Scanner input = new Scanner(System.in);
		Random rand = new Random();
		int first, last, middle, a, b;
		int noOfElements = 10; // variable declaration for the
set size, this value was changed for differing set sizes
		int arr[] = new int[noOfElements];

		for (a = 0; a < noOfElements; a ++)
		{
			arr[a] =  rand.nextInt(1000000); // insertion of
randomized values
			System.out.println(arr[a]);
		}

		System.out.println("");
```

```
            System.out.println("Sorted list of elements:");
            System.out.println("");

            Arrays.sort(arr); // in-built function to sort the array

            for (b = 0; b < noOfElements; b ++)
            {
                System.out.println(arr[b]);
            }
            System.out.println("Search for an element from the list:
");
            int search = input.nextInt(); // input of element to be
searched

            first  = 0; // index of first array element
            last   = noOfElements - 1; // index of last array
element
            middle = (first + last)/2;
            long startTime = System.nanoTime(); // start time of the
algorithm
            BSIte(arr, 0, noOfElements - 1, (first+last)/2, search);
            long endTime = System.nanoTime(); // end time of the
algorithm
            long durationInNano = endTime - startTime; // runtime of
the algorithm
            System.out.println("Time taken to execute this program:
" + durationInNano);
    }
}
```

11.4.2 Output of the Iterative Program

Total numbers in the array: 1000

```
995005
997883
998165
998771
998911
999216
Search for an element from the list:
999216
999216 found at location 1000
Time taken to execute this program: 84200
```

11.4.3 Recursive

```
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
class BinarySearchRec {

    int binarySearch(int arr[], int l, int r, int x) {
        if (r >= l) {
            int mid = l + (r - l) / 2;
            if (arr[mid] == x)
                return mid; // returns the middle element of the 
array that is continuously divided
            if (arr[mid] > x)
                return binarySearch(arr, l, mid - 1, x);
                return binarySearch(arr, mid + 1, r, x); // function 
calling itself, hence, recursive
            }
        return -1; // returns -1 if element not present
        }

    public static void main(String args[]) {
           BinarySearchRec ob = new BinarySearchRec();
            Random randomInt = new Random();

            int noOfElements = 10000; // declaring variable for 
number of elements
            int arr[] = new int[noOfElements];
            for (int m = 0; m < noOfElements; m ++)
            {
                arr[m] = randomInt.nextInt(1000000); // randomizing 
values and storing them in array
            }

            Arrays.sort(arr); // sorting array

            for (int l = 0; l < noOfElements; l ++)
                System.out.println(arr[l]);

            int n = arr.length; // obtaining array length

            Scanner input = new Scanner(System.in);
            int elementSearched = input.nextInt();

            long startTime = System.nanoTime(); // start time of 
algorithm
            int result = ob.binarySearch(arr, 0, n - 1, 
elementSearched);
            if (result == -1)
```

```
                System.out.println("Element not present");
            else
                System.out.println("Element found at index " +
(result+1));
            long endTime = System.nanoTime(); // end time of
algorithm
            long durationInNano = endTime - startTime; // run time
of algorithm
            System.out.println("Time taken to execute this program:
" + durationInNano);
    }
}
```

11.4.4 Output of the Recursive Program

Total numbers in the array: 1000

```
993001
995178
996235
996594
997008
997116
998844
998844
Element found at index 1000
Time taken to execute this program: 94800
```

11.5 Insertion Sort Java Code

11.5.1 Iterative

```
import java.util.Arrays;
import java.util.Scanner;
import java.util.Random;
import java.util.concurrent.TimeUnit;
class InsertionSortIte
{
    public static void insertionSort(int[] arr) // method for
iterative insertion sort
    {
        for (int i = 1; i < arr.length; i++)
        {
            int value = arr[i];
            int j = i;
            while (j > 0 && arr[j - 1] > value) // use of loops,
covered in theory
            {
                arr[j] = arr[j - 1];
                j--;
            }
            arr[j] = value;
        }
    }
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.println("How many numbers do you want to sort?");
// input for total numbers to be sorted
        int j = input.nextInt();
        Random randomInt = new Random();
        int[] arr = new int[j];
        for (int i = 0; i < j; i ++)
        {
            arr[i] = randomInt.nextInt(100000000); // generating and
storing random numbers in an array
        }
        long startTime = System.nanoTime(); // start time of
algorithm
        insertionSort(arr); // calling the insertion sort function
        long endTime = System.nanoTime(); // end time of algorithm
        long durationInNano = endTime - startTime; // runtime of
algorithm
        System.out.println("Time taken for the numbers to be sorted:
" + durationInNano);
    }
}
```

11.5.2 Output of the Iterative Program

Total numbers sorted: 1000

```
How many numbers do you want to sort?
1000
Time taken for the numbers to be sorted: 1777500
```

11.5.3 Recursive

```
import java.util.Arrays;
import java.util.Scanner;
import java.util.concurrent.TimeUnit;
import java.util.Random;
class InsertionSortRec
{
    public static void insertionSort(int[] arr, int i, int n) //
recursive method for insertion sorts
    {
        int value = arr[i];
        int j = i;
        while (j > 0 && arr[j - 1] > value)
        {
            arr[j] = arr[j - 1];
            j--;
        }

        arr[j] = value;
        if (i + 1 <= n) {
            insertionSort(arr, i + 1, n); // calling the function
inside the function, hence, recursive
        }
    }

    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.println("How many numbers do you want to sort?");
        int j = input.nextInt();
        Random randomInt = new Random();
        int[] arr = new int[j];
        for (int i = 0; i < j; i ++)
        {
            arr[i] = randomInt.nextInt(100000); // randomizing
numbers
        }
        long startTime = System.nanoTime(); // start time of
algorithm
```

```
        insertionSort(arr, 1, arr.length - 1); // calling the
recursive insertion sort function
        long endTime = System.nanoTime(); // end time of algorithm
        long durationInNano = endTime - startTime; // runtime of
algorithm
        System.out.println("Time taken for the numbers to be sorted:
" + durationInNano);
    }
}
```

11.5.4 <u>Output of the Recursive Program</u>

Total numbers sorted: 1000

```
How many numbers do you want to sort?
1000
Time taken for the numbers to be sorted: 1088900
```

11.6 Searching in Binary Search Trees Java Code

11.6.1 Iterative

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;
class Node1 // node class
{
	int data;
	Node1 left = null, right = null;

	Node1(int data) {
		this.data = data;
	}
}
class BinarySearchTreeIte
{	static Node1 root;
	// Recursive function to insert a key into BST
	public static Node1 insert(Node1 root, int key)
	{
		// if the root is null, create a new node and return it
		if (root == null) {
			return new Node1(key);
		}

		// if given key is less than the root node, recur for
left subtree
		if (key < root.data) {
			root.left = insert(root.left, key);
		}

		// if given key is more than the root node, recur for
right subtree
		else {
			root.right = insert(root.right, key);
		}

		return root;
	}

	// Iterative function to search in given BST
	public static void searchIterative(Node1 root, int key) {
		// start with root node
		Node1 curr = root;

		// pointer to store parent node of current node
		Node1 parent = null;
```

```
            // traverse the tree and search for the key
            while (curr != null && curr.data != key)
            {
                  // update parent node as current node
                  parent = curr;

                  // if given key is less than the current node, go to
left subtree
                  // else go to right subtree
                  if (key < curr.data) {
                        curr = curr.left;
                  } else {
                        curr = curr.right;
                  }
            }

            // if key is not present in the key
            if (curr == null) {
                  System.out.print("Key Not found");
                  return;
            }

            if (parent == null) {
                  System.out.println("The node with key " + key + " is
root node");
            }
            else if (key < parent.data) {
                  System.out.println("Given key is left node of node
with key "
                                          + parent.data);
            }
            else {
                  System.out.println("Given key is right node of node
with key "
                                          + parent.data);
            }
      }

      Node1 sortedArrayToBST(int arr[], int start, int end) {

         if (start > end) {
             return null;
         }

         /* Get the middle element and make it root */
         int mid = (start + end) / 2;
         Node1 node = new Node1(arr[mid]);

         /* Recursively construct the left subtree and make it
```

```
         left child of root */
        node.left = sortedArrayToBST(arr, start, mid - 1);

        /* Recursively construct the right subtree and make it
         right child of root */
        node.right = sortedArrayToBST(arr, mid + 1, end);

        return node;
    }

      // Search given key in BST
      public static void main(String[] args)
      {
            BinarySearchTreeIte tree = new BinarySearchTreeIte();
            Random random = new Random();
            Scanner input = new Scanner(System.in);
            System.out.println("Enter no. of elements: ");
            int noOfElements = input.nextInt(); // // input for total
number of nodes in the tree
            int array[] = new int[noOfElements];
            int k = 0;
            for (int i = 0; i < noOfElements; i++) {
                  k = random.nextInt(10000);
                  array[i] = k; // storing in an array
            }
            System.out.println("Enter element: ");
            int find = input.nextInt(); // input element to be
searched
            Arrays.sort(array);

            for (int i = 0; i < noOfElements; i++) {
                 root = insert(root, array[i]); // insertion of nodes
in BST
            }
            tree.sortedArrayToBST(array, 0, noOfElements-1); //
changing to a balanced BST
            long startTime = System.nanoTime(); // start time of
algorithm
            searchIterative(root, find);  // iterative search in
binary search tree
            long endTime = System.nanoTime(); // end time of
algorithm
        long durationInNano = endTime - startTime; // runtime of
algorithm
        System.out.println("Time taken for the node to be searched:
" + durationInNano);
      }
}
```

11.6.2 Output of the Iterative Program

Total nodes in the tree: 1000

```
7946
1742
9874
1451
1850
3686
Enter element:
3686
Given key is right node of node with key 3676
Time taken for the node to be searched: 105300
```

11.6.3 Recursive

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

class Node2 {

    int data;
    Node2 left, right;

    Node2(int d) {
        data = d;
        left = right = null;
    }
}

// A binary tree node
class BinarySearchTreeRecursively {

    static Node2 root;

    /* A function that constructs Balanced Binary Search Tree
     from a sorted array */
    Node2 sortedArrayToBST(int arr[], int start, int end) {

        /* Base Case */
        if (start > end) {
            return null;
        }

        /* Get the middle element and make it root */
        int mid = (start + end) / 2;
        Node2 node = new Node2(arr[mid]);
```

```
        /* Recursively construct the left subtree and make it
         left child of root */
        node.left = sortedArrayToBST(arr, start, mid - 1);

        /* Recursively construct the right subtree and make it
         right child of root */
        node.right = sortedArrayToBST(arr, mid + 1, end);

        return node;
    }

 // Recursive function to search in given BST
      public static void search(Node2 root, int key, Node2 parent)
      {
            // if key is not present in the key
            if (root == null)
            {
                  System.out.print("Key Not found");
                  return;
            }

            // if key is found
            if (root.data == key)
            {
                  if (parent == null) {
                        System.out.println("The node with key " + key
+ " is root node");
                  }

                  else if (key < parent.data) {
                        System.out.println("Given key is left node of
node with key "
                                                                  +
parent.data);
                  }
                  else {
                        System.out.println("Given key is right node of
node with key "
                                                                  +
parent.data);
                  }

                  return;
            }

            // if given key is less than the root node, recur for
left subtree
            // else recur for right subtree
```

```
            if (key < root.data) {
                  search(root.left, key, root);
            }
            else {
                  search(root.right, key, root);
            }
      }

    public static void main(String[] args) {
        BinarySearchTreeRecursively tree = new
BinarySearchTreeRecursively();
        Random random = new Random();
        int noOfElements = 1000; //number of nodes in the tree
        int array[] = new int[noOfElements]; // array of size
noOfElements declared
            int k = 0;
            for (int i = 0; i < noOfElements; i++) {
                  k = random.nextInt(10000);
                  array[i] = k; // inserting random values in an array
                  System.out.println(k);
            }

            Arrays.sort(array); // sorting array

        root = tree.sortedArrayToBST(array, 0, noOfElements - 1); //
sorted array to balanced BST
        Scanner input = new Scanner(System.in);
        System.out.println("Enter an element to be searched");
        int find = input.nextInt(); // element to be searched input
        long startTime = System.nanoTime(); // start time of
algorithm
        search(root, find, null); // calling the recursive function
to search for node
        long endTime = System.nanoTime(); // end time of algorithm
        long durationInNano = endTime - startTime; // runtime of
algorithm
        System.out.println("Time taken for the node to be searched:
" + durationInNano);
    }
}
```

11.6.4 Output of the Recursive Program

Total nodes in the tree: 1000

```
3892
9143
3486
9999
2665
3709
Enter an element to be searched
3709
Given key is right node of node with key 2492
Time taken for the node to be searched: 84900
```