Fixed Recursive Binary Search

```java
import java.util.Arrays;

    public class RecBinarySearch {

        public int search(int[] a, int e) {
            return search(a, e, 0, a.length - 1);
        }

        private int search(int[] a, int e, int left, int right) {
            if (left > right) {
                return -1;
            }

            int midpoint = left + (right - left) / 2;

            if (e == a[midpoint]) {
                return midpoint;
            } else if (e < a[midpoint]) {
                return search(a, e, left, midpoint - 1);
            } else {
                return search(a, e, midpoint + 1, right);
            }
        }
}
```

Breakdown with Comments

```java
import java.util.Arrays;

    public class RecBinarySearch {

        public int search(int[] a, int e) {
            if(a == null) {
                return -1;
            }
        //This if statement is used to check if the input array a is null

            return search(a, e, 0, a.length - 1);
        }
```

```java
        private int search(int[] a, int e, int left, int right) {
        //Takes 4 arguments just like psuedocode.
        //a is the array we are searching through
        //e is our target integer
        //left is the leftmost(0) index in array a
        //right is the rightmost(a.length - 1) index in array a
        //left and right are used to act as placeholder for length of array a
```

```java
            if (left > right) {
                return -1;
            }
        //left is greater than right if element is not in the array
        //copied directly from psuedocode
```

```java
            int midpoint = left + (right - left) / 2;
        //used to calculate midpoint
        //same as psuedocode
```

```java
           if (e == a[midpoint]) {
               return midpoint;
           }
        //first condition
        //if the target value is equal to the element in the middle index return index
        //in this case the index is the midpoint
        //if the element is in the array then this condition will always be met
```

```java
            else if (e < a[midpoint]) {
                return search(a, e, left, midpoint - 1);
            }
        //second condition
        //if target is less than value at midpoint
        //recursively calls the function, updating the rightmost index to the midpoint - 1
```

```
46              else {
47                  return search(a, e, midpoint + 1, right);
48              }
49          //third condition
50          //if target is greater than value at midpoint
51          //recursively calls the function, updating the leftmost index to the midpoint + 1
52
53          }
54  }
```

Testing Time Complexity

```
27     public double timeFind(int[] a, int e) {
28
29         long start = System.nanoTime();
30         int x = search(a,e);
31         long end = System.nanoTime();
32         double diff = (end-start);
33         return diff;
34         //return "Size: " + a.length + " Time elapsed(nanoseconds): " + diff;
35     }
36 }
37
```

We designed the above method to test the runtime of our program in nanoseconds

```
1
2  public class Driver {
3
4      public static void main(String[] args) {
5
6
7          int[] sizes = {1,2,10,20,100,200,1000,2000,10000,20000,100000,200000,1000000,2000000,100
8          RecBinarySearch a = new RecBinarySearch();
9
10         int[] arr_setup = {0};
11         System.out.println("Setup --" + a.timeFind(arr_setup, 0));
12         System.out.println();
13
14         for(int size:sizes) {
15               int[] arr = new int[size];
16               for (int i = 0; i < size; i++) {
17               arr[i] = i;
18         }
19
20
21        //int target = (int) (Math.random() * size);
22
23 //       double ret_avg = 0;
24 //
25 //       for(int i = 0; i < 10;i++) {
26 //           ret_avg += a.timeFind(arr, 0);
27 //       }
28 //
29 //       double ret = ret_avg/10;
30
31         System.out.println("Size: " + size + " Average runtime(nanoseconds): " + a.timeFind(arr,
32         //System.out.println("Size: " + size + " Average runtime(nanoseconds): " + ret);
33
34
35     }
36
37 }
38 }
```

We updated the driver so that it will test the program for input sizes ranging from 2 to 2 billion

When we run the program we get data as follows:

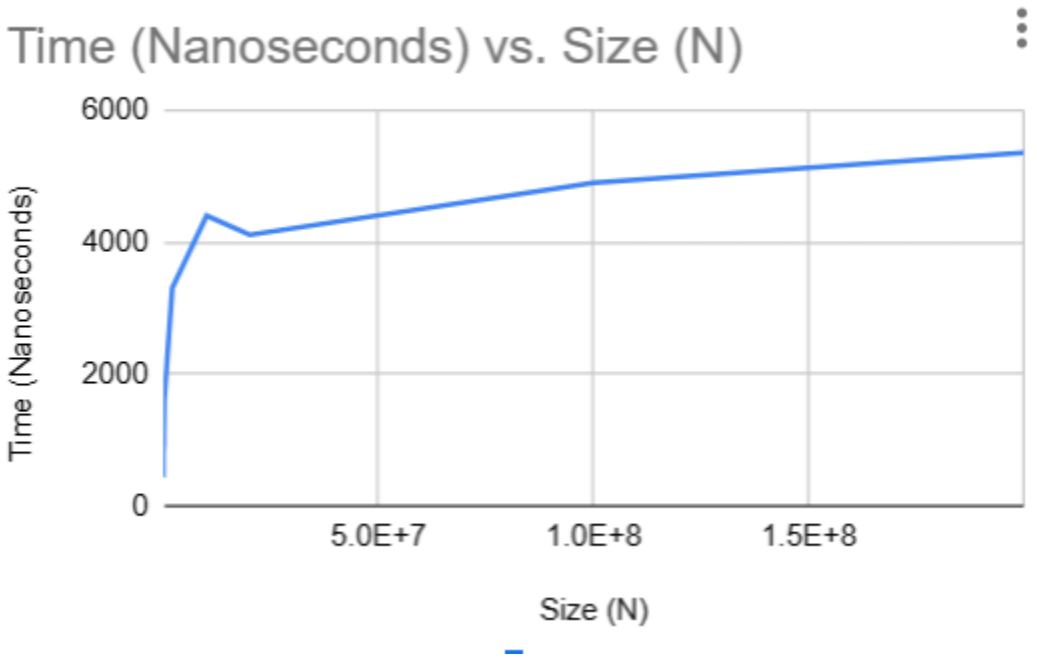
```
Setup --1700.0

Size: 1 Runtime(nanoseconds): 500.0
Size: 2 Runtime(nanoseconds): 400.0
Size: 10 Runtime(nanoseconds): 700.0
Size: 20 Runtime(nanoseconds): 700.0
Size: 100 Runtime(nanoseconds): 600.0
Size: 200 Runtime(nanoseconds): 700.0
Size: 1000 Runtime(nanoseconds): 800.0
Size: 2000 Runtime(nanoseconds): 800.0
Size: 10000 Runtime(nanoseconds): 800.0
Size: 20000 Runtime(nanoseconds): 800.0
Size: 100000 Runtime(nanoseconds): 1100.0
Size: 200000 Runtime(nanoseconds): 2100.0
Size: 1000000 Runtime(nanoseconds): 4600.0
Size: 2000000 Runtime(nanoseconds): 4100.0
Size: 10000000 Runtime(nanoseconds): 5700.0
Size: 20000000 Runtime(nanoseconds): 5400.0
Size: 100000000 Runtime(nanoseconds): 4800.0
Size: 200000000 Runtime(nanoseconds): 6500.0
```

The data varies every time we run the program, so we run it five times to find average data points which are as follows:

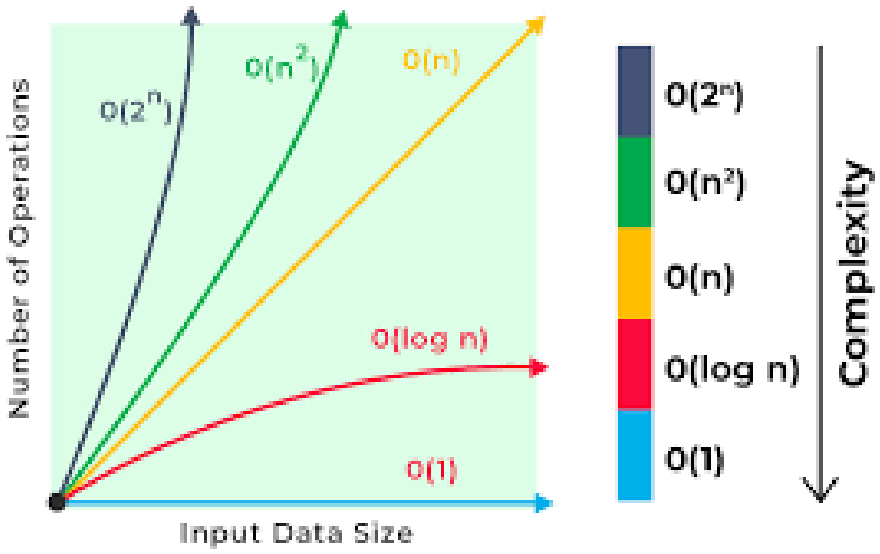
Input Size(N)	**Time(Nanoseconds)**
1	440
2	460
10	600
20	610
100	610
200	700
1000	780
2000	810
10000	950

20000	1010
100000	1100
200000	1610
1000000	2360
2000000	3300
10000000	4400
20000000	4110
100000000	4900
200000000	5350

When graphing our data we get the following:



We can compare this to a graph of log(n), which is red in the following picture:



When Considering the graphs, as well as analyzing the data points we can show that the time complexity of the recursive binary search is O(log(n)).

Test Code

```
1
2  public class RecBinarySearchTest {
3
4  public static void main(String[] args) {
5
6      RecBinarySearch a1 = new RecBinarySearch();
7
8      int[] a = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
9
10     int target = 6;
11
12     int target2 = 0;
13
14     int target3 = 20;
15
16     int target4 = -1;
17
18     int target5 = 1;
19
20     int target6 = 15;
21
22     System.out.println(a1.search(a, target) + " = " + 5);
23     System.out.println(a1.search(a, target2) + " = " + -1);
24     System.out.println(a1.search(a, target3) + " = " + -1);
25     System.out.println(a1.search(a, target4) + " = " + -1);
26     System.out.println(a1.search(a, target5) + " = " + 0);
27     System.out.println(a1.search(a, target6) + " = " + 14);
28
29
30
31
32
33
34
35
36
37
38
39
40     }
41 }
42
```

```
5 = 5
-1 = -1
-1 = -1
-1 = -1
0 = 0
14 = 14
```

This test code has test cases coded in. If the statements above are all true, proving that out implementation of the pseudo code works.