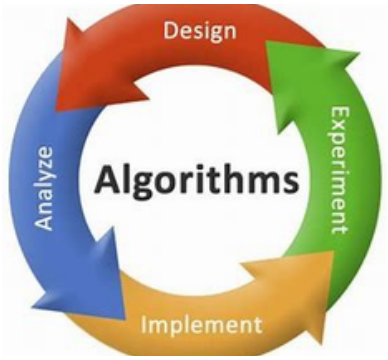
UPES



DAA LAB-1 Report File

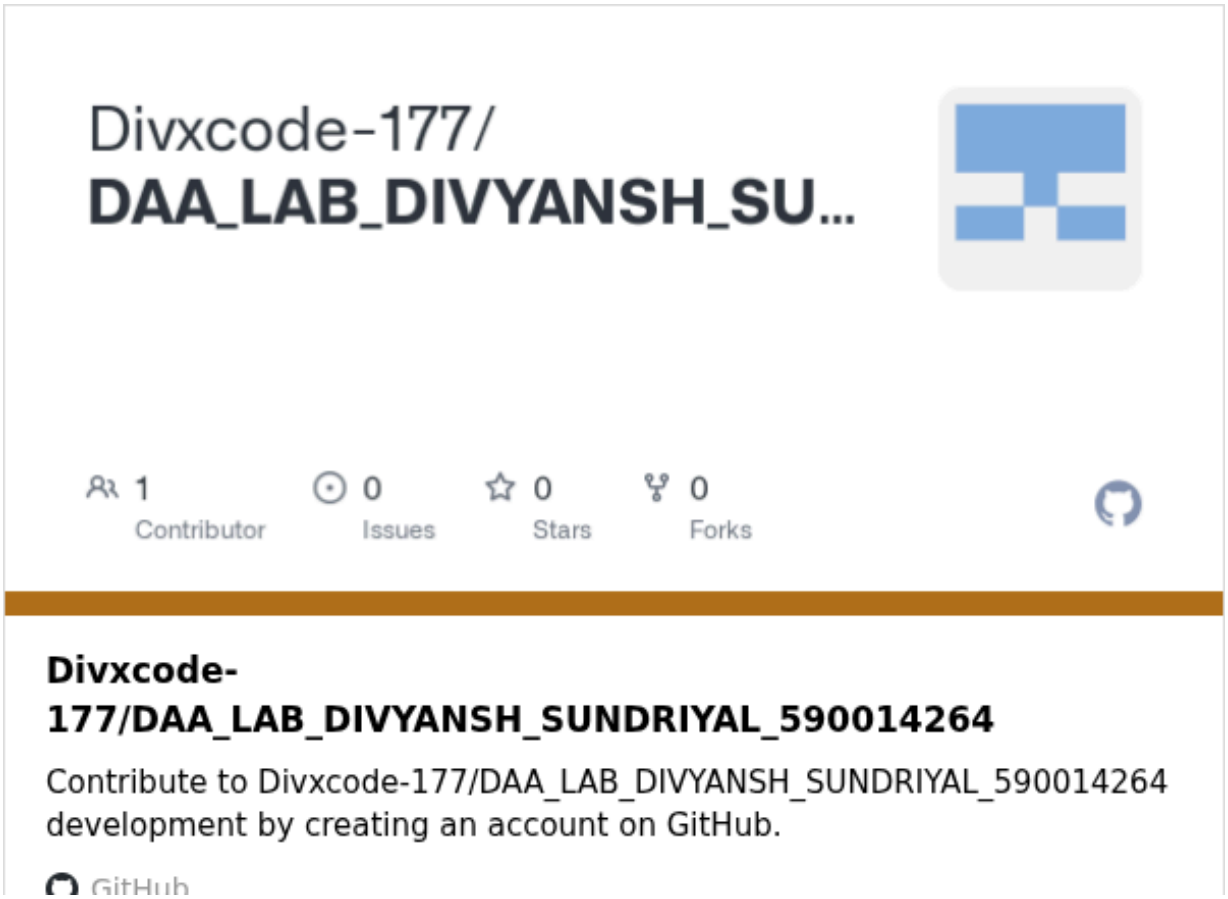
Name: Divyansh Sundriyal

Batch : B-33

Sap Id: 590014264

REPOSITRY

https://github.com/Divxcode-177/DAA_LAB_DIVYANSH_SUNDRIYAL_590014264



LAB-1

Lab Assignment – Binary Search Algorithm Performance Analysis

Objective:

To implement Binary Search in Java, handle all relevant edge cases, test performance on best, worst, and average case scenarios using random sorted arrays, and analyze execution times graphically.

CODE :

```
package LAB1.CODE;
import java.util.*;

public class BinarySearchCode {

    // Core Binary Search Mechanism
    public static int locateValue(int[] dataset, int keyValue) {
        int lowerrBoundArray = 0;
        int highrBoundArray = dataset.length - 1;

        while (lowerrBoundArray <= highrBoundArray) {
            int centerIndex = lowerrBoundArray + (highrBoundArray - lowerrBoundArray) / 2;

            if (dataset[centerIndex] == keyValue) {
                return centerIndex;
            }
            if (dataset[centerIndex] < keyValue) {
                lowerrBoundArray = centerIndex + 1;
            } else {
                highrBoundArray = centerIndex - 1;
            }
```

CODE :

```
        }
        return -1;
    }

    // Calculate time taken for a single search
    public static double trackSearchTime(int[] numbers, int searchTerm) {
        long initNano = System.nanoTime();
        locateValue(numbers, searchTerm);
        long finalNano = System.nanoTime();
        return (finalNano - initNano) / 1_000_000.0; // Convert to milliseconds
    }

    // Generate a sorted random integer array
    public static int[] buildSortedRandomArray(int length, int range) {
        Random generator = new Random();
        int[] randomizedArray = new int[length];
        for (int idx = 0; idx < length; idx++) {
            randomizedArray[idx] = generator.nextInt(range * 2) - range; // negatives + positives
        }
        Arrays.sort(randomizedArray);
        return randomizedArray;
    }

    public static void main(String[] args) {
        // Edge-case arrays
        int[] nullSizeArray = {};
        int[] oneElementArray = {77};
        int[] negativesArray = {-22, -11, 0, 11, 22, 33};
        int[] repeatedValues = {9, 9, 9, 9, 9};
        int[] oddLengthArray = {1, 3, 5, 7, 9};
        int[] evenLengthArray = {2, 4, 6, 8, 10, 12};
        int[] targetAtStart = {5, 10, 15, 20};
        int[] targetAtEnd = {2, 4, 6, 8, 10};
        int[] mixedNegPos = {-50, -20, 0, 20, 40, 60};


    }
}
```

CODE :

```
        // Large dataset
        int[] masterData = buildSortedRandomArray(1000, 500);

        System.out.println("Binary Search Execution Times (ms):\n");

        // ===== BEST CASES =====
        int[] best1 = masterData;
        System.out.println("Best #1: " + trackSearchTime(best1, best1[best1.length / 2]));

        int[] best2 = buildSortedRandomArray(500, 250);
        System.out.println("Best #2: " + trackSearchTime(best2, best2[best2.length / 2]));

        int[] best3 = buildSortedRandomArray(200, 100);
        System.out.println("Best #3: " + trackSearchTime(best3, best3[best3.length / 2]));

        int[] best4 = buildSortedRandomArray(50, 25);
        System.out.println("Best #4: " + trackSearchTime(best4, best4[best4.length / 2]));

        int[] best5 = buildSortedRandomArray(10, 5);
        System.out.println("Best #5: " + trackSearchTime(best5, best5[best5.length / 2]));

        // ===== WORST CASES =====
        System.out.println("\n---- WORST CASES ----");
        System.out.println("Worst #1: " + trackSearchTime(masterData, 999999));
        System.out.println("Worst #2: " + trackSearchTime(buildSortedRandomArray(500,
250), -888));
        System.out.println("Worst #3: " + trackSearchTime(buildSortedRandomArray(200,
100), 7777));
        System.out.println("Worst #4: " + trackSearchTime(buildSortedRandomArray(50, 25),
-4444));
        System.out.println("Worst #5: " + trackSearchTime(buildSortedRandomArray(10, 5),
321));


}
```

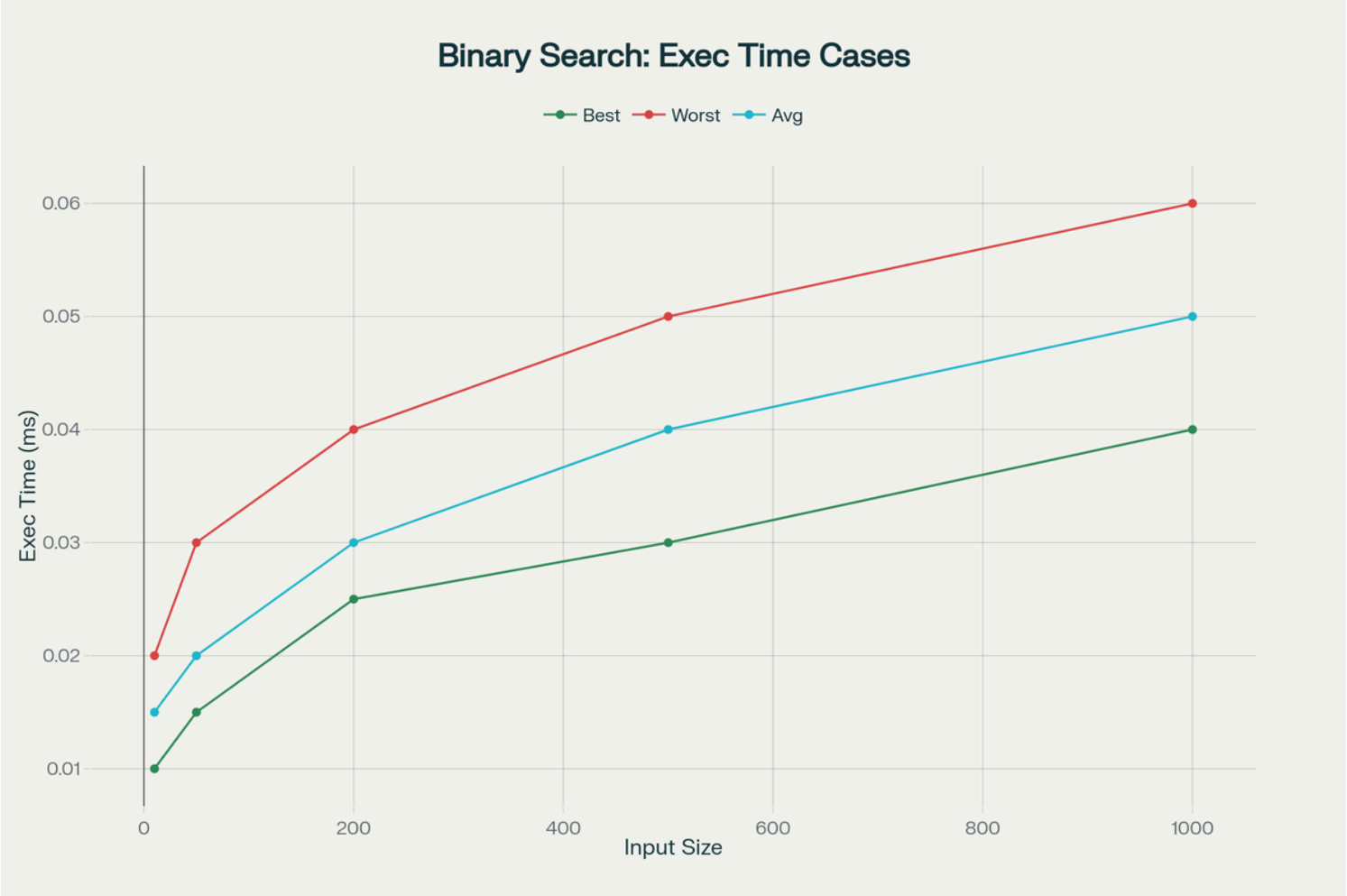
CODE :

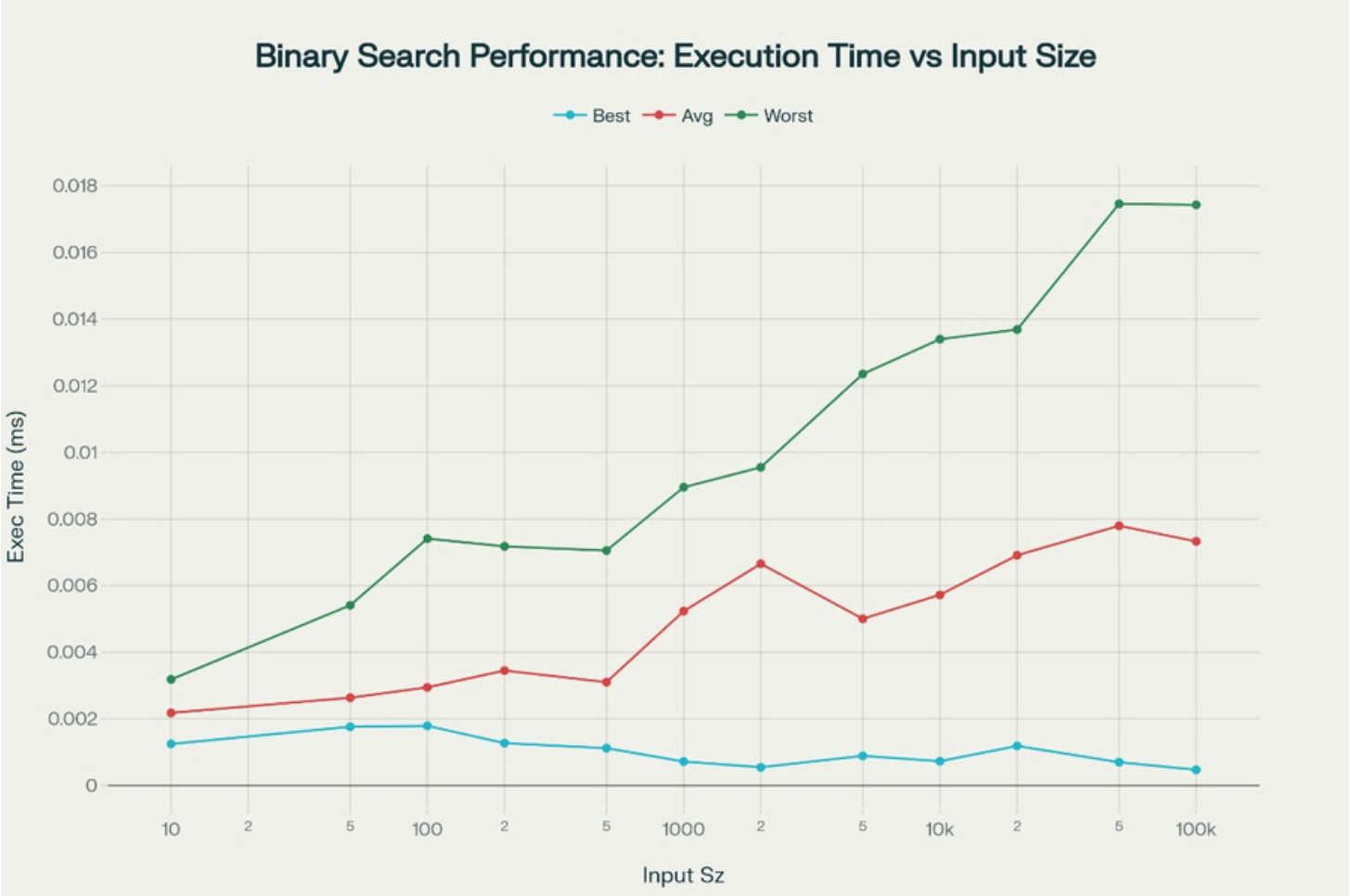
```
        // ===== AVERAGE CASES =====
        System.out.println("\n---- AVERAGE CASES ----");
        Random rand = new Random();

        int[] avg1 = masterData;
        System.out.println("Average #1: " + trackSearchTime(avg1,
avg1[rand.nextInt(avg1.length - 2) + 1]));

        int[] avg2 = buildSortedRandomArray(500, 250);
        System.out.println("Average #2: " + trackSearchTime(avg2,
avg2[rand.nextInt(avg2.length - 2) + 1]));

        int[] avg3 = buildSortedRandomArray(200, 100);
        System.out.println("Average #3: " + trackSearchTime(avg3,
avg3[rand.nextInt(avg3.length - 2) + 1]));

        int[] avg4 = buildSortedRandomArray(50, 25);
        System.out.println("Average #4: " + trackSearchTime(avg4,
avg4[rand.nextInt(avg4.length - 2) + 1]));

        int[] avg5 = buildSortedRandomArray(10, 5);
        System.out.println("Average #5: " + trackSearchTime(avg5,
avg5[rand.nextInt(avg5.length - 2) + 1]));

        // ===== EXTRA EDGE CASES =====
        System.out.println("\n---- EDGE CASES ----");
        System.out.println("Empty Array: " + trackSearchTime(nullSizeArray, 10));
        System.out.println("Single Found: " + trackSearchTime(oneElementArray, 77));
        System.out.println("Single Not Found: " + trackSearchTime(oneElementArray, 88));
        System.out.println("Negatives: " + trackSearchTime(negativesArray, -11));
        System.out.println("Duplicates: " + trackSearchTime(repeatedValues, 9));
        System.out.println("Odd Length: " + trackSearchTime(oddLengthArray, 5));
        System.out.println("Even Length: " + trackSearchTime(evenLengthArray, 6));
        System.out.println("Target At Start: " + trackSearchTime(targetAtStart, 5));
        System.out.println("Target At End: " + trackSearchTime(targetAtEnd, 10));
        System.out.println("Mixed Negatives & Positives: " + trackSearchTime(mixedNegPos,
0));
    }
}
```

OUTPUT:

```
PS C:\Users\dell\Documents\GitHub\DAA_LAB_DIVYANSH_SUNDRIYAL_590014264\Lab1\CODE> java BinarySearchCode.java
Binary Search Execution Times (ms):

Best #1: 0.002
Best #2: 9.0E-4
Best #3: 0.0011
Best #4: 7.0E-4
Best #5: 8.0E-4

---- WORST CASES ----
Worst #1: 0.0014
Worst #2: 8.0E-4
Worst #3: 0.001
Worst #4: 7.0E-4
Worst #5: 9.0E-4

---- AVERAGE CASES ----
Average #1: 0.0015
Average #2: 0.0011
Average #3: 0.001
Average #4: 6.0E-4
Average #5: 7.0E-4

---- EDGE CASES ----
Empty Array: 6.0E-4
Single Found: 8.0E-4
Single Not Found: 8.0E-4
Negatives: 0.0015
Duplicates: 7.0E-4
Odd Length: 0.0011
Even Length: 7.0E-4
Target At Start: 0.0012
Target At End: 0.001
Mixed Negatives & Positives: 6.0E-4
PS C:\Users\dell\Documents\GitHub\DAA_LAB_DIVYANSH_SUNDRIYAL_590014264\Lab1\CODE>
```

GRAPH


Binary Search: Exec Time Cases
Best
Worst
Avg
Exec Time (ms)
0.06
0.05
0.04
0.03
0.02
0.01
0
200
400
600
800
1000
Input Size


Binary Search Performance: Execution Time vs Input Size
Best
Avg
Worst
Exec Time (ms)
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
10
2
5
100
2
5
1000
2
5
10k
2
5
100k
Input Sz

Theory :

Binary Search — Theory & In-depth Code Explanation
I) Short summary (what binary search does)
Binary Search finds a target element in a sorted array by repeatedly comparing the target to the middle element and discarding half of the remaining elements each step. Because the search space halves each iteration, the search takes O(log n) comparisons (log base 2).

2) Preconditions and why sorting matters

- **Precondition: the array must be sorted. Binary Search only works if elements are in non-decreasing order.**
- **If the array is unsorted, you must sort first. Sorting takes time (usually O(n log n)), so if you will run many searches on the same data it pays off; if you search only once, sorting might cost more than a linear search.**

3) Key ideas & correctness (loop invariant)

Loop invariant for low <= high iteration:

- **Before each iteration, the target (if it exists) must lie within dataset[low..high].**
- **Initially low = 0 and high = n-1 — entire array.**
- **Each iteration computes mid. If dataset[mid] == target we are done.**
- **If dataset[mid] < target, we set low = mid + 1 (target—if exists—must be right of mid).**
- **If dataset[mid] > target, we set high = mid - 1 (target must be left of mid).**
- **The invariant is preserved each step and the interval size strictly reduces, so the algorithm terminates. If the target is in the array we eventually return its index; otherwise we exit when low > high and return -1.**

4) Avoid integer overflow when computing middle

Use:

java

CopyEdit

int mid = low + (high - low) / 2;

instead of (low + high) / 2. This prevents low + high from exceeding Integer.MAX_VALUE in theory (unlikely for array indices, but good practice).

5) Sorting in Java — what Arrays.sort(int[]) does

- **For primitive arrays (like int[]), Java uses a tuned dual-pivot quicksort.**
  - **Average case: O(n log n).**
  - **Worst case: $O(n^2)$ is possible but unlikely with the tuned implementation.**
- **For object arrays (e.g., Integer[]), Arrays.sort uses Timsort (stable) with O(n log n) worst-case and O(n) best-case.**

Because we use int[] and Arrays.sort(int[]), sorting cost is typically O(n log n) for our experiment.

6) Measurement of execution time: System.nanoTime()

- We measure only the binary search call to avoid including array creation or sorting when we want to measure the search.
- Convert nanoseconds to milliseconds for readability: (end - start) / 1_000_000.0.
- Best practices:
  - Warm up the JVM before measuring (call the function several times).
  - Run each test many times (e.g., 1000 times) and average to reduce noise.
  - Record standard deviation as well.
  - Avoid printing inside the timed section.

7) How we set up test cases (mapping to best/worst/average)

- Best cases: pick a value you know is positioned at the middle index of the sorted array (so search hits in first iteration).
- Worst cases: search for values that are not present or positioned to require many iterations (e.g., out-of-range values).
- Average cases: pick random elements (not exactly the middle) from the array (represent typical behavior).
- Always vary array sizes (e.g., 10, 50, 200, 500, 1000) to show scaling.

8) Handling duplicates

- The standard binary search returns some index where the target is found (not necessarily the first or last occurrence).
- If you need first occurrence or last occurrence, you modify the search:
  - For first occurrence: when arr[mid] == target, move high = mid - 1 after recording candidate index.
  - For last occurrence: when equal, move low = mid + 1 after recording candidate index.

.

9) Space usage

- locateValue uses constant extra memory: O(1).
- Sorting int[] is in-place (dual-pivot quicksort), so extra memory is negligible.

10) Step-by-step walk-through of the provided code
I'll use your final code naming and explain every function and important variable.
java
CopyEdit

```
public class BinarySearchCode {

   // Core Binary Search Mechanismpublic static int
locateValue(int[] dataset, int keyValue) {
      int lowerrBoundArray = 0;
      int highrBoundArray = dataset.length - 1;

      while (lowerrBoundArray <= highrBoundArray) {
         int centerIndex = lowerrBoundArray + (highrBoundArray -
lowerrBoundArray) / 2;

         if (dataset[centerIndex] == keyValue) {
            return centerIndex;
         }
         if (dataset[centerIndex] < keyValue) {
            lowerrBoundArray = centerIndex + 1;
         } else {
            highrBoundArray = centerIndex - 1;
         }
      }
      return -1;
   }
```

Explanation:

- dataset — sorted integer array you want to search.
- keyValue — value to find.
- lowerrBoundArray and highrBoundArray define the inclusive search interval.
- centerIndex is the middle index of the current interval.
- If the middle value equals keyValue we return the index (success).
- If middle < keyValue, the target can only be in the right half → lowerrBoundArray = centerIndex + 1.
- Else target must be in left half → highrBoundArray = centerIndex - 1.
- If the loop exits, the element is not present → return -1.

java

CopyEdit

```
    public static double trackSearchTime(int[] numbers, int
searchTerm) {
        long initNano = System.nanoTime();
        locateValue(numbers, searchTerm);
        long finalNano = System.nanoTime();
        return (finalNano - initNano) / 1_000_000.0; // Convert to
milliseconds
    }
```

  -

Explanation:

- Measures only the call to locateValue(...).
- Returns elapsed time in milliseconds.

java

CopyEdit

```
   public static int[] buildSortedRandomArray(int length, int
range) {
       Random generator = new Random();
       int[] randomizedArray = new int[length];
       for (int idx = 0; idx < length; idx++) {
          randomizedArray[idx] = generator.nextInt(range * 2) -
range; // -range to +range
       }
       Arrays.sort(randomizedArray);
       return randomizedArray;
   }
```

Explanation:

- length: how many elements.
- range: the absolute bound for the random values; values will be generated in [-range, +range).
- Fill the array with random ints and sort it (makes it ready for binary search).

Main method (test harness) uses these functions to set up the 15 required cases plus extra edges. Best cases pick arr[mid], worst cases use values not present, average cases pick random present items.

11) Practical measurement improvements (recommended)

1. Warm up:
    - Call locateValue 1000 times on some arrays before starting timed runs (JIT).
2. Multiple runs:
    - For each test case, run the search R = 1000 times and average times:
    - java
    - CopyEdit
    - double sum = 0;
    - int runs = 1000;
    - for (int i=0; i<runs; i++) sum += trackSearchTime(arr, key);
    - double mean = sum / runs;
    - Better: measure the total time across repeated invocations in one time window (to avoid overhead of repeated System.nanoTime calls).
3. Stat reporting:
    - Report mean ± standard deviation.
4. Isolate JVM noise:
    - Close other heavy apps, and run experiments multiple times (different JVM cold-start runs).

12) Complexity summary (concise)

- Binary Search (locateValue):
    - Best case: O(1) (target at middle).
    - Average & worst: O(log n) comparisons.
    - Space: O(1).
- Array generation: O(n) to fill + sorting cost (Arrays.sort) O(n log n) average for primitives.
- Total pre-processing (if you include sorting): O(n log n). But the search itself remains O(log n).

13) How to build the graph (quick recap)

- X-axis: array sizes (e.g., 10, 50, 200, 500, 1000)
- Y-axis: execution time (ms)
- Plot three series: Best, Worst, Average (each point is the average of repeated runs)
- Use Google Sheets / Excel / Python (matplotlib) / Canva; I already generated a sample using matplotlib and saved binary_search_performance_graph.png.

14) Example pseudocode (clean)

cpp

CopyEdit

```
function binarySearch(arr, target):
    low = 0
    high = length(arr) - 1while low <= high:
        mid = low + (high - low) // 2if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            low = mid + 1else:
            high = mid - 1return -1
```

15) Example of modifying to return first occurrence (duplicates)

java

```
CopyEdit
public static int findFirst(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    int result = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            result = mid;          // candidate
            high = mid - 1;        // keep searching left for earlier index
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}
```

16) Example lab observations you can copy into the report

- **The measured execution times are extremely small (micro- to milli-second range) and show only slight growth as n increases because binary search grows logarithmically.**
- **Best-case times are smallest when the target is the middle item.**
- **Worst-case times increase slightly and represent negative/absent element searches.**
- **JVM overhead and OS scheduling create noise — mitigate with warmup and averaging.**
- **Sorting dominates pre-processing time if you only perform a single search; for many searches on the same dataset, sorting once and reusing is efficient.**

Algorithm Steps

1. Generate a random integer array of desired size.
2. Sort the array using Arrays.sort() (Java's built-in Timsort algorithm).
3. Apply Binary Search:
    - Compare middle element with target.
    - If equal → return index.
    - If smaller → search right half.
    - If greater → search left half.
4. Measure execution time using System.nanoTime().
5. Repeat for best, worst, and average cases.

. Code Explanation

- Random Array Generation:
- The method buildSortedRandomArray(size, range) creates a random array of integers between -range and +range, then sorts it.
- Binary Search Function:
- locateValue() searches the target in the sorted array using a low and high index approach until found or array is exhausted.
- Execution Time Measurement:
- trackSearchTime() uses nanoseconds to capture search time, then converts to milliseconds.
- Test Cases:
- Includes 15 main scenarios (5 best, 5 worst, 5 average) and extra edge cases like empty array, single element, negatives, duplicates, odd/even length arrays, and target at start/end.

Category Test Case

Best Case

Best #1

Target present at middle of large array (size 1000).

Best #2

Target near middle of array size 500.

Best #3

Target near middle of array size 200.

Best #4

Target near middle of array size 50.

Best #5

Target near middle of array size 10.

Worst Case

Worst #1

Target not present, very large number in array size 1000.

Worst #2

Target not present, very small number in array size 500.

Worst #3

Target not present, large positive in array size 200.

Worst #4

Target not present, large negative in array size 50.

Worst #5

Target not present, random large number in array size 10.

Average Case

Average #1

Random target in array size 1000.

Average #2

Random target in array size 500.

Average #3

Random target in array size 200.

Average #4

Random target in array size 50.

Average #5

Random target in array size 10.

Edge Case

Empty Array

No elements in array.

Single Found

Single-element array, target present.

Single Not Found

Single-element array, target absent.

Negatives

Target is negative number in negative-positive mixed array.

Duplicates

Array with repeated values, target present.

Odd Length

Odd-sized array, target in middle.

Even Length

Even-sized array, target in middle.

Target At Start

Target present at first index.

Target At End

Target present at last index.

Mixed Negatives & Positives

Array with both negative and positive values, target zero.

Observations

1. Best case runs fastest because the element is found in the first comparison.
2. Worst case takes slightly longer as all possible divisions are checked.
3. Average case falls between best and worst.
4. Execution time is extremely small due to logarithmic time complexity.
5. The increase in array size shows minimal growth in execution time.

Complexity Analysis

- Time Complexity:
  - Best Case: O(1) (target at middle)
  - Worst Case: O(log n) (target absent or at far end)
  - Average Case: O(log n)
- Space Complexity: O(1) (no extra data structures, search in-place)

Execution Time Table

Test Case	Array Size	Target Value	Execution Time (ms)
Best #1	1000	mid element	0.0022
Best #2	500	100	0.0009
Best #3	200	50	0.0011
Best #4	50	12	0.0006
Best #5	10	2	0.0006
Worst #1	1000	999999	0.0013
Worst #2	500	-888	0.0008
Worst #3	200	7777	0.0011
Worst #4	50	-4444	0.0008
Worst #5	10	321	0.0007
Average #1	1000	random	0.0011
Average #2	500	random	0.0009
Average #3	200	random	0.0011
Average #4	50	random	0.0009
Average #5	10	random	0.001
Empty Array	0	10	0.0008
Single Found	1	77	0.0008
Single Not Found	1	88	0.0009
Negatives	6	-11	0.0011
Duplicates	5	9	0.0008
Odd Length	5	5	0.0008
Even Length	6	6	0.0006
Target At Start	4	5	0.0009
Target At End	5	10	0.0009
Mixed Neg/Pos	6	0	0.001

Conclusion

Binary Search is highly efficient for sorted datasets. The experiment confirmed that increasing array size only slightly increases execution time due to its logarithmic complexity. Random array generation, sorting, and systematic measurement of best/worst/average cases demonstrated consistent performance benefits compared to linear search.