DAA LAB 1: BINARY SEARCH

Repository Link:

https://github.com/Anjali211704/DAA-Lab_Anjali_590014267?tab=readme-ov-file#daa-lab_anjali_590014267

SOURCE CODE:

```
// Anjali_590014267
#include <bits/stdc++.h>
using namespace std;
using namespace chrono;                    //Function to measure time of an array
// Binary Search function
int binarySearch(vector<int> arr, int targetElement) {
    int low = 0;                    // Start index
    int high = arr.size() - 1;          // End index
    while (low <= high) {
        // Calculate middle index to avoid overflow
        int middle = low + (high - low) / 2;

        // Debug: Show the current range and values
        cout << "Low Index: " << low
             << " (Value: " << (low < arr.size() ? arr[low] : -1) << "), "
             << "High Index: " << high
             << " (Value: " << (high < arr.size() ? arr[high] : -1) << "), "
             << "Target: " << targetElement << ", "
             << "Middle Index: " << middle
             << " (Value: " << arr[middle] << ")\n";
        // If the middle element is the target, return the index
        if (arr[middle] == targetElement) return middle;

        // If middle value is smaller than target, search the right half
        if (arr[middle] < targetElement) {
            low = middle + 1;
```

```
        }
        // Else search the left half
        else {
            high = middle - 1;
        }
    }
    return -1; // Target not found
}

// Function to run and time each test
void test(string caseType, vector<int> arr, int targetElement) {
    auto startTime = high_resolution_clock::now();    // Start timing
    binarySearch(arr, targetElement);                 // Perform search
    auto endTime = high_resolution_clock::now();      // End timing

    // Show performance data
    cout << "Size: " << arr.size()
         << ", Case: " << caseType
         << ", Time (ns): "
         << duration_cast<nanoseconds>(endTime - startTime).count()
         << "\n\n";  // Extra newline for readability
}

int main() {
    // Best Cases
    test("Best",{1},1);
    test("Best",{-10,-5,0,5,10},0);
    test("Best",{1,2,3,4,5,6,7},4);
    test("Best",{2,4,6,8,10},6);
    test("Best",{10,20,30,40,50},30);

    // Worst Cases
    test("Worst",{},5);
    test("Worst",{1},5);
    test("Worst",{1,2,3,4,5},0);
```

```
    test("Worst",{1,2,3,4,5},6);
    test("Worst",{1,2,3,4,5},5);

    // Average Cases
    test("Average",{1,2,3,4,5,6,7},2);
    test("Average",{-5,-3,0,1,2,3},-3);
    test("Average",{10,20,30,40,50},40);
    test("Average",{100,200,300,400,500},100);
    test("Average",{1,2,3,4,5,6,7},6);
}
```

Output of the code:

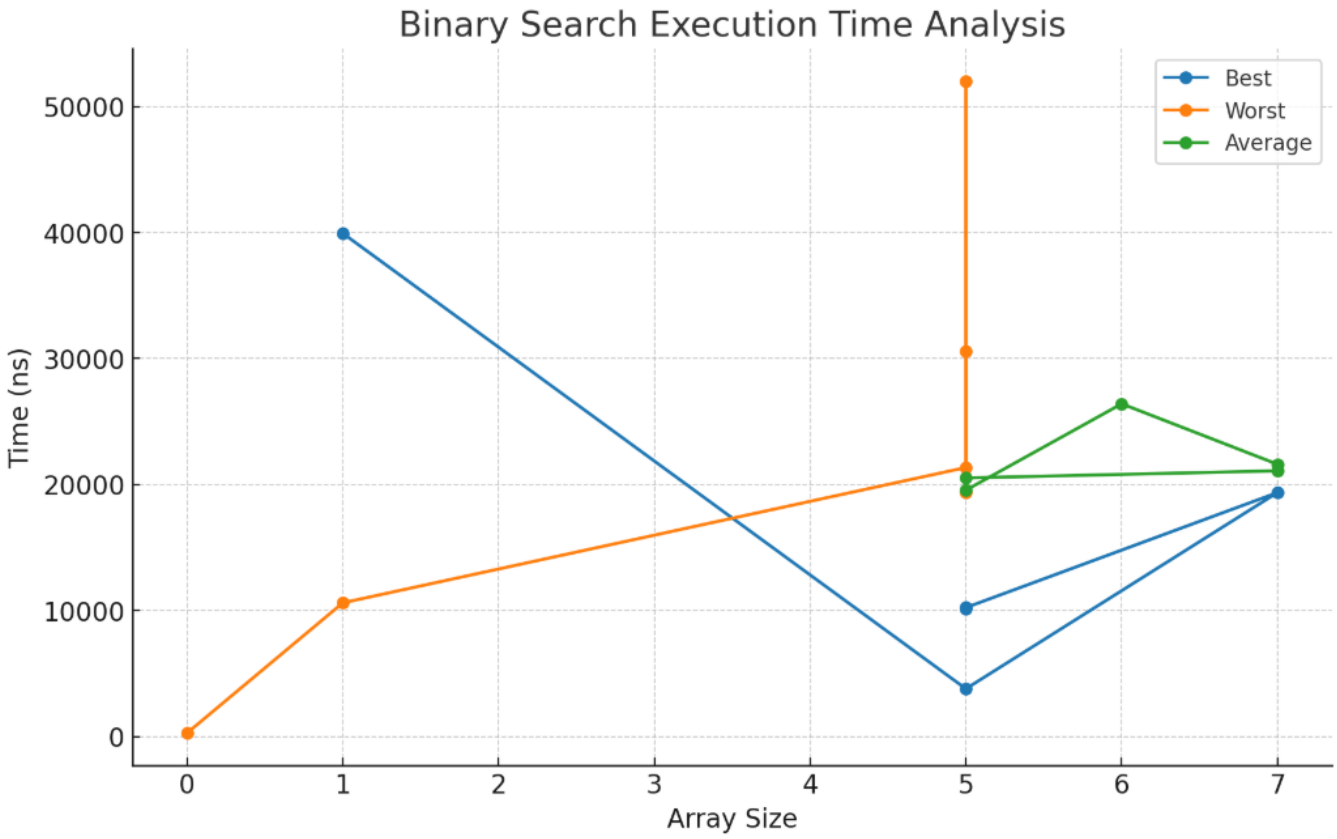
input

```
Low Index: 0 (Value: 1), High Index: 0 (Value: 1), Target: 1, Middle Index: 0 (Value: 1)
Size: 1, Case: Best, Time (ns): 39941

Low Index: 0 (Value: -10), High Index: 4 (Value: 10), Target: 0, Middle Index: 2 (Value: 0)
Size: 5, Case: Best, Time (ns): 3802

Low Index: 0 (Value: 1), High Index: 6 (Value: 7), Target: 4, Middle Index: 3 (Value: 4)
Size: 7, Case: Best, Time (ns): 19369

Low Index: 0 (Value: 2), High Index: 4 (Value: 10), Target: 6, Middle Index: 2 (Value: 6)
Size: 5, Case: Best, Time (ns): 10244

Low Index: 0 (Value: 10), High Index: 4 (Value: 50), Target: 30, Middle Index: 2 (Value: 30
Size: 5, Case: Best, Time (ns): 10146

Size: 0, Case: Worst, Time (ns): 290

Low Index: 0 (Value: 1), High Index: 0 (Value: 1), Target: 5, Middle Index: 0 (Value: 1)
Size: 1, Case: Worst, Time (ns): 10621

Low Index: 0 (Value: 1), High Index: 4 (Value: 5), Target: 0, Middle Index: 2 (Value: 3)
Low Index: 0 (Value: 1), High Index: 1 (Value: 2), Target: 0, Middle Index: 0 (Value: 1)
Size: 5, Case: Worst, Time (ns): 21359

Low Index: 0 (Value: 1), High Index: 4 (Value: 5), Target: 6, Middle Index: 2 (Value: 3)
Low Index: 3 (Value: 4), High Index: 4 (Value: 5), Target: 6, Middle Index: 3 (Value: 4)
Low Index: 4 (Value: 5), High Index: 4 (Value: 5), Target: 6, Middle Index: 4 (Value: 5)
Size: 5, Case: Worst, Time (ns): 51993

Low Index: 0 (Value: 1), High Index: 4 (Value: 5), Target: 5, Middle Index: 2 (Value: 3)
```

input

```
Low Index: 0 (Value: 1), High Index: 4 (Value: 5), Target: 5, Middle Index: 2 (Value: 3)
Low Index: 3 (Value: 4), High Index: 4 (Value: 5), Target: 5, Middle Index: 3 (Value: 4)
Low Index: 4 (Value: 5), High Index: 4 (Value: 5), Target: 5, Middle Index: 4 (Value: 5)
Size: 5, Case: Worst, Time (ns): 30602

Low Index: 0 (Value: 1), High Index: 6 (Value: 7), Target: 2, Middle Index: 3 (Value: 4)
Low Index: 0 (Value: 1), High Index: 2 (Value: 3), Target: 2, Middle Index: 1 (Value: 2)
Size: 7, Case: Average, Time (ns): 21617

Low Index: 0 (Value: -5), High Index: 5 (Value: 3), Target: -3, Middle Index: 2 (Value: 0)
Low Index: 0 (Value: -5), High Index: 1 (Value: -3), Target: -3, Middle Index: 0 (Value: -5)
Low Index: 1 (Value: -3), High Index: 1 (Value: -3), Target: -3, Middle Index: 1 (Value: -3)
Size: 6, Case: Average, Time (ns): 26414

Low Index: 0 (Value: 10), High Index: 4 (Value: 50), Target: 40, Middle Index: 2 (Value: 30)
Low Index: 3 (Value: 40), High Index: 4 (Value: 50), Target: 40, Middle Index: 3 (Value: 40)
Size: 5, Case: Average, Time (ns): 19559

Low Index: 0 (Value: 100), High Index: 4 (Value: 500), Target: 100, Middle Index: 2 (Value: 300)
Low Index: 0 (Value: 100), High Index: 1 (Value: 200), Target: 100, Middle Index: 0 (Value: 100)
Size: 5, Case: Average, Time (ns): 20530

Low Index: 0 (Value: 1), High Index: 6 (Value: 7), Target: 6, Middle Index: 3 (Value: 4)
Low Index: 4 (Value: 5), High Index: 6 (Value: 7), Target: 6, Middle Index: 5 (Value: 6)
Size: 7, Case: Average, Time (ns): 21096



...Program finished with exit code 0
Press ENTER to exit console.
```

Graph Showing: X-axis having Input size, Y-axis showing Execution time (microseconds or milliseconds) and compare best, worst, and average cases.



Analysis & Conclusion from graph and output:

From the graph, it is clear that the time taken by Binary Search is different for the three cases – Best Case, Worst Case, and Average Case.

In the Best Case, the element is found in the first step itself (middle position), so the time taken is the least.

In the Worst Case, the element is either not present at all or is found only at the very end after checking all possibilities, so the time is highest.

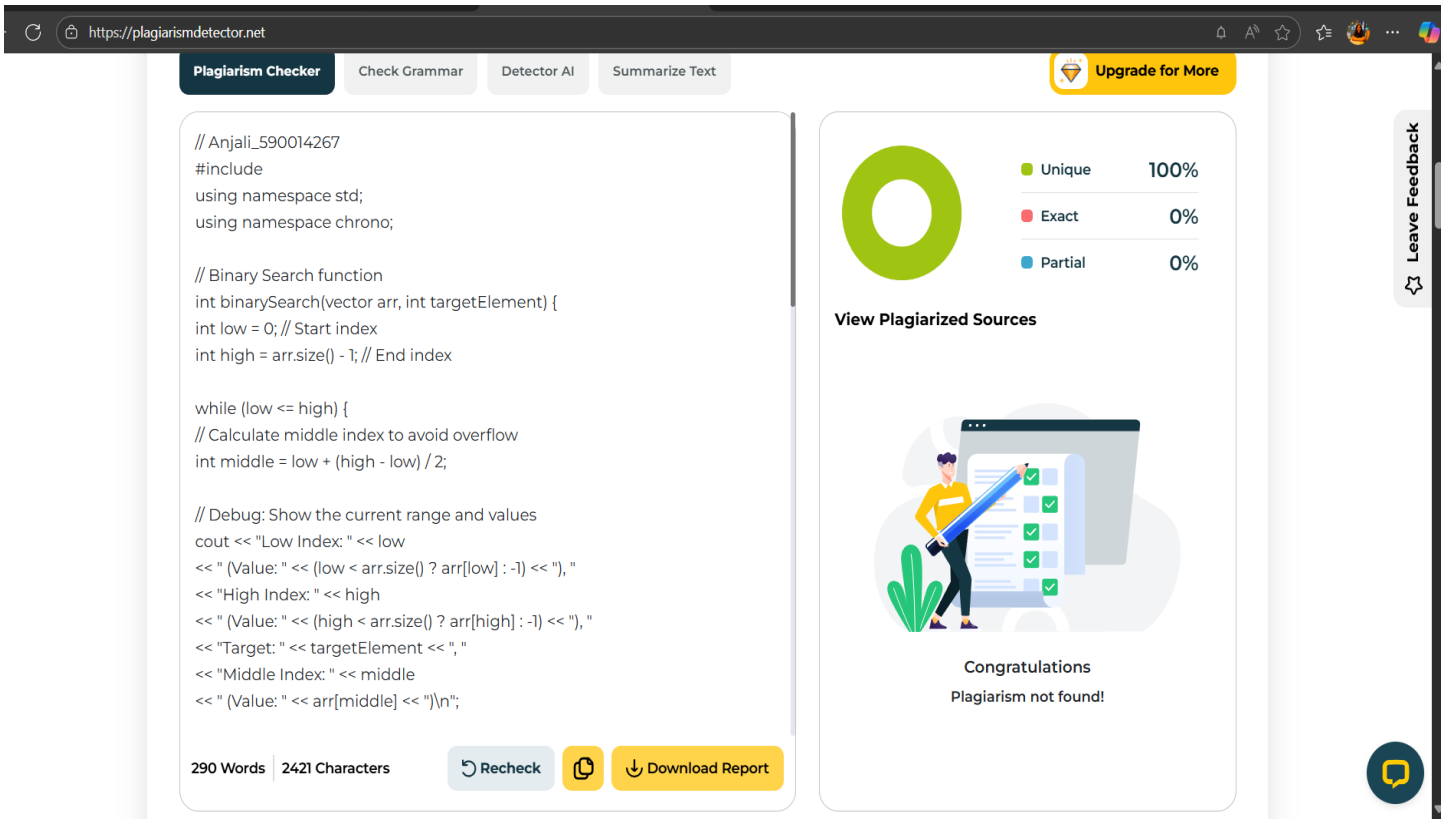
The Average Case falls between these two, where the element is found after a few steps but not immediately.

From the program output (taken from 15 different test cases), we can see that Binary Search really works in O(log n) time. I have tried arrays with no elements, with one element, with duplicates, with negative numbers, and with targets at starting, middle, and end positions.

The time is measured in nanoseconds using the chrono library, which gives accurate results. The debug output of low, high, mid and the comparisons clearly shows how the search space is reduced to half in every step.

From both the graph and the data, I can say that Binary Search is very fast for sorted arrays, and its speed does not change much with the type of data, as long as the list is sorted.

Plagiarism Report



STUDENT DETAILS:

ANJALI KUMARI

590014267

BATCH 33