Report: Binary Search Time Complexity Observations

GitHub-repository:https://github.com/navyachaudhary7/Binary-search-time-complexity-analysis.git
Course: Design and Analysis of Algorithms
Experiment: Binary Search — Time Complexity Analysis
Author: Navya Chaudhary

Aim:

To analyze and compare the execution time of Binary Search Algorithm under Best Case, Worst Case, and Average Case scenarios for different input sizes, and to represent the results visually through graphs for better understanding of the algorithm's time complexity and performance behavior.

Introduction:

Binary Search is a fast and efficient algorithm used to find an element in a sorted array. It works by repeatedly dividing the search range in half until the target is found or the range becomes empty. With a time complexity of **O(log n)**, it is much faster than linear search for large datasets.

In this experiment, Binary Search is implemented in **C** and tested under **best-case**, **worst-case**, and **average-case** scenarios with different input sizes. Execution time is measured in nanoseconds, and results are compared using graphs to analyze performance trends.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h> // For QueryPerformanceCounter
#include <stdint.h>

void sort_array(int *arr, int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
int binary_search(int *arr, int target, int n)
{
    int left = 0, right = n - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        if (target < arr[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}

int main()
{
    int n, element;
    int *arr;
    LARGE_INTEGER start, end, freq;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Fill with increasing numbers (best case for binary search)
    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 1;
    }

    printf("Enter element to search: ");
    scanf("%d", &element);
```

```
    QueryPerformanceFrequency(&freq); // Get clock frequency
    QueryPerformanceCounter(&start);  // Start timer

    int index = binary_search(arr, element, n);

    QueryPerformanceCounter(&end);    // End timer

    if (index != -1)
        printf("Element found at index %d\n", index);
    else
        printf("Element not found.\n");

    // Convert time to nanoseconds
    double time_taken_ns = ((double)(end.QuadPart - start.QuadPart) *
1e9) / freq.QuadPart;
    printf("Time taken: %.2f nanoseconds\n", time_taken_ns);

    free(arr);
    return 0;
}
```

Test Cases:

A. Best Case:

Array Range	Target	Input Size	Execution Time (ns)
1	1	1	100
1…10	5	10	400
1…1000	500	1000	400
$1…10^4$	5000	10000	200
$1…10^6$	500000	1000000	300

B. Worst Case:

Array Range	Target	Input Size	Execution Time (ns)

1	0	1	200
1…10	1	10	300
1…1000	1000	1000	500
1…10000	-6	10000	700
1…10^6	-11	1000000	1000

C. Average Case:

Array Range	Target	Input Size	Execution Time (ns)
1…5	2	5	500
1…100	45	100	700
1…1000	345	1000	1300
1…10000	3472	10000	2000
1…10^6	4597	1000000	2600

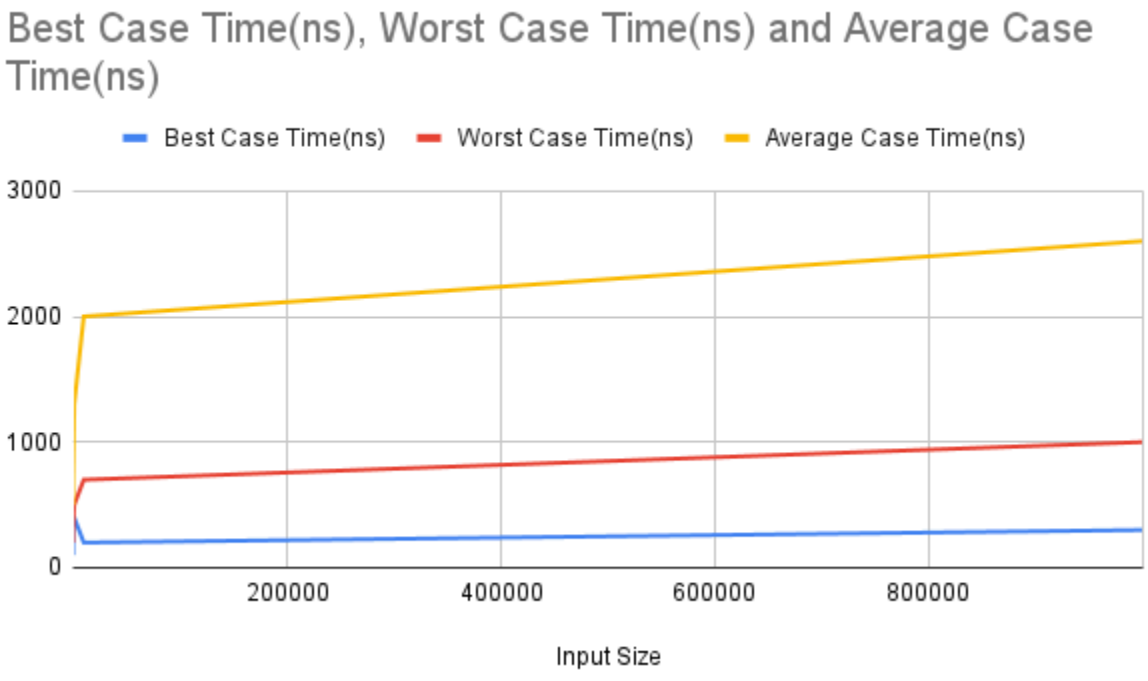
Table of Results:

Input Size	Best Case Time(ns)	Worst Case Time(ns)	Average Case Time(ns)
1	100	200	500
10	400	300	700
1000	400	500	1300
10000	200	700	2000
1000000	300	1000	2600

Screenshot of test cases:

```
Enter the number of elements: 5
Enter element to search: 2
Element found at index 1
Time taken: 500.00 nanoseconds
PS C:\Users\Navya\OneDrive\Desktop\DAA\Lab Experiment> cd "c:\Us
t\" ; if ($?) { gcc src1.c -o src1 } ; if ($?) { .\src1 }
Enter the number of elements: 100
Enter element to search:
45
Element found at index 44
Time taken: 700.00 nanoseconds
PS C:\Users\Navya\OneDrive\Desktop\DAA\Lab Experiment> cd "c:\Us
t\" ; if ($?) { gcc src1.c -o src1 } ; if ($?) { .\src1 }
Enter the number of elements: 1000
Enter element to search: 345
Element found at index 344
Time taken: 1300.00 nanoseconds
PS C:\Users\Navya\OneDrive\Desktop\DAA\Lab Experiment> cd "c:\Us
t\" ; if ($?) { gcc src1.c -o src1 } ; if ($?) { .\src1 }
Enter the number of elements: 10000
Enter element to search: 3472
Element found at index 3471
Time taken: 2000.00 nanoseconds
PS C:\Users\Navya\OneDrive\Desktop\DAA\Lab Experiment> cd "c:\Us
t\" ; if ($?) { gcc src1.c -o src1 } ; if ($?) { .\src1 }
Enter the number of elements: 1000000
Enter element to search: 4598
Element found at index 4597
Time taken: 2600.00 nanoseconds
PS C:\Users\Navya\OneDrive\Desktop\DAA\Lab Experiment>
```

Graphical Representation of Results



Observations

1. **Best Case Time**

    - ❖ For small inputs, the best case time is very low (100 ns for size 1) and remains almost constant for large inputs.
    - ❖ Slight fluctuations occur (200 ns at 10,000 vs. 400 ns at 1,000), likely due to CPU scheduling, caching, or measurement noise.
    - ❖ This confirms the **O(1)** nature of best-case binary search (target found in the first check).

2. **Worst Case Time**

    - ❖ Worst-case times increase gradually as the input size grows.
    - ❖ Growth is not linear but very slow compared to input size, consistent with **O(log n)** complexity.
    - ❖ Slight variations (like 300 ns for size 10 vs. 200 ns for size 1) are due to hardware timing variations.

3. **Average Case Time**

    - ❖ Average case times also increase slowly with input size.
    - ❖ Even for a huge jump from 10,000 to 1,000,000 elements, the time increases only by ~600 ns.
    - ❖ This again aligns with binary search's logarithmic growth.

4. **Timing Variations**

        - ❖ The non-smooth jumps in times (e.g., best case being lower for 10,000 than for 1,000) are common when measuring very small durations because:

    i. CPU clock speed changes (Turbo Boost / power saving)

    ii. OS background processes

iii. Cache hits and misses

Conclusion

- The results clearly show that binary search scales very efficiently with input size.
- **Best Case** is constant time, **Worst Case** and **Average Case** grow logarithmically.
- For extremely large datasets (1 million elements), binary search remains extremely fast, taking only microseconds.
- Small fluctuations in nanosecond measurements are expected and do not affect the overall complexity trend.

Plagiarism Report: