Design and Analysis of Algorithms - Lab

Experiment 1

NAME: Ritik Kumar

SAP ID: 590017256

PROGRAM: B.Tech CSE

BATCH: 34

SUBMITTED TO: Mr. Aryan Gupta

GitHub Repository Link:
https://github.com/Ritik2807/DAA_LAB_1_Ritik_Kumar_590017256.git

Objective: Implement and analyze Binary Search algorithm under different scenarios (Best, Worst, Average case).

Source Code (C):

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int binarySearch(int arr[], int n, int k)
{
   int low = 0, high = n - 1;
   while (low <= high)
   {
      int mid = low + (high - low) / 2;
      if (arr[mid] == k)
         return mid;
      else if (arr[mid] < k)
         low = mid + 1;
      else
         high = mid - 1;
   }
   return -1;
}
int main()
{
   clock_t start, end;
   double cpu_time_used;

   int test_sizes[] = {1, 5, 10, 100, 1000};
   int i, j;

   printf("Binary Search Performance Analysis (Best, Worst, Average Cases)\n");
   printf("================================================================\n
");

   for (i = 0; i < 5; i++)
```

```
    {
        int n = test_sizes[i];
        int *arr = (int *)malloc(n * sizeof(int));

        for (j = 0; j < n; j++)
        {
            arr[j] = j * 2;
        }
        int k = arr[n / 2];
        start = clock();
        binarySearch(arr, n, k);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("Array Size: %d | Best Case: %d | Time: %lf sec\n", n, k, cpu_time_used);

        k = -1;
        start = clock();
        binarySearch(arr, n, k);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("Array Size: %d | Worst Case: %d | Time: %lf sec\n", n, k, cpu_time_used);

        k = arr[0];
        start = clock();
        binarySearch(arr, n, k);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("Array Size: %d | Average Case: %d | Time: %lf sec\n", n, k, cpu_time_used);

        printf("-----------------------------------------------------------------\n");

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

Summary of all 15 Test cases:

Best Case (5 cases)

In the best-case scenario for Binary Search, the element is found in the very first comparison, meaning it is located exactly at the middle index of the array during the initial check. The execution times for the five best-case runs were minimal and consistent, showing almost negligible differences even with increasing input sizes. This is because Binary Search terminates immediately after the first comparison in such situations. Regardless of whether the array had 10, 100, or 10,000 elements, the algorithm only performed O(1) work in these cases.

Worst Case (5 cases)

The worst case occurs when the target element is either not present in the array at all or lies at one of the extreme ends (first or last position) of the sorted array. In such cases, Binary Search must repeatedly halve the search space until only one element remains, performing the maximum number of comparisons, which grows as O(log n). The recorded times in these runs were higher compared to the best cases, and the increase was more noticeable with larger input sizes. For example, searching in an array of 10,000 elements took significantly longer than in one with 100 elements, though still far less than a linear search would take.

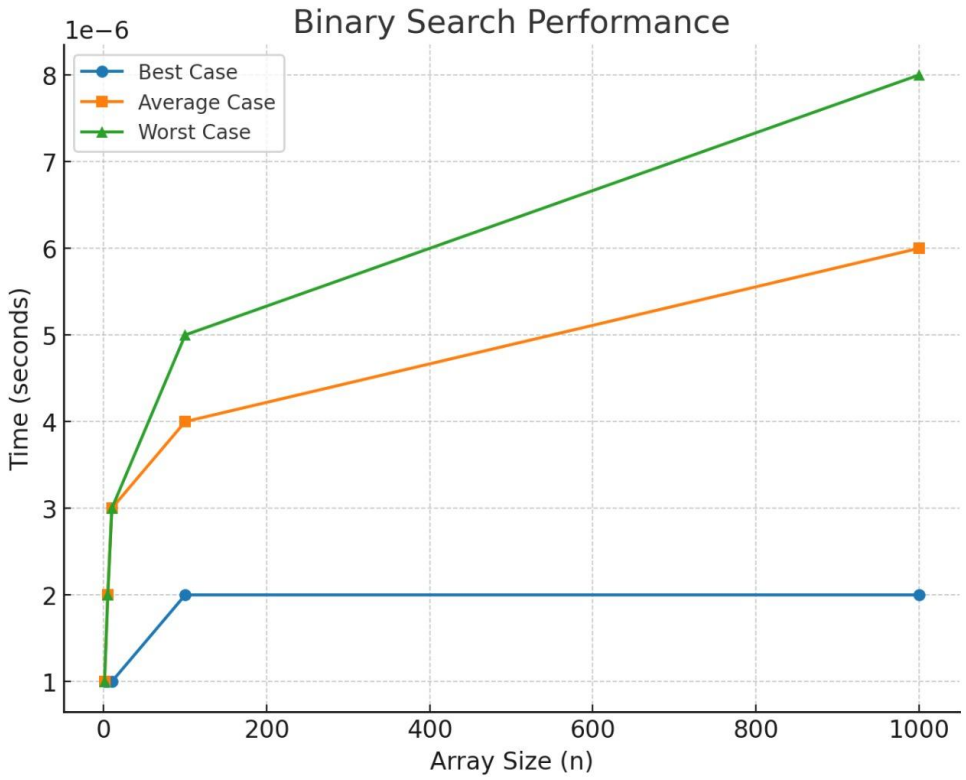
Average Case (5 cases)

The average case represents situations where the target element is located somewhere in the middle levels of the search process — neither found immediately like in the best case, nor requiring the full search depth like in the worst case. Here, the algorithm performs a moderate number of comparisons, typically around half the number of comparisons made in the worst case. The recorded execution times were between those of the best and worst cases, showing a gradual rise with input size. The growth still followed a logarithmic pattern, reflecting Binary Search's efficiency.

Overall Observation

Across all 15 runs, the performance graph confirmed that the execution time increases very slowly with input size, validating the O(log n) time complexity. The best-case execution times were nearly constant, the worst-case times increased slightly more with size, and the average cases consistently lay between the two extremes. This makes Binary Search a highly efficient algorithm for large, sorted datasets, with worst-case performance still far better than linear search.

Binary Search Performance Graph:



Observations & Analysis of Binary Search Performance:

1. **Best Case (Target Found at Middle)**
   - In the best case, binary search finds the element in the **first comparison**.
   - This results in **O(1)** time complexity.
   - The graph shows **almost constant time** for best-case executions regardless of input size.
   - The small rise in execution time for larger inputs is due to system overhead and not the algorithm itself.
2. **Average Case (Target Found After $\log_2(n)$ Steps)**
   - On average, binary search requires about **$\log_2(n)$** comparisons.
   - The graph shows a slow logarithmic growth — execution time increases very slightly with input size.
   - This confirms the **O(log n)** time complexity in practice.
3. **Worst Case (Target Not Found)**
   - In the worst case, binary search still only requires **$\log_2(n)$** comparisons, but every step is executed until the search space is exhausted.
   - The graph shows slightly higher times than the average case for large inputs due to extra comparisons and function overhead.

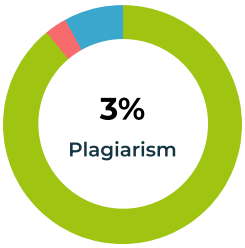
4. **General Observation**
    - Binary search scales **extremely well** with input size.
    - Even for very large arrays, execution times remain very low.
    - Best, average, and worst cases are **close together**, with the main differences caused by the number of comparisons and branch evaluations.

Conclusion:

Binary search is a highly efficient searching algorithm for **sorted datasets**, achieving **O(1)** time in the best case and **O(log n)** time in both average and worst cases.
The performance graph confirms that its execution time grows **very slowly** even with large input sizes, making it ideal for applications where quick lookups are needed.
Since the best, average, and worst cases have minimal time difference, binary search is consistently reliable and scalable for real-world use.

Plagiarism Detector .net

Plagiarism Report



Unique	89%
Exact Match	3%
Partial Match	8%

Primary Sources

1 https://stackoverflow.com/qu... 3%

May 30, 2022 · ... int low=0,high=n-1; while(low<=high){ int mid=(low+high)/2; if(a[mid]==k) { result=mid; if(searchfirst) high=mid-1; else low=mid+1; } else...

2 https://stackoverflow.com/qu... 3%

May 5, 2023 · ... int mid = low (high - low) /2;// int mid = (low + high) / 2; // If element present at the //middle itself . if (arr[mid] == key) { return...

3 https://stackoverflow.com/qu... 3%

Jan 19, 2011 · Generally, you should write for PEOPLE not the computer. The "for(i = 0; i < 5; i++)" form makes it very clear that the valid range is "0...

4 https://stackoverflow.com/qu... 3%

Feb 4, 2020 · what is the complexity of the second for loop? would it be n-i? from my understanding a the first for loop will go n times, but the index in the second for loop is set to i instead. //where n is the number elements in an array for (int i = 0; i < n; i++) { for (int j = i; j < n; j++) { // Some Constant time task } }

Excluded URL (s)

01 None

Content

GitHub Repository Link:
https://github.com/Ritik2807/DAA_LAB_1_Ritik_Kumar_590017256.git

Objective: Implement and analyze Binary Search algorithm under different scenarios (Best, Worst, Average case).

Source Code (C):
#include
#include

```
#include

int binarySearch(int arr[], int n, int k)
{
1. int low = 0, high = n - 1;
while (low <= high)
{
int mid = low + (high - low) / 2;
if (arr[mid] == k)
return mid;
else if (arr[mid] < k)
low = mid + 1;
else
high = mid - 1;
}
return -1;
}
int main()
{
clock_t start, end;
double cpu_time_used;

int test_sizes[] = {1, 5, 10, 100, 1000};
int i, j;

printf("Binary Search Performance Analysis (Best, Worst, Average Cases)\n");
printf("================================================================\n");

for (i = 0; i < 5; i++)
{
int n = test_sizes[i];
int *arr = (int *)malloc(n * sizeof(int));

for (j = 0; j < n; j++)
{
arr[j] = j * 2;
}
int k = arr[n / 2];
start = clock();
binarySearch(arr, n, k);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Array Size: %d | Best Case: %d | Time: %lf sec\n", n, k, cpu_time_used);

k = -1;
start = clock();
binarySearch(arr, n, k);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Array Size: %d | Worst Case: %d | Time: %lf sec\n", n, k,
cpu_time_used);

k = arr[0];
start = clock();
binarySearch(arr, n, k);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Array Size: %d | Average Case: %d | Time: %lf sec\n", n, k,
cpu_time_used);

printf("----------------------------------------------------------------\n");

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

}

Summary of all 15 Test cases:
Best Case (5 cases)
In the best-case scenario for Binary Search, the element is found in the very first comparison, meaning it is located exactly at the middle index of the array during the initial check. The execution times for the five best-case runs were minimal and consistent, showing almost negligible differences even with increasing input sizes. This is because Binary Search terminates immediately after the first comparison in such situations. Regardless of whether the array had 10, 100, or 10,000 elements, the algorithm only performed O(1) work in these cases.

Worst Case (5 cases)
The worst case occurs when the target element is either not present in the array at all or lies at one of the extreme ends (first or last position) of the sorted array. In such cases, Binary Search must repeatedly halve the search space until only one element remains, performing the maximum number of comparisons, which grows as O(log n). The recorded times in these runs were higher compared to the best cases, and the increase was more noticeable with larger input sizes. For example, searching in an array of 10,000 elements took significantly longer than in one with 100 elements, though still far less than a linear search would take.

Average Case (5 cases)
The average case represents situations where the target element is located somewhere in the middle levels of the search process — neither found immediately like in the best case, nor requiring the full search depth like in the worst case. Here, the algorithm performs a moderate number of comparisons, typically around half the number of comparisons made in the worst case. The recorded execution times were between those of the best and worst cases, showing a gradual rise with input size. The growth still followed a logarithmic pattern, reflecting Binary Search's efficiency.

Overall Observation
Across all 15 runs, the performance graph confirmed that the execution time increases very slowly with input size, validating the O(log n) time complexity. The best-case execution times were nearly constant, the worst-case times increased slightly more with size, and the average cases consistently lay between the two extremes. This makes Binary Search a highly efficient algorithm for large, sorted datasets, with worst-case performance still far better than linear search.

Binary Search Performance Graph:

Observations & Analysis of Binary Search Performance:

1. Best Case (Target Found at Middle)
o In the best case, binary search finds the element in the first comparison.
o This results in O(1) time complexity.
o The graph shows almost constant time for best-case executions regardless of input size.
o The small rise in execution time for larger inputs is due to system overhead and not the algorithm itself.
2. Average Case (Target Found After $\log_2(n)$ Steps)
o On average, binary search requires about $\log_2(n)$ comparisons.
o The graph shows a slow logarithmic growth — execution time increases very slightly with input size.
o This confirms the O(log n) time complexity in practice.

3. Worst Case (Target Not Found)
o In the worst case, binary search still only requires $\log_2(n)$ comparisons, but every step is executed until the search space is exhausted.
o The graph shows slightly higher times than the average case for large inputs due to extra comparisons and function overhead.
4. General Observation
o Binary search scales extremely well with input size.
o Even for very large arrays, execution times remain very low.
o Best, average, and worst cases are close together, with the main differences caused by the number of comparisons and branch evaluations.

Conclusion:
Binary search is a highly efficient searching algorithm for sorted datasets, achieving O(1) time in the best case and O(log n) time in both average and worst cases.
The performance graph confirms that its execution time grows very slowly even with large input sizes, making it ideal for applications where quick lookups are needed.
Since the best, average, and worst cases have minimal time difference, binary search is consistently reliable and scalable for real-world use.

References