

# ADA PRACTICAL 1

**1A) Objective:** The goal of the **Linear Search** algorithm is to:

- Find a specified value (called the key) by examining each item in an array or list, one by one.
- Return the position (index) of the key if it is found.
- Indicate clearly if the key does not exist in the list.
- Linear Search is an easy-to-understand method that requires no data sorting and works for both sorted and unsorted data.

**Algorithm:** Linear Search.

1. Start from the first element (index 0) in the array.
2. Check the current element:
  - If it matches the key, print the position and stop.
  - Otherwise, move to the next element.
3. Repeat this process until you reach the end of the array.
4. If no element matches the key by the end of the array, print "Element not found".

**Pseudo Code :** Linear Search.

Algorithm LinearSearch(A, n, key)

Step 1: Set  $i \leftarrow 0$

Step 2: Repeat while  $i < n$

    If  $A[i] = \text{key}$  then

        Print "Element found at position",  $i$

        Exit

    EndIf

$i \leftarrow i + 1$

Step 3: If  $i = n$  then

    Print "Element not found"

End Algorithm

# ADA PRACTICAL 1

## Code: Linear Search in C.

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

int linear_search(int arr[], int key, int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

void generateRandomArr(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100000;
    }
}

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

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

    generateRandomArr(arr, n);
    int key = arr[rand() % n];

    int repetitions = 1000000;

    clock_t start = clock();
    for (int i = 0; i < repetitions; i++) {
        linear_search(arr, key, n);
    }
    clock_t end = clock();

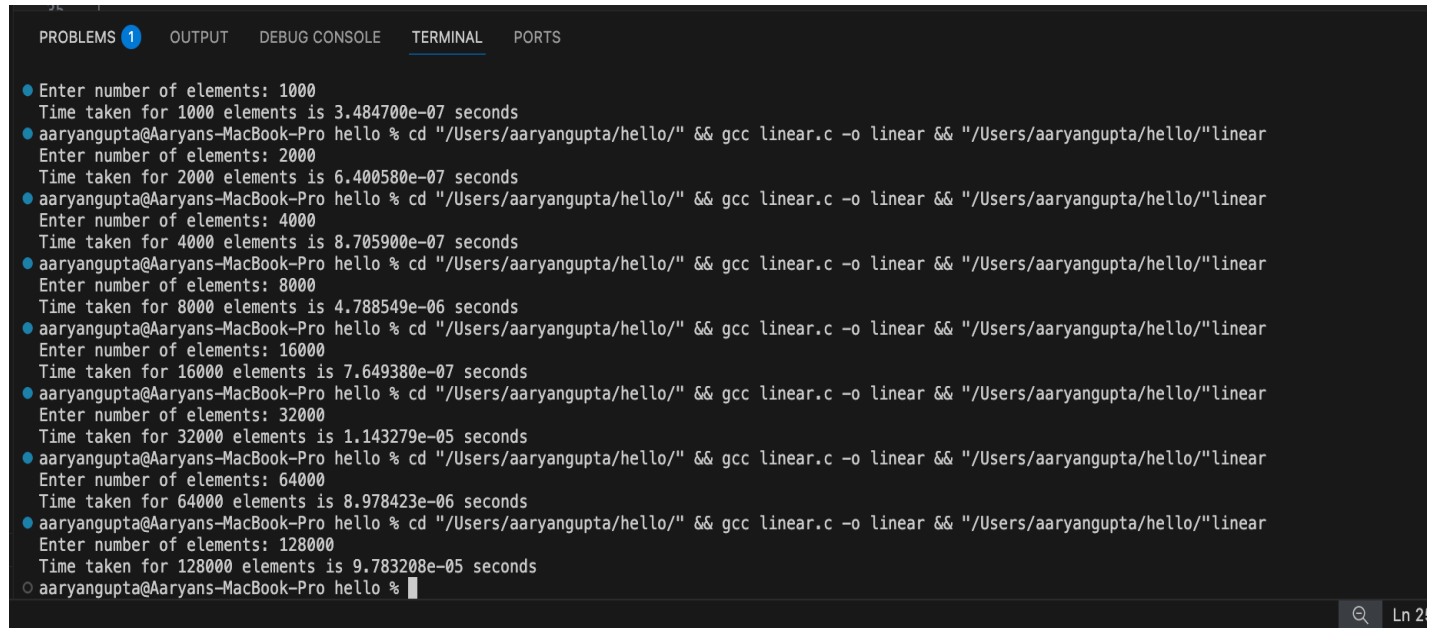
    double total_time = (double)(end - start) / CLOCKS_PER_SEC;
    double time_per_search = total_time / repetitions;
```

# ADA PRACTICAL 1

```
printf("Time taken for %d elements is %e seconds\n", n, time_per_search);

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

## Output:



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

• Enter number of elements: 1000
  Time taken for 1000 elements is 3.484700e-07 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 2000
  Time taken for 2000 elements is 6.400580e-07 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 4000
  Time taken for 4000 elements is 8.705900e-07 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 8000
  Time taken for 8000 elements is 4.788549e-06 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 16000
  Time taken for 16000 elements is 7.649380e-07 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 32000
  Time taken for 32000 elements is 1.143279e-05 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 64000
  Time taken for 64000 elements is 8.978423e-06 seconds
• aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc linear.c -o linear && "/Users/aaryangupta/hello/"linear
  Enter number of elements: 128000
  Time taken for 128000 elements is 9.783208e-05 seconds
○ aaryangupta@Aaryans-MacBook-Pro hello %
```

## Python for plotting the graph: Linear Search.

```
import matplotlib.pyplot as plt

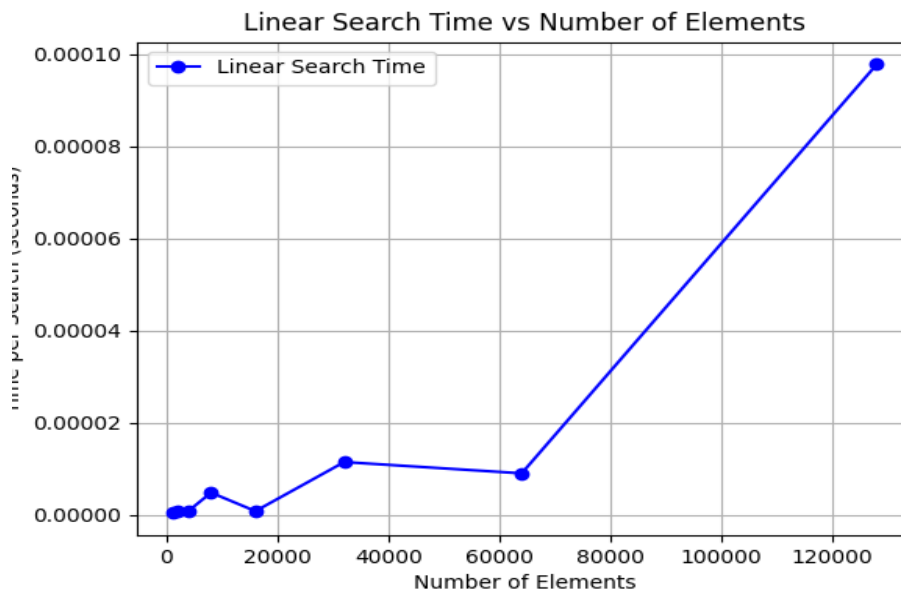
elements = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]
times = [
    3.484700e-07,
    6.400580e-07,
    8.705900e-07,
    4.788549e-06,
    7.649380e-07,
    1.143279e-05,
    8.978423e-06,
    9.783208e-05
]

plt.plot(elements, times, marker='o', color='blue', label='Linear Search Time')
plt.xlabel('Number of Elements')
```

# ADA PRACTICAL 1

```
plt.ylabel('Time per Search (seconds)')
plt.title('Linear Search Time vs Number of Elements')
plt.grid(True)
plt.legend()
plt.show()
```

## Graph : Linear Search



## Factors Contributing to Irregularities in the Linear Search Timing Graph

- **Background System Processes:**  
The processor simultaneously manages other applications and system tasks, leading to fluctuations in the measured search times.
- **Variable Element Positions:**  
The location of the target element within the array can differ widely—sometimes early, sometimes late, or absent—causing variations in how long each search takes.
- **Memory Access Differences:**  
Data retrieval speeds vary depending on whether the information is fetched from the CPU cache or from main memory, affecting overall timing results.
- **Randomized Test Data:**  
Experiments conducted on different randomly generated arrays mean the target's position changes each time, resulting in timing inconsistencies.

# ADA PRACTICAL 1

## **1B) Objective: Binary Search:**

The aim of the Binary Search algorithm is to quickly locate a specific value (the target) in a sorted array or list. Instead of checking each item sequentially as in linear search, binary search works by repeatedly splitting the search range in half and focusing only on the relevant half. This approach significantly speeds up the search process, resulting in a time complexity of  $O(\log n)$ . Binary search is applicable only to data that is already sorted.

## **Algorithm: Binary Search:**

1. Begin by setting two variables, one pointing to the start of the array (let's call it low) and the other to the end (call it high).
2. Repeat the following steps as long as low does not surpass high:
  - Find the middle element's position by taking the average of low and high.
  - If the item at the middle position matches the value you're searching for, report its position and end the search.
  - c. If the middle value is less than the desired value, move the low pointer just after the middle, so your new search range is the right half.
  - d. If the middle value is greater than the target, move the high pointer just before the middle, narrowing the search to the left half.
3. If you exit the loop without finding your item, it means the value isn't present in the array.

## **Pseudocode of Binary Search:-**

BinarySearch(array, size, target):

Input: array (sorted), size (number of elements), target (item to find)

Output: index of target if present, otherwise -1

start  $\leftarrow$  0

end  $\leftarrow$  size - 1

while start  $\leq$  end:

    middle  $\leftarrow$  (start + end) // 2

    if array[middle] == target:

        return middle // target found at index middle

    else if array[middle] < target:

        start  $\leftarrow$  middle + 1

    else:

        end  $\leftarrow$  middle - 1

return -1 // target not present in array

# ADA PRACTICAL 1

## Code in C: Binary Search:

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

int binary_search(int a[], int x, int size) {
    int l = 0, r = size - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (a[m] == x)
            return 1;
        if (a[m] > x)
            r = m - 1;
        else
            l = m + 1;
    }
    return 0;
}

void randarr(int a[], int n) {
    a[0] = rand() % 10;
    for (int i = 1; i < n; i++) {
        a[i] = a[i - 1] + rand() % 10;
    }
}

int main() {
    int n;
    printf("enter no of elements: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid value of n\n");
        return 1;
    }

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    randarr(arr, n);
    int x = arr[rand() % n];
    clock_t start = clock();
    for (int i = 0; i < 1000000; i++)
        binary_search(arr, x, n);
    clock_t end = clock();
    double time_per_search = ((double)(end - start) / CLOCKS_PER_SEC) / 1000000.0;
```

# ADA PRACTICAL 1

```
printf("Time taken for %d elements is %e\n", n, time_per_search);
free(arr);
return 0;
}
```

## OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

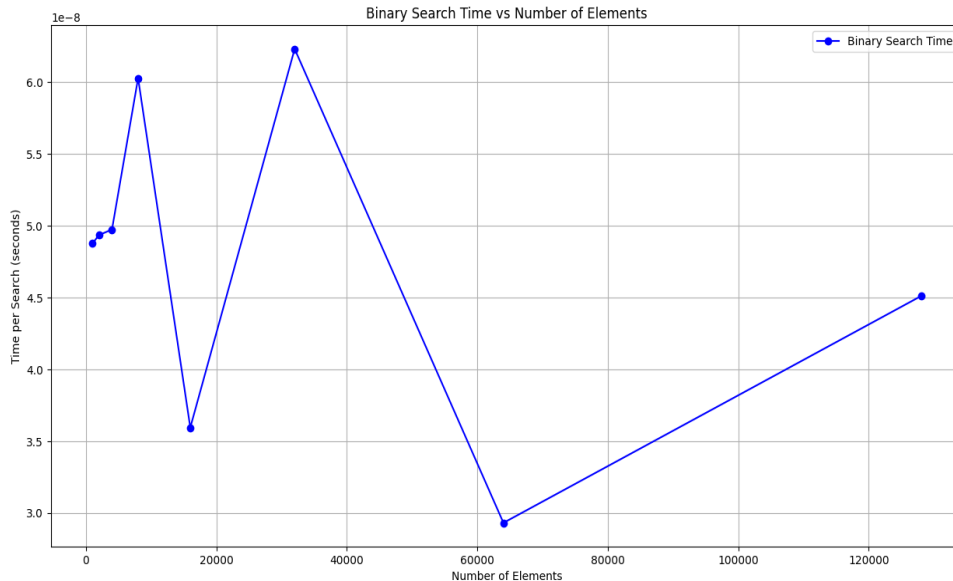
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 1000
Time taken for 1000 elements is 4.876600e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 2000
Time taken for 2000 elements is 4.934900e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 4000
Time taken for 4000 elements is 4.974400e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 8000
Time taken for 8000 elements is 6.027200e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 16000
Time taken for 16000 elements is 3.595500e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 32000
Time taken for 32000 elements is 6.230600e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 64000
Time taken for 64000 elements is 2.932600e-08
● aaryangupta@Aaryans-MacBook-Pro hello % cd "/Users/aaryangupta/hello/" && gcc binary.c -o binary && "/Users/aaryangupta/hello/"binary
enter no of elements: 128000
Time taken for 128000 elements is 4.510500e-08
○ aaryangupta@Aaryans-MacBook-Pro hello %
```

## Python for plotting the graph

```
binary.py > ...
1  import matplotlib.pyplot as plt
2
3  elements = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]
4  times = [4.876600e-08, 4.934900e-08, 4.974400e-08, 6.027200e-08, 3.595500e-08, 6.230600e-08, 2.932600e-08, 4.510500e-08]
5
6  plt.plot(elements, times, marker='o', color='blue', label='Binary Search Time')
7  plt.xlabel('Number of Elements')
8  plt.ylabel('Time per Search (seconds)')
9  plt.title('Binary Search Time vs Number of Elements')
10 plt.legend()
11 plt.grid(True)
12 plt.show()
```

# ADA PRACTICAL 1

## Graph: Binary Search



## Conclusion for Binary Search Timing Experiment

- The experimental graph shows how the time taken by binary search changes with the number of elements in a sorted array.
- Theoretically, binary search has a logarithmic time complexity  $O(\log n)$ , which means the search time grows very slowly even when the array becomes very large.
- This slow increase in time makes binary search highly efficient and suitable for searching within large sorted datasets compared to linear search.
- Although small variations in the graph exist due to system noise or measurement inaccuracies, the overall trend confirms that binary search maintains consistently fast performance.
- The experimental results support the expected logarithmic complexity, demonstrating binary search as an optimal method for searching sorted arrays.
- In summary, both practical measurements and theoretical understanding agree that binary search is a quick and dependable algorithm for large sorted data.