# 1(A) LINEAR SEACH

## C CODE:

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

// Linear Search function
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i; // return index if found
    }
    return -1; // not found
}

int main() {
    int n, key, result;
    clock_t start, end;
    double cpu_time_used;

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

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }


    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

)
    key = n;

    start = clock();
    result = linearSearch(arr, n, key);
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

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

    printf("Time taken: %f seconds\n", cpu_time_used);

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

# PYTHON SCRIPT:

```
import matplotlib.pyplot as plt


n_values = [1000, 5000, 10000, 20000, 50000, 100000]
time_values = [0.00001, 0.00004, 0.00009, 0.00018, 0.00045, 0.0009]  # in seconds

plt.plot(n_values, time_values, marker='o')
plt.title("Linear Search: Time vs Number of Elements")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.grid(True)
plt.show()
```

# PSEUDO CODE:

**C CODE:**

```
Procedure LinearSearch(Array, n, key)
    For i ← 0 to n-1
        If Array[i] = key Then
            Return i   // Found at index i
        End If
    End For
    Return -1
End Procedure


Main Program:
    Input n
    Create Array of size n
    Fill Array with values (e.g., 1 to n)
    key ← n

    Start timer
    result ← LinearSearch(Array, n, key)
    Stop timer

    If result ≠ -1 Then
        Print "Element found at index", result
    Else
        Print "Element not found"
    End If

    Print "Time taken:", (Stop - Start)
End Program
```

**PYTHON CODE:**

Import matplotlib library

Initialize list n_values = [1000, 5000, 10000, 20000, 50000, 100000]
Initialize empty list time_values

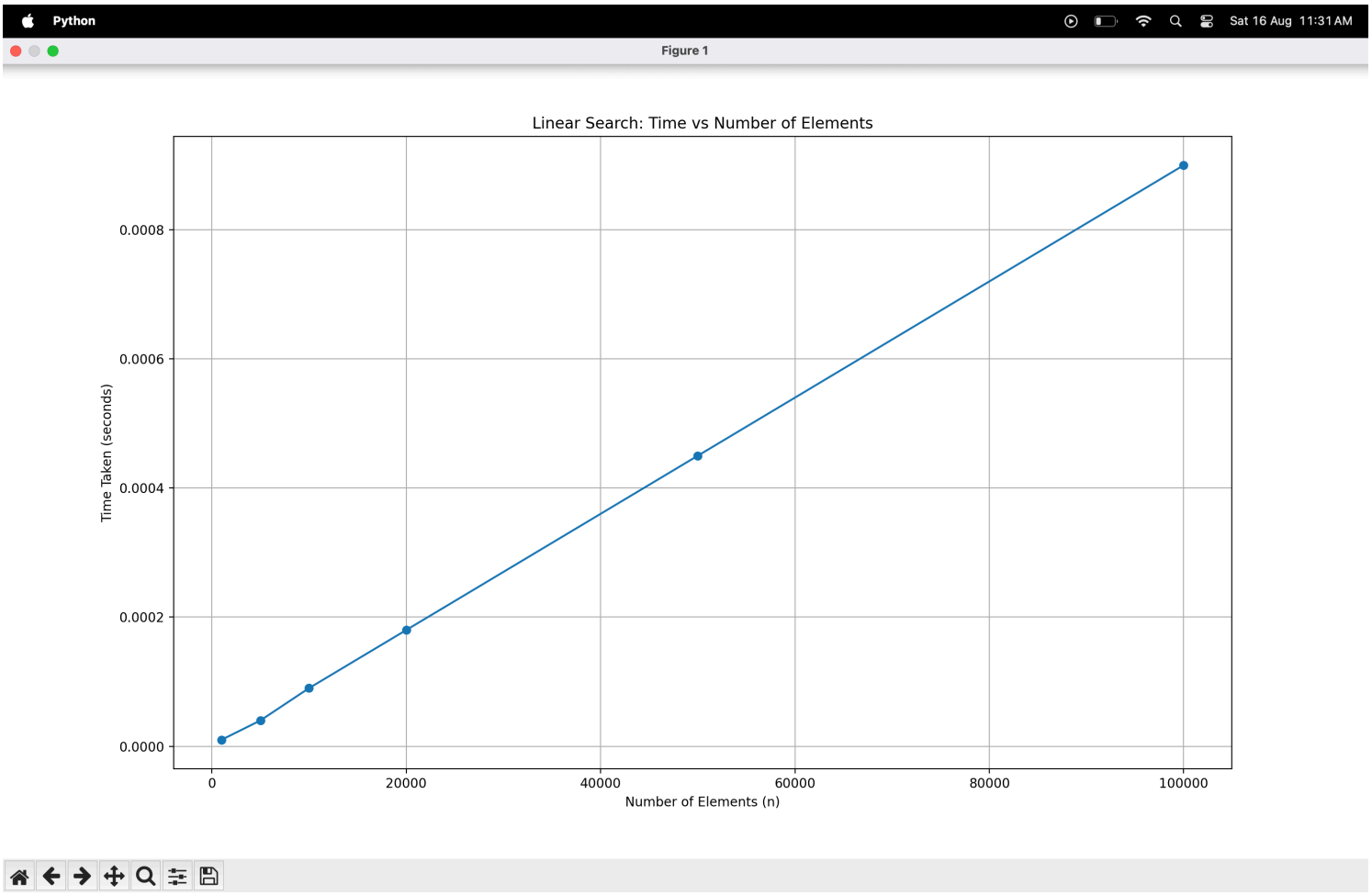
For each n in n_values:
    Create an array of numbers from 1 to n
    Set key = n (worst case)

    Start timer
    Perform Linear Search on array
    Stop timer

    Append (Stop - Start) to time_values

Plot graph with:
    X-axis = n_values
    Y-axis = time_values
    Title = "Linear Search: Time vs Number of Elements"
    Labels = ("Number of Elements", "Time Taken (seconds)")

Display graph



# 1(B) BINARY SEARCH

## C CODE:

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

// Binary Search function
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int n, key, result;
    clock_t start, end;
    double cpu_time_used;

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

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }
    key = n;
    start = clock();
    result = binarySearch(arr, n, key);
    end = clock();

    cpu_time_used = ((double)(end - start))

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

    printf("Time taken: %f seconds\n", cpu_time_used);

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

# PYTHON SCRIPT:

```
import matplotlib.pyplot as plt

# Example data (replace with your actual recorded values)
n_values = [1000, 5000, 10000, 20000, 50000, 100000]
time_values = [0.000001, 0.000002, 0.000003, 0.000003, 0.000004, 0.000004]  # in seconds

plt.plot(n_values, time_values, marker='o', label="Binary Search")
plt.title("Binary Search: Time vs Number of Elements")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.grid(True)
plt.legend()
plt.show()
```

# PSEUDO CODE:

**C CODE:**

```
Procedure BinarySearch(Array, n, key)
    low ← 0
    high ← n - 1
    While low ≤ high Do
        mid ← (low + high) / 2
        If Array[mid] = key Then
            Return mid        // Element found
        Else If Array[mid] < key Then
            low ← mid + 1
        Else
            high ← mid - 1
        End If
    End While
    Return -1
End Procedure


Main Program:
    Input n
    Create Array of size n
    Fill Array with values (1 to n) in sorted order
    key ← n

    Start timer
    result ← BinarySearch(Array, n, key)
    Stop timer

    If result ≠ -1 Then
        Print "Element found at index", result
    Else
        Print "Element not found"
    End If

    Print "Time taken:", (Stop - Start)
End Program
```

**PYTHON CODE:**

Import matplotlib library

Initialize list n_values = [1000, 5000, 10000, 20000, 50000, 100000]
Initialize empty list time_values

For each n in n_values:
    Create sorted array of numbers from 1 to n
    Set key = n (worst case)

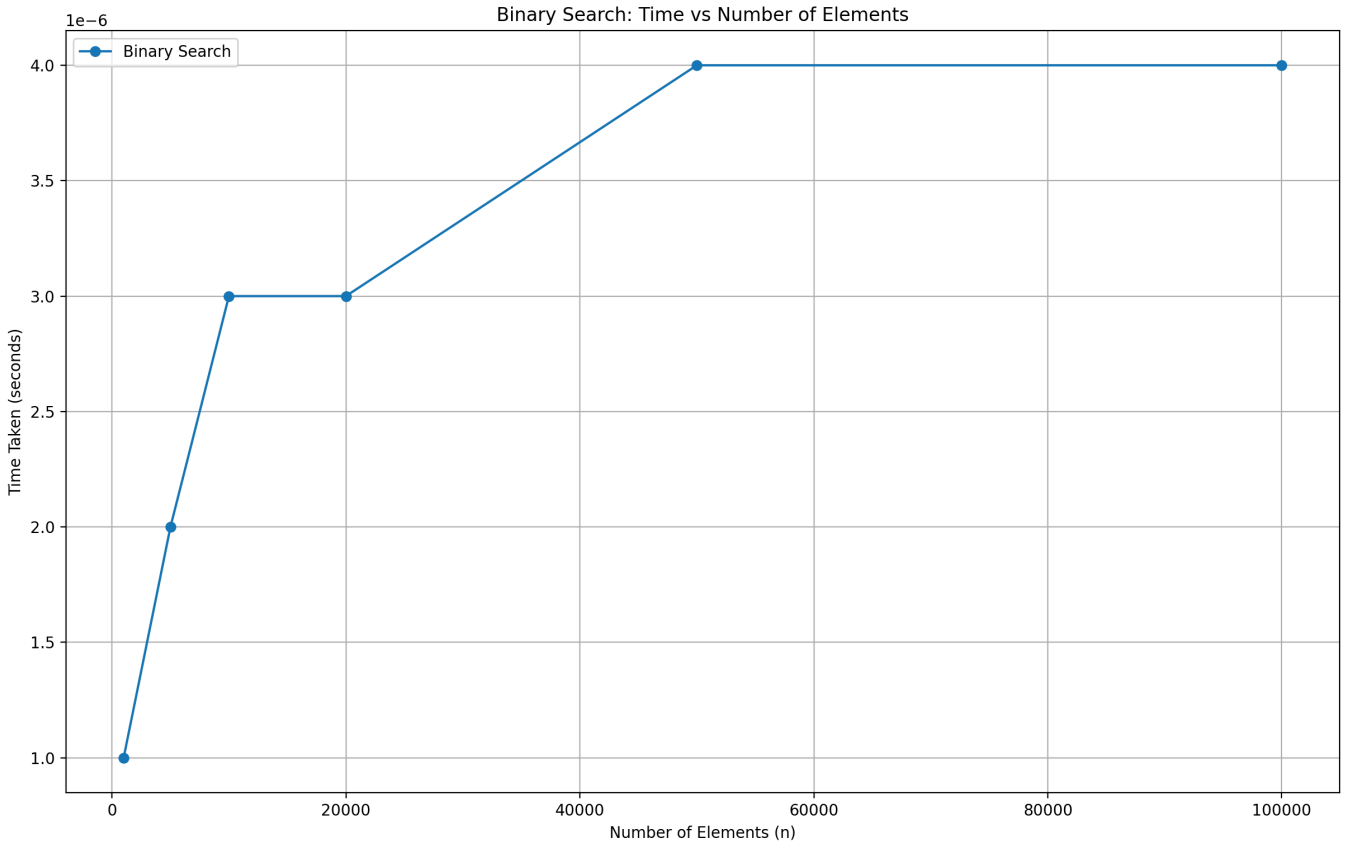
    Start timer
    Perform BinarySearch on array
    Stop timer

    Append (Stop - Start) to time_values

Plot graph with:
    X-axis = n_values
    Y-axis = time_values
    Title = "Binary Search: Time vs Number of Elements"
    Labels = ("Number of Elements", "Time Taken (seconds)")

Display graph

---



# CONCLUSION:

From the two experiments, we clearly see a difference in how **Linear Search** and **Binary Search** perform as the number of elements (`n`) grows.

- **Linear Search** goes through each element one by one until it finds the target.
    - This means if the element is at the end, it will check almost the entire list.
    - The time taken **increases directly with `n`** (O(n)).
    - On the graph, the line keeps rising steadily as `n` gets larger.
- **Binary Search** works in a much smarter way.
    - Since the list is sorted, it keeps dividing the search space into halves.
    - This reduces the work drastically — even for a huge list, it takes only a few steps.
    - The time grows **very slowly** with `n` (O(log n)), almost flat in the graph compared to linear search.

## 🔷 What We Learn

1. **Efficiency matters** – When the dataset is small, both searches feel equally fast. But as the dataset grows, linear search becomes slower, while binary search still remains quick.
2. **Sorted data unlocks better algorithms** – Binary search only works on sorted lists. If we can arrange our data in order, searching becomes far more efficient.
3. **Theory matches reality** – The experiment confirms what we learn in algorithm analysis:
    - Linear Search: O(n)
    - Binary Search: O(log n)