Chandpur Science And Technology University

Lab Report

Course Code : CSE 2201

Course Title : Algorithm Design and Analysis

Experiment no : 01

Experiment Name: Introduction to Algorithm Design & Complexity Analysis

Submitted By:

Name: Turja Chakraborty

ID: B210101001

Program: B.Sc. in CSE

Submitted To:

Name: Mustafizur Rahaman

Lecturer Dept. of CSE

Chandpur science and technology university

Experiment # 1: *Linear Search & Step Analysis*

Objective:

To implement the Linear Search algorithm in C, analyze its performance by counting the number of steps taken to find an element in arrays of different sizes (n = 10, 100, 1000), and compare theoretical and practical results for worst-case scenarios.

Algorithm:

1) Start from the first element of the array.
2) Compare the current element with the key:
   - If the element matches the key, return its index (position).
3) If the element does not match, move to the next element.
4) Repeat steps 2 and 3 until:
   - The key is found, or
   - The end of the array is reached.
5) If the key is not found after scanning the entire array, return -1 to indicate "not found".

Theoretical Solution of given problem :

- Best Case: Key is at the first index → Steps = 1
- Average Case: Key is in the middle → Steps = n / 2
- Worst Case: Key is at the last index or not found → Steps = n

Expected Worst-Case Step Counts:

Array Size (n)	Expected Steps (Worst Case)
10	10
100	100
1000	1000

Practical Work:

a. Pseudocode:

```
FUNCTION LinearSearch(array, size, key):
    SET steps = 0
    FOR i FROM 0 TO size-1 DO:
        steps = steps + 1
        IF array[i] == key THEN:
```

RETURN (i, steps)
END FOR
RETURN (-1, steps)

FUNCTION TestSearch(size, key):
CREATE array of given size
FILL array with sorted values from 1 to size
CALL LinearSearch(array, size, key)
PRINT "Steps taken"

Source Code in C :

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int key, int* steps) {
    for (int i = 0; i < n; i++) {
        (*steps)++;
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

```c
void testSearch(int n, int key) {
    int arr[n];
    int steps = 0;
```

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

```c
    int index = linearSearch(arr, n, key, &steps);
```

```c
    printf("Array size: %d, Search key: %d\n", n, key);
    if (index != -1)
        printf("Key found at index %d\n", index);
```

```
    else
        printf("Key not found.\n");

    printf("Steps taken: %d\n\n", steps);
}
// worst case
int main() {
    testlinearsearch(10, 10);
    testlinearsearch(100, 100);
    testlinearsearch(1000, 1000);

    return 0;
}
```

Output:

Array size: 10, Search key: 10
Key found at index 9
Steps taken: 10

Array size: 100, Search key: 100
Key found at index 99
Steps taken: 100

Array size: 1000, Search key: 1000
Key found at index 999
Steps taken: 1000

Observations:

- Step Count Grows Linearly
  As expected, the number of steps taken is directly proportional to the array size. For every additional element, one more comparison is needed in the worst case.

- Experimental vs. Theoretical Results
  The experimental results matched the theoretical analysis exactly: - For n = 10 → Steps = 10
  - For n = 100 → Steps = 100
  - For n = 1000 → Steps = 1000

- Predictable Behavior
  Linear Search behaves consistently — the steps are predictable and increase at a steady rate as the size of the array increases.

Challenges:

- Handling Large Input Sizes

When testing with large arrays (e.g., n = 1000 or more), memory management becomes crucial, especially in environments with limited stack size. Using variable-length arrays can lead to stack overflow in some compilers.

- Worst-Case Identification

Ensuring that the search key is placed at the worst-case position (i.e., last index) during testing is important for accurate step analysis. Any mistake here can give misleading performance data

Conclusion :

n this project, we successfully implemented the Linear Search algorithm in C and analyzed its performance across different input sizes (n = 10, 100, 1000). By counting the number of steps (comparisons) taken during the search, we validated the theoretical time complexity of Linear Search.

Key takeaways:

- Linear Search has a time complexity of O(n), meaning its performance degrades linearly with the size of the array.
- The number of steps in the worst case is equal to the size of the array.
- While it is simple and easy to implement, Linear Search is not suitable for large datasets due to its inefficiency.
- More optimized algorithms like Binary Search are recommended when the data is sorted and speed is essential.

Experiment # 2: Binary *Search & Step Analysis:*

Objective:

To implement the Binary Search algorithm in C, count the number of steps (comparisons) taken for different input sizes (n = 10, 100, 1000), and compare the practical step count with the theoretical analysis for worst-case scenarios.

Algorithm:

Binary Search is an efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half. If the key is less than the middle element, it searches in the left half; otherwise, it searches in the right half.

Steps:

Start with two pointers: low and high.

While low ≤ high:

Calculate mid = (low + high) / 2.

If key == arr[mid], return mid.

If key < arr[mid], search the left half.

If key > arr[mid], search the right half.

If the key is not found, return -1.

Theoretical Solution of the Problem:

Time complexity of Binary Search:

1)Best Case: O(1) (when the key is at the middle)

2)Average & Worst Case: O($\log_2 n$)

Theoretical Step Counts (Worst Case):

Array Size (n)	$\log_2(n)$ (approx)	Steps (Worst Case)
10	≈ 4	4
100	≈ 7	7

Array Size (n)	$\log_2(n)$ (approx)	Steps (Worst Case)
1000	≈ 10	10

Practical Work:

a. Pseudocode:

```
WHILE low ≤ high DO

    INCREMENT steps_ref by 1

    SET mid ← (low + high) / 2

    IF array[mid] = key THEN

        RETURN mid  // Key found
```

```
    ELSE IF array[mid] < key THEN

        SET low ← mid + 1  // Search right half

    ELSE

        SET high ← mid - 1  // Search left half

END WHILE

RETURN -1  // Key not found
```

b. Source Code in C:

```c
#include <stdio.h>


int binarySearch(int arr[], int n, int key, int* steps) {
    int low = 0, high = n - 1;
```

```c
    while (low <= high) {
        (*steps)++;
        int mid = (low + high) / 2;
```

```c
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
```

```c
    return -1;
}
```

```c
void testbinarysearch(int n, int key) {
    int arr[n];
    int steps = 0;
```

```c

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

```c
    int index = binarySearch(arr, n, key, &steps);
```

```c
    printf("Array size: %d, Search key: %d\n", n, key);
    if (index != -1)
    printf("Key found index %d\n", index);
    else
    printf("Key not found.\n");
```

```c
    printf("Steps taken: %d\n\n", steps);
}
```

```c
// worst case
int main() {
    testbinarysearch(10, 10);
    testbinarysearch(100, 100);
   testbinarysearch(1000, 1000);
```

```c
    return 0;
```

```
}
```

Output:

Key found at index 9

Steps taken: 4

Array size: 100, Search key: 100

Key found at index 99

Steps taken: 7

Array size: 1000, Search key: 1000

Key found at index 999

Observations:

The number of steps grows logarithmically with the size of the array.

Experimental results confirm the theoretical estimates:

For n = 10 → Steps ≈ 4

For n = 100 → Steps ≈ 7

For n = 1000 → Steps ≈ 10

Binary Search is extremely efficient compared to Linear Search for large sorted arrays.

Challenges:

Ensuring that the array is sorted before applying Binary Search is mandatory.

Correctly handling integer division when calculating the mid index.

In systems where array sizes are very large, memory handling and stack limits might need attention.

Conclusion:

Binary Search significantly reduces the number of comparisons needed to find an element in a sorted array. Its logarithmic time complexity makes it far more efficient than Linear Search for large datasets. The implementation successfully matched theoretical expectations, confirming that Binary Search is optimal for sorted data.

Let me know if you'd like this formatted into a Word or PDF report — or need a comparison table between linear and binary search included!

Experiment # 3: Bubble Sort – Complexity Analysis:

Objective :

To implement the Bubble Sort algorithm in C, analyze the number of steps (comparisons and swaps) for different input sizes, and compare practical results with the theoretical time complexity.

Algorithm:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process continues until the array is fully sorted.

Steps:

Repeat (n – 1) times:

For i = 0 to n – j – 1:

If arr[i] > arr[i + 1], swap them.

The largest unsorted element "bubbles" to the end of the array in each pass.

Theoretical Solution of the Problem:

Time Complexity of Bubble Sort:

Case	Comparisons	Time Complexity
Best Case	**n – 1**	**O(n) (if optimized with a flag)**
Worst Case	**n(n – 1)/2**	**O(n^2)**
Average	**n(n – 1)/2**	**O(n^2)**

Practical Work:

a. Pseudocode :

1) Set steps ← 0

2) For i ← 0 to n - 2 do a. Set swapped ← false b. For j ← 0 to n - i - 2 do i. steps ← steps + 1 ii. If arr[j] > arr[j + 1] then A. Swap arr[j] and arr[j + 1] B. Set swapped ← true c. If swapped = false then i. Break // array is already sorted

3) Return steps

b. Source Code in C:

```
void bubbleSort(int arr[], int n, int* steps)
{
    for (int i = 0; i < n - 1; i++)

    {
        int swapped = 0;
for (int j = 0; j < n - i - 1; j++) {
    (*steps)++;
```

```
    if (arr[j] > arr[j + 1]) {

        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
        swapped = 1;
    }
}
```

```
if (swapped == 0)
    break;
}
}
```

```
void testbubblesort(int n)
{
    int arr[n];
    int steps = 0;
for (int i = 0; i < n; i++)
{
    arr[i] = n - i;
}
```

```
bubbleSort(arr, n, &steps);

printf("Array size: %d\n", n);
printf("Steps : %d\n\n", steps);
}

int main()
{ testbubblesort(10);
    testbubblesort(100);
    testbubblesort(1000);
    return 0;
}
```

Output:

Array size: 10

Steps (comparisons): 45

Array size: 100

Steps (comparisons): 4950

Array size: 1000

Steps (comparisons): 499500

Observations:

Array Size (n)	Expected Comparisons	Actual Comparisons
10	45	45
100	4950	4950
1000	499500	499500

Challenges:

Bubble Sort is inefficient for large datasets.

Without optimization (e.g., break early if no swaps), even already sorted arrays go through full iterations.

Manually counting steps and verifying swap behavior for small datasets is straightforward, but harder with large n.

Conclusion:

Bubble Sort is a fundamental sorting algorithm used to demonstrate basic sorting principles. While easy to implement and understand, it is inefficient for large inputs. The practical results closely match the theoretical complexity of $O(n^2)$. For performance-critical applications, advanced algorithms like Merge Sort or Quick Sort are preferred.