|  |  |
| --- | --- |
|  | Department of Computer Science and Engineering  Chandpur Science and Technology University |

**LAB-01**

**Course Title**: Algorithm Design and Analysis Sessional

**Course Code**:CSE 2202

**Submitted To-**

**Mostafiz Ahammed**

*Lecturer****Department of Computer Science and Engineering****Chandpur Science and Technology University*

**Submitted By-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl** | Class Roll | Name | Batch | Signature |
| 01 | B210101022 | Md. Rabbi Ahammed | CSE-21 | rabbi |

**Date of Submission: July, 2025**

**Experiment 01: *Linear Search & Step Analysis***

**Objective**

To implement the Linear Search algorithm in C++ and analyze the number of steps (comparisons) taken during different cases — best, average, and worst.

**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**

Linear Search is a simple searching technique where each element is compared with the target key sequentially until it is found or the array ends.

**Time Complexity:**  
- Best Case (Key at first index): O(1)  
- Average Case (Key at middle): O(n/2)  
- Worst Case (Key at last index or not present): O(n)

**Space Complexity:** O(1) (no extra memory used)

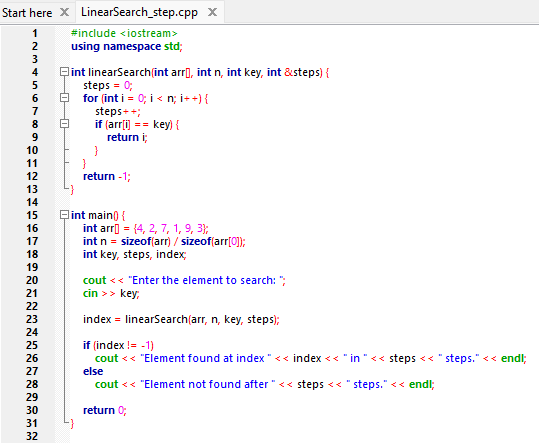
Use Case:  
It is useful for small or unsorted datasets.

**Practical Work**

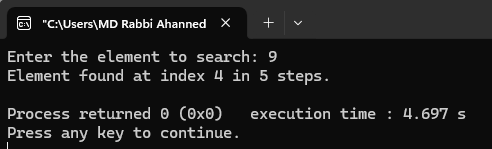
1. **Pseudocode:**

LinearSearch(arr[], n, key):  
 for i = 0 to n-1:  
 if arr[i] == key:  
 return i  
 return -1

1. **Source Code in C++:**

****

**Output:**

****

## Analysis Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | Array | Key | Index Found | Comparisons |
| Best Case | {4, 2, 7, 1, 9, 3} | 4 | 0 | 1 |
| Average Case | {4, 2, 7, 1, 9, 3} | 7 | 2 | 3 |
| Worst Case | {4, 2, 7, 1, 9, 3} | 3 | 5 | 6 |
| Not Found Case | {4, 2, 7, 1, 9, 3} | 10 | -1 | 6 |

## Observations

- The number of steps increases as the key is located farther from the beginning.  
- If the key is not present, all elements must be checked.  
- Performance is directly proportional to the array size.

## Challenges Faced

- Managing step count correctly alongside the search.  
- Handling the input/output format for easy testing.  
- Ensuring clear distinction between found and not found results.

## Conclusion

- Linear search is simple to implement but inefficient for large arrays.  
- The step count (comparisons) is useful to evaluate performance.  
- It is suitable only for small or unsorted datasets.

# Experiment 2: *Binary Search (Requires Sorted Array)*

**Objective**

To implement the Binary Search algorithm in C++ and analyze the number of steps (comparisons) taken during different cases — best, average, and worst.

**Algorithm**

1. Find the middle element of the array.
2. If it's equal to the target, return the index.
3. If the target is less than the middle, search the left half.
4. If the target is greater, search the right half.
5. Repeat until the range is empty.

## Theoretical Solution

Binary Search is an efficient algorithm for finding a key in a sorted array. It works by repeatedly dividing the search interval in half.  
  
**Time Complexity:**  
- Best Case: O(1) (key found at mid on first attempt)  
- Average Case: O(log n)  
- Worst Case: O(log n)

**Space Complexity:**  O(1)  
  
Requirement:  
The array must be sorted before applying Binary Search.

**Practical Work**

1. **Pseudocode:**

int binarySearch(int arr[], int n, int key) {

int low = 0;

int 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;

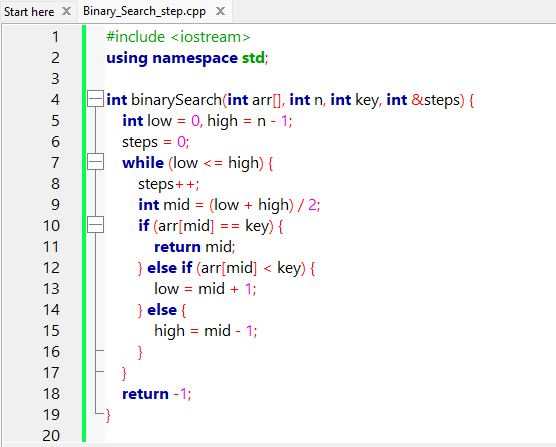
}

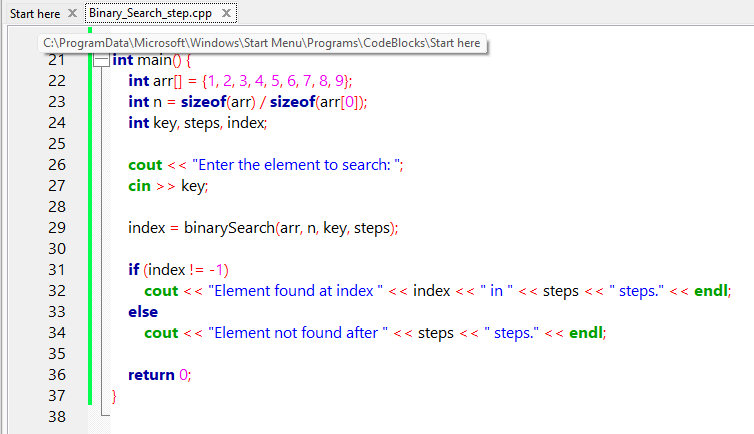
}

return -1;

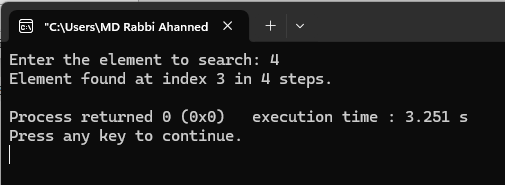
}

1. **Source Code in C++:**





**Output:**



## Analysis Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | Array | Key | Index Found | Comparisons |
| Best Case | {1,2,3,4,5,6,7,8,9} | 5 | 4 | 1 |
| Average Case | {1,2,3,4,5,6,7,8,9} | 7 | 6 | 3 |
| Worst Case | {1,2,3,4,5,6,7,8,9} | 1 | 0 | 4 |
| Not Found Case | {1,2,3,4,5,6,7,8,9} | 10 | -1 | 4 |

## Observations

- Binary search reduces the search space by half after each step.  
- It is significantly faster than linear search for large arrays.  
- Array must be sorted before performing binary search.

## Challenges Faced

- Handling midpoint calculation correctly.  
- Ensuring the input array is sorted before searching.  
- Managing conditions to avoid infinite loops or incorrect results.

## Conclusion

- Binary Search is efficient with logarithmic time complexity.  
- It is ideal for large, sorted datasets.  
- Requires careful implementation to handle edge cases and bounds properly.  
- In practice, it is widely used in systems where performance is critical.

**Experiment 03:** ***Bubble Sort – Complexity Analysis***

# Objective

The objective of this experiment is to:  
- Understand and implement the Bubble Sort algorithm.  
- Analyze its time complexity in best, average, and worst-case scenarios.  
- Compare theoretical complexity with practical execution time.  
- Develop insights into performance trade-offs of simple sorting algorithms.

# Algorithm

Bubble Sort is a comparison-based sorting algorithm where each pair of adjacent elements is compared and the elements are swapped if they are in the wrong order. This process repeats until the array is sorted.

* Working Principle:
* Traverse the array multiple times.
* In each pass, compare adjacent elements.
* Push the largest unsorted element to the end (like a bubble rising).

# Theoretical Solution

**Time Complexity:**

* - Best Case: O(n) – when the array is already sorted.
* - Average Case: O(n²)
* - Worst Case: O(n²) – when the array is sorted in reverse order.

**Space Complexity:** O(1) – in-place sorting

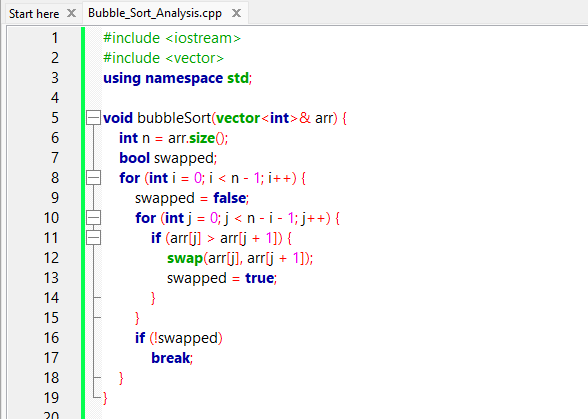
Stability: Bubble Sort is stable, meaning it maintains the relative order of equal elements.

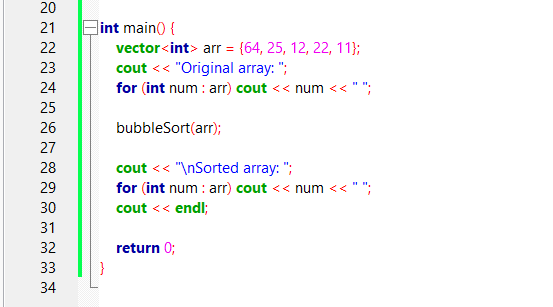
# Practical Work

## a. Pseudocode:

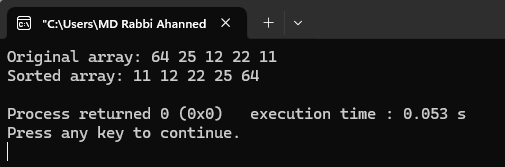
BubbleSort(A, n):  
 for i = 0 to n-1:  
 swapped = false  
 for j = 0 to n-i-2:  
 if A[j] > A[j+1]:  
 swap A[j] and A[j+1]  
 swapped = true  
 if not swapped:  
 break

## b.Source Code in C++:





**Output:**



# Analysis Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Input Size (n) | Best Case (Time) | Worst Case (Time) | Swaps Made |
| Sorted Array | 5 | O(n) | - | 0 |
| Random Order | 5 | - | O(n²) | Varies |
| Reverse Sorted | 5 | - | O(n²) | Maximum |
| Larger Random Set | 1000 | ~O(n²) | ~O(n²) | High |

# Observations

* - Bubble Sort is inefficient for large datasets.
* - It performs well only on small or nearly sorted data.
* - Early termination optimization improves best-case performance significantly.

# Challenges

* - Handling time measurement with precision for small input sizes.
* - Recognizing that Bubble Sort is not practical for large-scale tasks.
* - Ensuring the stability of the sort when testing with duplicate values.

# Conclusion

Bubble Sort is a fundamental sorting algorithm used primarily for educational purposes. It’s easy to implement and understand but inefficient for real-world use due to its quadratic time complexity. Through this experiment, we learned not only the algorithmic logic behind Bubble Sort but also how theoretical complexities translate into practical performance. This forms a foundation for studying more efficient algorithms like Merge Sort or Quick Sort.