**Table Of Index**

|  |  |  |  |
| --- | --- | --- | --- |
| Sl | Excrement | Page | Remark |
| 1 | Linear Search Algorithm |  |  |
| 2 | Binary Search Algorithm |  |  |
| 3 | Insertion Sort Algorithm |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |
| 8 |  |  |  |
| 9 |  |  |  |
| 10 |  |  |  |
| 11 |  |  |  |
| 13 |  |  |  |
| 14 |  |  |  |
| 15 |  |  |  |
|  |  |  |  |

**Experiment Name: Linear Search Algorithm**

**Objective:**

To understand and implement the Linear Search algorithm to find an element in an unsorted list of numbers.

**Theory:**

Linear Search is one of the simplest searching algorithms. It checks each element of the array or list sequentially until the desired element is found or the list ends.

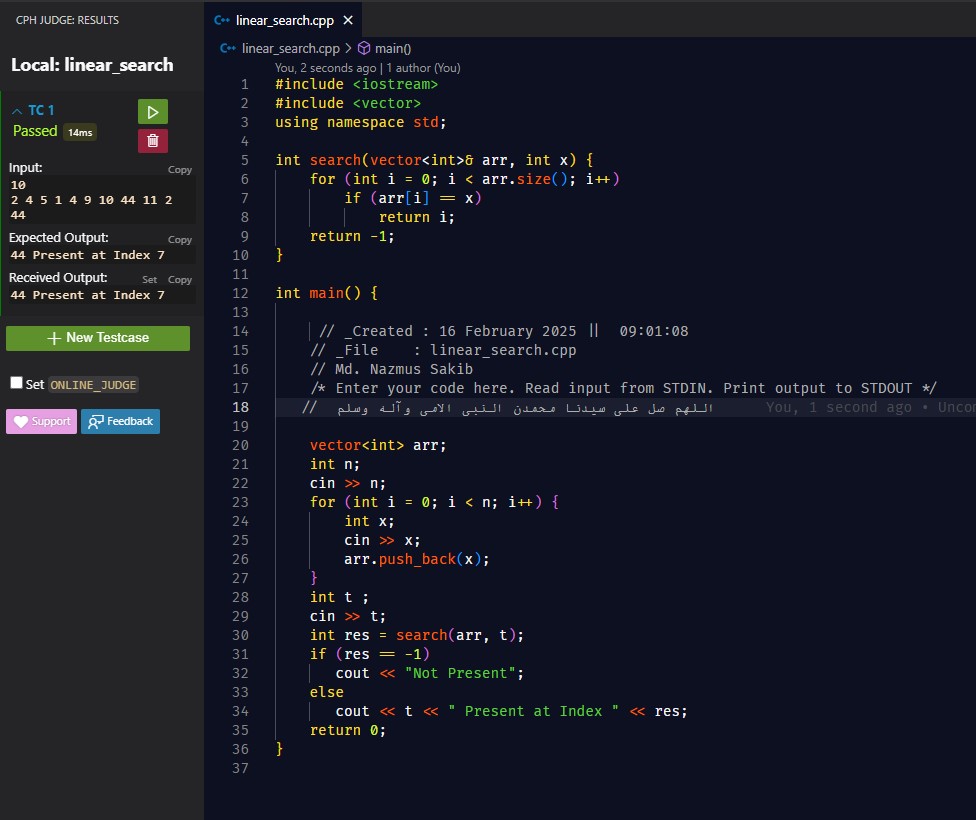
* **Working Principle:**
  + Start from the first element of the list.
  + Compare each element with the target value.
  + If the target value is found, return its position.
  + If the target is not found by the end of the list, return an indication of failure (e.g., -1).
* **Complexity:**
  + **Time Complexity:** O(n), where n is the number of elements in the list.
  + **Space Complexity:** O(1), as no additional data structures are used.
* **Advantages:**
  + Simple and easy to implement.
  + No need for sorted data.
* **Disadvantages:**
  + Not efficient for large datasets.
  + Average and worst-case time complexity is O(n).

**Algorithm Steps:**

1. Start from the first element of the list.
2. Compare the current element with the target value.
3. If they are equal, return the index of the current element.
4. If they are not equal, move to the next element.
5. Repeat steps 2-4 until the end of the list.
6. If the target value is not found, return -1.

**Time Complexity:**  
The time complexity of the Linear Search algorithm is **O(n)**, where n is the number of elements in the list. This is because the algorithm checks each element sequentially, and in the worst-case scenario, it has to examine all n elements.

**Space Complexity:**  
The space complexity of Linear Search is **O(1)**. This is because the algorithm does not require any additional data structures and uses only a few variables, independent of the size of the input list.

In this way, while Linear Search is simple and easy to implement, it may not be very efficient for large datasets.

**Experiment Name:** Binary Search Algorithm  
**Objective:**  
To understand and implement the Binary Search algorithm to efficiently find an element in a sorted list of numbers.

**Theory:**  
Binary Search is a highly efficient search algorithm that works by repeatedly dividing the search interval in half. If the value of the target element is less than the value in the middle of the interval, the search continues in the lower half. Otherwise, it continues in the upper half. This process repeats until the target value is found or the interval is empty.

**Working Principle:**

1. Start with the middle element of the sorted list.
2. If the middle element is the target value, return its index.
3. If the target value is less than the middle element, narrow the search to the lower half of the list.
4. If the target value is greater than the middle element, narrow the search to the upper half of the list.
5. Repeat the process until the target is found or the search interval becomes empty.

**Complexity:**

* **Time Complexity:** O(log n), where n is the number of elements in the list. Since the search space is halved at each step, the number of comparisons grows logarithmically with the size of the list.
* **Space Complexity:** O(1) for the iterative version, as no extra space is required other than the variables used for indexing. If a recursive implementation is used, the space complexity can be O(log n) due to the call stack.

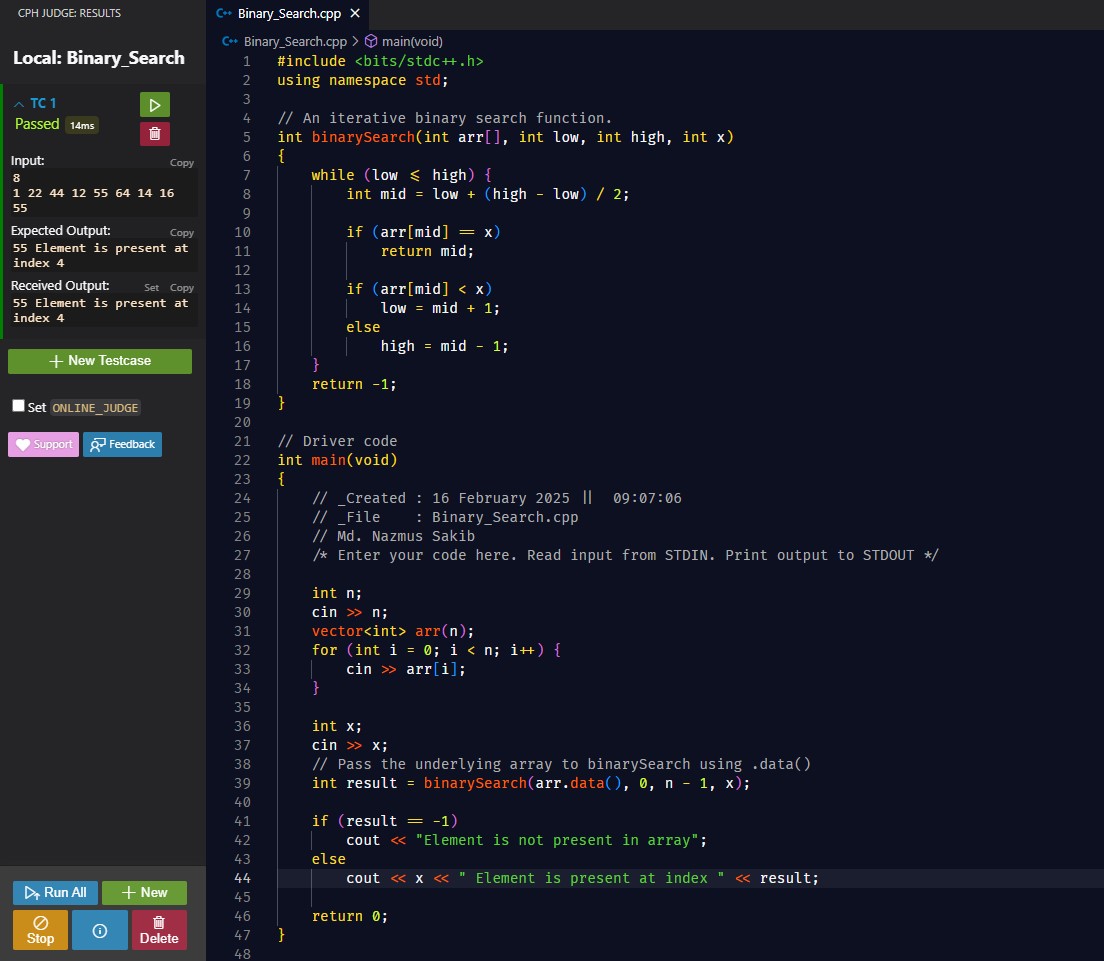
**Advantages:**

* Very efficient for large datasets.
* Time complexity is logarithmic (O(log n)), making it faster than linear search for sorted data.

**Disadvantages:**

* Requires the list to be sorted before searching.
* Not suitable for unsorted data.

This provides an overview of the Binary Search algorithm, explaining how it operates on sorted lists and its efficiency compared to other searching methods.



**Experiment Name:** Insertion Sort Algorithm  
**Objective:**  
To understand and implement the Insertion Sort algorithm to sort a list of numbers in ascending order.

**Theory:**  
Insertion Sort is a simple comparison-based sorting algorithm. It builds the final sorted array one element at a time. It works similarly to how we might sort playing cards in our hands: we take one card at a time and place it in the correct position relative to the already sorted cards.

**Working Principle:**

1. Start from the second element of the list.
2. Compare the current element with the previous elements, moving the larger elements one position to the right.
3. Insert the current element at the correct position in the sorted part of the list.
4. Repeat steps 2-3 for each element until the entire list is sorted.

**Complexity:**

* **Time Complexity:**
  + Best case: O(n) when the list is already sorted.
  + Worst case: O(n²) when the list is in reverse order.
  + Average case: O(n²), as it typically requires comparing each element with every other element.
* **Space Complexity:** O(1), as it sorts the list in place and doesn't require any extra space other than a few variables.

**Advantages:**

* Simple to implement and understand.
* Efficient for small datasets or nearly sorted data.
* Performs well when the list is partially sorted.

**Disadvantages:**

* Not efficient for large datasets due to its O(n²) time complexity in the worst case.
* Comparisons and shifts can be expensive for large inputs.

This provides a brief overview of the Insertion Sort algorithm, how it works, and its time and space complexities.

