PRACTICAL NO.01&02
Arrays Operations
Searching Operations

PLO	CLO	LL
4	1	C-3

Lab Outcomes

By the end of this lab, students will be able to:

- Create and manipulate arrays.
- Understand and implement array traversal and element access.
- Perform dry runs and code Linear Search and Binary Search.
- Modify searching algorithms and analyzing efficiency.
- Use ChatGPT as a learning partner for debugging, asking “why”, and validating logic.

Part 1: Arrays
Array Operations – Traversal, Insertion, and Deletion

- **Traversal:** Accessing and processing each element of an array sequentially.
- **Insertion:** Adding a new element at a specified position in the array (beginning, middle, or end).
- **Deletion:** Removing an element from a specified position, shifting the remaining elements accordingly.
- **Focus:** Implementation in C++ with dry runs and practical exercises.

1. Traversal

- Traversal means visiting each element of the array to display or process it.
- Time Complexity: **O(n)**.

C++ Implementation

```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;

    cout << "Array Traversal: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Tasks

- Modify program to print only **even numbers** in the array.
- Print the **sum and average** of array elements.
- Reverse the array using traversal.

2. Insertion

- Adding a new element at a specific position.
- It requires shifting elements to the right.

Time Complexity:

- Best Case (insert at end): **O(1)**
- Worst Case (insert at beginning): **O(n)**

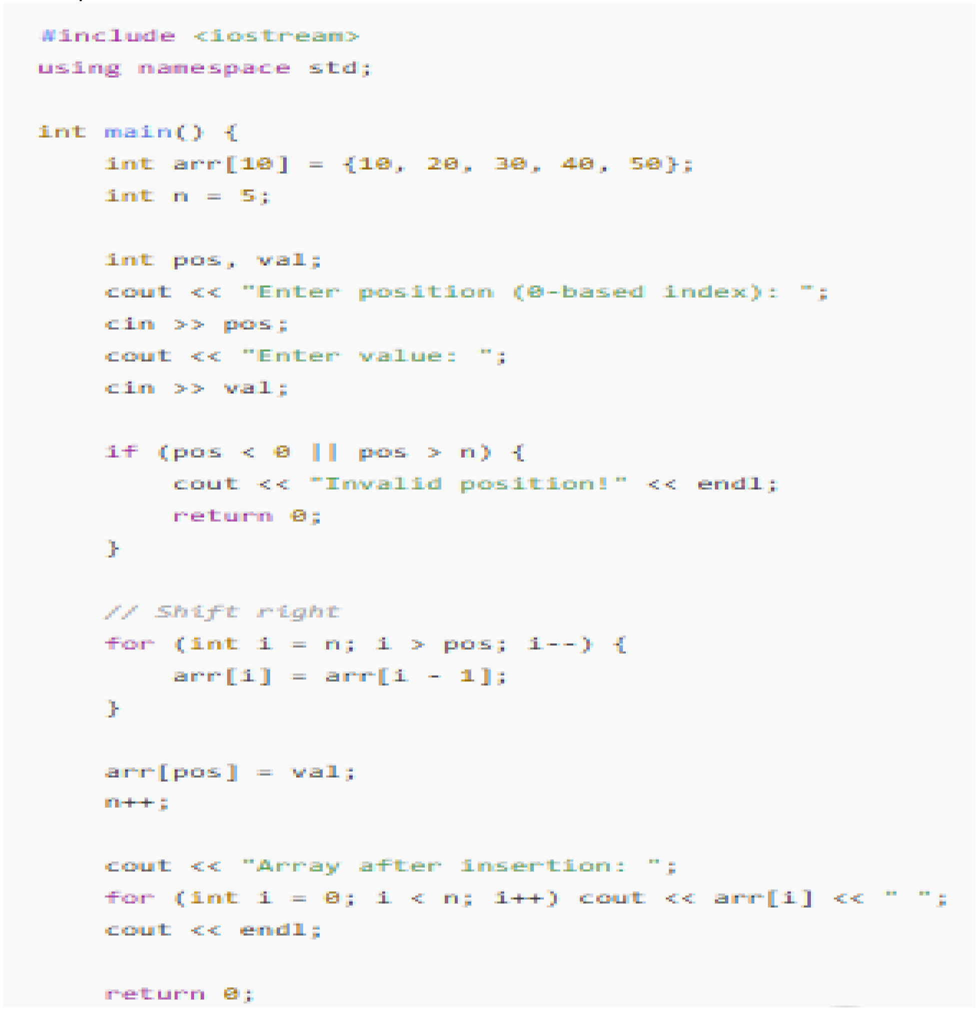
Dry Run Example

Array: {10, 20, 30, 40, 50}, n = 5

Insert 25 at position 2 (0-based index).

Result: {10, 20, 25, 30, 40, 50}

C++ Implementation



```
#include <iostream>
using namespace std;

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int n = 5;

    int pos, val;
    cout << "Enter position (0-based index): ";
    cin >> pos;
    cout << "Enter value: ";
    cin >> val;

    if (pos < 0 || pos > n) {
        cout << "Invalid position!" << endl;
        return 0;
    }

    // Shift right
    for (int i = n; i > pos; i--) {
        arr[i] = arr[i - 1];
    }

    arr[pos] = val;
    n++;

    cout << "Array after insertion: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;

    return 0;
```

Tasks

- Insert element at the **beginning**.
- Insert element at the **end**.
- Try inserting at an **invalid position** (test error handling).

3. Deletion

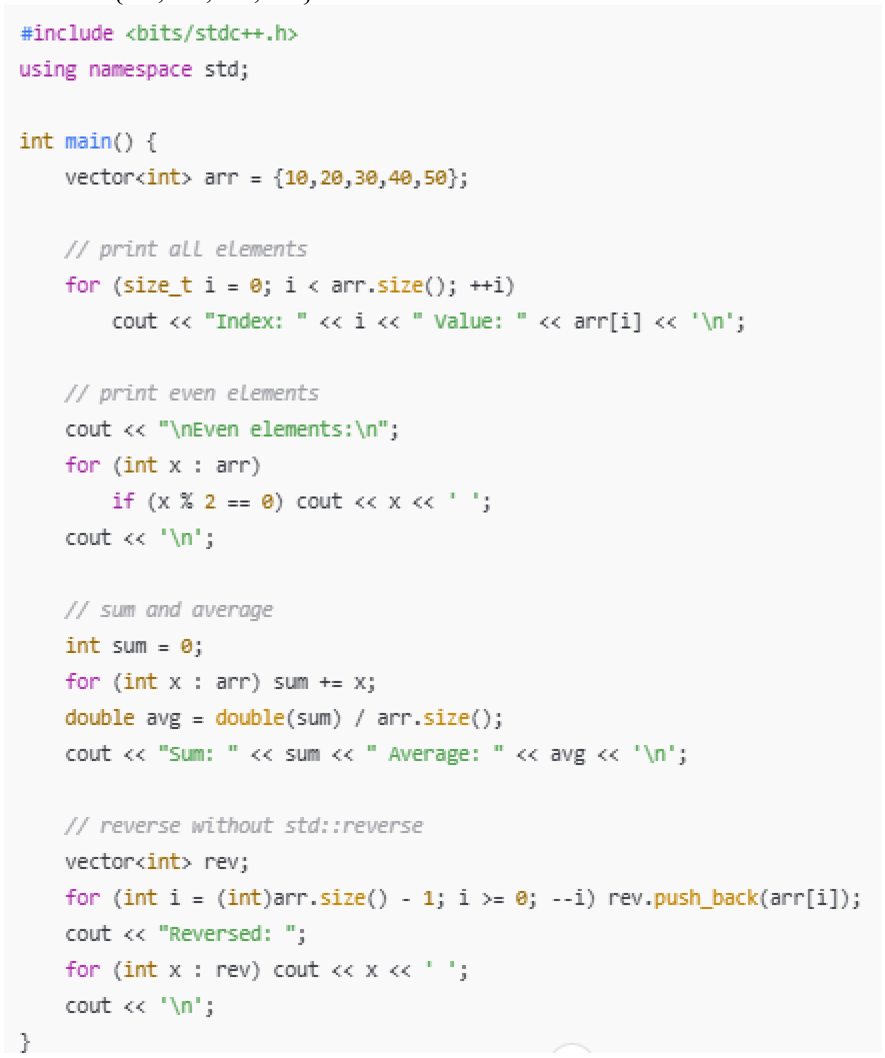
- Removing an element from a specific position.
- Requires shifting elements to the left.

Time Complexity

Best Case (delete last element): **O(1)**

Worst Case (delete first element): **O(n)**

Dry Run Example
Array: {10, 20, 30, 40, 50}, n = 5
Delete element at position 2 (0-based index).
Result: {10, 20, 40, 50}



```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> arr = {10,20,30,40,50};

    // print all elements
    for (size_t i = 0; i < arr.size(); ++i)
        cout << "Index: " << i << " Value: " << arr[i] << '\n';

    // print even elements
    cout << "\nEven elements:\n";
    for (int x : arr)
        if (x % 2 == 0) cout << x << ' ';
    cout << '\n';

    // sum and average
    int sum = 0;
    for (int x : arr) sum += x;
    double avg = double(sum) / arr.size();
    cout << "Sum: " << sum << " Average: " << avg << '\n';

    // reverse without std::reverse
    vector<int> rev;
    for (int i = (int)arr.size() - 1; i >= 0; --i) rev.push_back(arr[i]);
    cout << "Reversed: ";
    for (int x : rev) cout << x << ' ';
    cout << '\n';
}
```

Small challenges
Implement (write and run):

- Find **max** and **min** element.
- Count elements greater than 25.
- Input an element from user and check existence.

Part 2 — Linear Search

Linear Search = check elements one by one until target is found or array ends.

Dry Run Exercise

Array: {12, 45, 7, 23, 89}

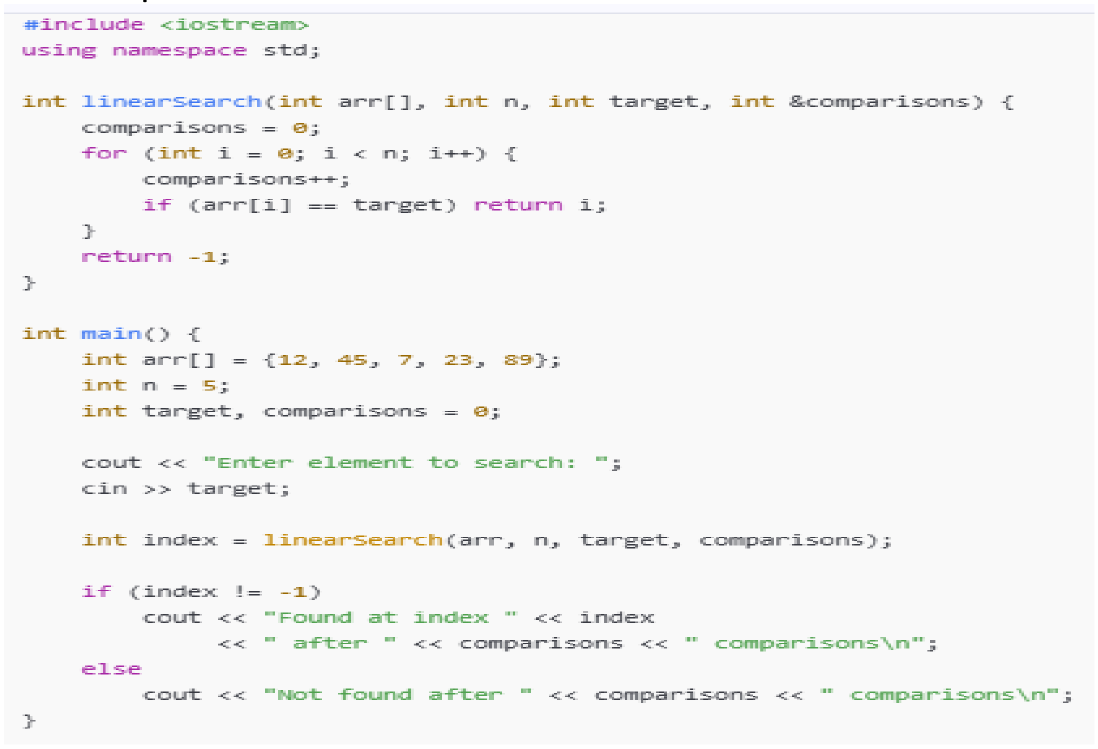
Target: 23

Step	Compared Value	Match?	Comparisons So Far
1	12	No	1
2	45	No	2
3	7	No	3
4	23	Yes	4

Task:

- Fill the table for **Target = 89**.
- How many comparisons? At which index found?
- Try **Target = 100 (not present)**. How many comparisons?

Basic C++ Implementation



```cpp
#include <iostream>
using namespace std;

int linearSearch(int arr[], int n, int target, int &comparisons) {
    comparisons = 0;
    for (int i = 0; i < n; i++) {
        comparisons++;
        if (arr[i] == target) return i;
    }
    return -1;
}

int main() {
    int arr[] = {12, 45, 7, 23, 89};
    int n = 5;
    int target, comparisons = 0;

    cout << "Enter element to search: ";
    cin >> target;

    int index = linearSearch(arr, n, target, comparisons);

    if (index != -1)
        cout << "Found at index " << index
             << " after " << comparisons << " comparisons\n";
    else
        cout << "Not found after " << comparisons << " comparisons\n";
}
```

Tasks to Try:

- Input 23, 89, and 100 → observe comparisons.
- Modify program to **print all indices** if number appears multiple times.

Exploration:

Students test with:

Empty array → what happens?

Array of identical elements (e.g., {5,5,5,5,5}) searching for 5.

Target at **first element** vs **last element**.

Part 3--Binary Search

Binary Search works only on **sorted arrays**.
Repeatedly divides the array into halves until target is found or search space becomes empty.
Time Complexity: O($\log_2 n$).

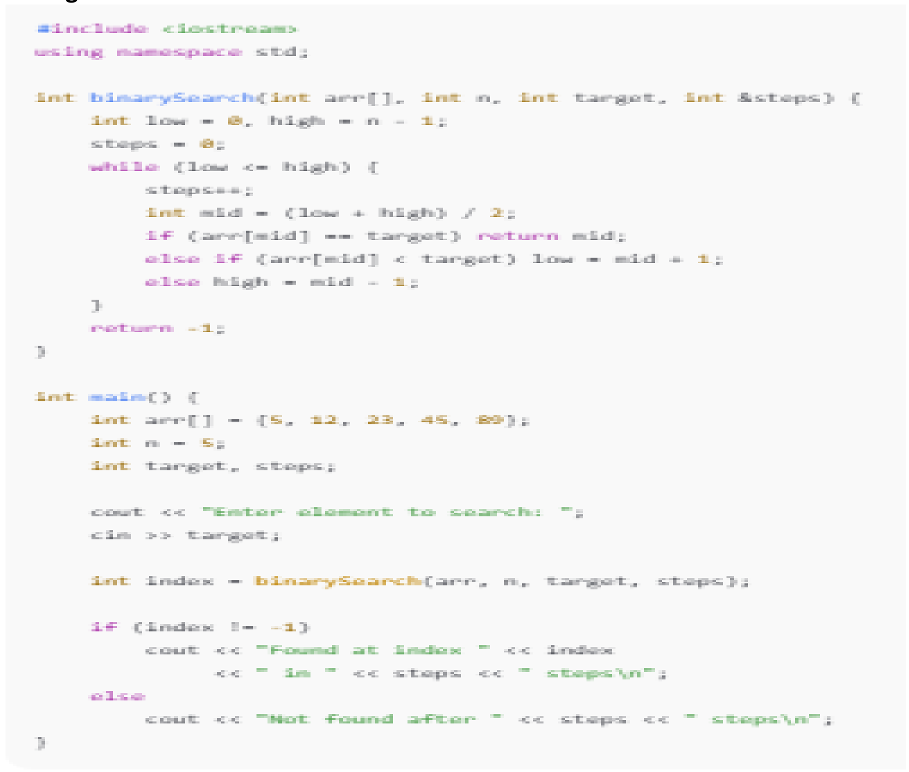
Dry Run Exercise

Array (sorted): {5, 12, 23, 45, 89}
Target = 23

Step	Low	High	Mid	arr[mid]	Match?
1	0	4	2	23	Yes

Task 1: Fill the table for **Target = 45**.
Task 2: Fill the table for **Target = 7 (not present)**.
How many steps were needed in each case?

Coding Task



```cpp
#include <iostream>
using namespace std;

int binarySearch(int arr[], int n, int target, int &steps) {
    int low = 0, high = n - 1;
    steps = 0;
    while (low <= high) {
        steps++;
        int mid = (low + high) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {5, 12, 23, 45, 89};
    int n = 5;
    int target, steps;

    cout << "Enter element to search: ";
    cin >> target;

    int index = binarySearch(arr, n, target, steps);

    if (index != -1)
        cout << "Found at index " << index
             << " in " << steps << " steps\n";
    else
        cout << "Not found after " << steps << " steps\n";
}
```

Tasks to Try:

- Search for 23, 45, and 7. Compare step counts.
- Try searching in a **large, sorted array** (manually extend array).
- Print the **mid value at each step** to see the halving process.

Explorations

Students test with:
Empty array.
Target at first element (e.g., 5).
Target at last element (e.g., 89).
Apply Binary Search on **unsorted array** → observe wrong results.

Part 4 — Efficiency Comparison

To compare **Linear Search** and **Binary Search** in terms of efficiency, using both **step counts** and **time complexity analysis**.

a) Linear **Search**:

Worst-case comparisons ≈ n.
Time Complexity = **O(n)**.

b) Binary **Search** (sorted arrays only):

Worst-case comparisons ≈ $\log_2(n)$.
Time Complexity = **O(log n)**.

Task A- Step Count Comparison

Fill in the table

Array Size (n)	Linear Search (Worst-Case Steps)	Binary Search (Worst-Case Steps)
10	10	~4
1000	1000	~10
1,000,000	1,000,000	~20

Note: $\log_2(1{,}000{,}000) \approx 20$

Task B — Coding Verification

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Linear Search
int linearSearch(vector<int> &arr, int target, int &steps) {
  steps = 0;
  for (int i = 0; i < arr.size(); i++) {
    steps++;
    if (arr[i] == target) return i;
  }
  return -1;
}
// Binary Search
int binarySearch(vector<int> &arr, int target, int &steps) {
  int low = 0, high = arr.size() - 1;
  steps = 0;
  while (low <= high) {
    steps++;
    int mid = (low + high) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
int main() {
  int n;
  cout << "Enter array size: ";
  cin >> n;
  vector<int> arr(n);
  for (int i = 0; i < n; i++) arr[i] = i + 1; // sorted data
  int target = n; // last element
```

```
    int stepsLinear, stepsBinary;
    linearSearch(arr, target, stepsLinear);
    binarySearch(arr, target, stepsBinary);
    cout << "Array Size: " << n << "\n";
    cout << "Linear Search steps: " << stepsLinear << "\n";
    cout << "Binary Search steps: " << stepsBinary << "\n";
}
```

Experiment:

Run the program with $n = 10$, 1000, and 1000000 (try smaller first).
Record the **steps** for both search methods.
Compare results with the table above.
Summary Table: Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Works On	Sorted or Unsorted arrays	Sorted arrays only
Approach	Check each element one by one	Divide array in half each step
Best Case	O(1) (target at first index)	O(1) (target at middle)
Worst Case	O(n)	$O(\log_2 n)$
Space Complexity	O(1)	O(1)
Advantages	Simple, no pre-processing needed	Very fast for large sorted data
Disadvantages	Slow for large arrays	Requires sorting, random access needed
Use Case	Small or unsorted arrays, streaming	Large static sorted datasets