Marwadi University Marwadi Chandarana Group	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: Design & Analysis of Algorithms (01CT0512)	**Aim: Implementing the Searching Algorithms and understanding the time and space complexities**	
Experiment No: 2	**Date: 03/08/2025**	**Enrollment No: 92301733049**

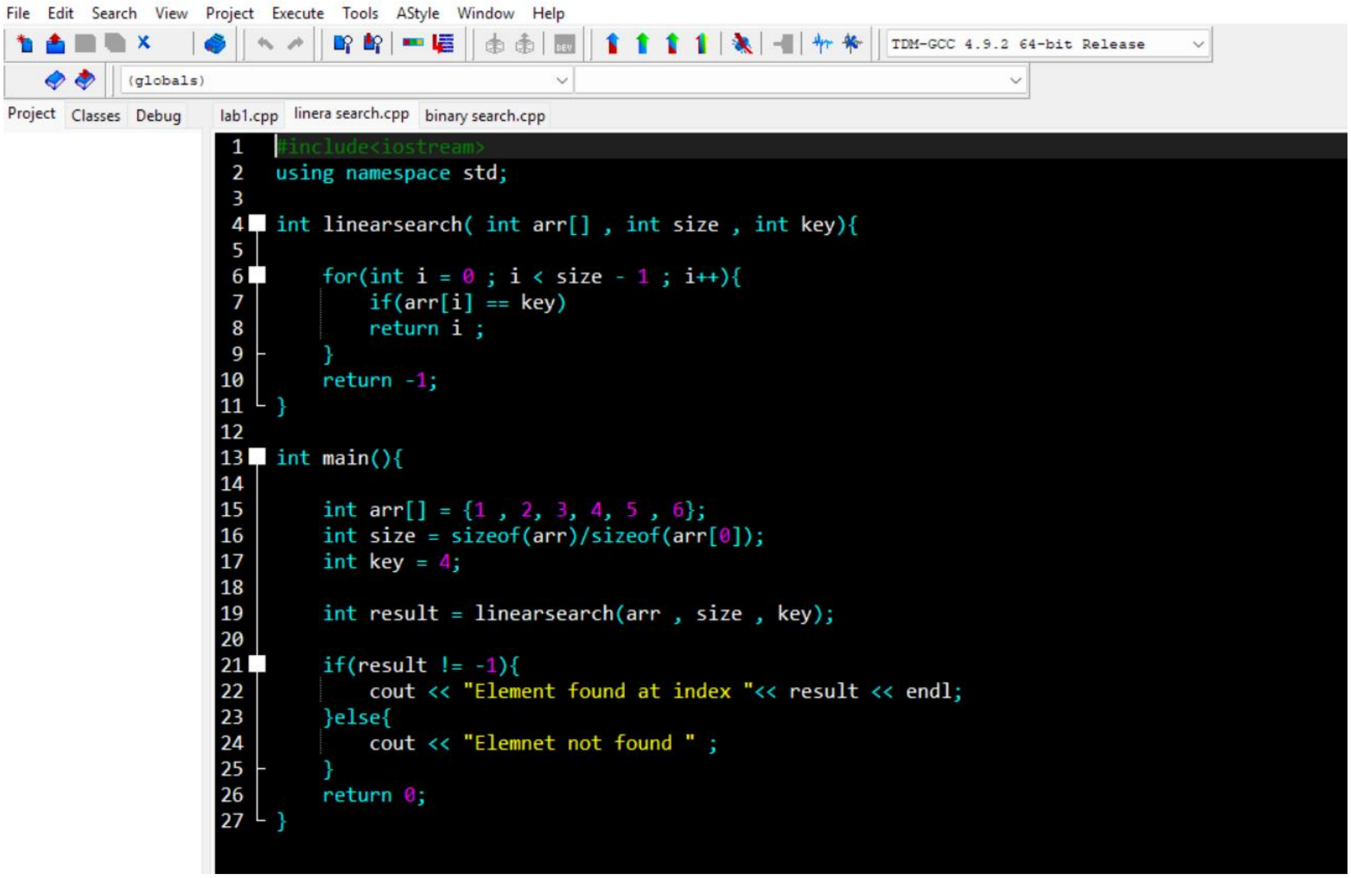
1. Linear Search

Theory

In Linear Search, we traverse the array from the beginning and compare each element with the target value. If a match is found, we return the index; otherwise, continue until the end of the array.

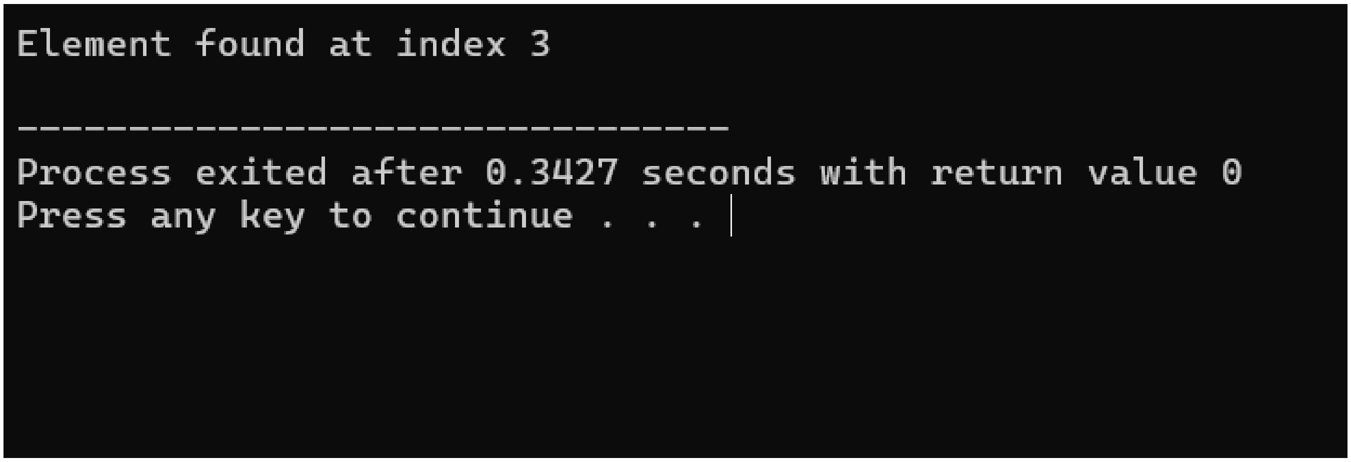
Programming Language: C++

Code:



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

int linearsearch( int arr[] , int size , int key){

    for(int i = 0 ; i < size - 1 ; i++){
        if(arr[i] == key)
        return i ;
    }
    return -1;
}

int main(){

    int arr[] = {1 , 2, 3, 4, 5 , 6};
    int size = sizeof(arr)/sizeof(arr[0]);
    int key = 4;

    int result = linearsearch(arr , size , key);

    if(result != -1){
        cout << "Element found at index "<< result << endl;
    }else{
        cout << "Elemnet not found " ;
    }
    return 0;
}
```

Output:



```
Element found at index 3

--------------------------------
Process exited after 0.3427 seconds with return value 0
Press any key to continue . . .
```

Space complexity:- O(1)
Justification:
Linear search operates directly on the input array without using any extra data structures, so it uses constant auxiliary space.

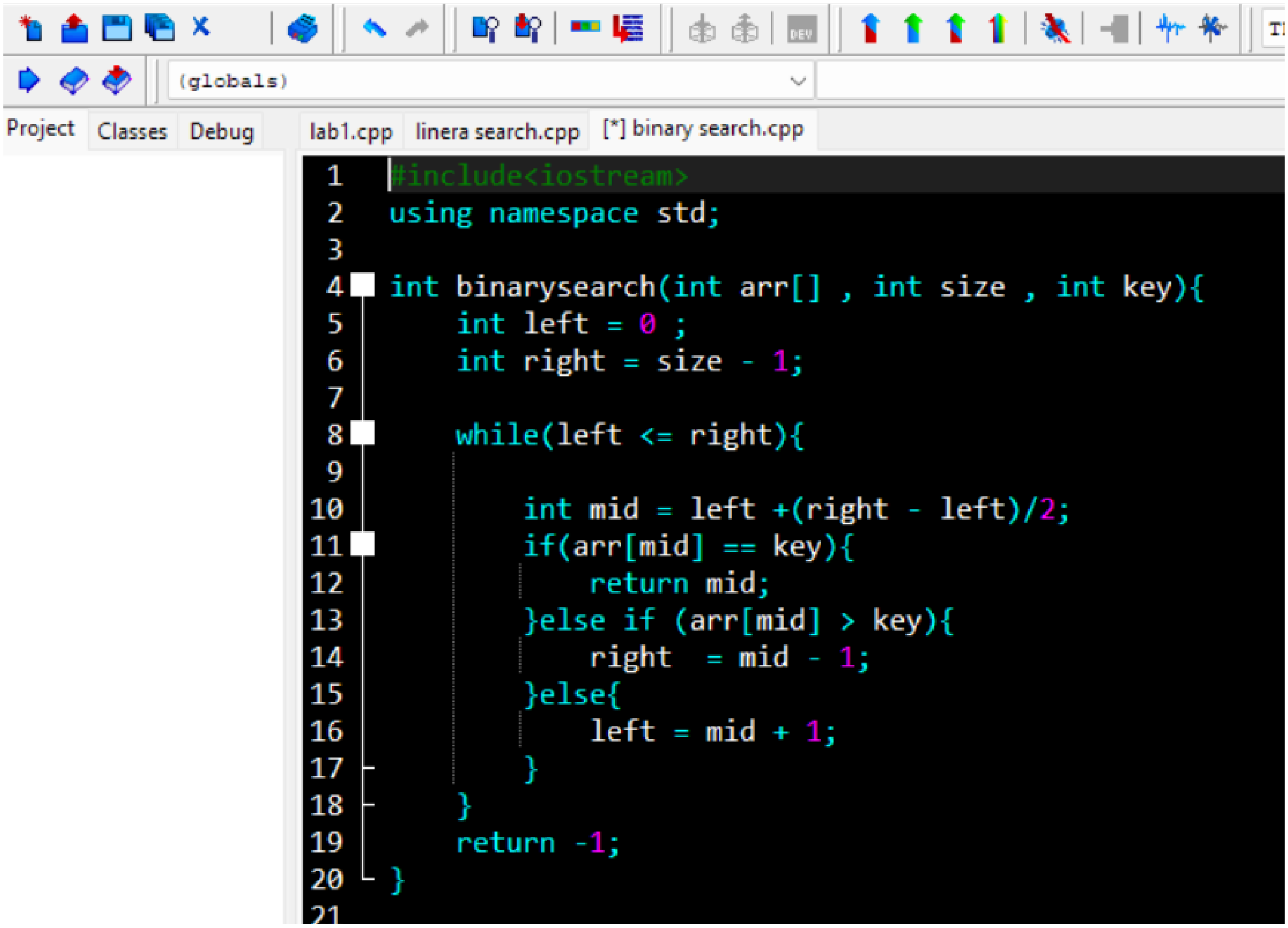
Time complexity:
– Best case time complexity: O(1)
Justification:
If the element to be found is at the first position, we find it in the first comparison.

– Worst case time complexity: O(n)
Justification:
If the element is at the last position or not present in the array, we have to check every element once, leading to linear time.

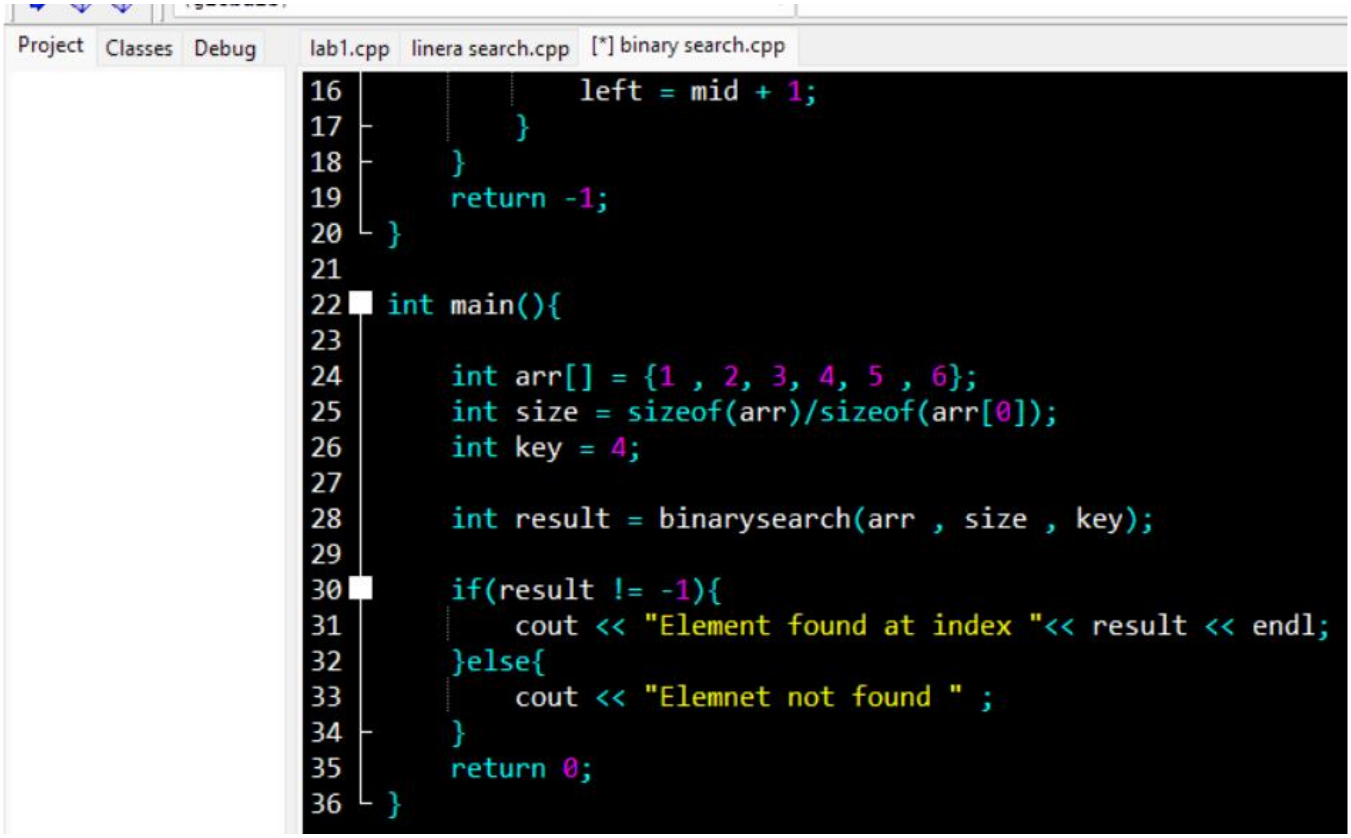
2. Binary Search

Theory

Binary Search is applied on a **sorted** array. It repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle, it narrows the interval to the lower half. Otherwise, to the upper half.

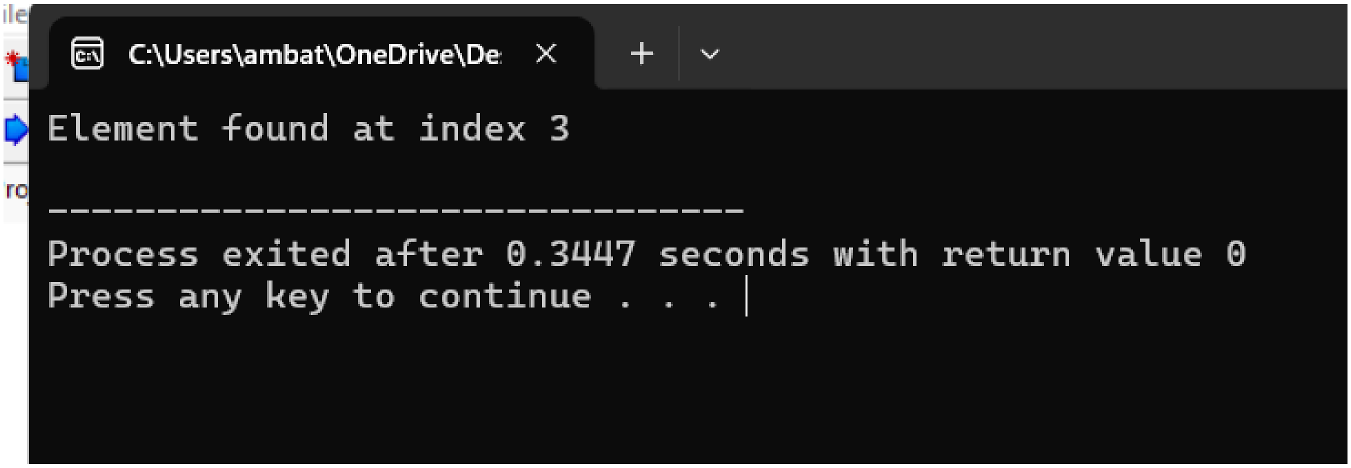


```
#include<iostream>
using namespace std;

int binarysearch(int arr[] , int size , int key){
    int left = 0 ;
    int right = size - 1;

    while(left <= right){

        int mid = left +(right - left)/2;
        if(arr[mid] == key){
            return mid;
        }else if (arr[mid] > key){
            right  = mid - 1;
        }else{
            left = mid + 1;
        }
    }
    return -1;
}
```



```
            left = mid + 1;
        }
    }
    return -1;
}

int main(){

    int arr[] = {1 , 2, 3, 4, 5 , 6};
    int size = sizeof(arr)/sizeof(arr[0]);
    int key = 4;

    int result = binarysearch(arr , size , key);

    if(result != -1){
        cout << "Element found at index "<< result << endl;
    }else{
        cout << "Elemnet not found " ;
    }
    return 0;
}
```

OUTPUT :-



```
Element found at index 3

--------------------------------
Process exited after 0.3447 seconds with return value 0
Press any key to continue . . .
```

Space complexity:
O(1) *(for iterative version)*
Justification:
Binary search only uses a few variables (like low, high, mid) for the search and does not require any extra space beyond that.

Time complexity:
– Best case time complexity: O(1)
Justification:
If the target element is at the middle index during the first check, it's found immediately.

– Worst case time complexity: O(log n)
Justification:
Each step of the binary search cuts the problem size in half. In the worst case, we keep dividing until one element is left, resulting in $\log_2(n)$ comparisons.