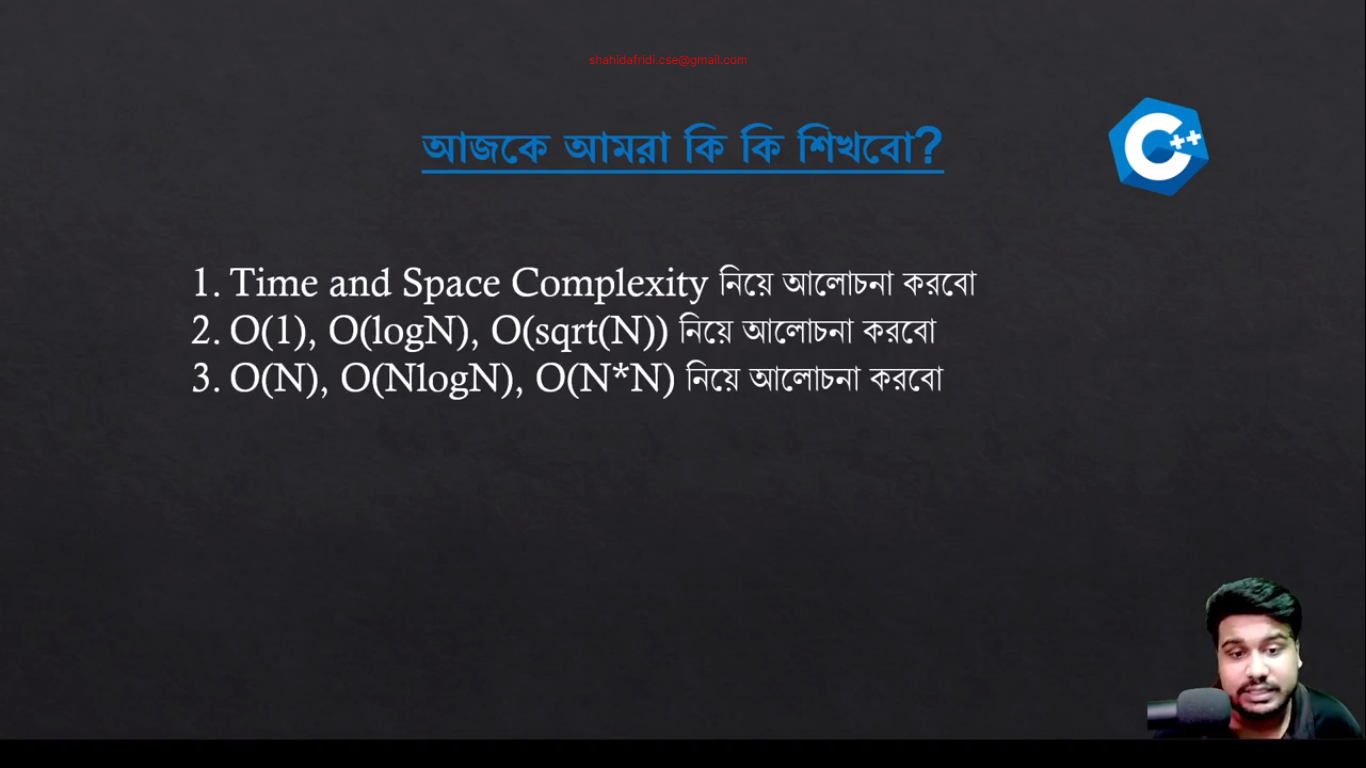
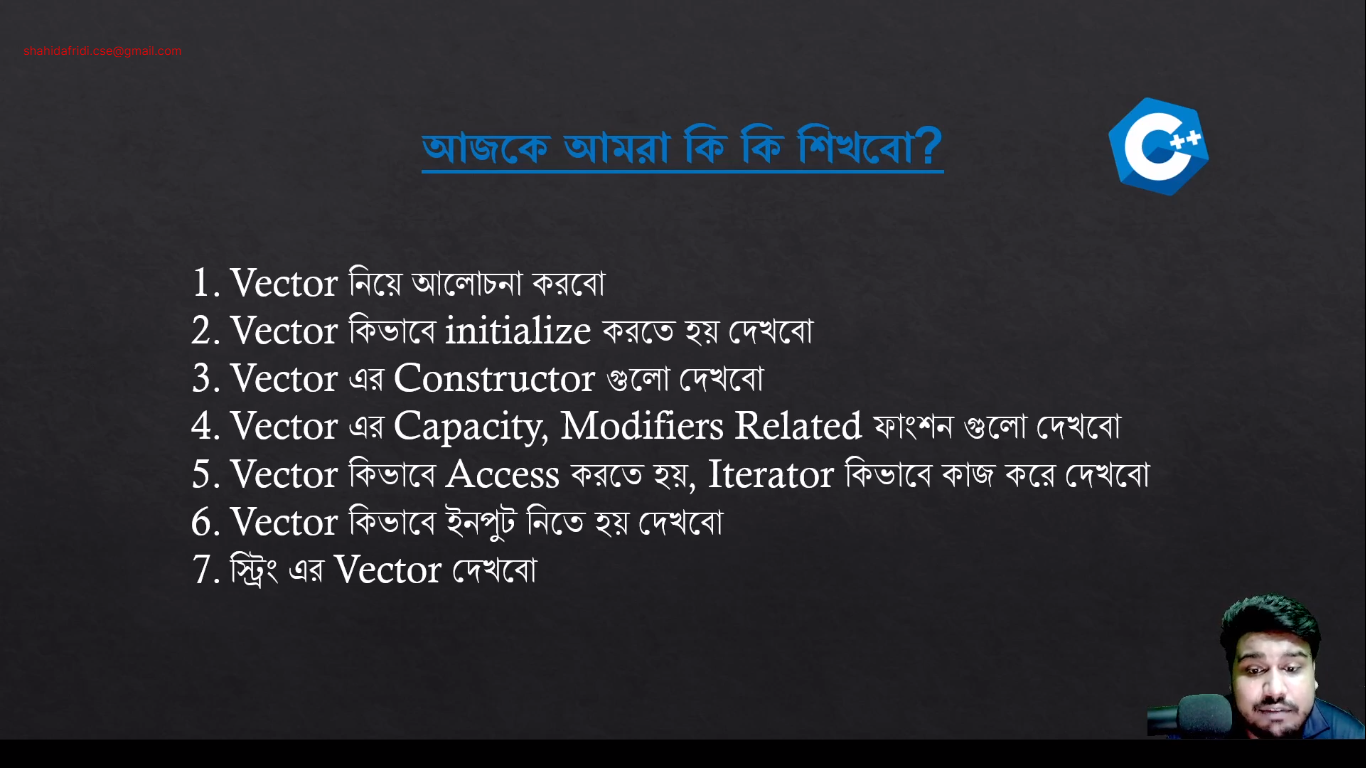
**Basic data Structure**

**Module – 1:**

****

**Module 2:**

****

**Vector Built-in Functions:**

1. **Constructor**

|  |  |  |
| --- | --- | --- |
| **Name** | **Details** | **Time Complexity** |
| **vector<type>v;** | Construct a vector with 0 elements. | O(1) |
| **vector<type>v(N);** | Construct a vector with N elements and the value will be garbage. | O(N) |
| **vector<type>v(N,V);** | Construct a vector with N elements and the value will be V. | O(N) |
| **vector<type>v(v2);** | Construct a vector by copying another vector v2. | O(N) |
| **vector<type>v(A,A+N);** | Construct a vector by copying all elements from an array A of size N. | O(N) |

1. **Capacity**

|  |  |  |
| --- | --- | --- |
| **Name** | **Details** | **Time Complexity** |
| **v.size()** | Returns the size of the vector. | O(1) |
| **v.max\_size()** | Returns the maximum size that the vector can hold. | O(1) |
| **v.capacity()** | Returns the current available capacity of the vector. | O(1) |
| **v.clear()** | Clears the vector elements. Do not delete the memory, only clear the value. | O(N) |
| **v.empty()** | Return true/false if the vector is empty or not. | O(1) |
| **v.resize()** | Change the size of the vector. | O(K); where K is the difference between new size and current size. |

1. **Modifiers**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Details** | | **Time Complexity** |
| **v= or v.assign()** | | Assign another vector. | O(N) if sizes are different, O(1) otherwise. |
| **v.push\_back()** | | Add an element to the end. | O(1) |
| **v.pop\_back()** | | Remove the last element. | O(1) |
| **v.insert()** | | Insert elements at a specific position. | O(N+K); where K is the number of elements to be inserted. |
| **v.erase()** | | Delete elements from a specific position. | O(N+K); where K is the number of elements to be deleted. |
| **replace(v.begin(),v.end(),value,replace\_value)** | | Replace all the value with replace value. Not under a vector. | O(N) |
| **find(v.begin(),v.end(),V)** | | Find the value V. Not under a vector. | O(N) |

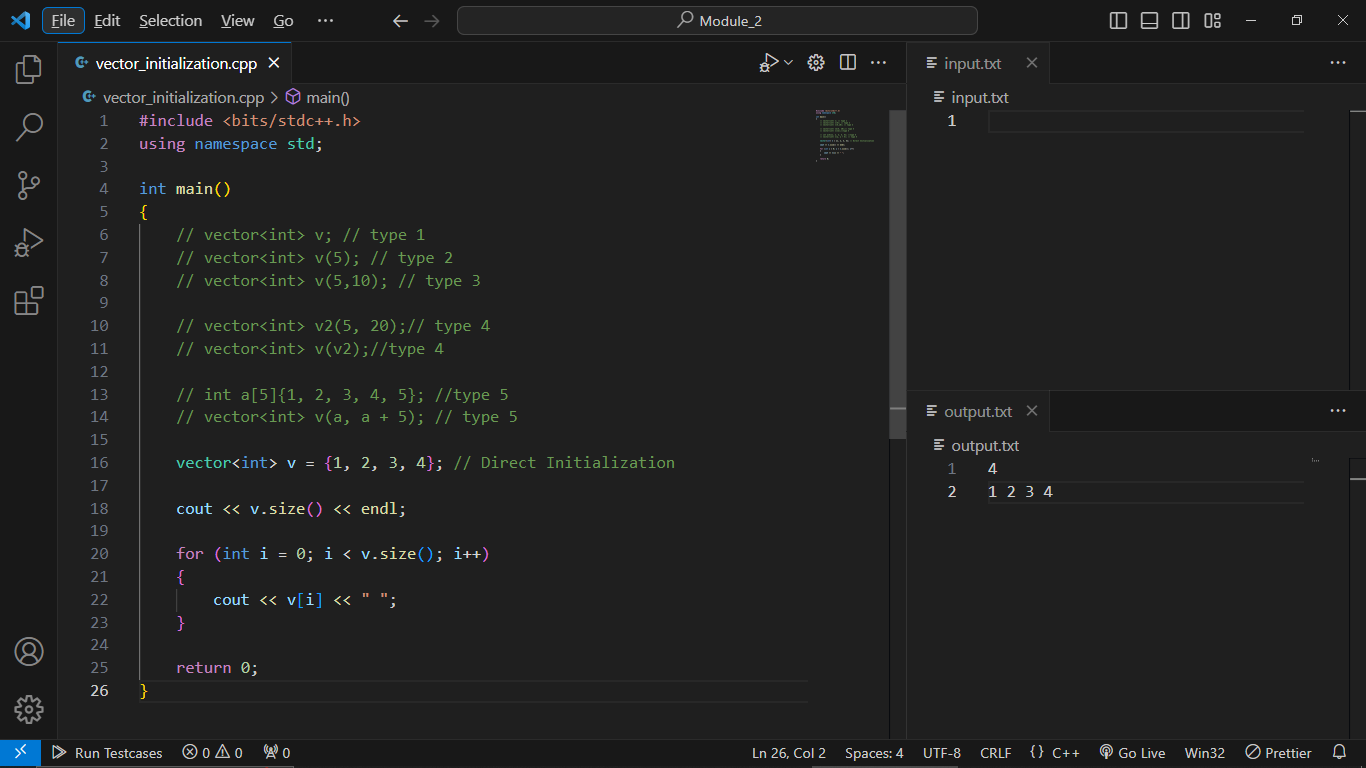
1. **Element access**

|  |  |  |
| --- | --- | --- |
| **Name** | **Details** | **Time Complexity** |
| **v[i]** | Access the ith element. | O(1) |
| **v.at(i)** | Access the ith element. | O(1) |
| **v.back()** | Access the last element. | O(1) |
| **v.front()** | Access the first element. | O(1) |

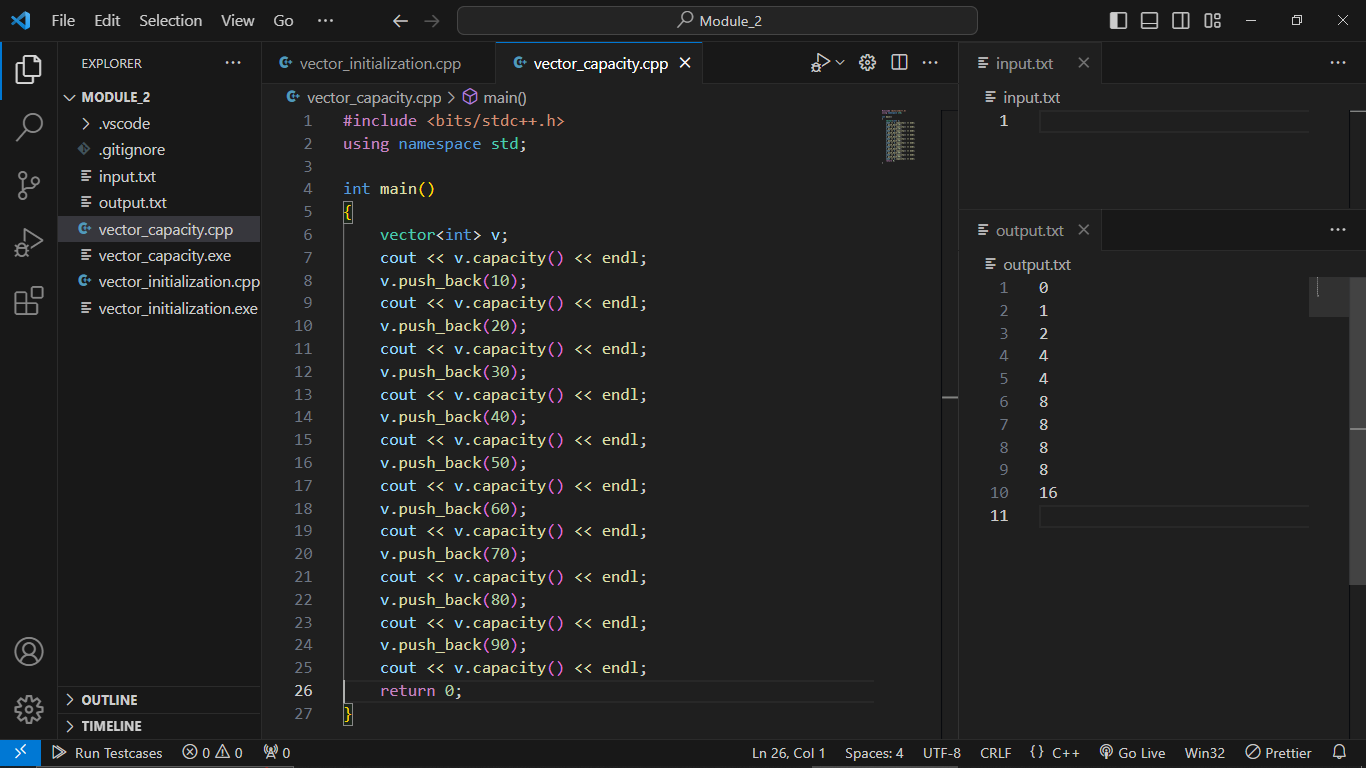
1. **Iterators**

|  |  |  |
| --- | --- | --- |
| **Name** | **Details** | **Time Complexity** |
| **v.begin()** | Pointer to the first element. | O(1) |
| **v.end()** | Pointer to the last element. | O(1) |

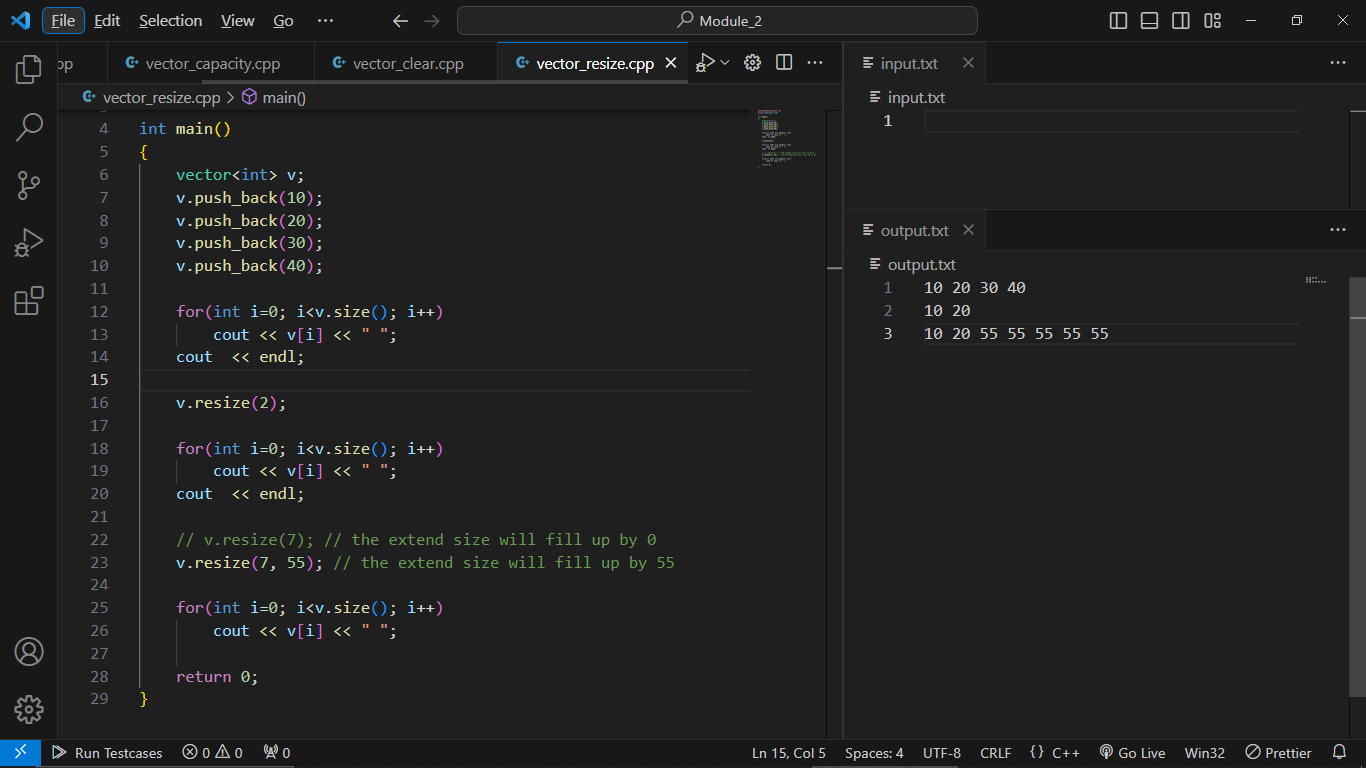
**Vector initialization:**

****

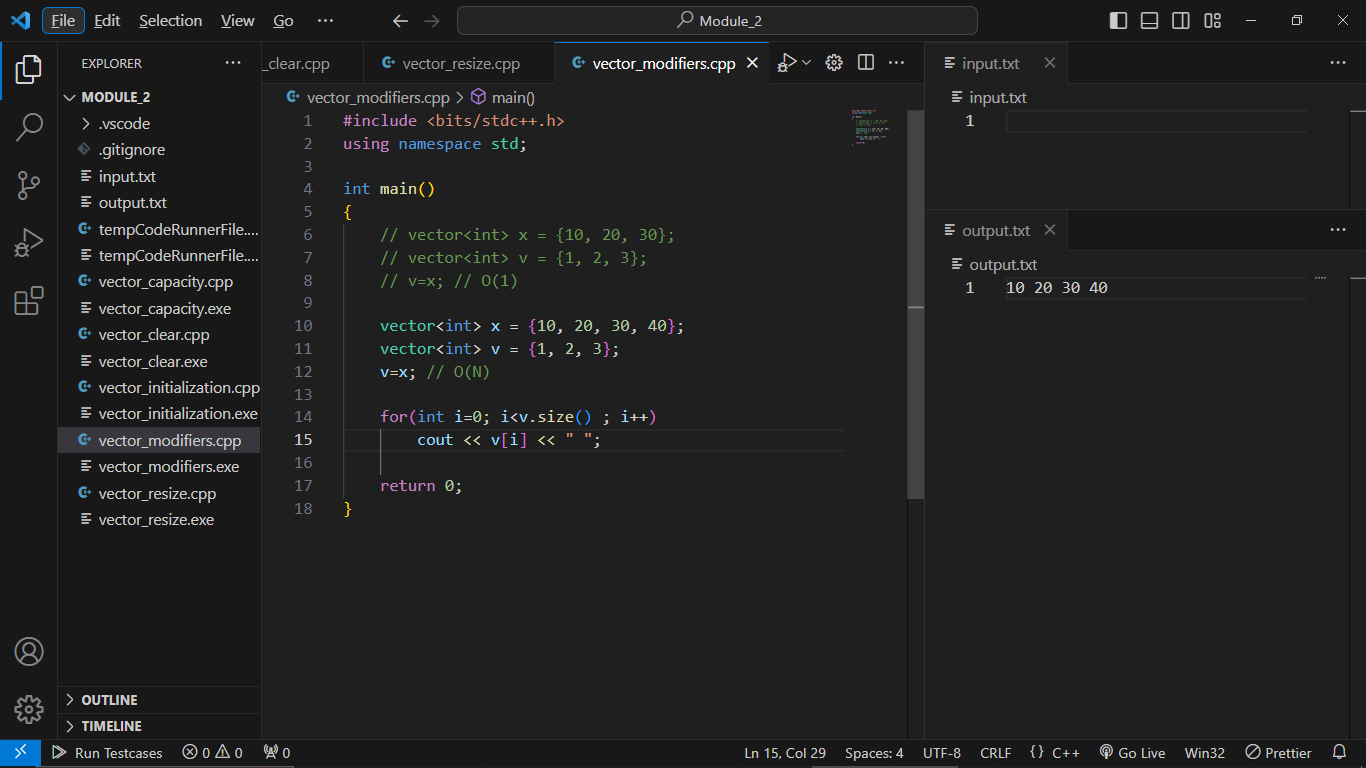
**Vector Capacity:**

****

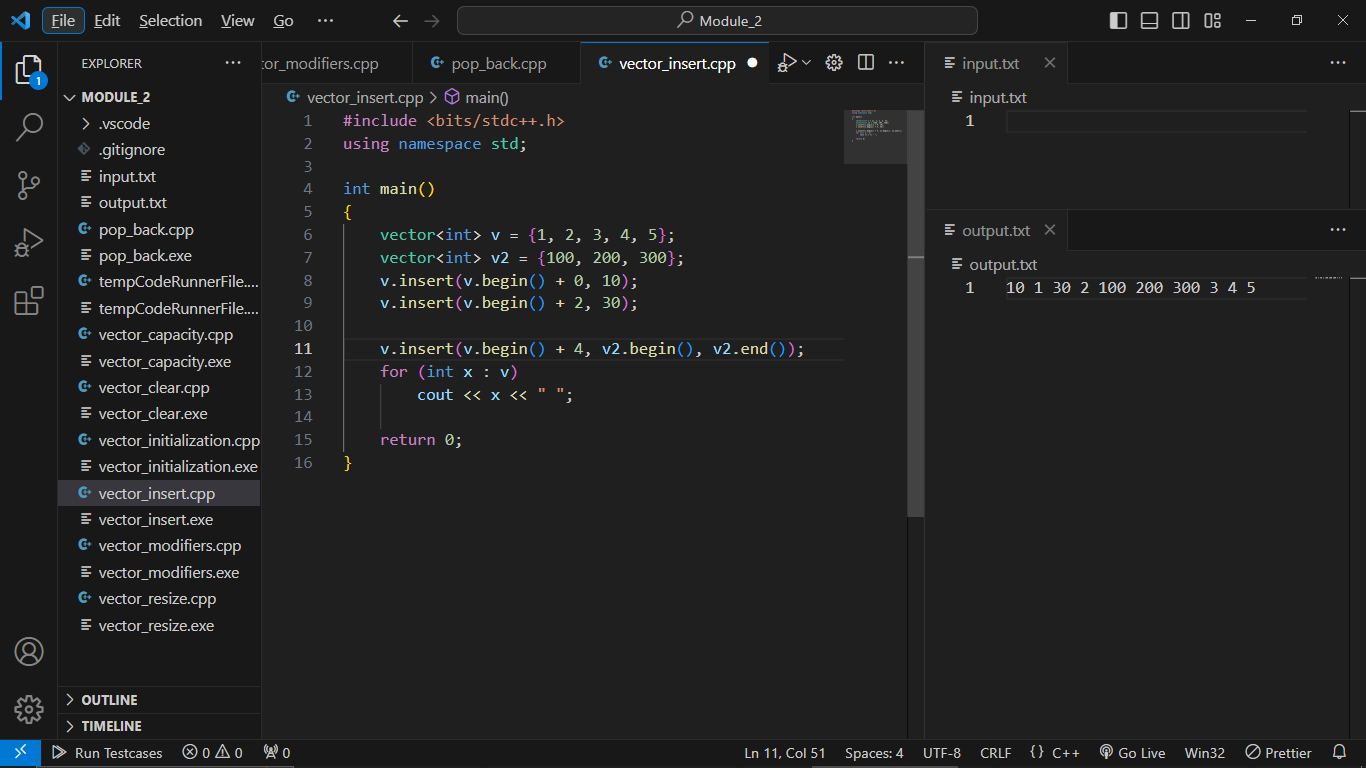
**Vector Resize:**

****

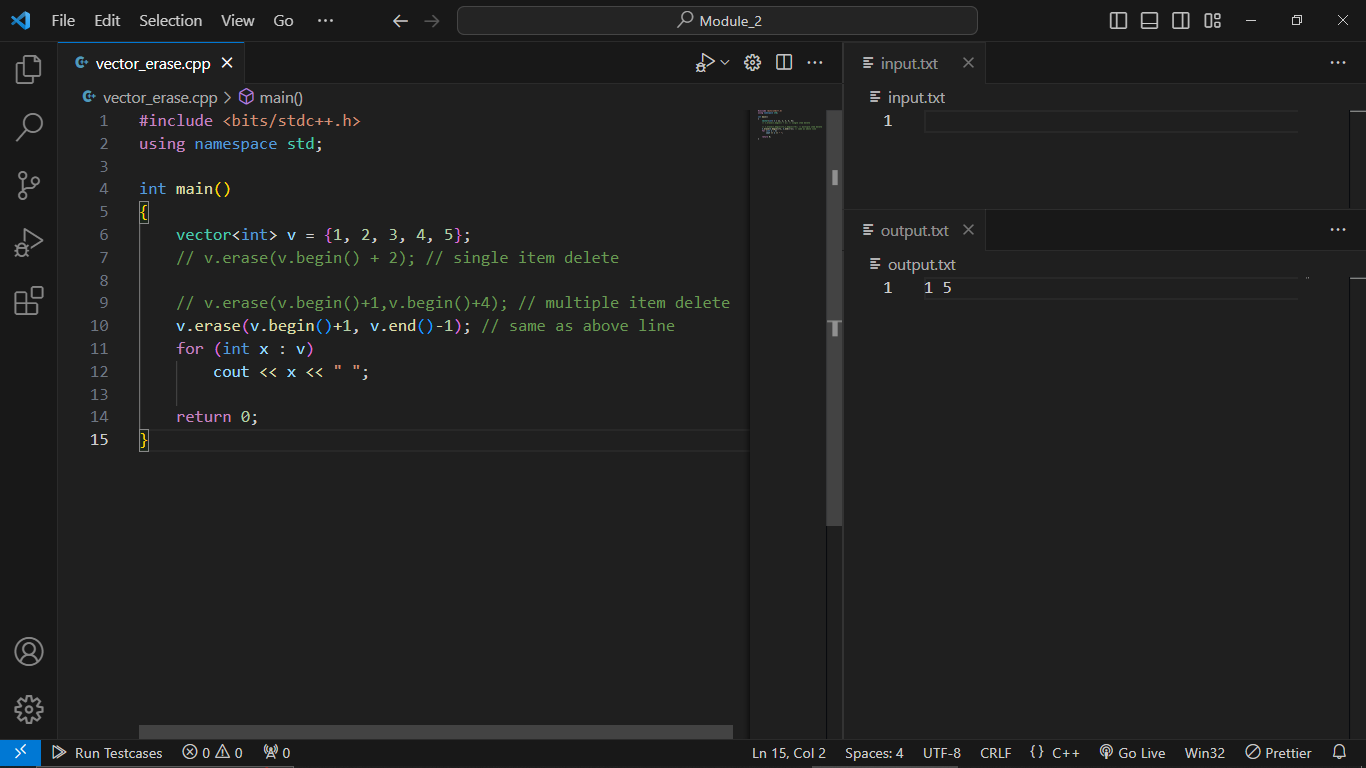
**Vector Modifiers -> Vector re-assign:**

****

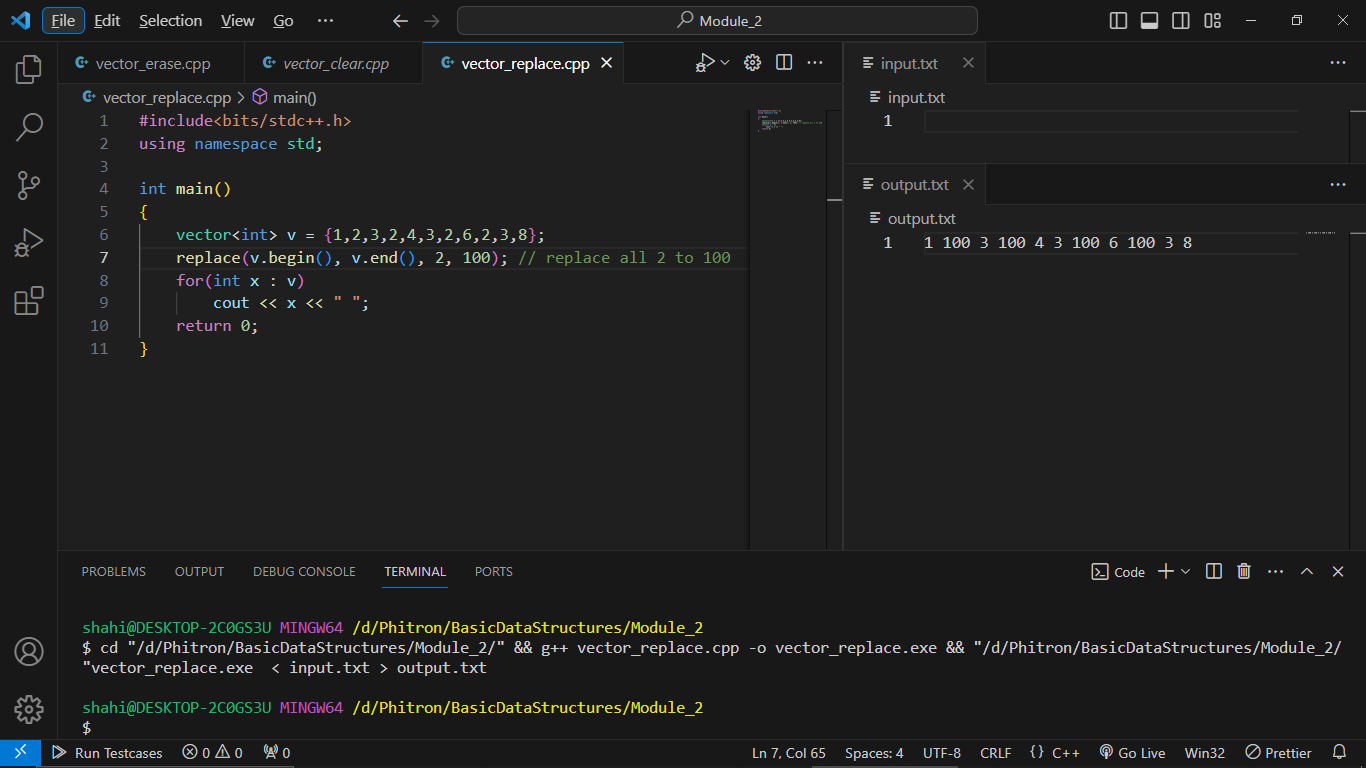
**Vector Insert:**

****

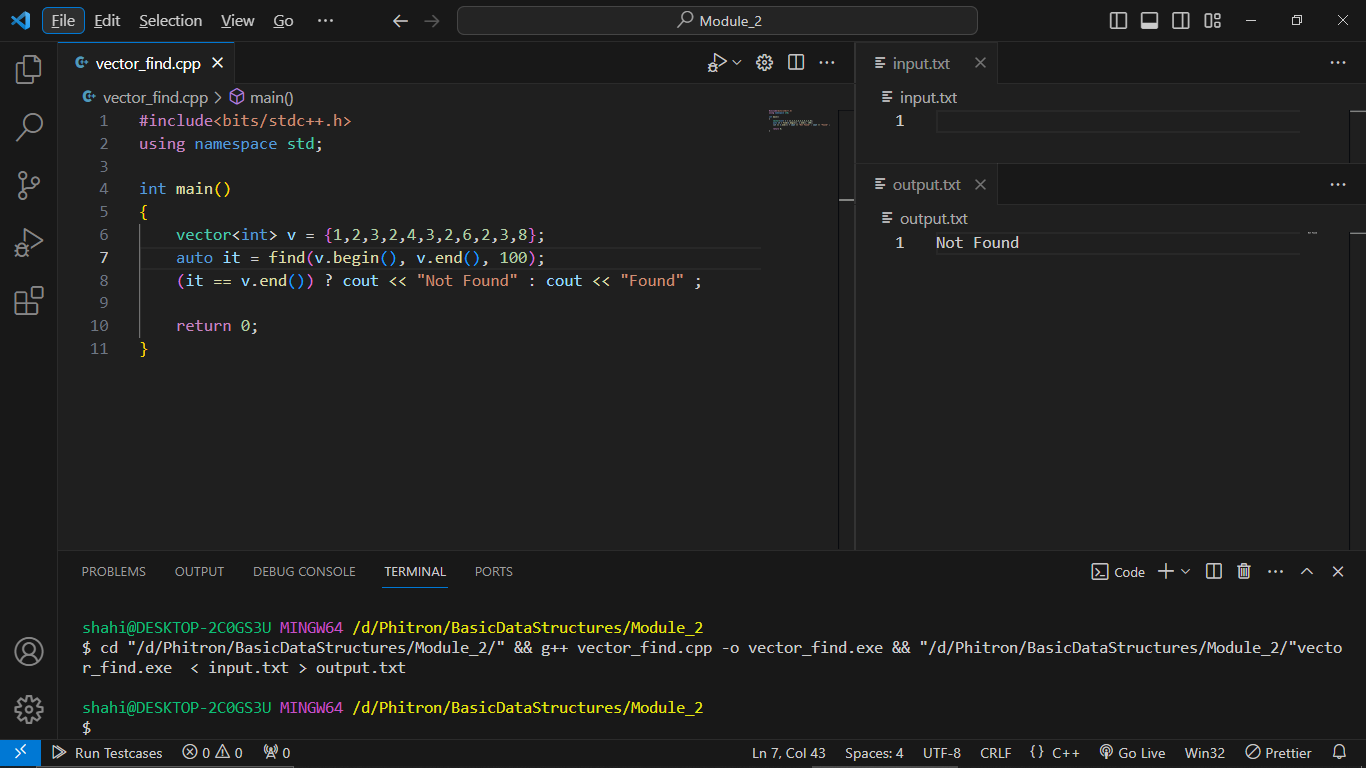
**Vector Erase:**

****

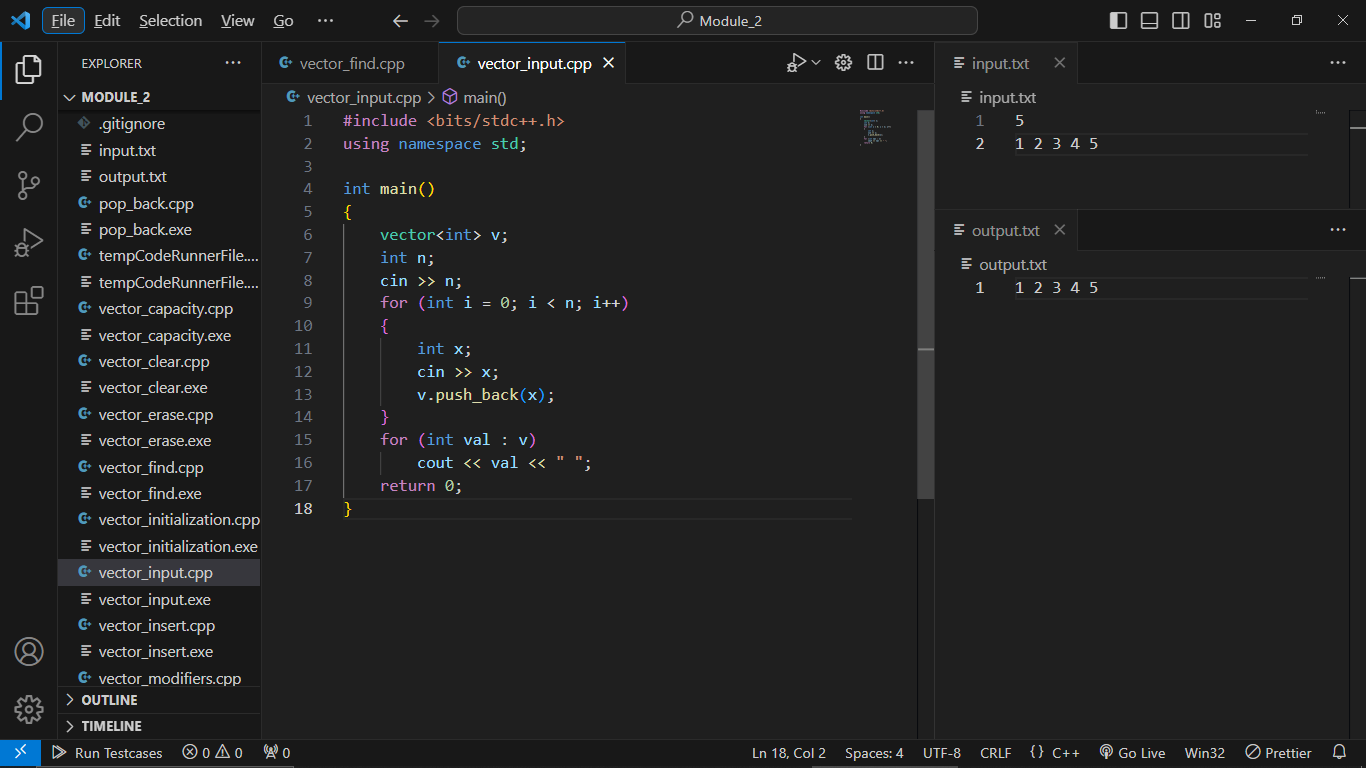
**Vector replace:**

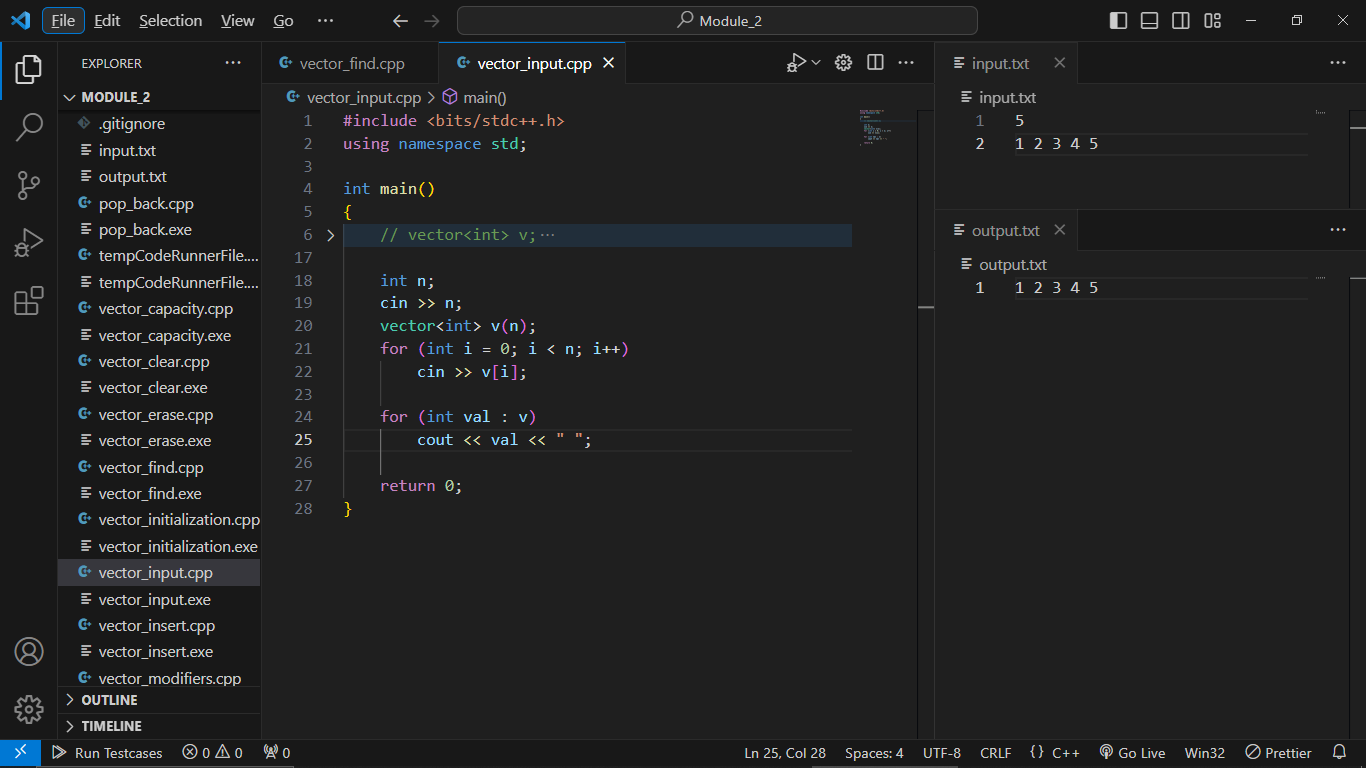
****

**Vector find:**

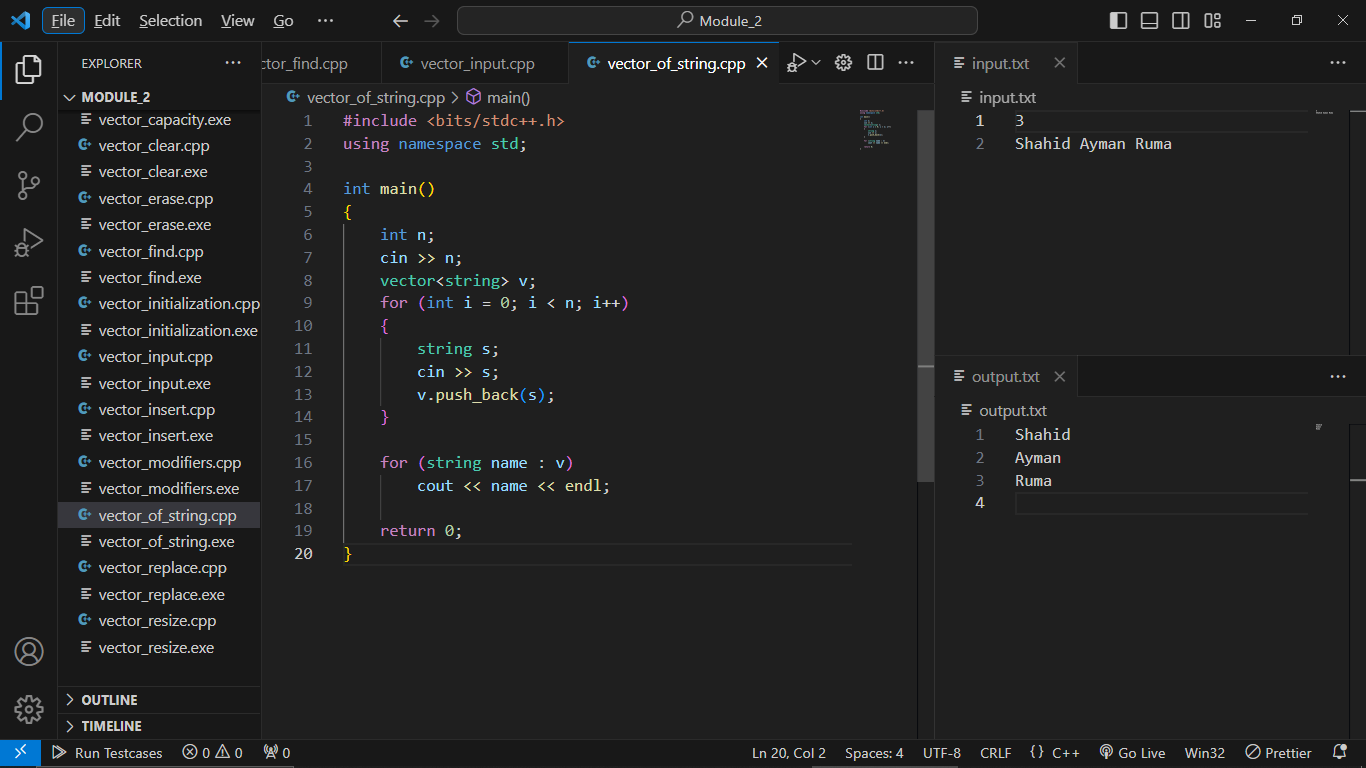
****

**Vector Input:**

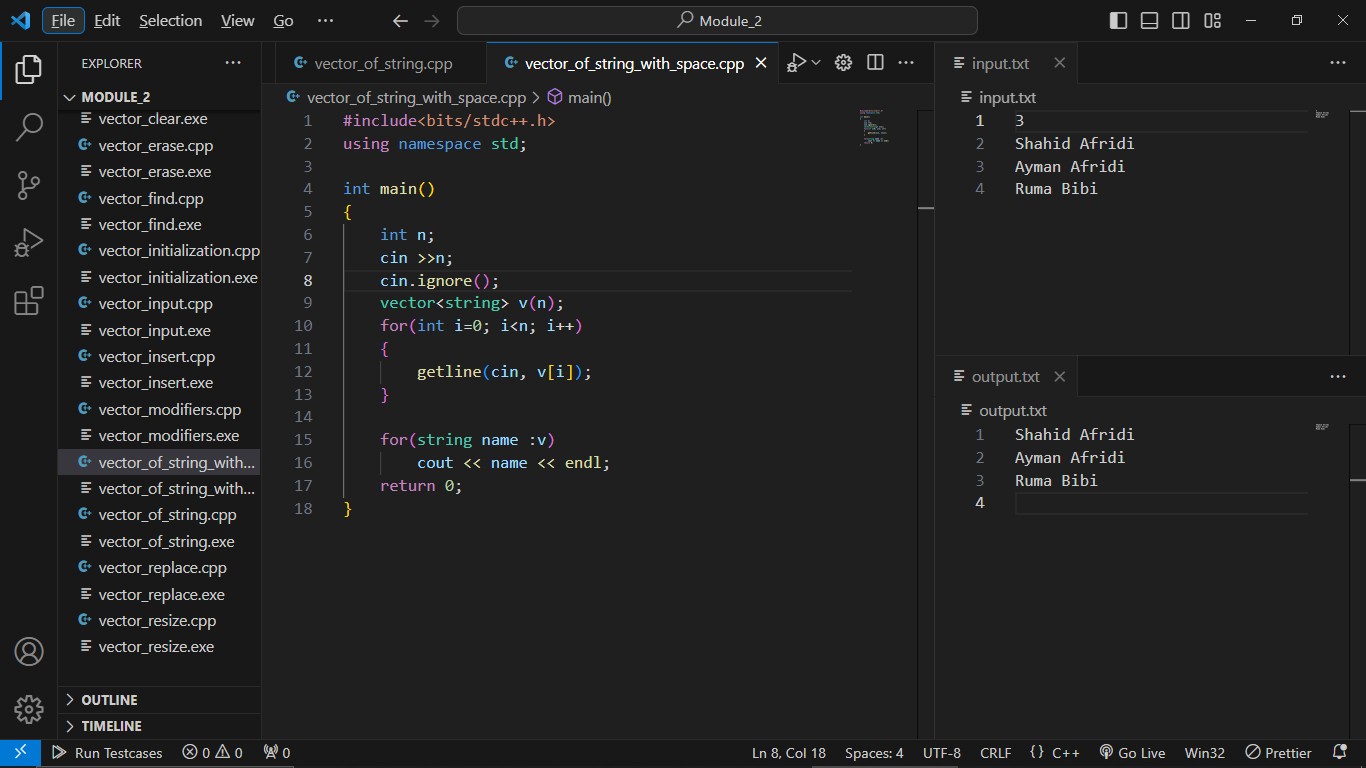
****

****

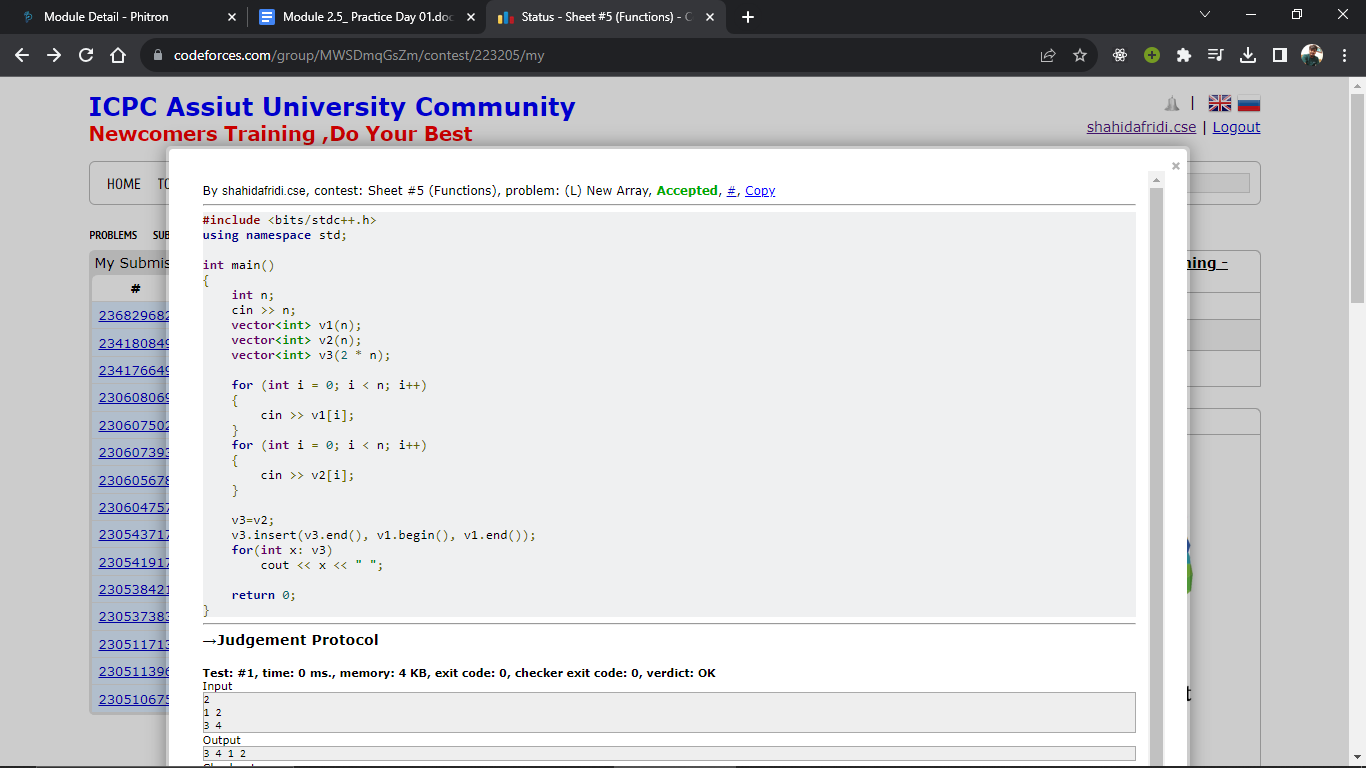
**Vector of String:**

****

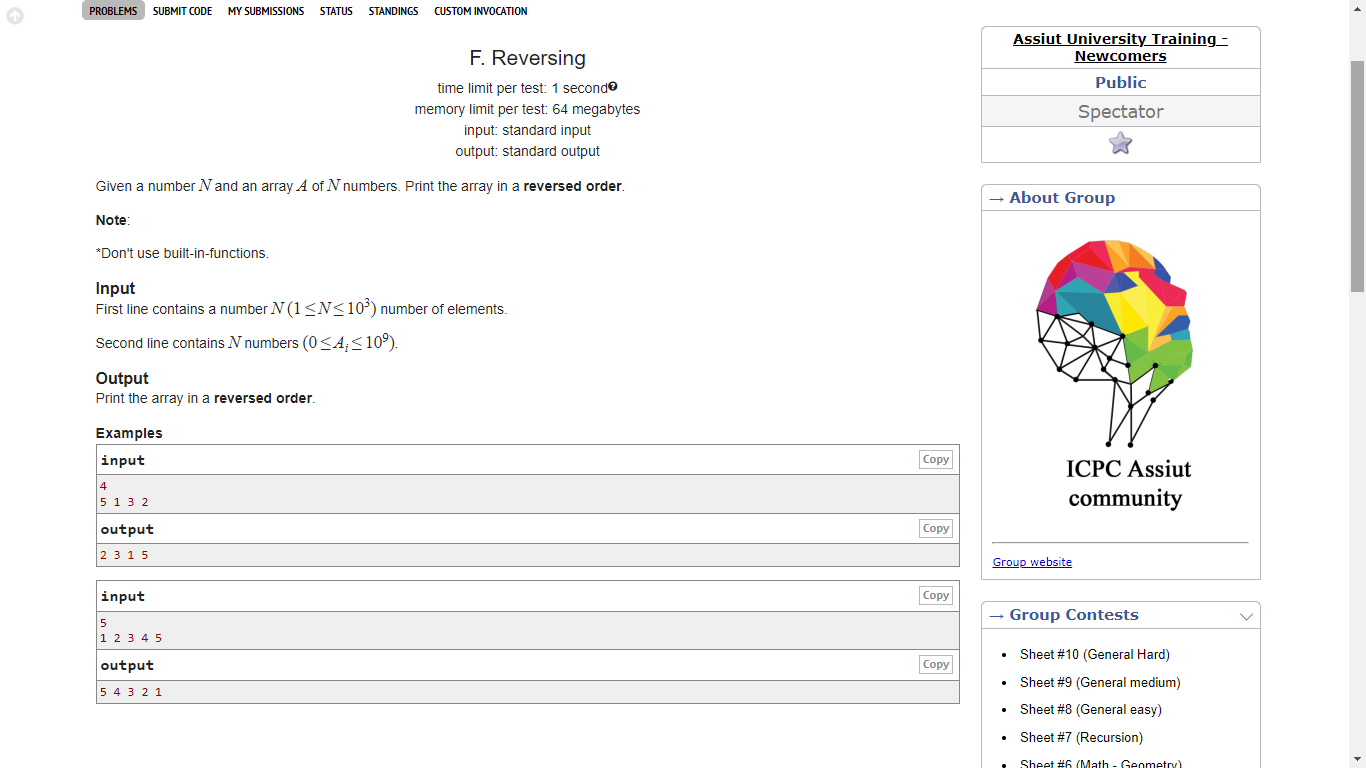
**Vector of String with Space:**

****

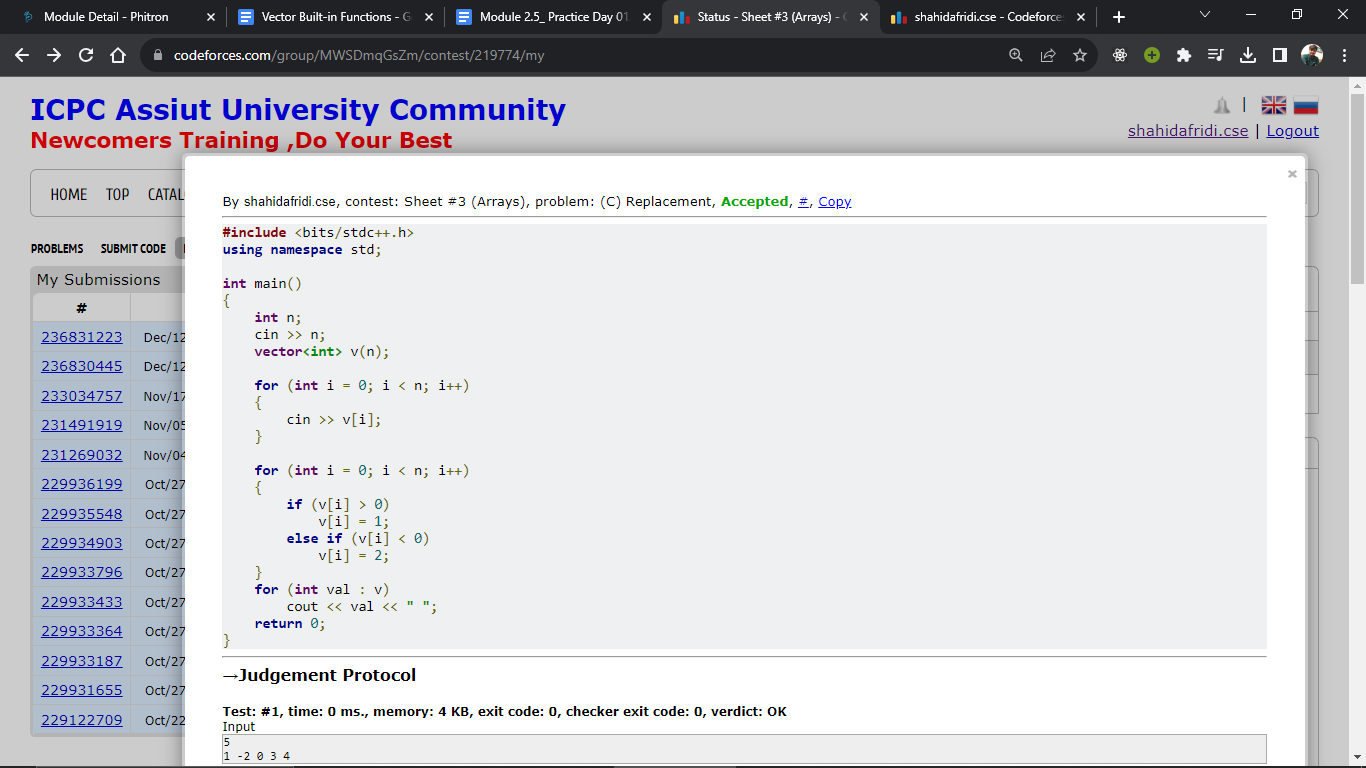
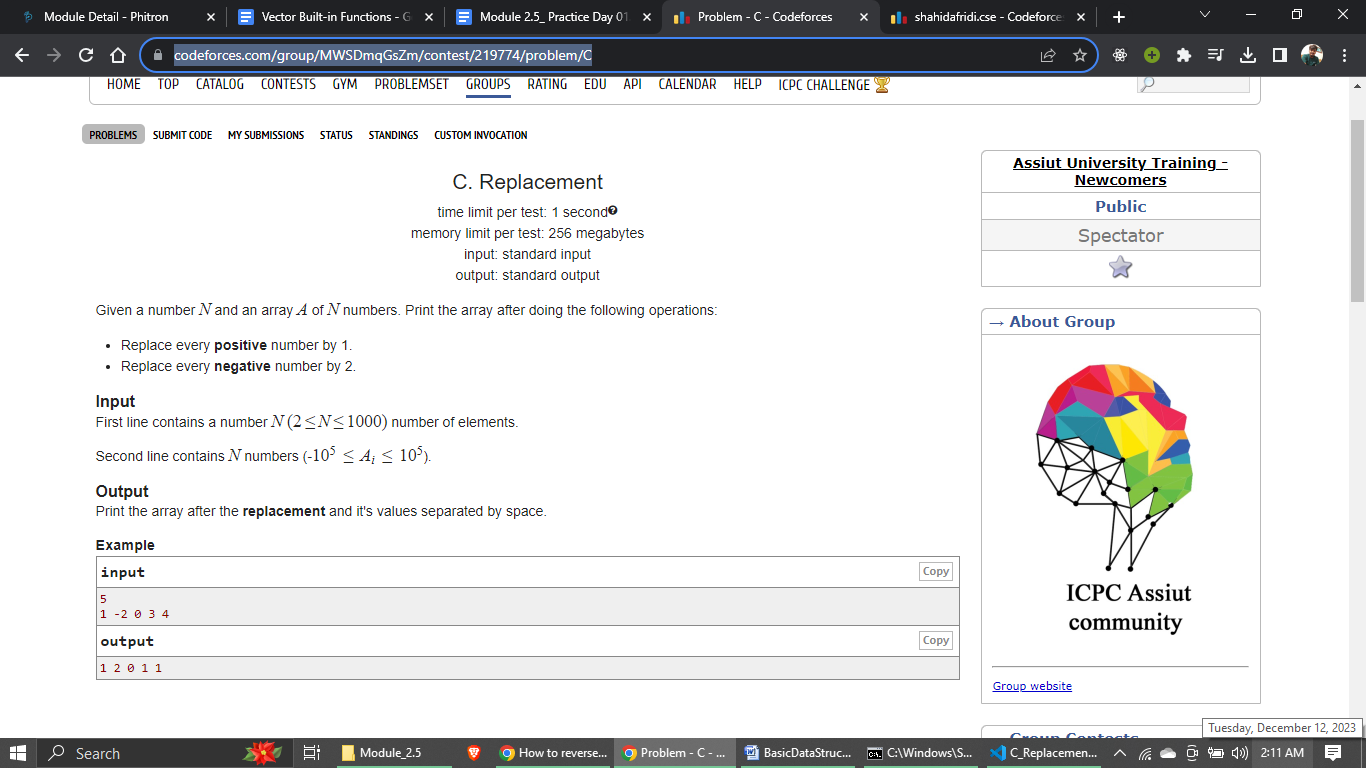
**Module 2.5:**

****

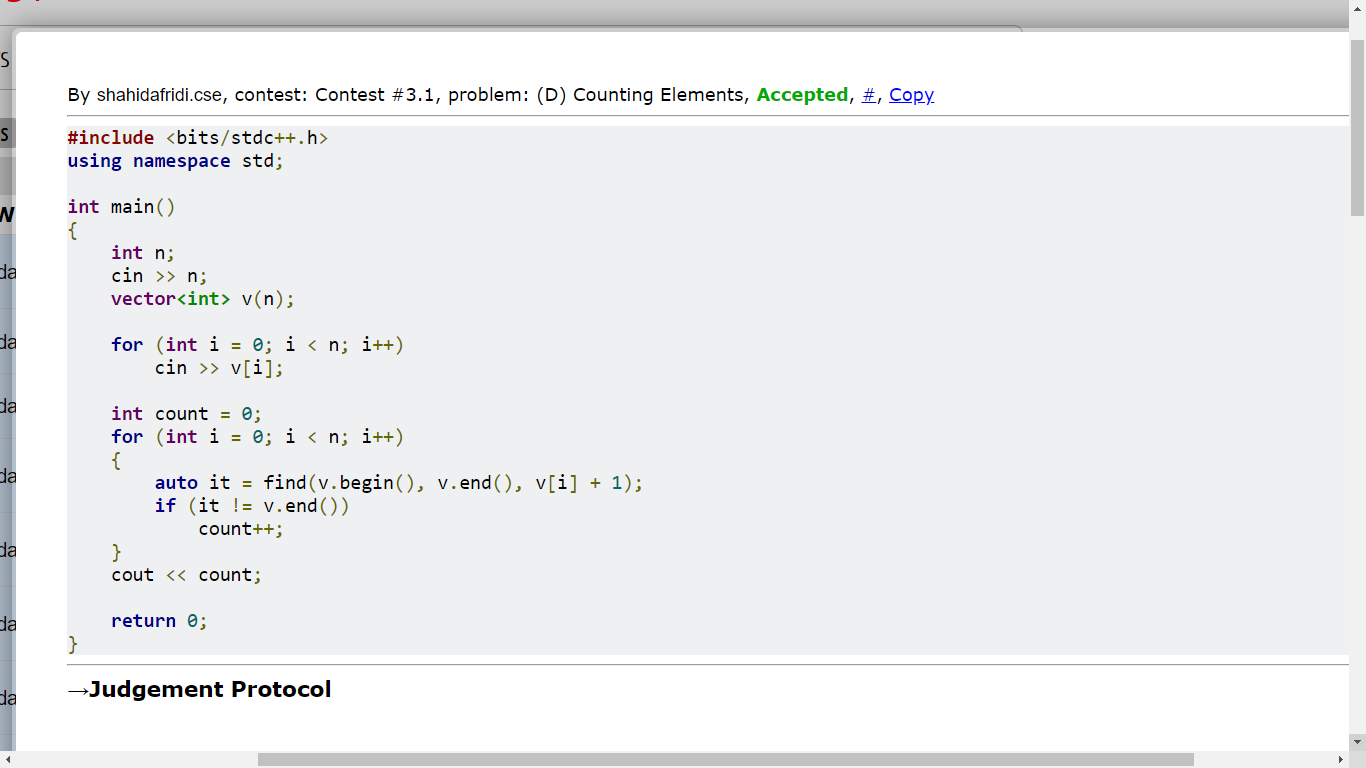
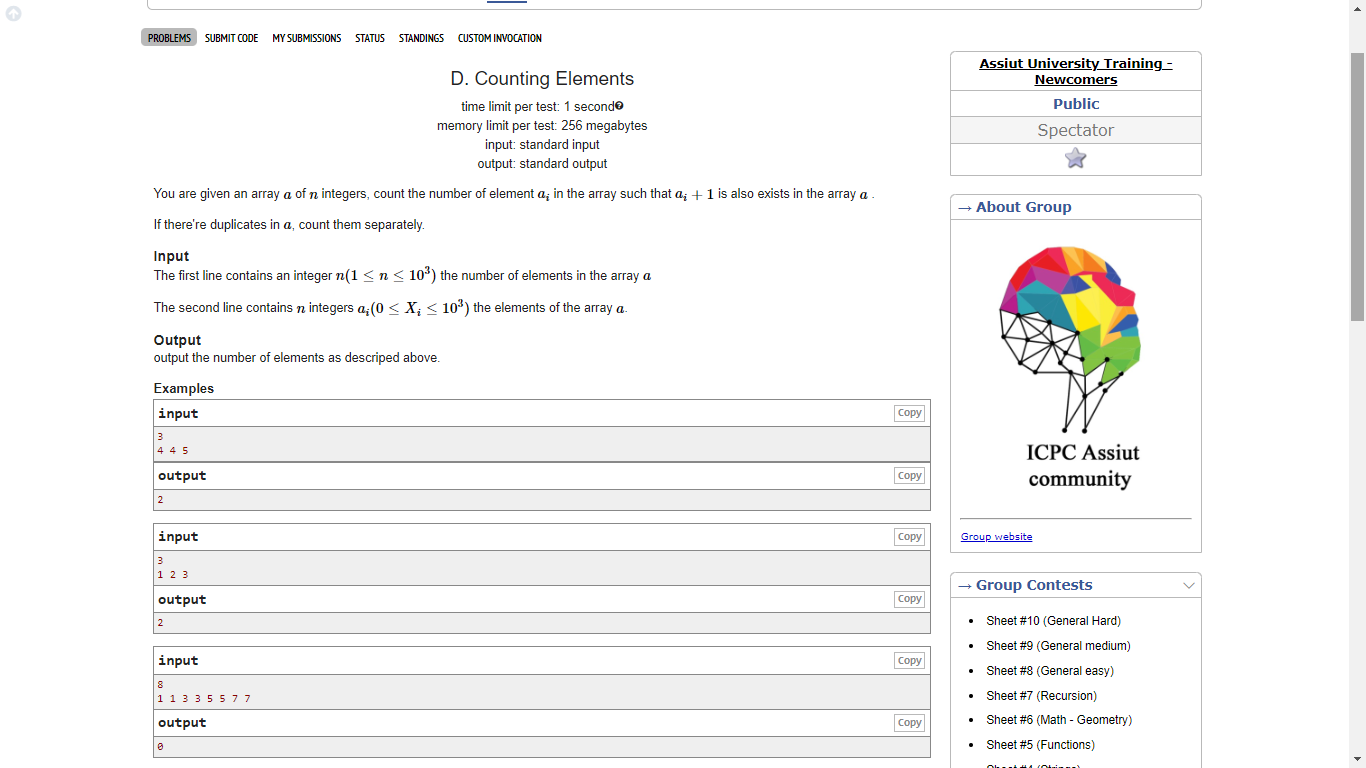
**Problem 2:**

****

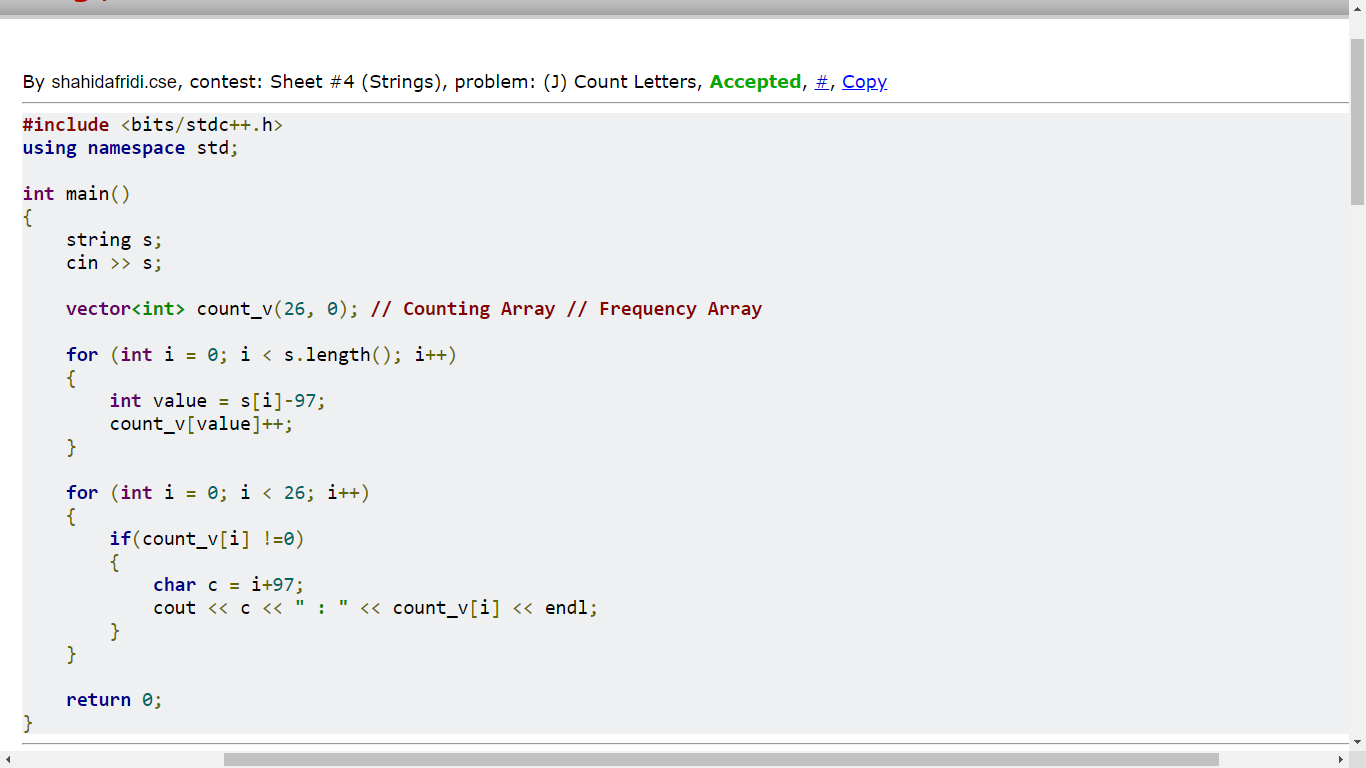
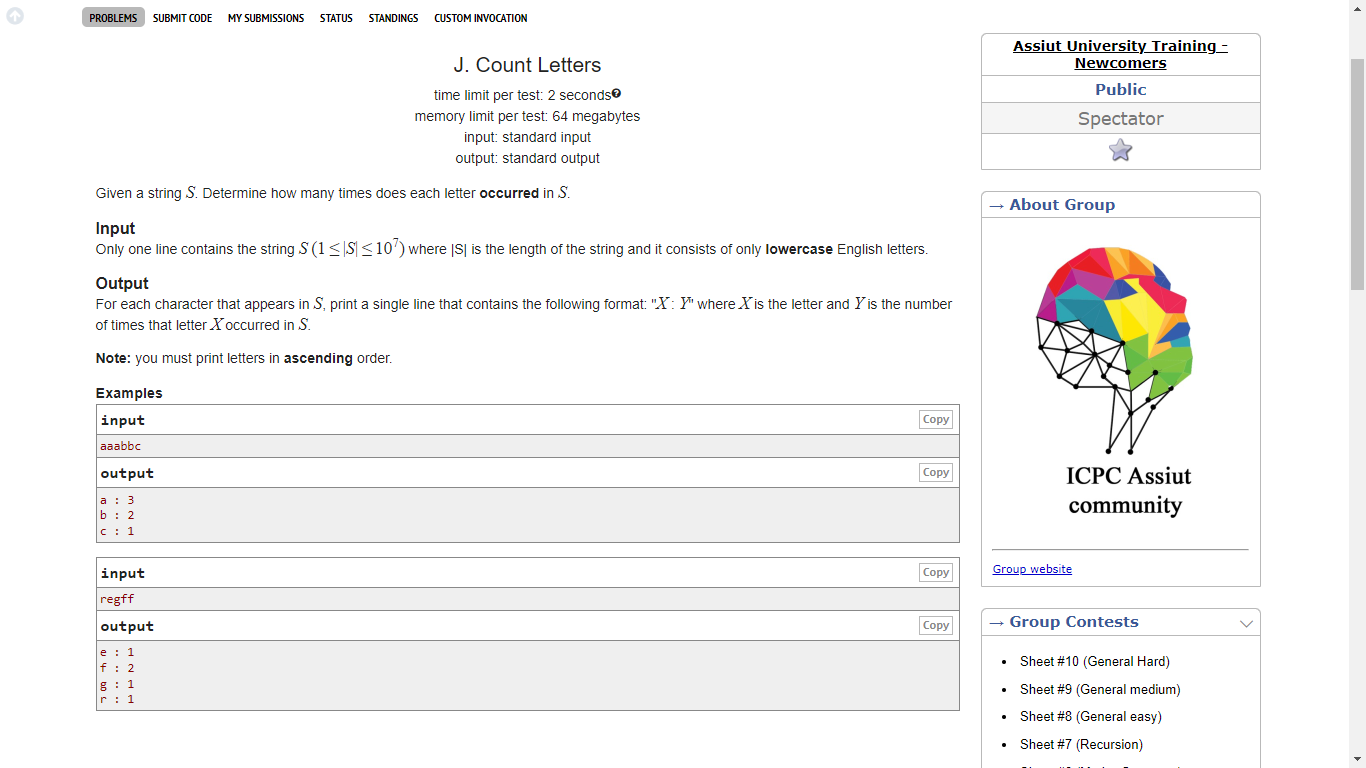
**Problem 3:**

****

**Problem 4:**

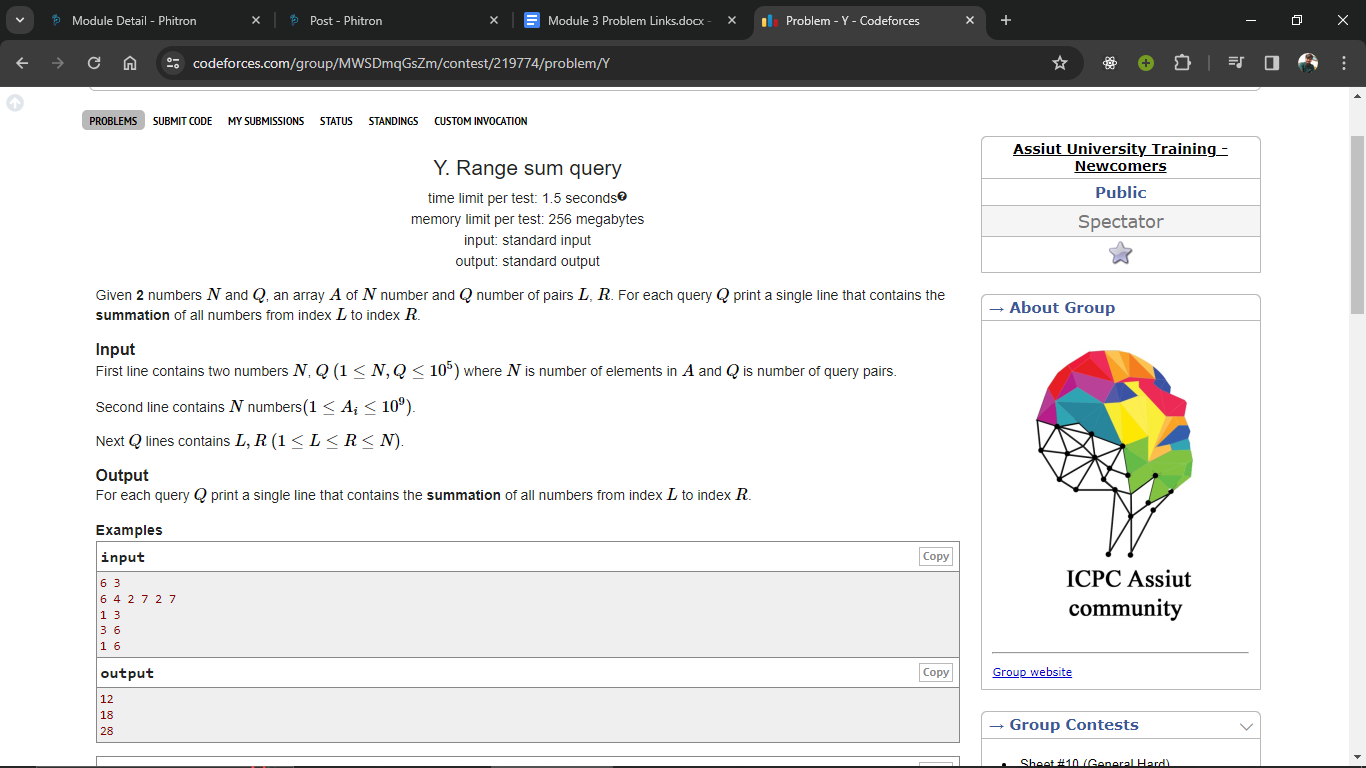
****

**Problem 5:**

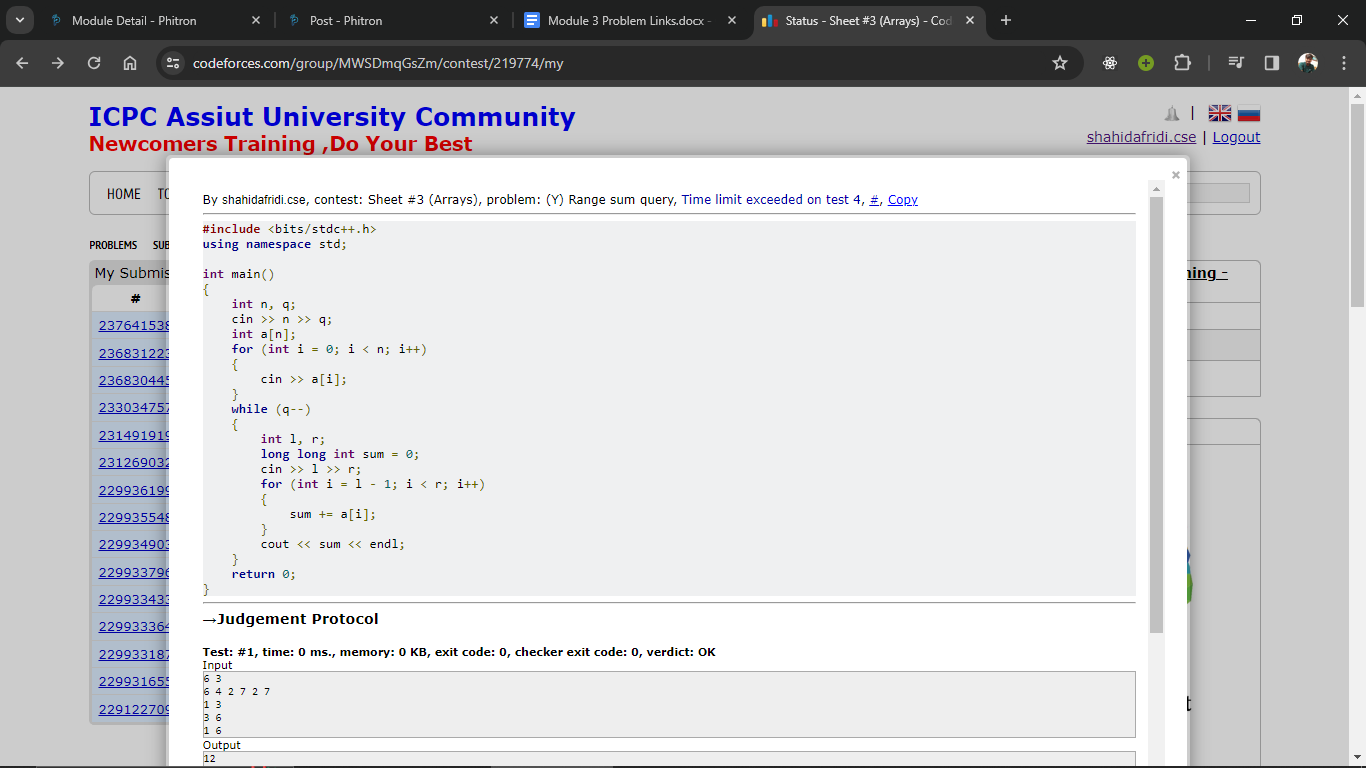
****

**Module 3:**

**Problem 1: (Prefix Sum Array)**

****

**Time limit exceeded in bruteforce**

****

**Here, it’s time complexity is O(Nq).**

* **N = 10^5 and q = 10^5**
* **Nq = 10^10 (It’s time complexity is more than 1.5 second)**

**That’s why it got a time limit exceeded error.**

Now using **Prefix sum Array** we will solve this problem.

**Prefix sum:** A prefix sum array, also known as a cumulative sum array, is an array that stores the cumulative sum of elements up to a certain index. This can be useful in various algorithms and problems where you need to efficiently compute the sum of a range of elements in an array.

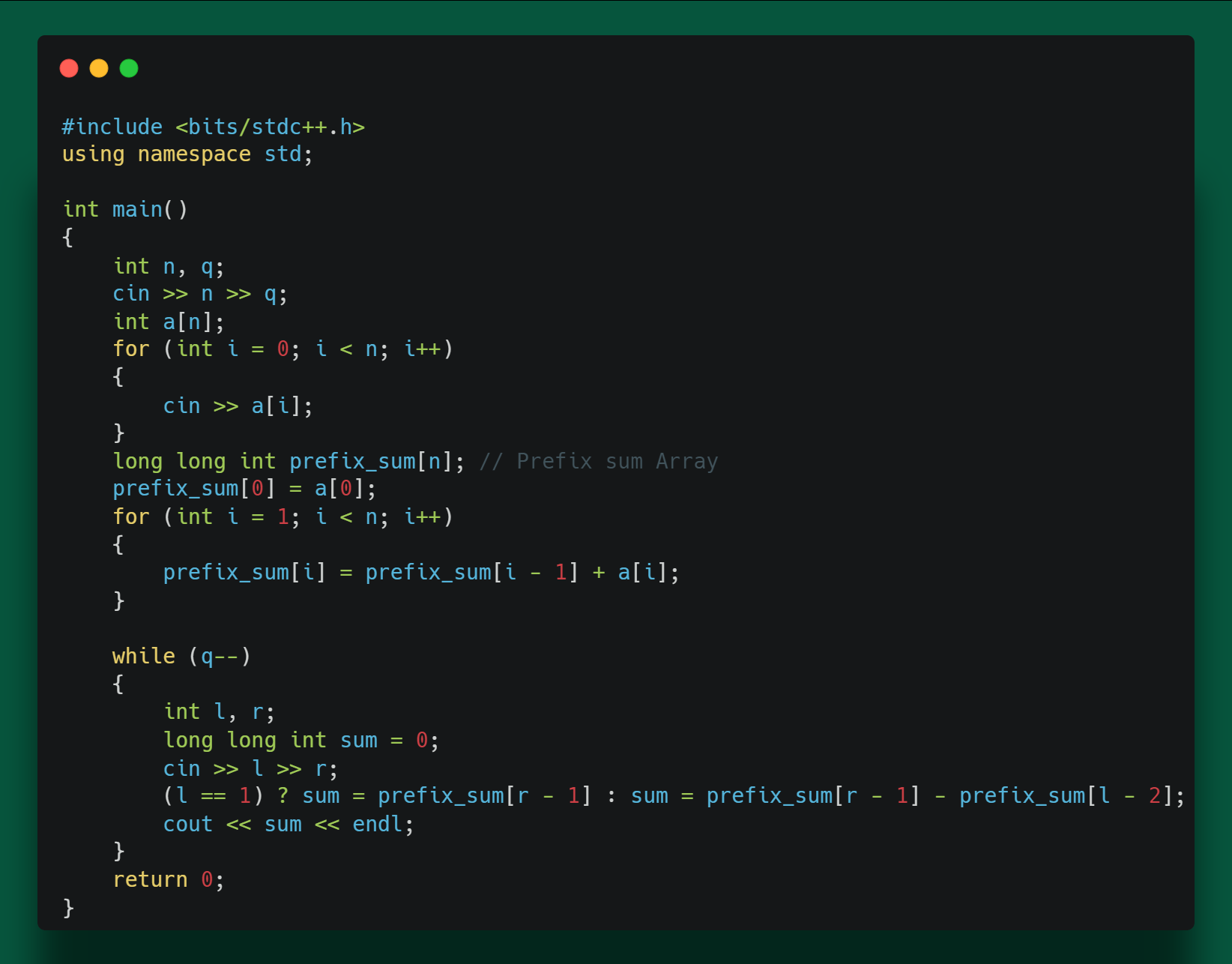
Let's illustrate the process of creating a prefix sum array with a table using the example array [1, 2, 3, 4, 5].

| **Index** | **Original Array** | **Prefix Sum Array** |
| --- | --- | --- |
| 0 | 1 | 1 |
| 1 | 2 | 1 + 2 = 3 |
| 2 | 3 | 1 + 2 + 3 = 6 |
| 3 | 4 | 1 + 2 + 3 + 4 = 10 |
| 4 | 5 | 1 + 2 + 3 + 4 + 5 = 15 |

Here,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Original Array = | 1 | 2 | 3 | 4 | 5 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Prefix Sum Array = | 1 | 3 | 6 | 10 | 15 |

****

**Algorithm Of this code is below:**

1. Input:

* Read integers n and q from the input, representing the size of the array and the number of queries, respectively.
* Read array elements a[i] for each i from 0 to n-1.

2. Prefix Sum Computation:

* Initialize an array **prefix\_sum** of type **long long int** to store the prefix sum.
* Calculate the prefix sum array **prefix\_sum** for the given array **a** using a loop.

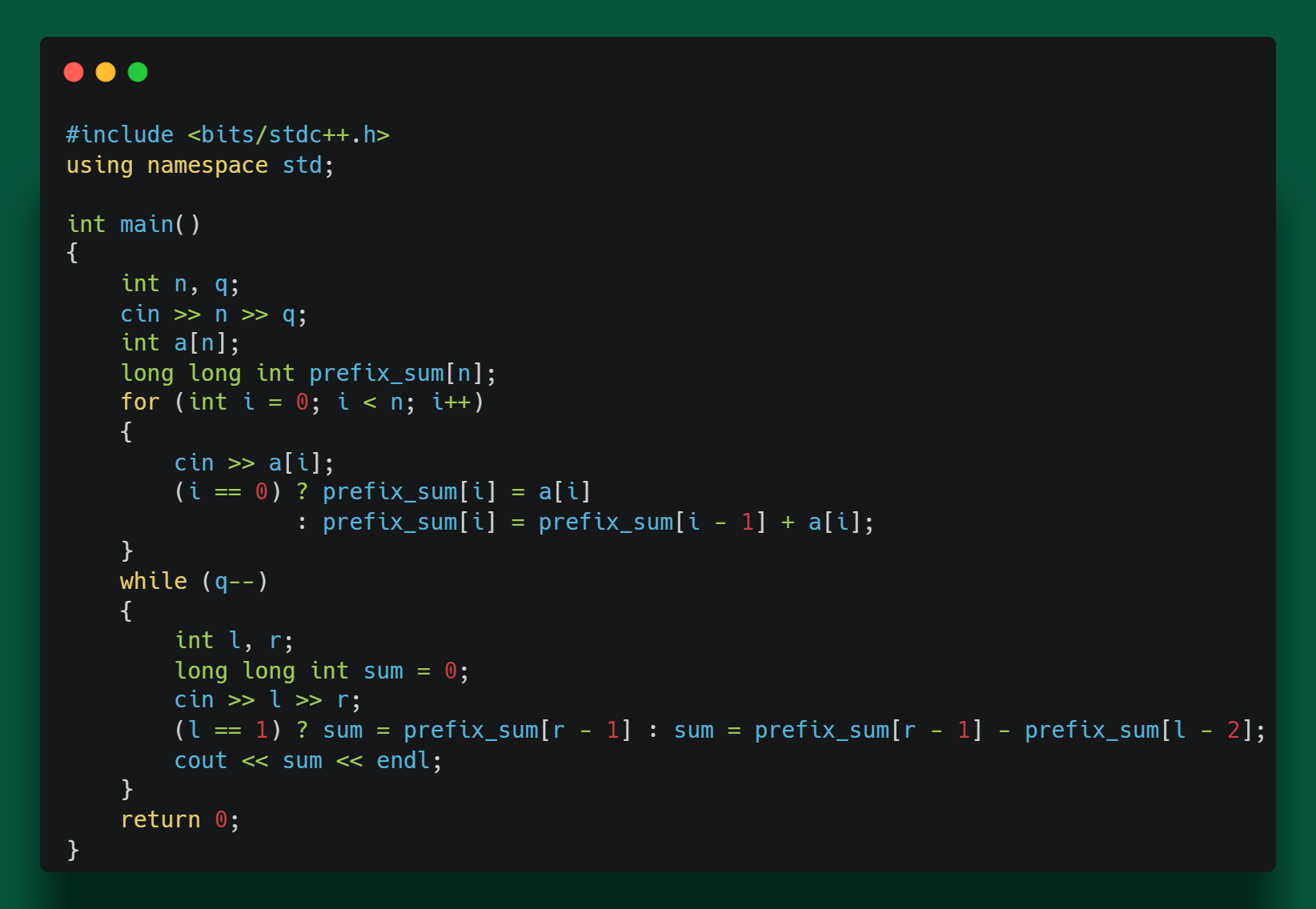
3. Query Processing:

* For each query (**q** times):
  + Read integers **l** and **r** from the input, representing the range of indices in the array.
  + Calculate the sum of elements in the range **[l, r]** using the prefix sum array:
    - If l is 1, set sum to prefix\_sum[r - 1].
    - If l is not 1, set sum to prefix\_sum[r - 1] - prefix\_sum[l - 2]. This handles the case where l is not at the beginning of the array.
  + Print the calculated sum.

4. Output:

* Print the sum for each query on a new line.

**Same approach just a little bit different style in code.**

****

**Module 3.5: Binary Search**

Let's use an example with the array **[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]** and the target value **12** for binary search.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Low** | **High** | **Mid** | **Value at Mid** | **Comparison** | **New Range** |
| **1** | **0** | **9** | **4** | **10** | **< target** | **10-9** |
| **2** | **5** | **9** | **7** | **16** | **> target** | **5-6** |
| **3** | **5** | **6** | **5** | **12** | **= target** | **Found!** |

## Explanation:

1. **Iteration 1:**

* Initial range: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] (low = 0, high = 9).
* Midpoint: mid = (0 + 9) / 2 = 4.
* Value at midpoint: 10 (less than the target).
* Adjust the range to the right half: low = 5, high = 9.

1. **Iteration 2:**

* Updated range: **[12, 14, 16, 18, 20]** (low = 5, high = 9).
* Midpoint: **mid = (5 + 9) / 2 = 7**.
* Value at midpoint: **16** (greater than the target).
* Adjust the range to the left half: **low = 5, high = 6**.

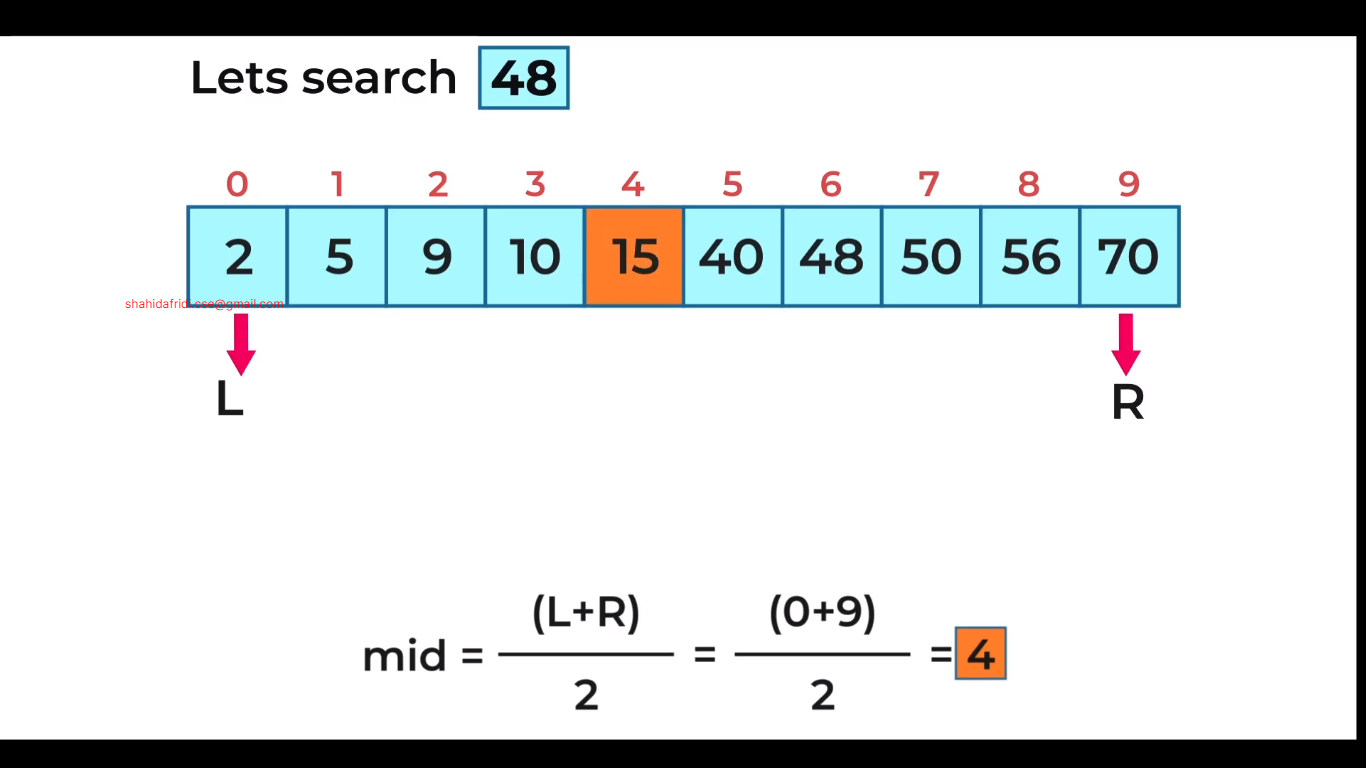
1. **Iteration 3:**

* Updated range: **[12]** (low = 5, high = 6).
* Midpoint: **mid = (5 + 6) / 2 = 5**.
* Value at midpoint: **12** (equal to the target).
* Target found!

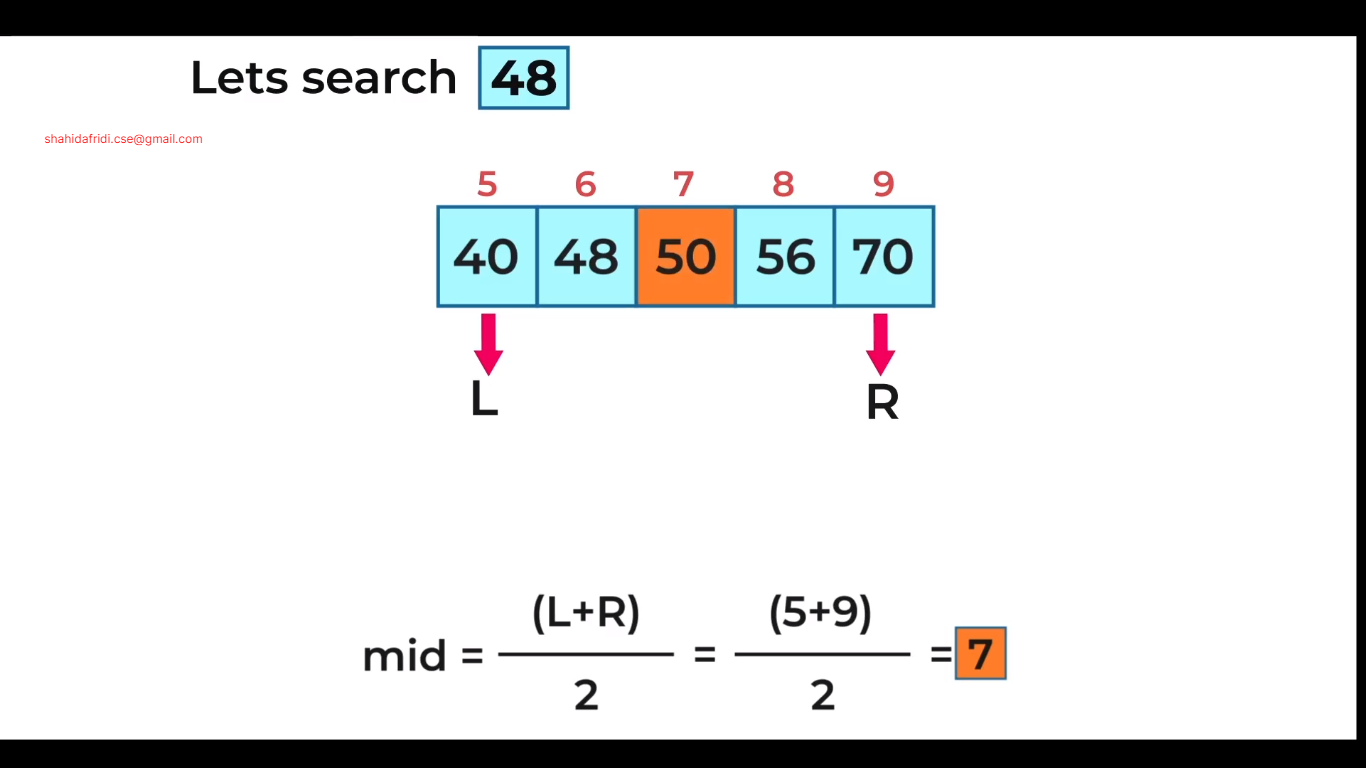
This example demonstrates how binary search efficiently narrows down the search range and finds the target value in just a few iterations.

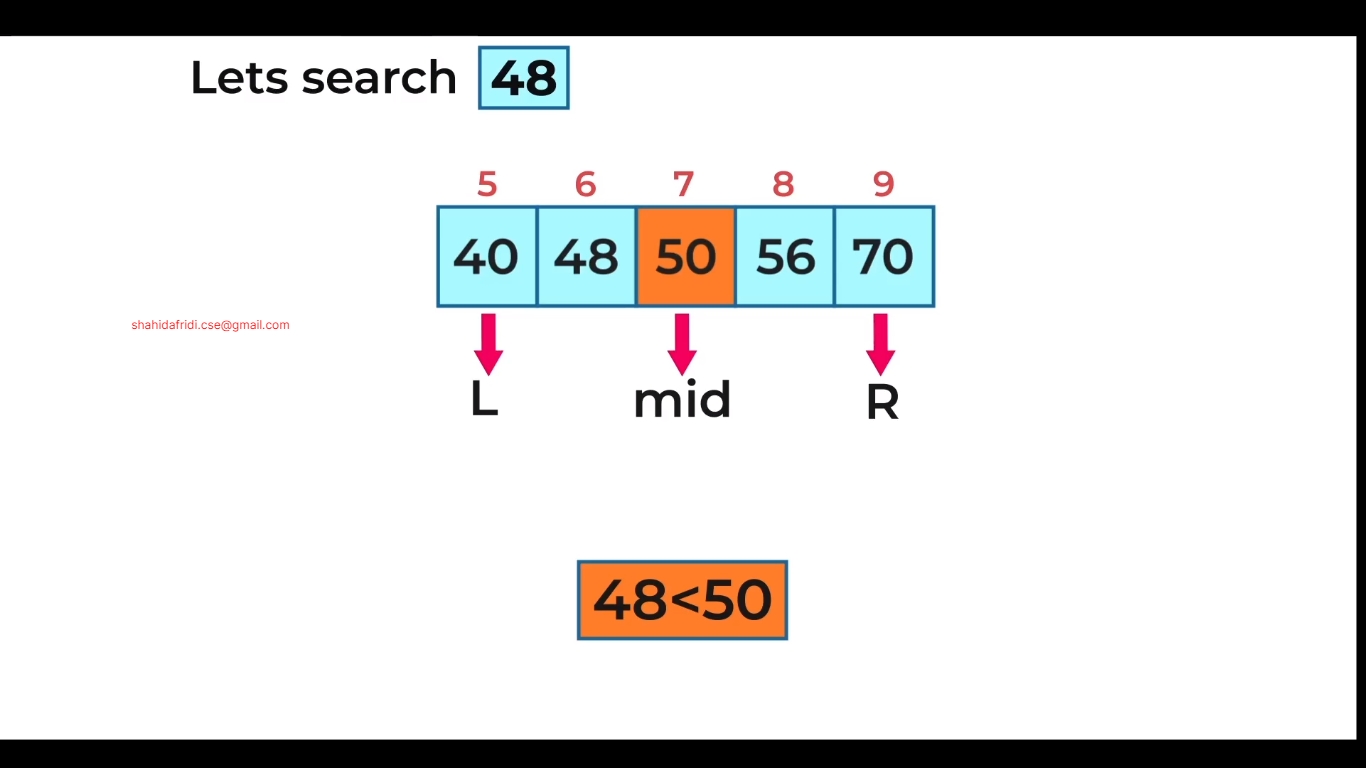
Top of Form

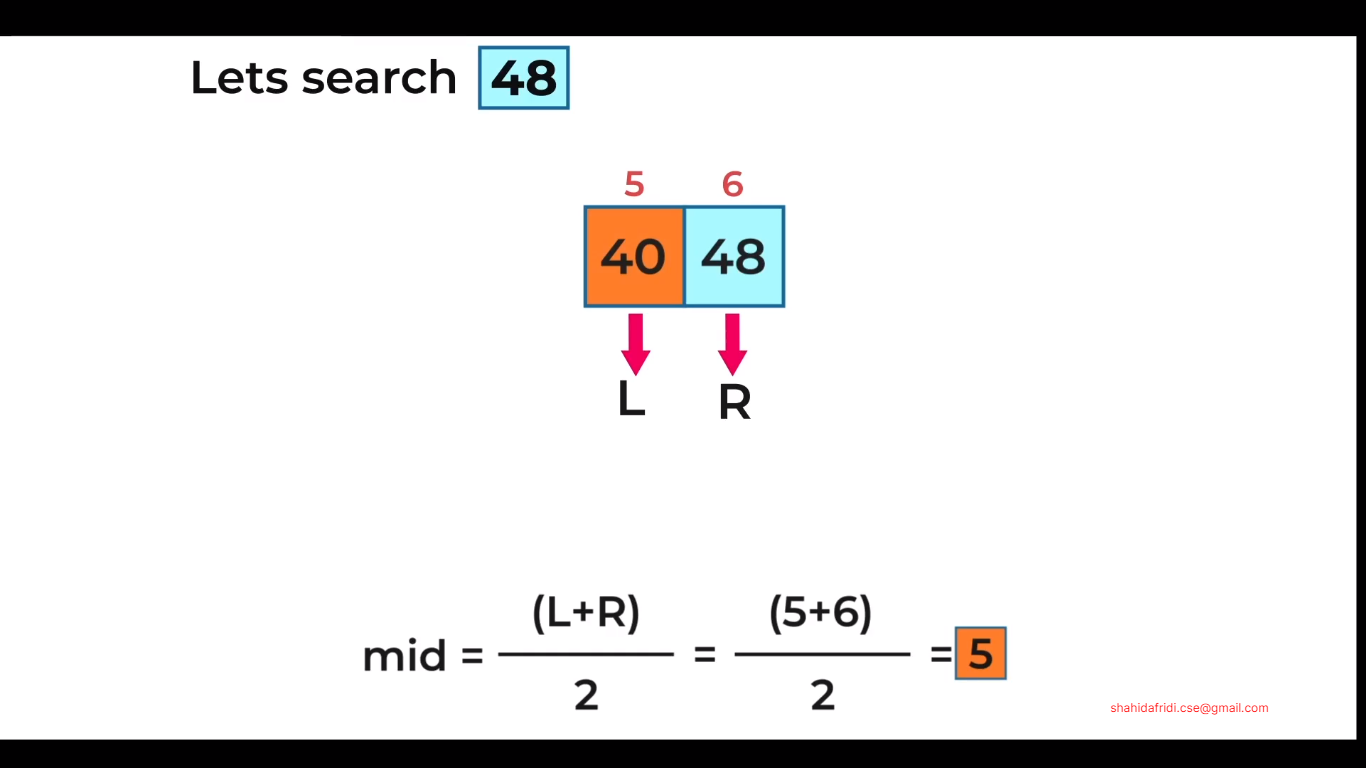
Let’s see another example with animated way.

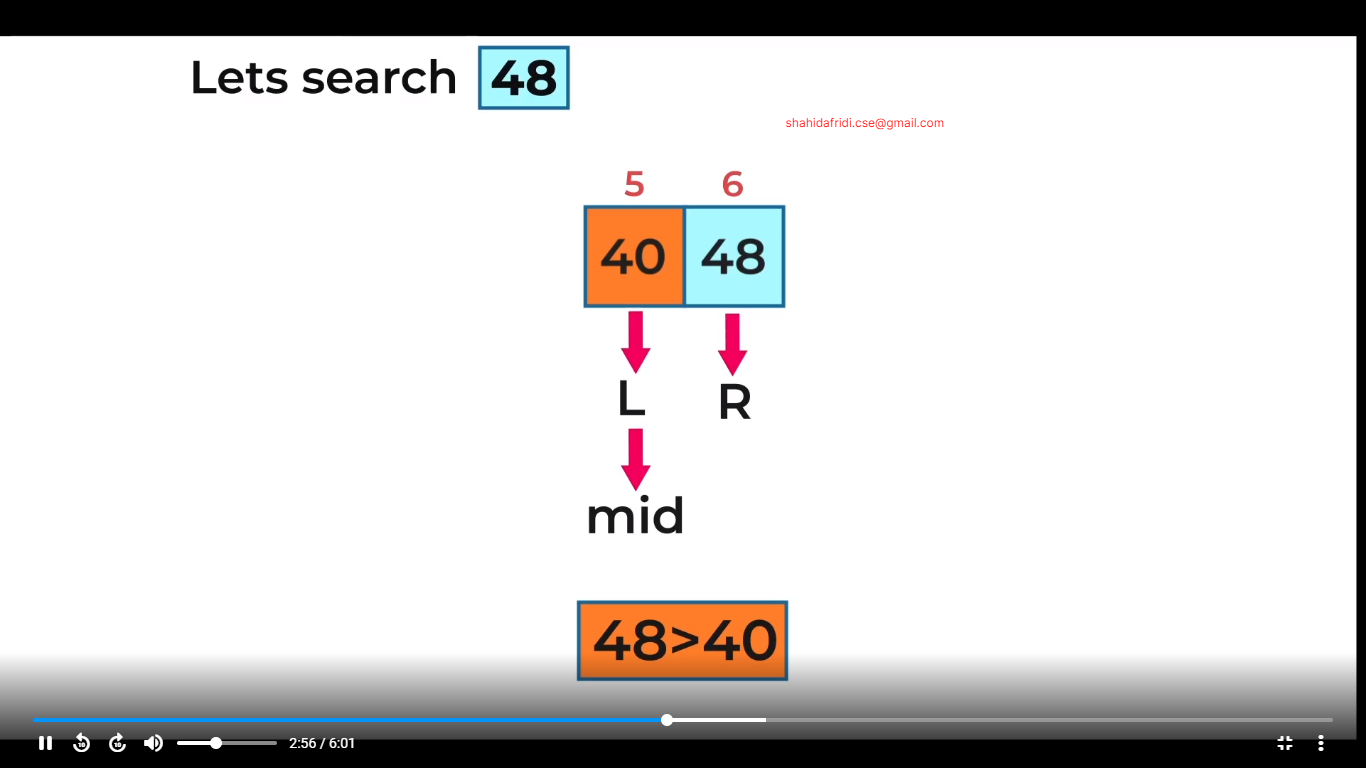




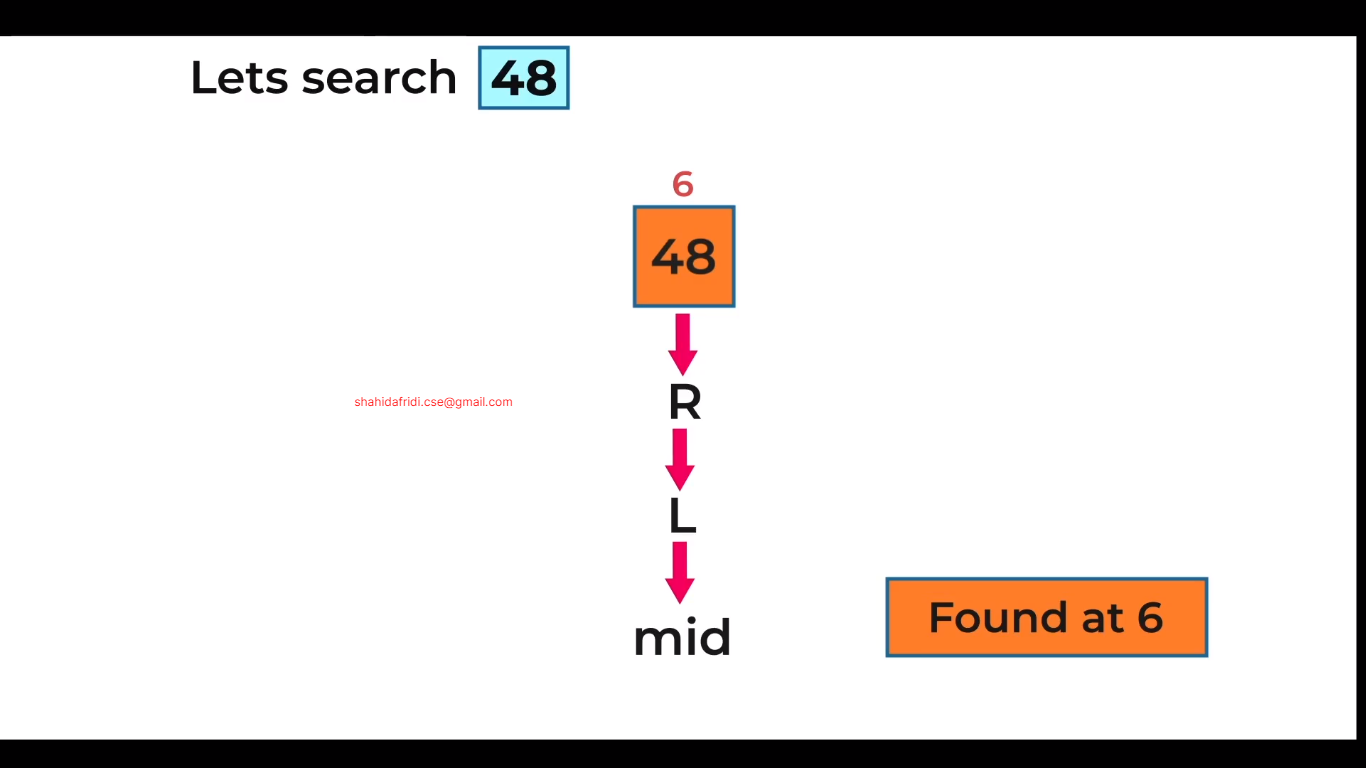




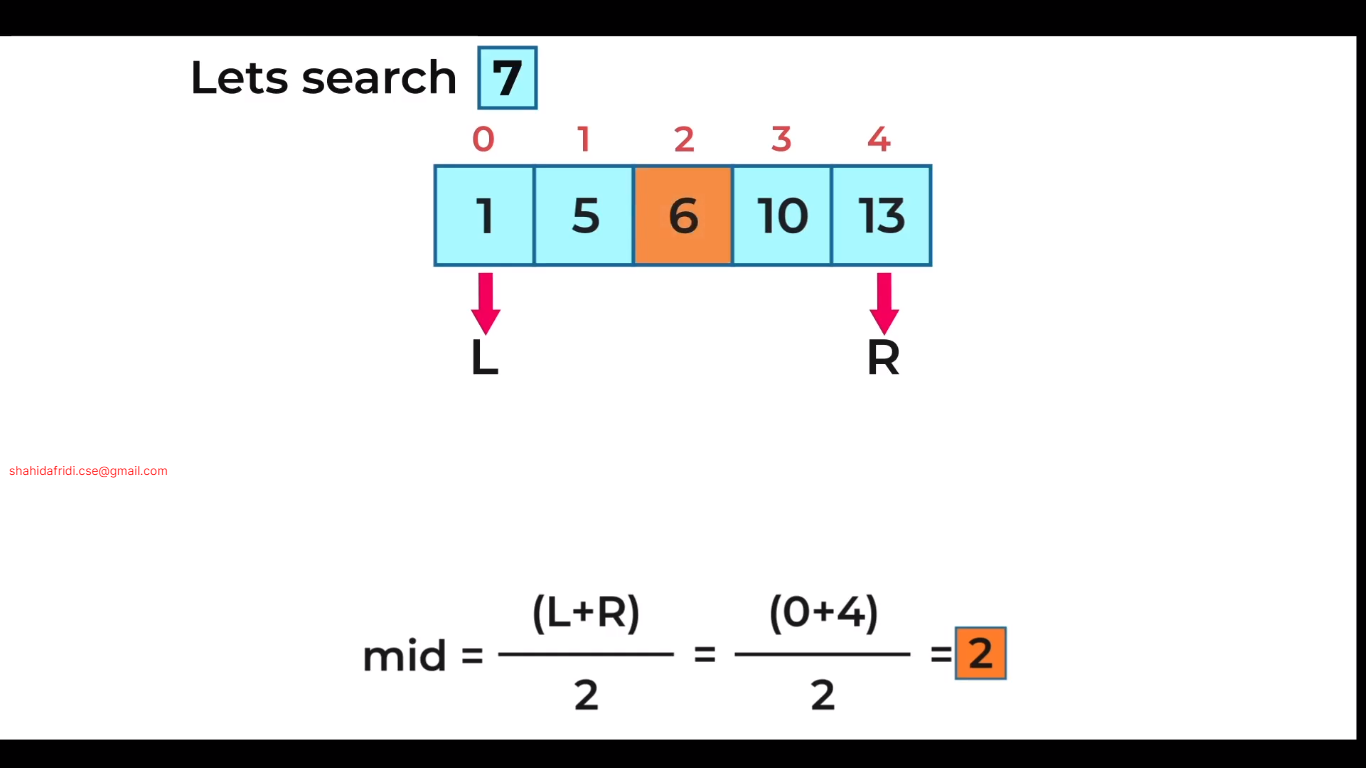


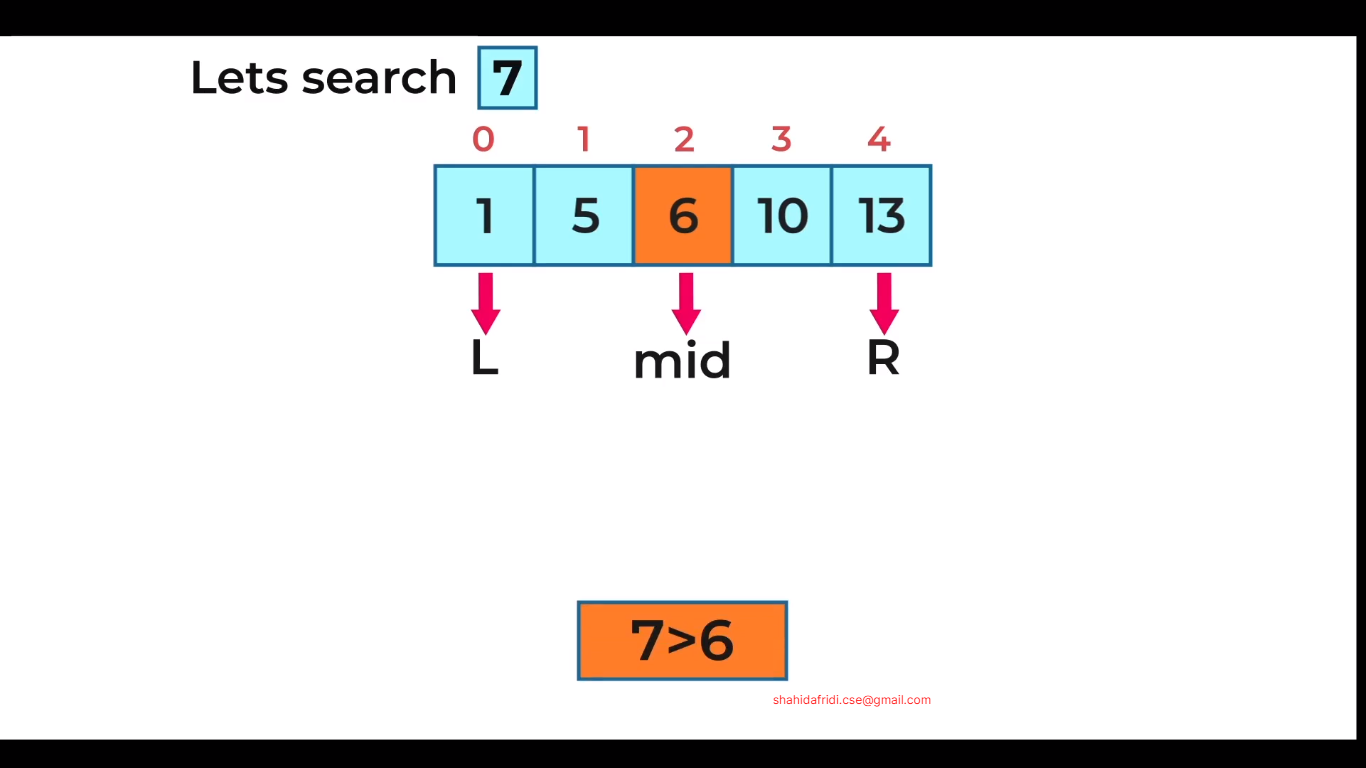


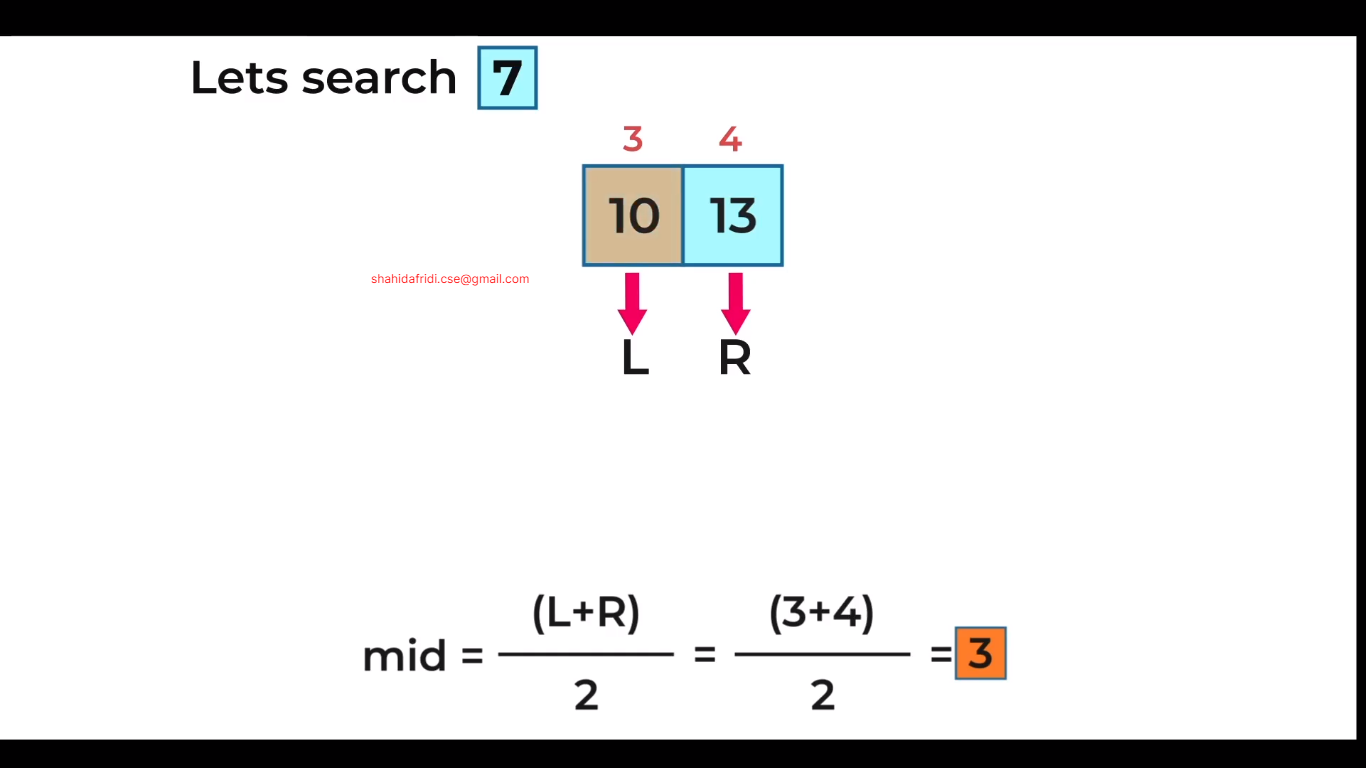


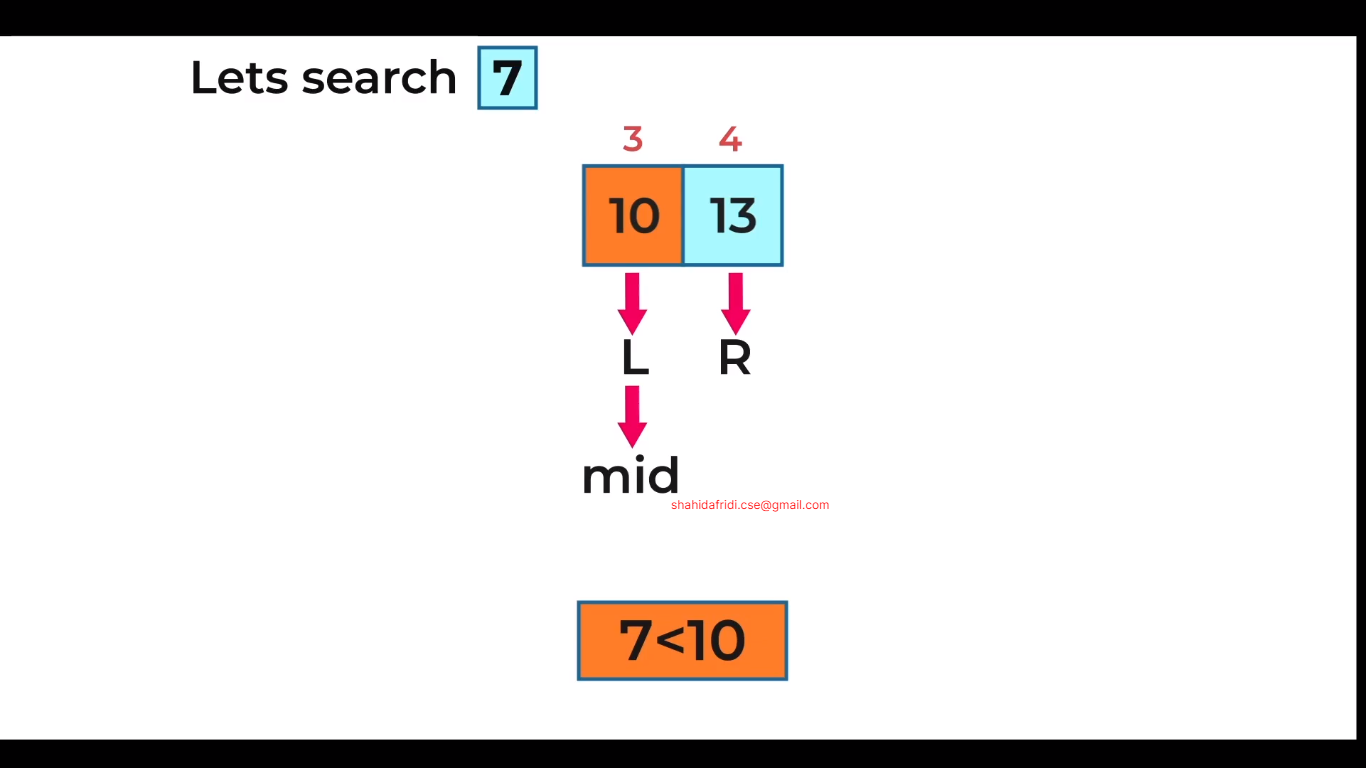


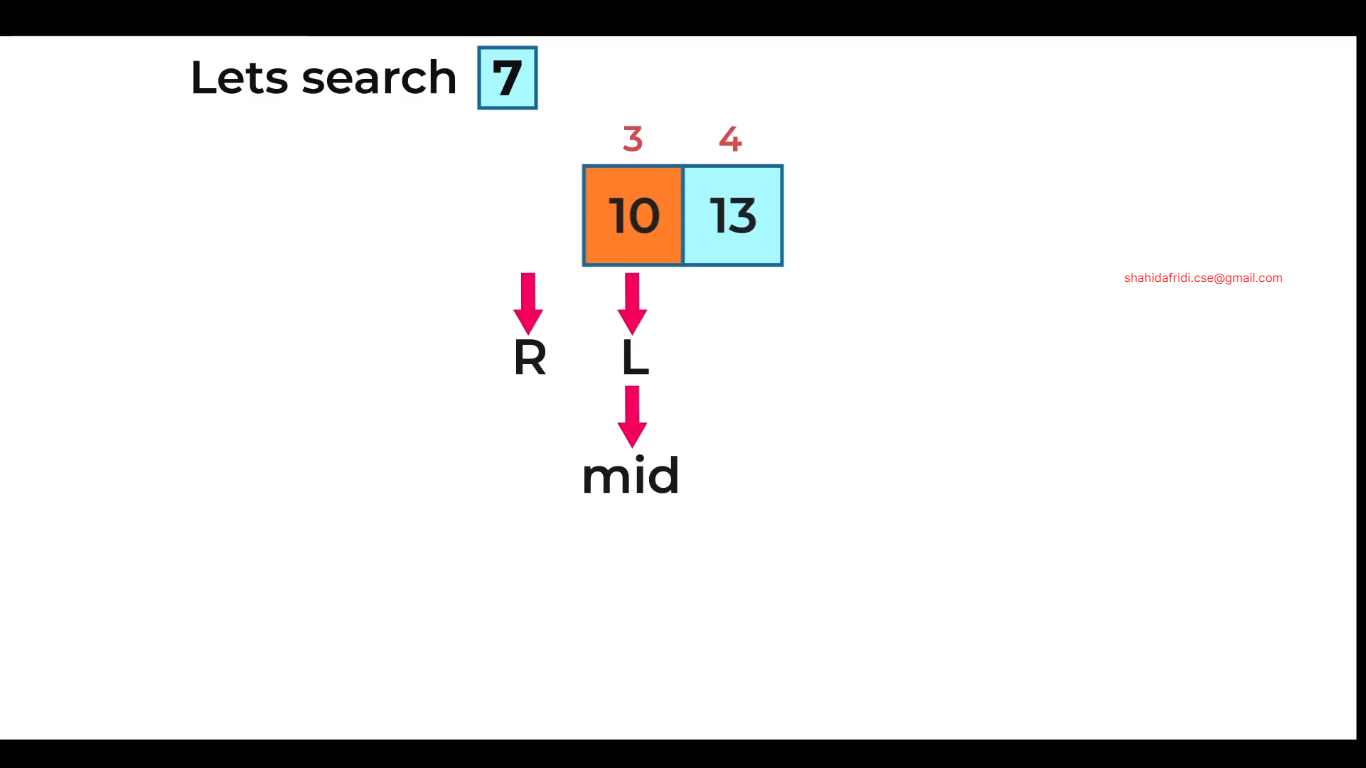
Another example where searching value is **not found.**

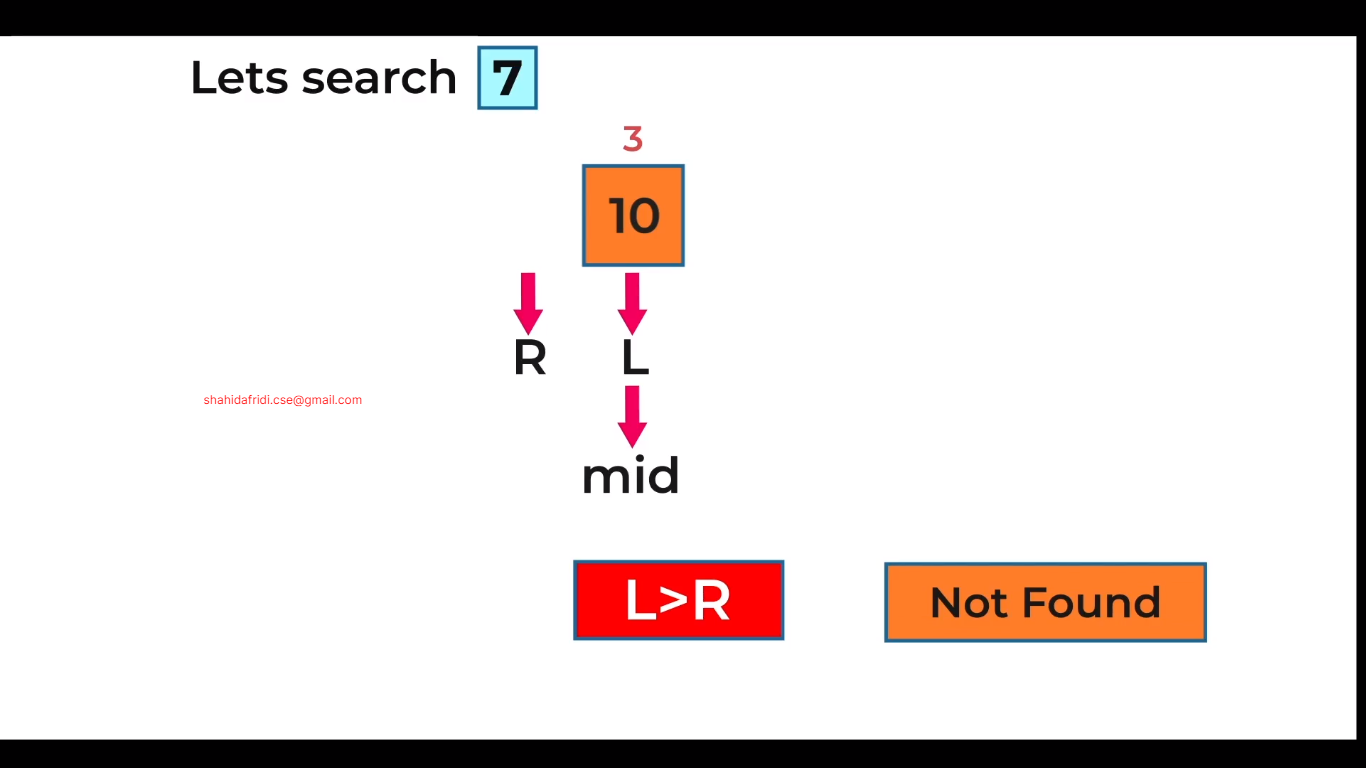
****

****

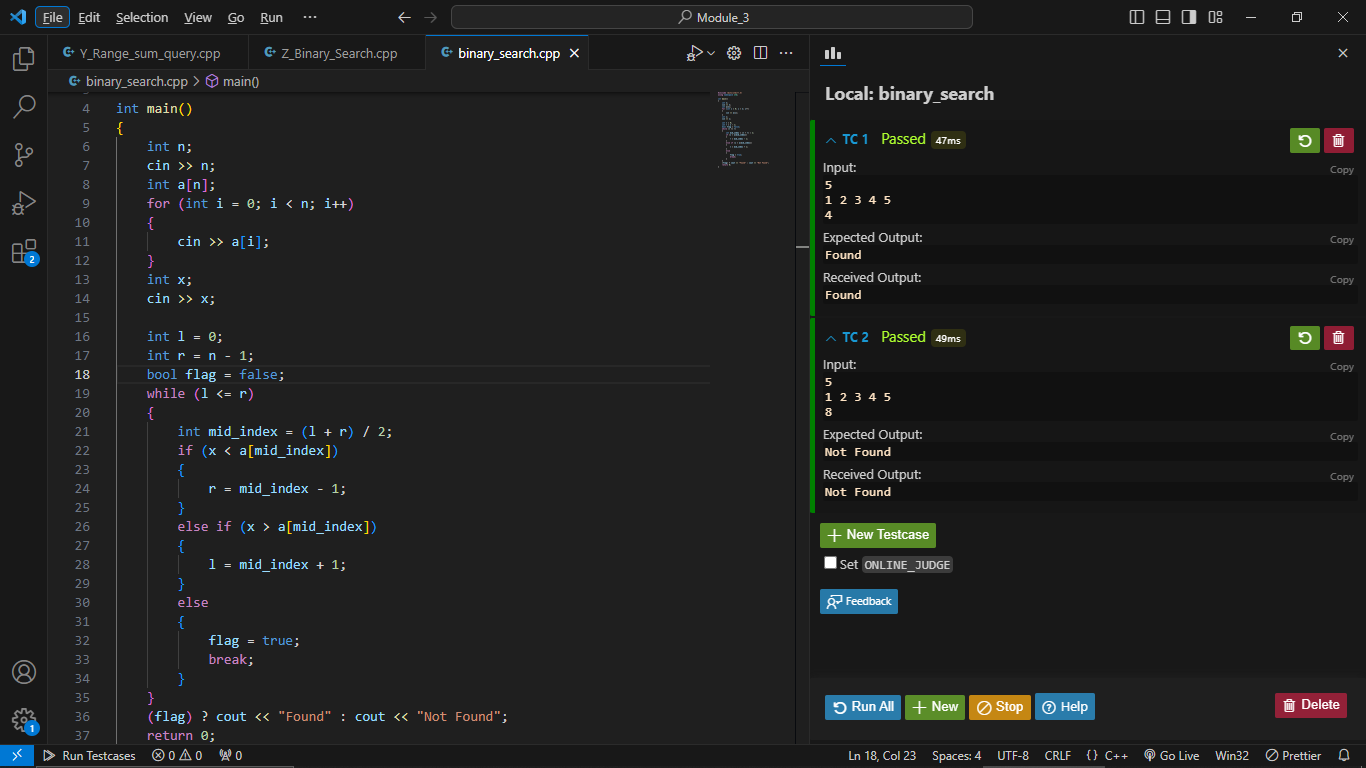
****

****

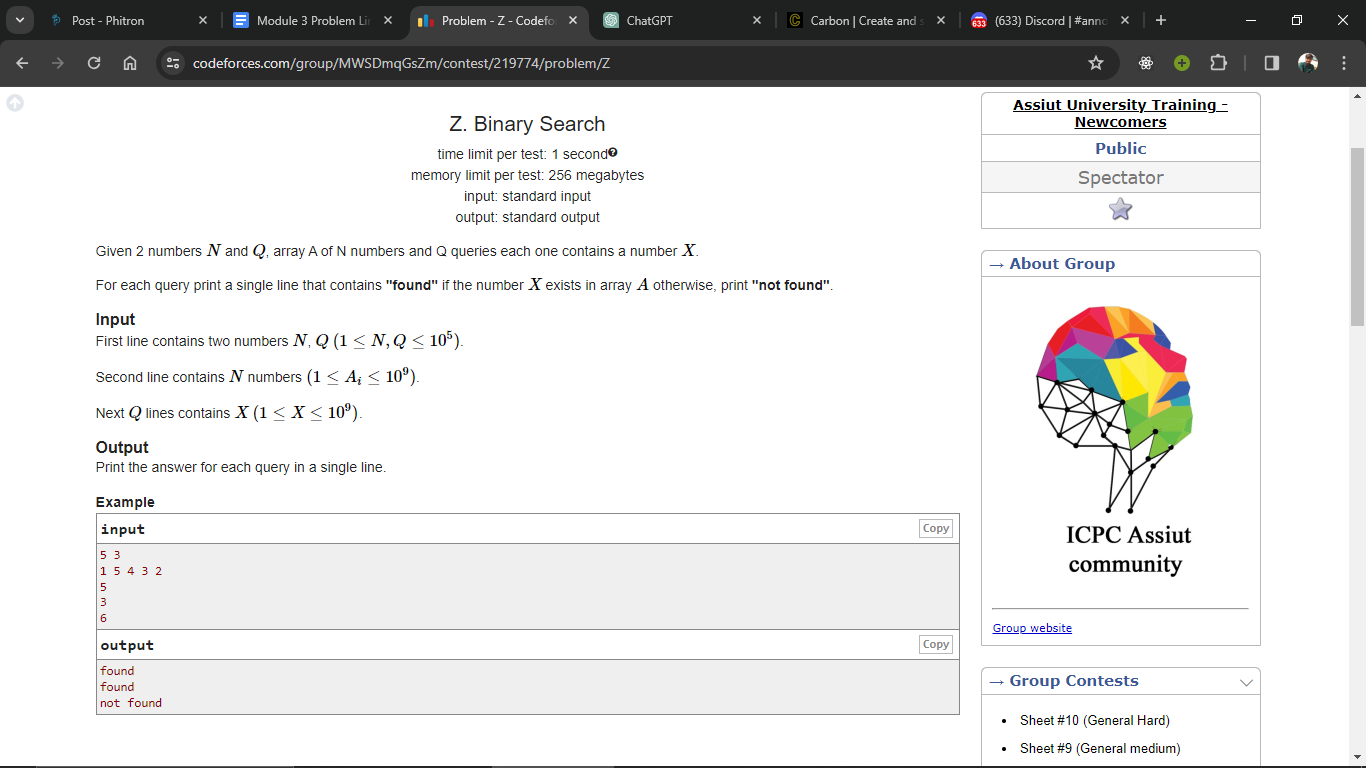




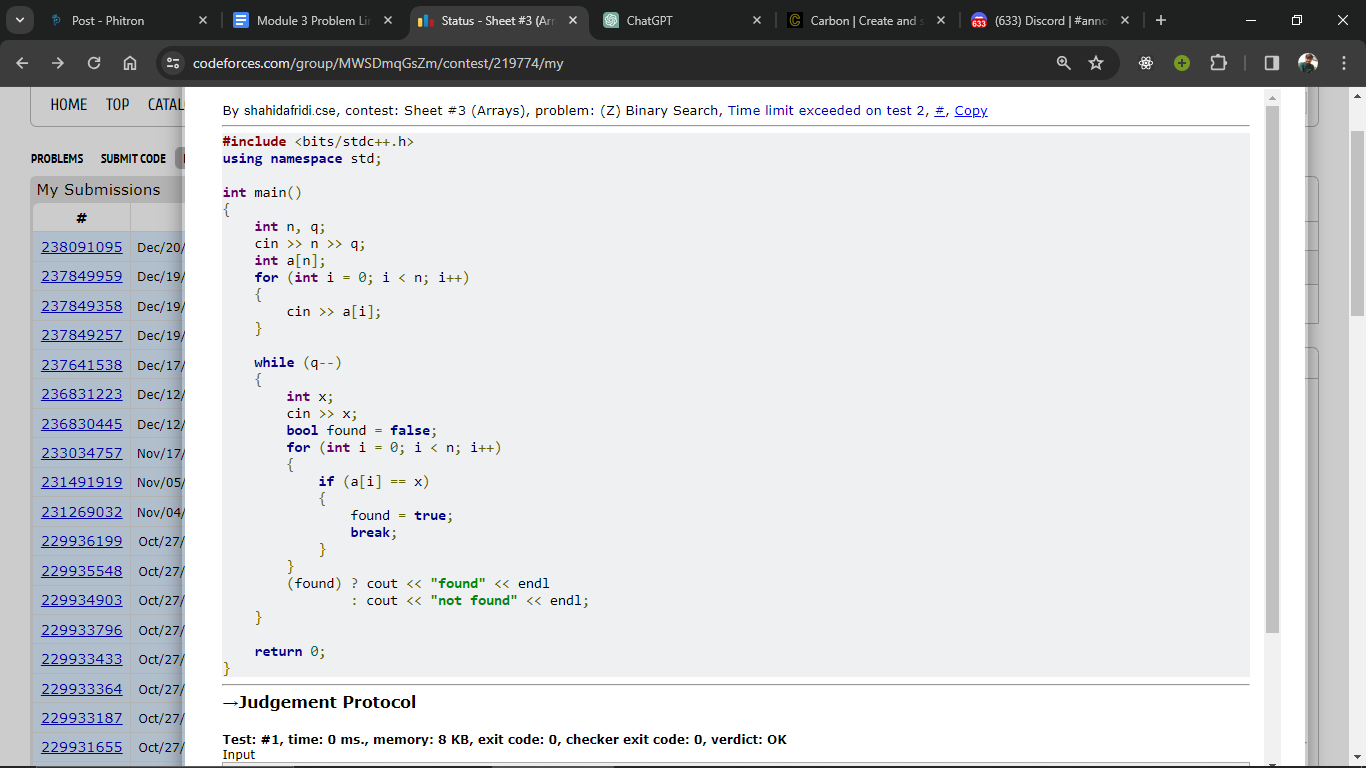
Binary Search Code in C++:



Problem:



Solution:



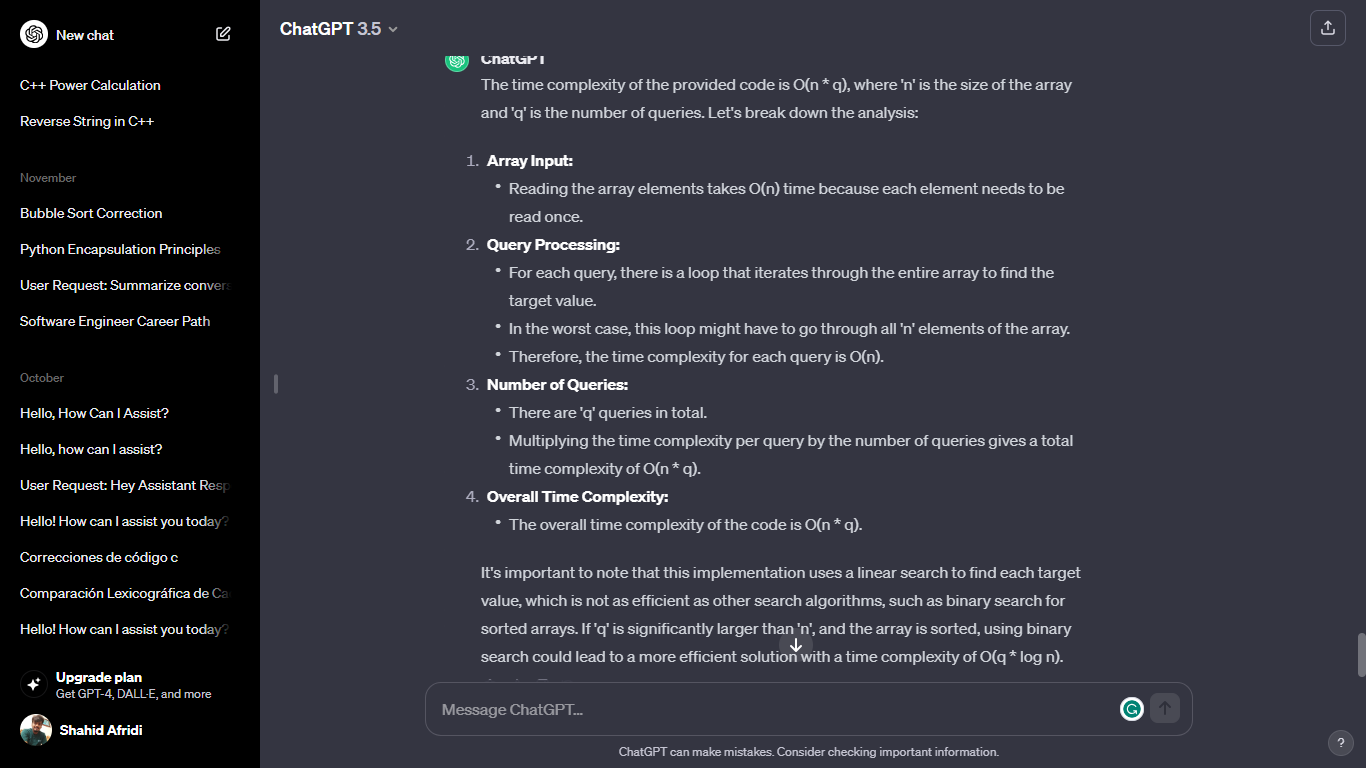
Here this problem is solved using **Brute Force Approach.** In this way it has been solved but the time limit exceeded. Because of its time complexity is O(N\*q). This is impossible to complete run in 1 second.

**Here, it’s time complexity is O(Nq).**

* **N = 10^5 and q = 10^5**
* **Nq = 10^10 (It’s time complexity is more than 1 second)**

**That’s why it got a time limit exceeded error.**

It’s Time Complexity is:



So, let’s try to solve it with Apply **Binary Search** for efficient time complexity.



**Time Complexity Analysis of above code:**

1. Array Sorting:

* Sorting the array using **sort(a, a + n)** takes O(n log n) time.

2. Binary Search:

* For each query (loop executed 'q' times), binary search is performed.
* Binary search has a time complexity of O(log n).
* Therefore, the overall time complexity for 'q' queries is O(q \* log n).

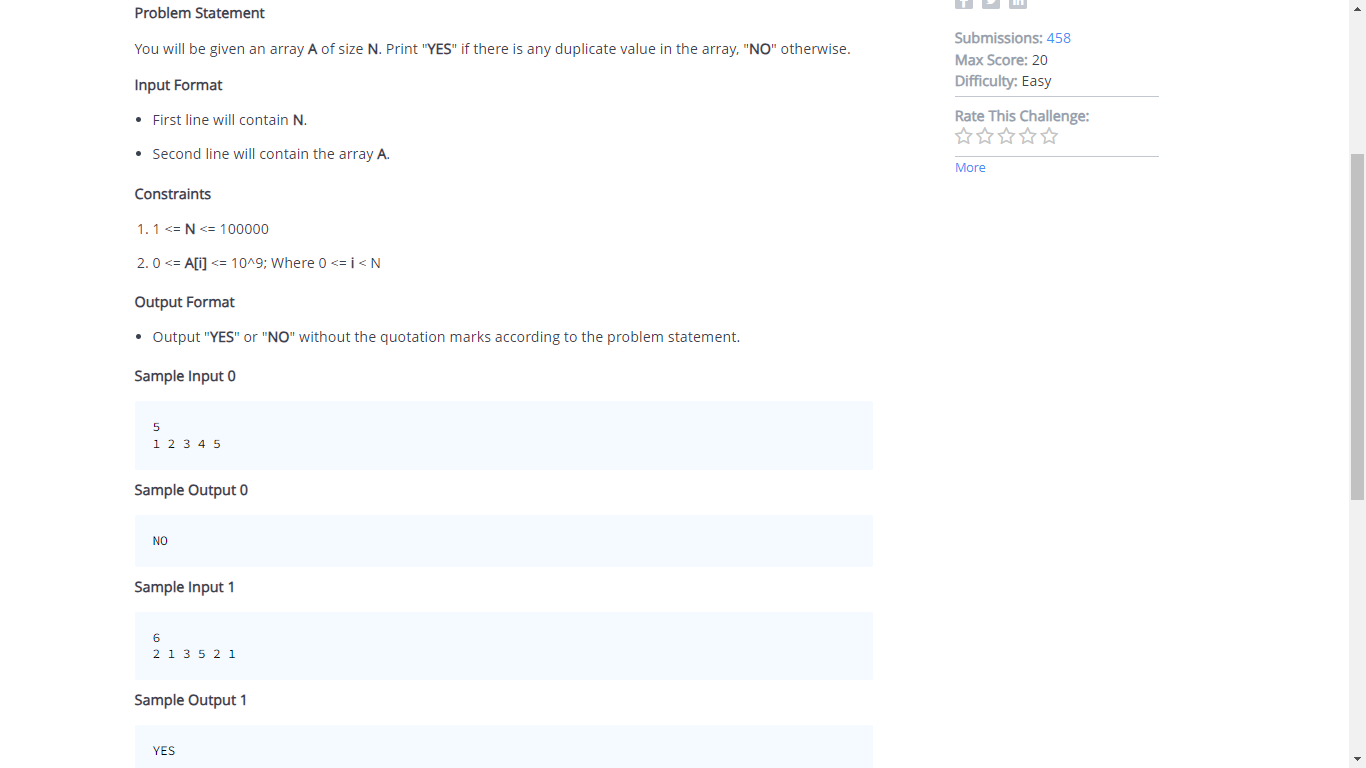
3. Overall Time Complexity:

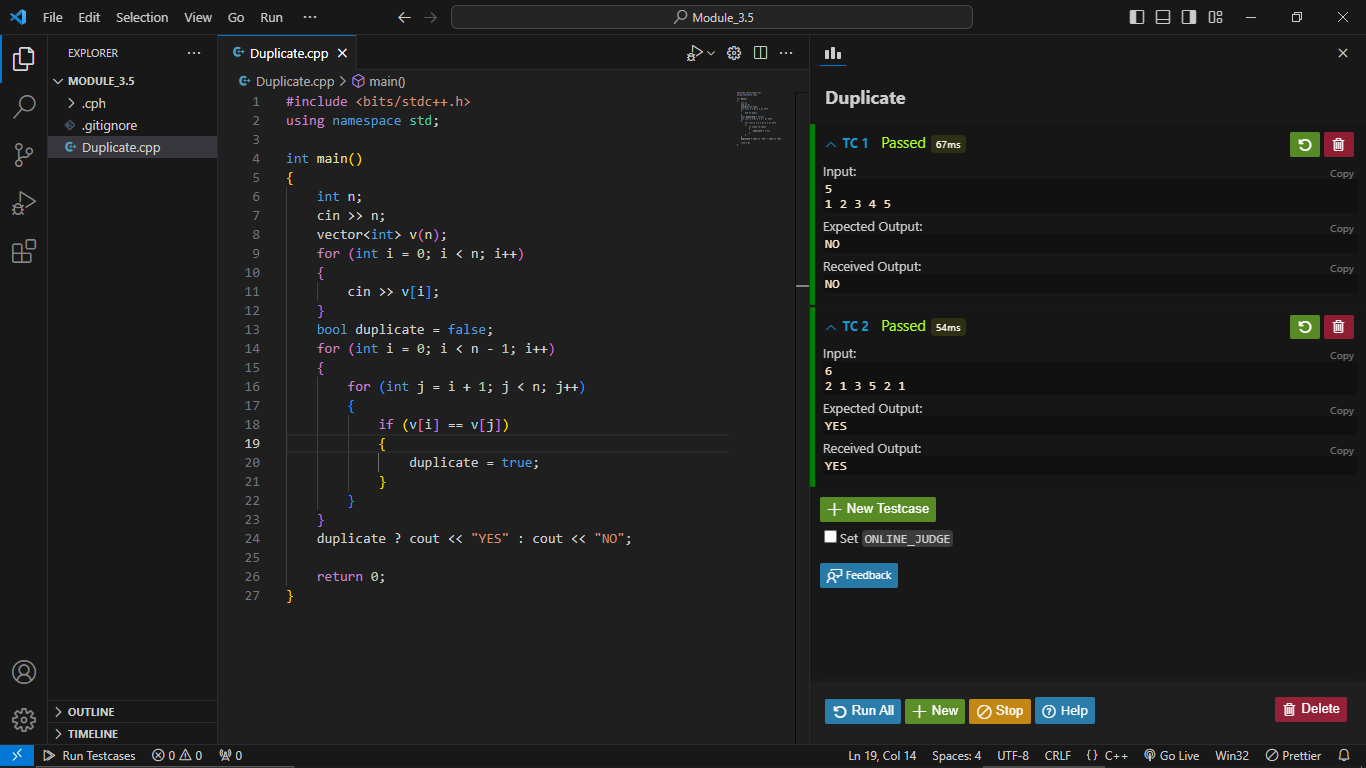
* The dominant factor is the binary search within the loop.
* The overall time complexity is O(n log n) (for sorting) + O(q log n) (for binary search within the loop).

In summary, the overall time complexity of the code is O(n log n + q log n). If 'q' is significantly smaller than 'n', this approach can be more efficient compared to the linear search in the previous example.

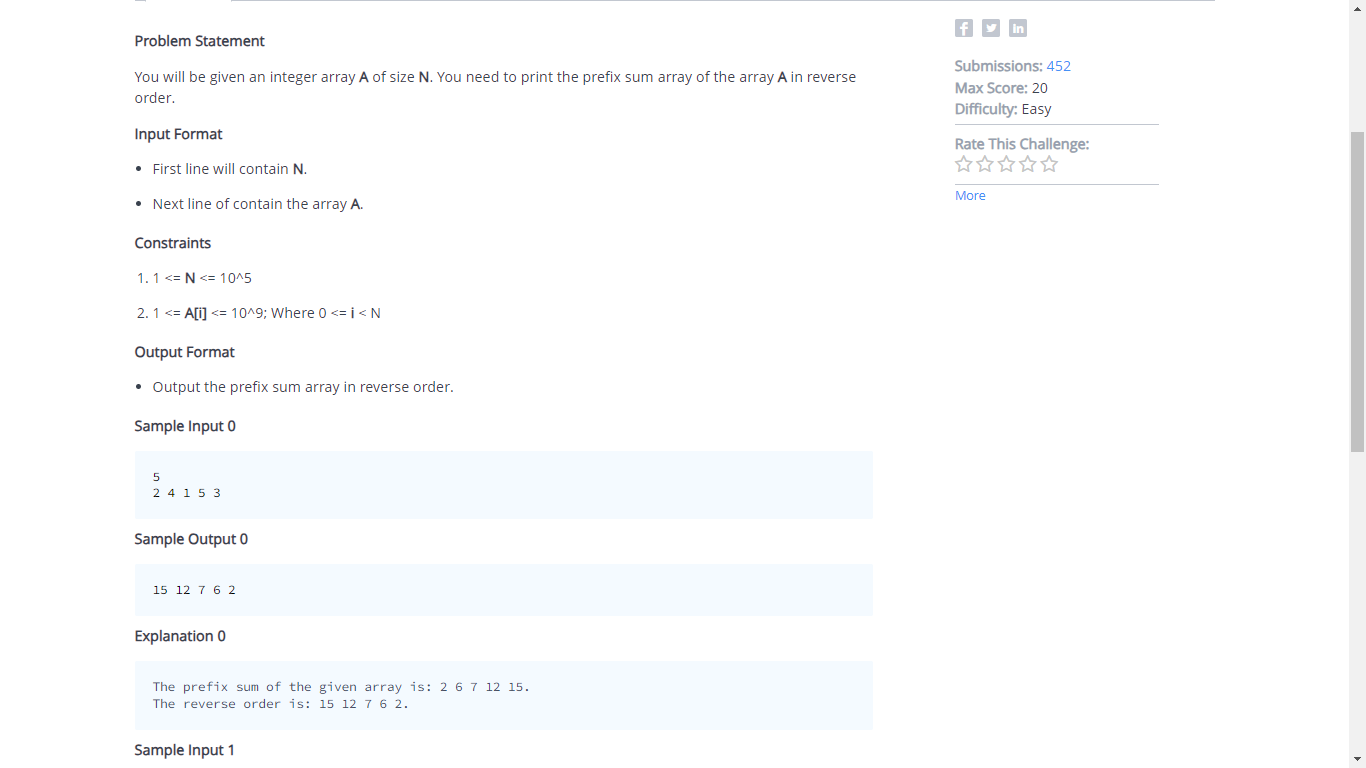
**Module 3.5**

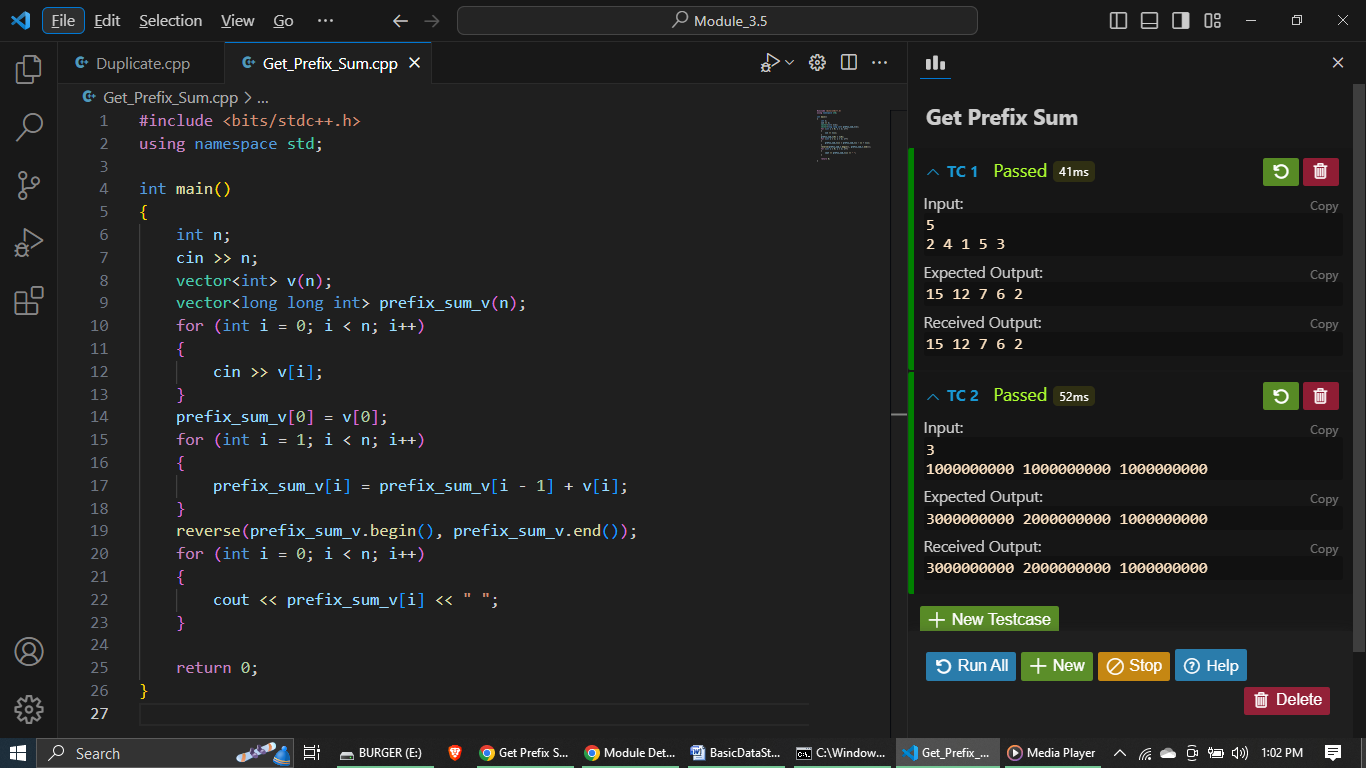
**Problem 1:**

****

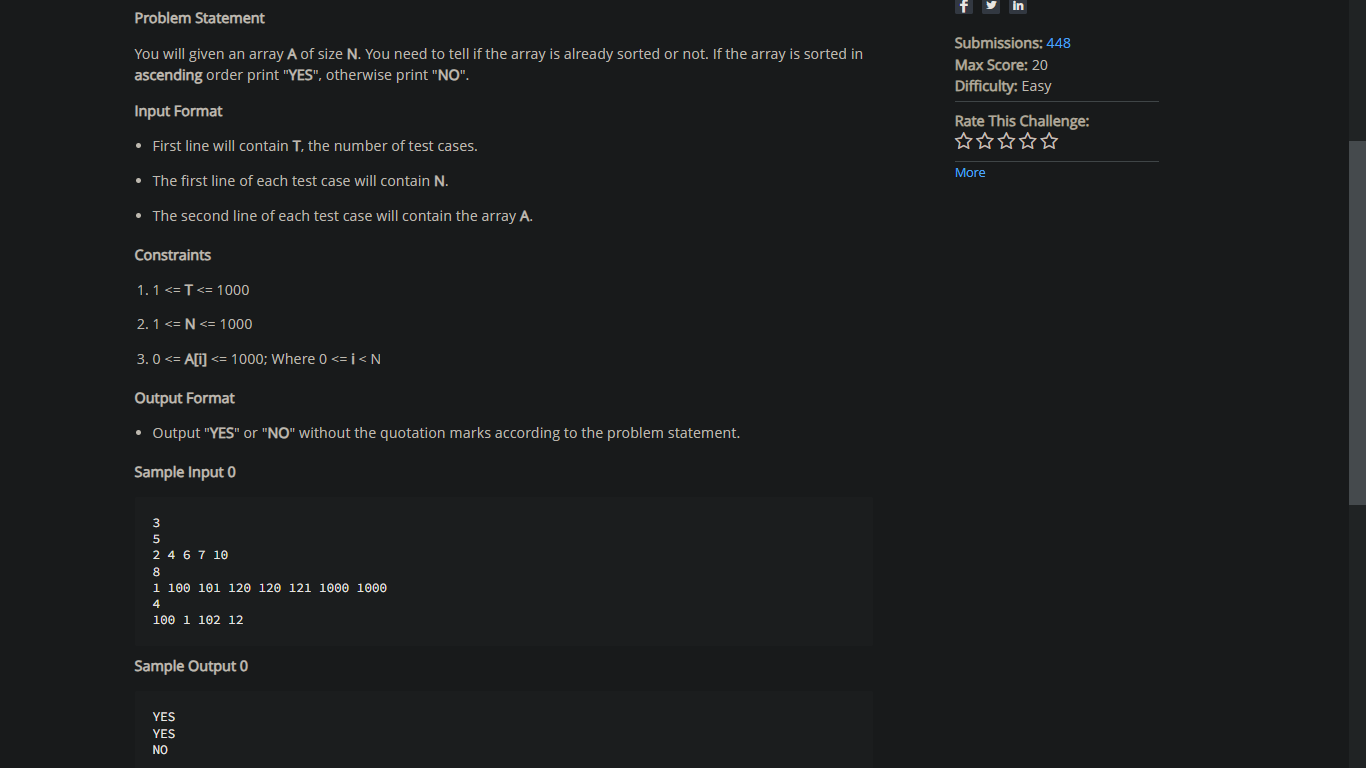
****

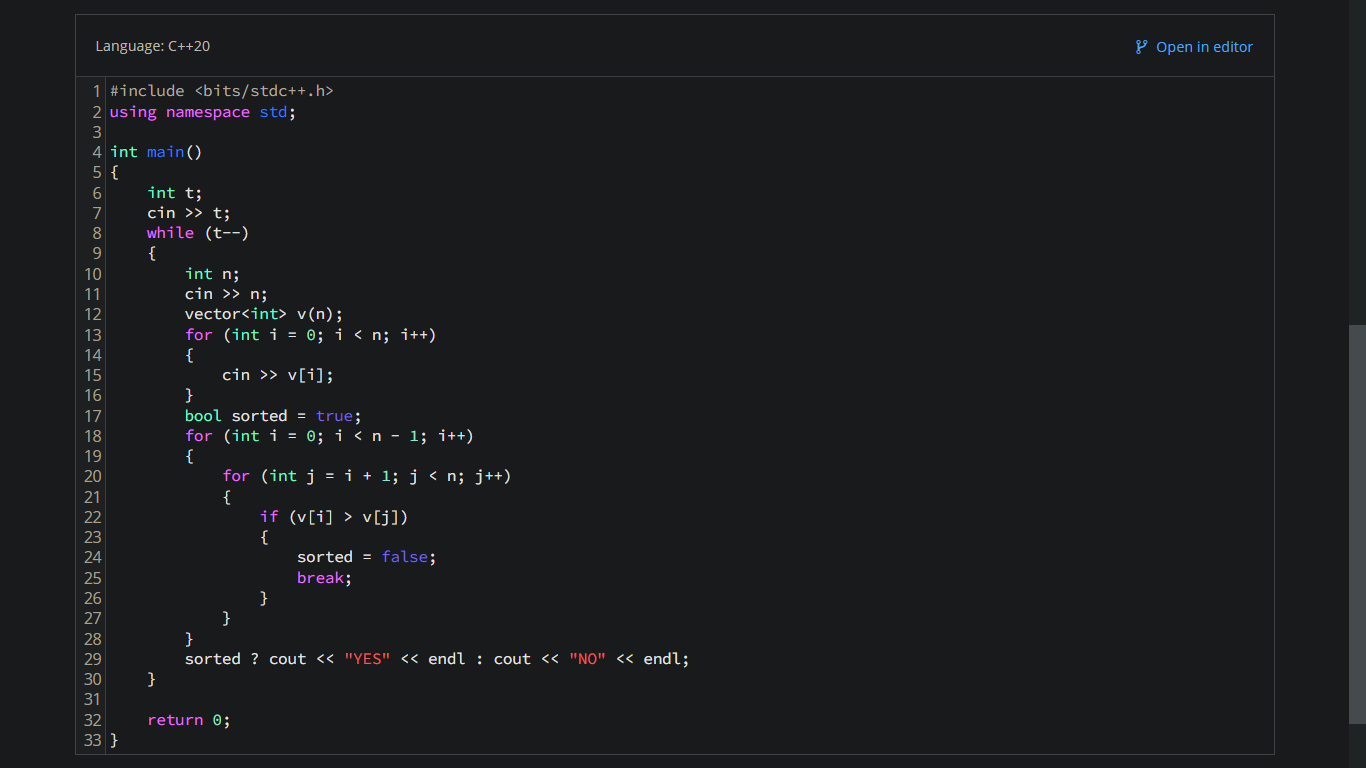
**Problem 2: Prefix Sum Array**

****

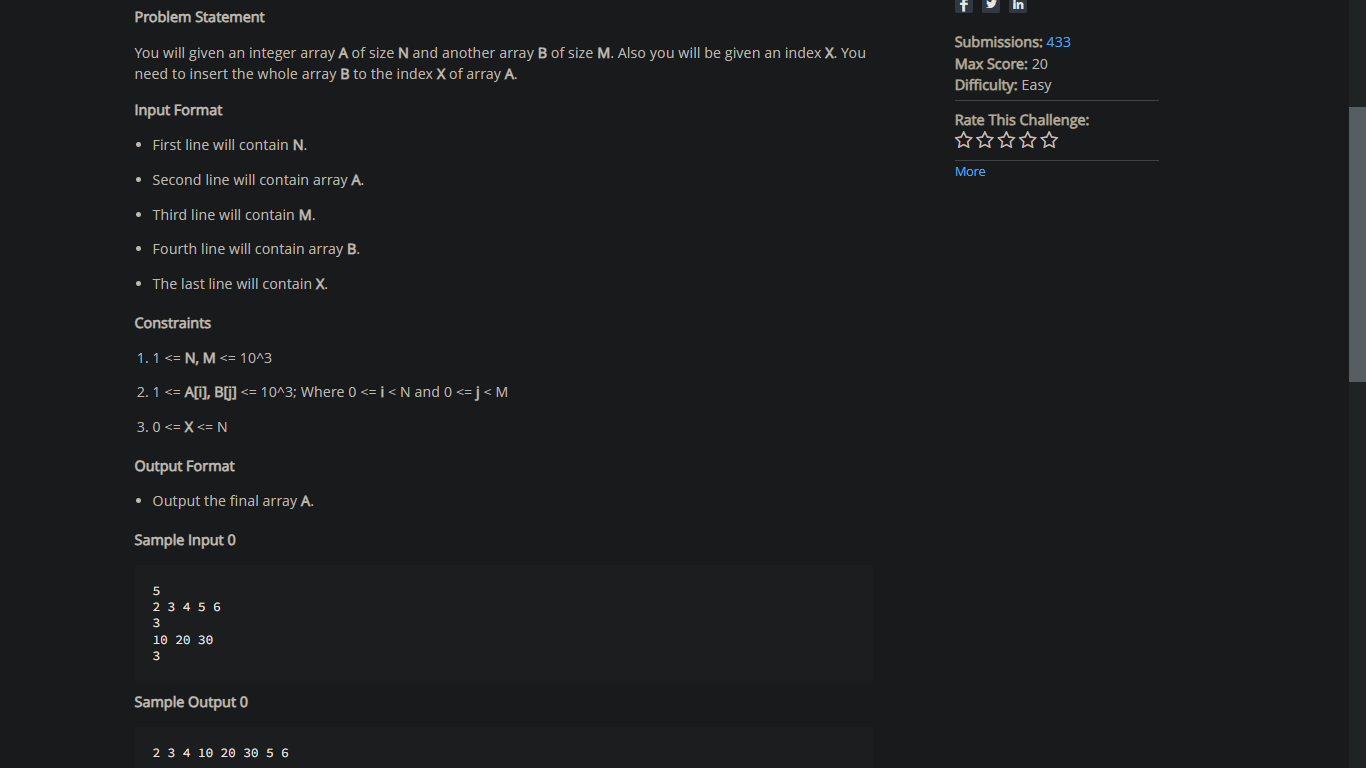
****

**Problem 3 :**

****

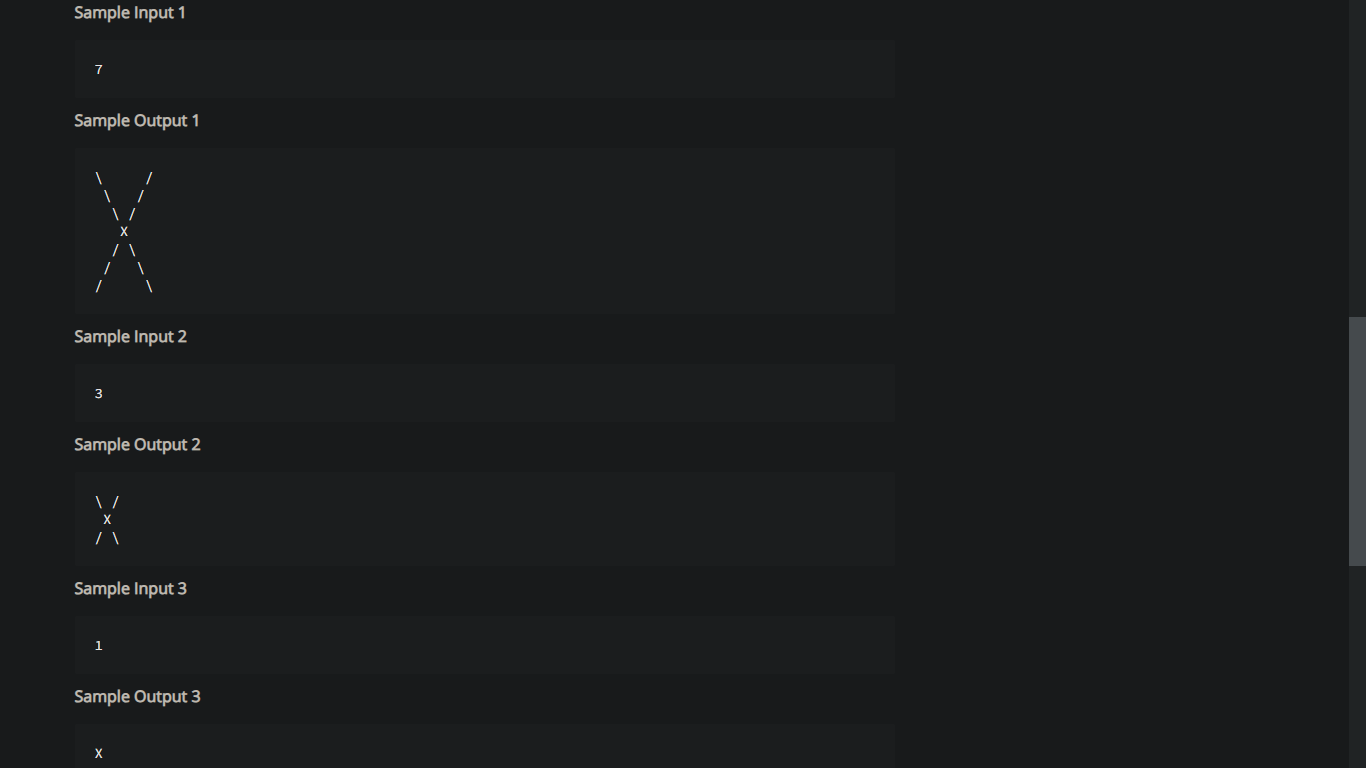
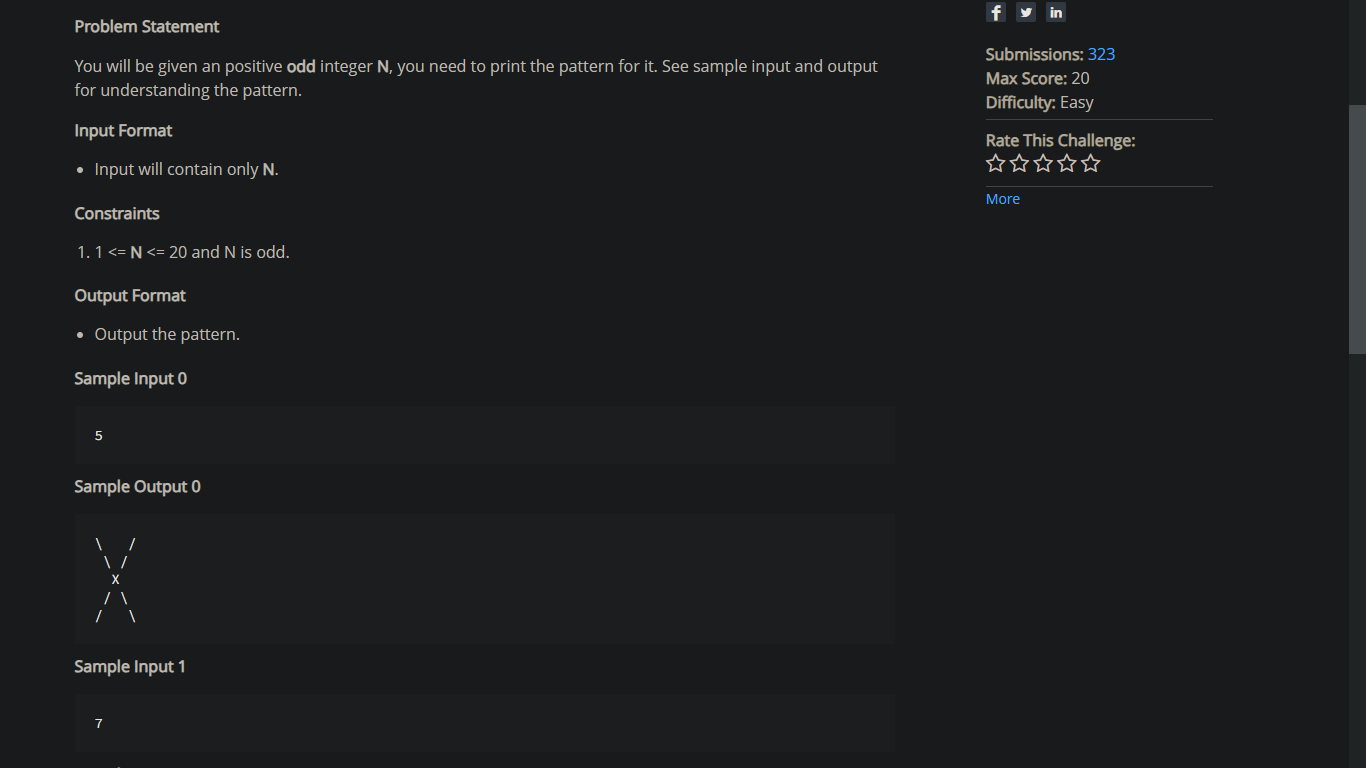
****

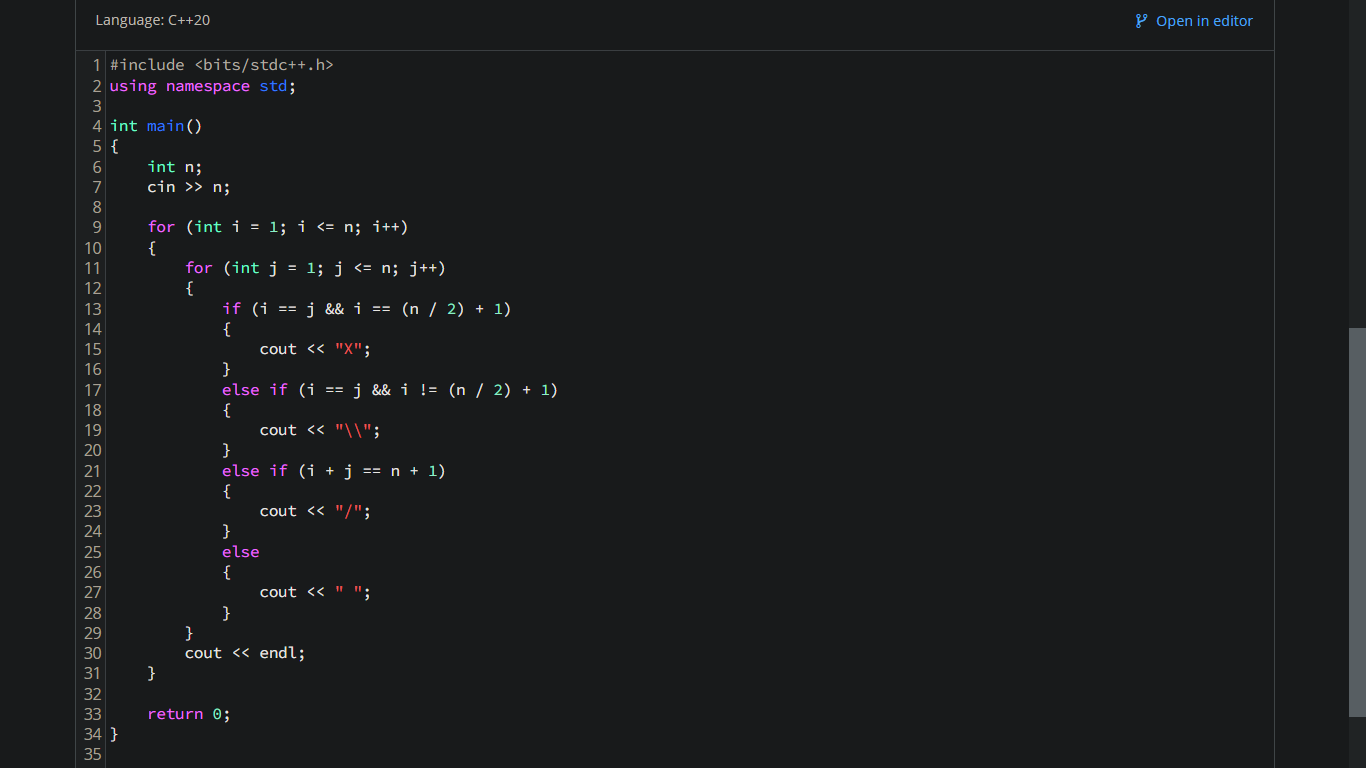
**Problem 4:**

****

****

**Problem 5:**

****

****