### **Table of Contents**

**Logic Building Problems and Flowchart……………………………1.**

1.1 **Easy Level**

**Recursion………………………………………………………………**

2.1 **BackTracking**

**\***Methods

\*When To use

2.2 **Problems**

**Sorting………………………………………………………………..**

3.1 **Merge Sort**

3.2 **Quick Sort**

**Arrays and Vectors………………………………………………….**

3.1 **Basics of Vectors in C++**

\*Dynamic Size and Memory Management

\*Initialization of Vectors

\*Accessing Elements via Indices

3.2 **Problems**

**Strings in C++.....................................................................................**

4.1 **Basics of Strings**

\*String Initialization and Operations

\*Common Functions in std::string

4.2 **Problems**

**Linked Lists………………………………………………………..**

5.1 **Basics of Linked Lists**

\*Singly Linked List

\*Doubly Linked List

\*Circular Linked List

5.2 **Problems**

**Stacks………………………………………………………………..**

6.1 **Basics of Stacks**

6.2**Problems**

**Queue………………………………………………………………..**

**Basics Of Queue**

**Problems**

**Binary Trees………………………………………………………..**

**Basics Of Trees**

**Problems**

HELLO Let start With the concept of DSA data science Algo-rithim

DSA (Data Structures and Algorithms) is the study of organizing data   
 efficiently using data structures like arrays, stacks, and trees, paired with step-by-step procedures (or algorithms) to solve problems effectively. Data structures manage how data is stored and accessed, while algorithms focus on

processing this data

**1 Logic Building Problems And Flowchart**

Logic building is about creating clear, step-by-step methods to solve problems

using simple rules and principles. It’s the heart of coding, enabling programs

to think, reason, and arrive at smart s just like we do.

Understand the problem: Read and understand the problem statement.

**• Generate Examples:** Create additional input and output cases for each

problem.

**• Draw observations**: Draw observations and patterns based on the

examples you created.

**• Start with Basic:** First, think of the basic way to solve the problem

and apply further conditions

Problems are of three level:

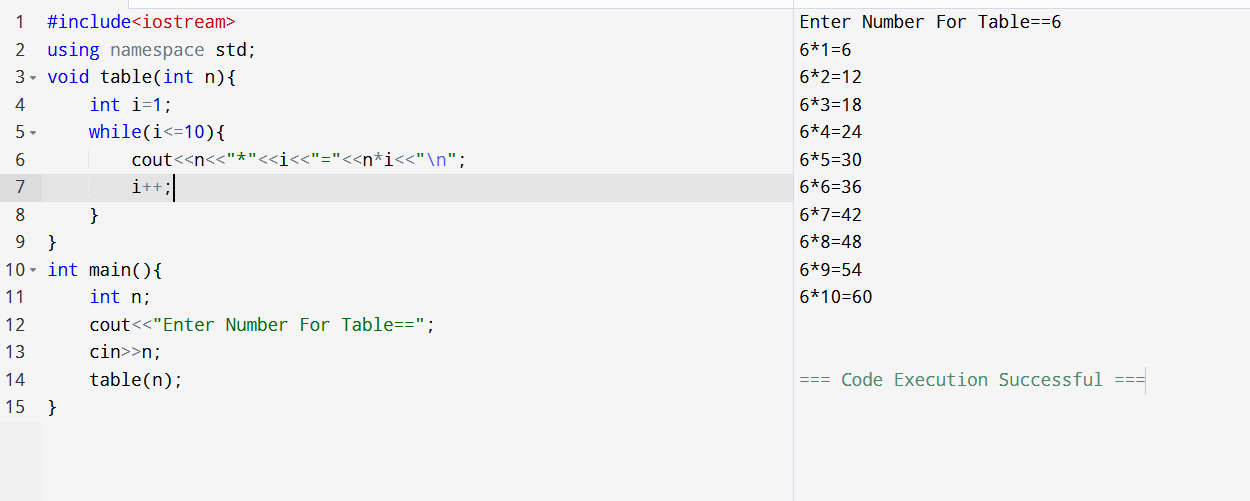
1. Easy.

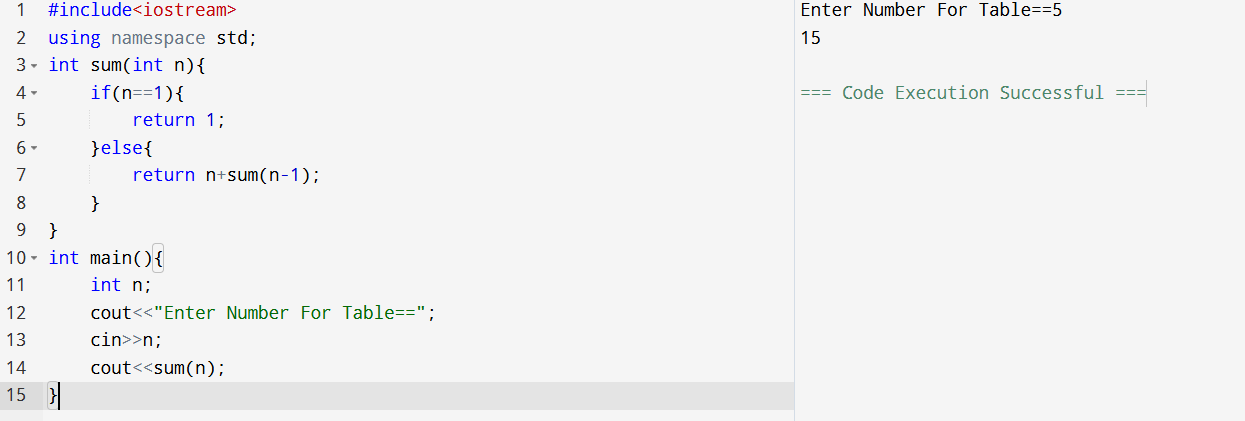
2. Medium.

3. Hard.

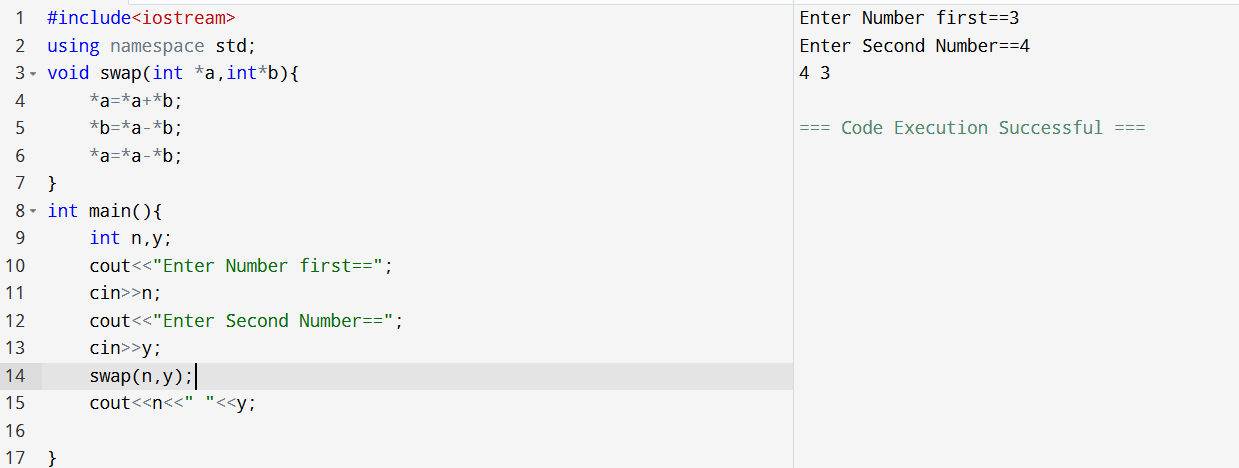
**1.0.1 Easy Level**

**QUESTION 2:Given a number n, we need to print its table.**

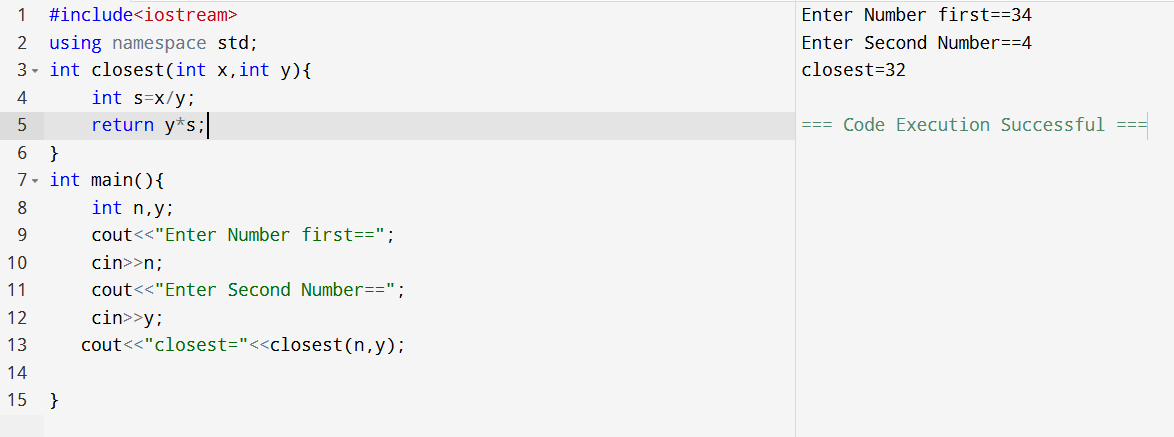
  **Question 3: SUM OF TWO NATURAL NUMBER**

****

**Question 4:Swap Two Numbers**

****

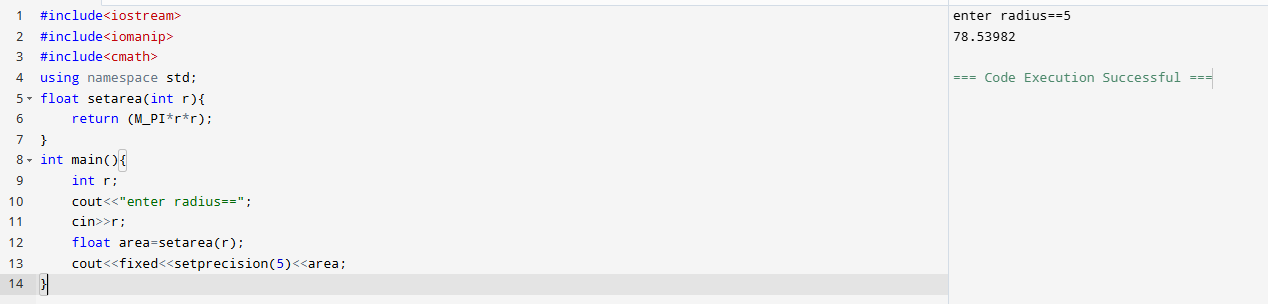
**Question 5:Closest number:**

****

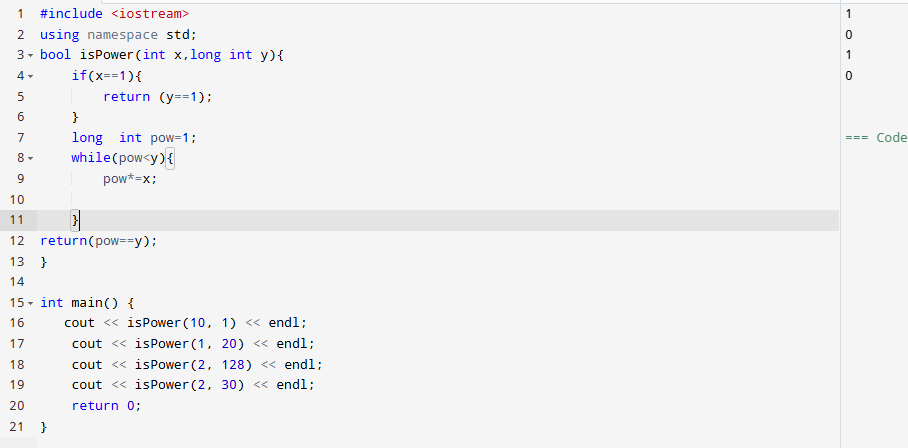
**Question 6: DICE OPPOSITE**

****

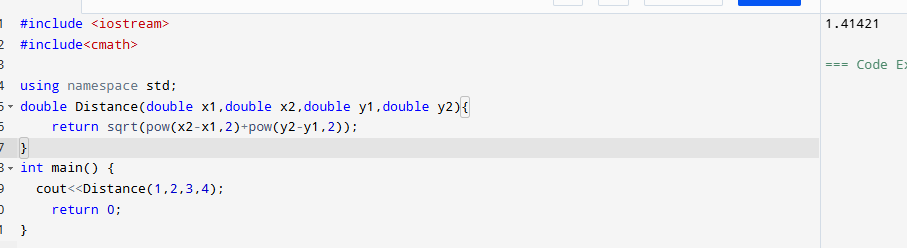
**Question 7:Finding Area Of Circle Upto 5 Digits**

****

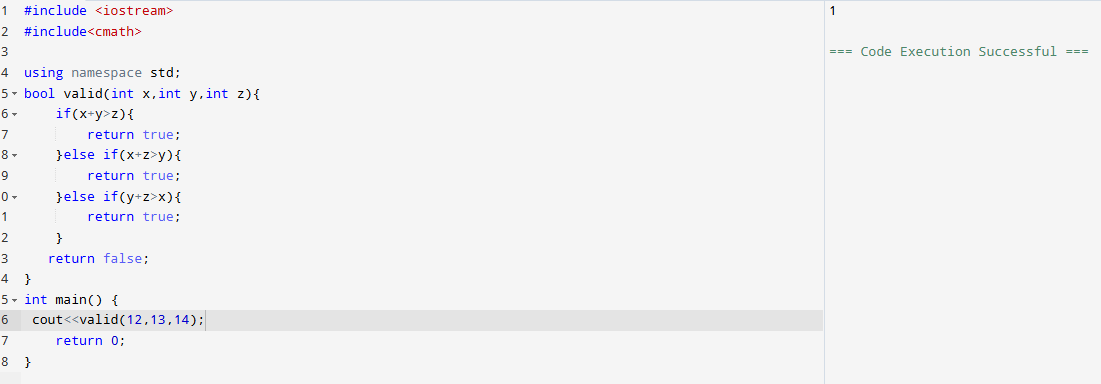
**Question 8: Check if a Number is A Power Of Another Number:**

****

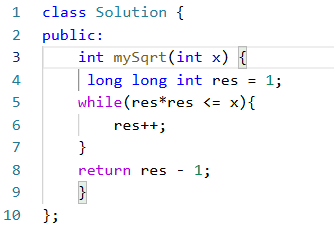
**Question 9: To calculate a Distance between two points**

****

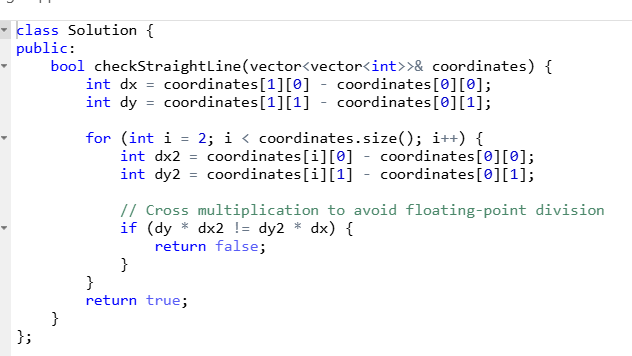
**Question 10 To check validity of A Triangle:**

****

**Question 11 To find sqrt without using any inbuilt function**

****

**Question Check if is a straight line:**

****

**RECURSION**

Recursion is a fundamental programming concept where a function calls itself in order to solve a problem. In the context of Data Structures and Algorithms (DSA), recursion is often used to solve problems that have a recursive structure or can be broken down into smaller subproblems of the same type.

In C++, recursion involves two key parts:

1. Base Case: This is the terminating condition where the recursion stops.
2. Recursive Case: This is the part where the function calls itself with a modified argument.

### **Key Concepts of Recursion**

1. Function Calling Itself: A function that calls itself directly or indirectly is known as a recursive function.
2. Base Case: Every recursive function must have a base case to stop the recursion and avoid infinite loops.
3. Recursive Case: The function should reduce the problem into a smaller version of the same problem.

**BACKTRACKING**

**Backtracking** is a general algorithmic technique used for finding all (or some) s to problems, particularly those that involve searching through a space. It is a trial-and-error approach where the algorithm builds the incrementally and abandons ("backtracks") a as soon as it determines that it cannot be completed.

The primary idea is to explore all possibilities by choosing an option, recursively solving the problem, and undoing the choice (backtracking) if it leads to an invalid or undesirable . This process continues until the is found or all possibilities are explored.

### **Key Characteristics of Backtracking:**

1. **Incremental construction:** The algorithm builds the step by step, making decisions at each step.
2. **Feasibility check:** If the current is invalid, the algorithm backtracks to try other possibilities.
3. **Explores all possibilities:** Backtracking can be used to generate all possible s to a problem (e.g., generating all permutations).

### **When to Use Backtracking:**

* **Searching for s in a space.**
* **Decision-making problems** where we need to consider multiple choices and find a correct configuration or sequence.
* **Combinatorial problems**, like generating permutations, combinations, and solving puzzles.

### **General Steps for Backtracking:**

1. **Choose:** Make a decision (choose an option).
2. **Explore:** Recursively attempt to solve the problem with the current decision.
3. **Backtrack:** If the current path doesn't lead to a valid , undo the last decision (backtrack) and try the next possibility.

**Question 1:**

### **1. Merge Sort**

#### **Overview:**

* **Merge Sort** is a divide-and-conquer algorithm that divides the unsorted list into smaller sublists, sorts those sublists, and then merges the sublists to produce the final sorted list.
* **Stable Sort**: Yes, Merge Sort is stable, which means that it preserves the relative order of equal elements.

#### **Steps of Merge Sort:**

1. Divide: Split the unsorted list into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the two halves back together in sorted order.

#### **Time Complexity:**

* Best, Average, and Worst Case: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements in the array.
* Space Complexity: O(n)O(n)O(n), because it requires extra space for merging the arrays.

**2. Quick Sort**

#### **Overview:**

* Quick Sort is another divide-and-conquer algorithm, but it works differently than Merge Sort. It selects a pivot element and partitions the array such that elements less than the pivot come before it, and elements greater than the pivot come after it.
* Unstable Sort: Quick Sort is not stable, meaning it does not necessarily preserve the relative order of equal elements.

#### **Steps of Quick Sort:**

1. Partition: Pick an element as a pivot and partition the array into two parts:
   * Elements smaller than the pivot are placed before it.
   * Elements greater than the pivot are placed after it.
2. Recurse: Recursively apply the same process to the subarrays before and after the pivot.

#### **Time Complexity:**

* Best and Average Case: O(nlog⁡n)O(n \log n)O(nlogn), when the pivot divides the array into nearly equal parts.
* Worst Case: O(n2)O(n^2)O(n2), when the pivot selection is poor (e.g., the array is already sorted).
* Space Complexity: O(log⁡n)O(\log n)O(logn) on average, due to the recursion stack (in place sorting, no extra space is required).

**ARRAY AND VECTOR**

### **Basics of Vectors in C++**

A vector in C++ is a dynamic array that can grow and shrink in size. Vectors are part of the C++ Standard Library, and they are defined in the vector header file (#include <vector>). Unlike traditional arrays, vectors can store elements dynamically, meaning that you don't need to know the size in advance.

### **Key Features of Vectors:**

1. **Dynamic Size:** Vectors can grow and shrink as needed, unlike arrays whose size is fixed after declaration.
2. **Random Access**: Vectors allow access to elements via indices, just like arrays.
3. **Memory Management:** Vectors handle memory allocation and deallocation automatically.
4. **Ease of Use:** Vectors provide member functions to manipulate elements (e.g., add, remove, access, resize, etc.).

**INITIALISATION**

#include <vector>

#include <iostream>

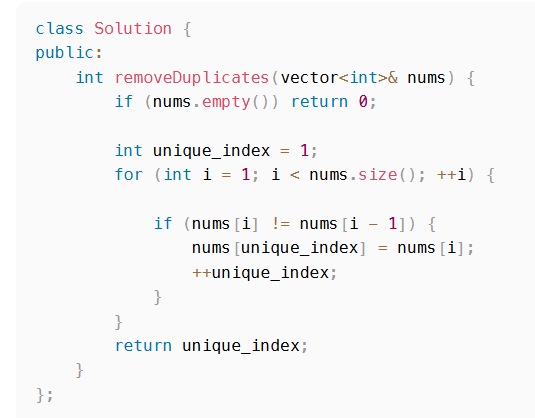
Using namespace std;

int main() {

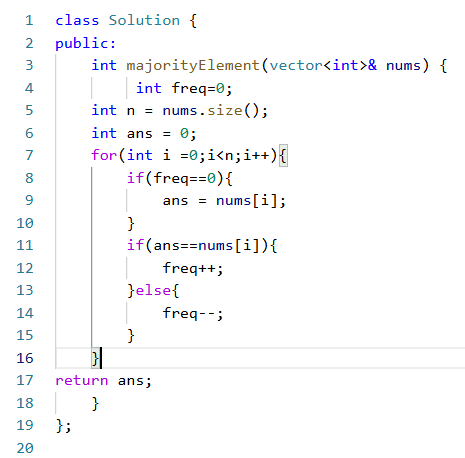
vector<int> vec(5, 10); // Vector with 5 elements, all initialized to 10

}

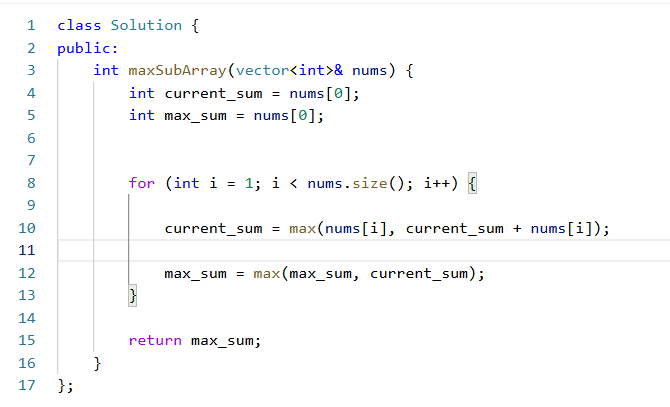
**Question 1: Remove Duplicates In An Array**

****

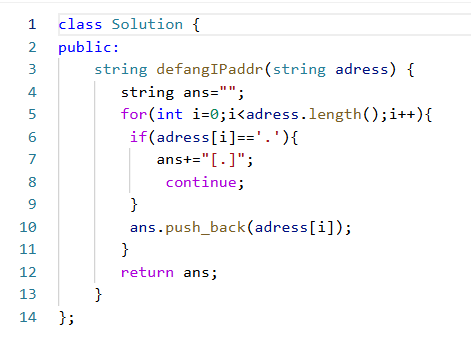
**Question 2 Majority Element**

****

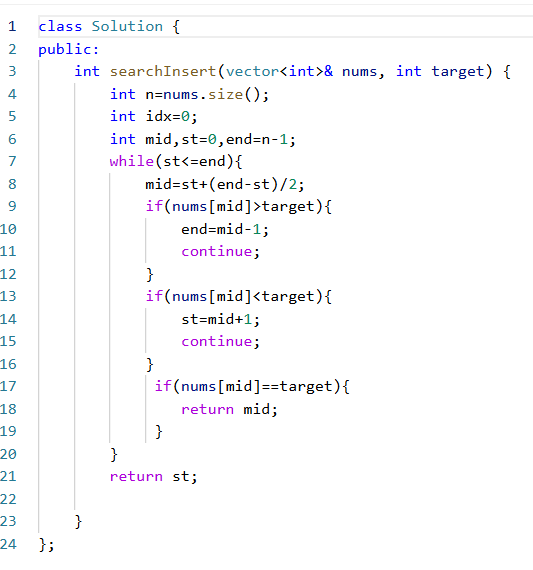
**Question3 Return Max Sum Of subarray**

****

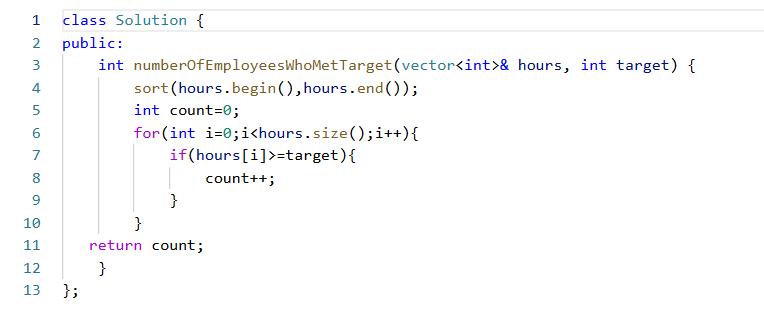
**Question 4 Defanging An IP address**

****

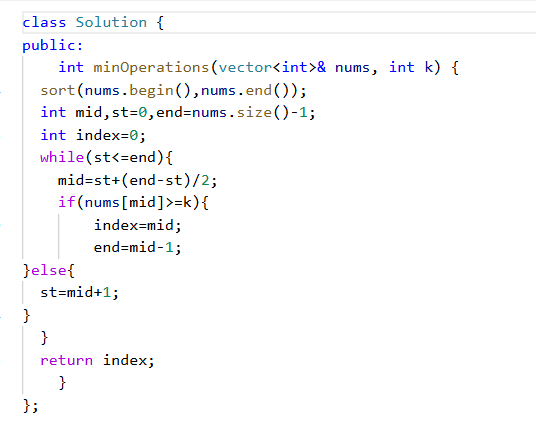
**Question 5: Insert Place**

****

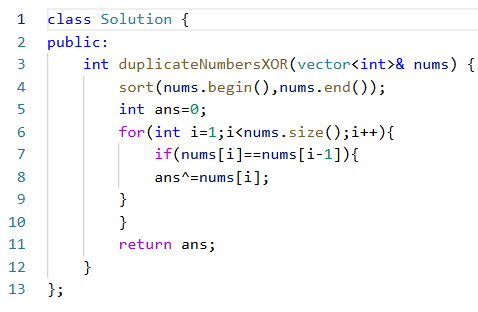
**Question 6: Employee Who met The target:**

****

**Question 7:**[**Minimum Operations to Exceed Threshold Value I**](https://leetcode.com/problems/minimum-operations-to-exceed-threshold-value-i/)

****

**Question 8:Find The XOR Of Number That Appear Twice**

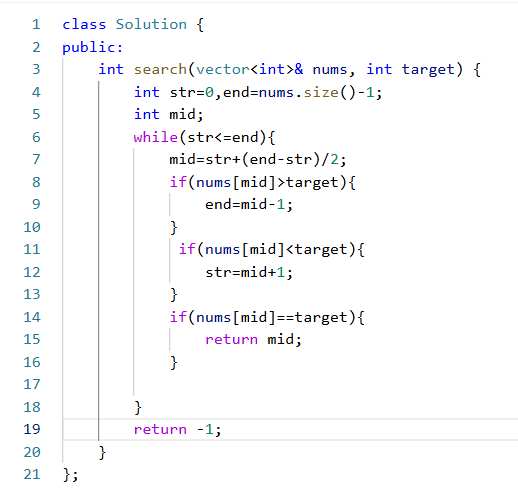
****

**Question 9:Find First And Last Position Of Target:**

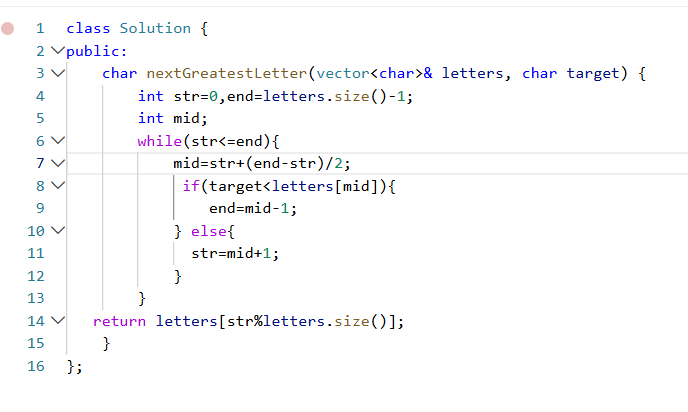
****

**Question 10: Container With Most Water:**

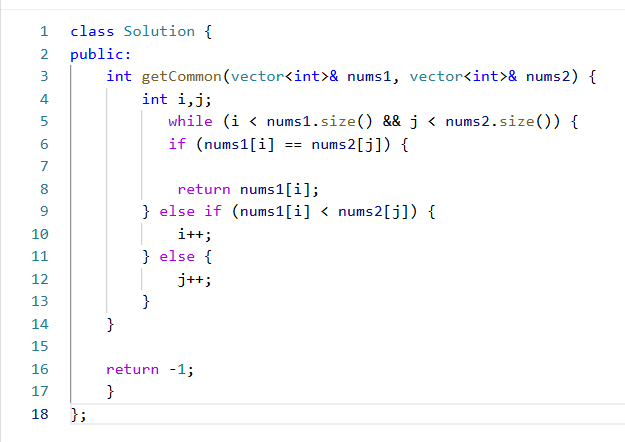
****

**Question 11 Binary Search:**

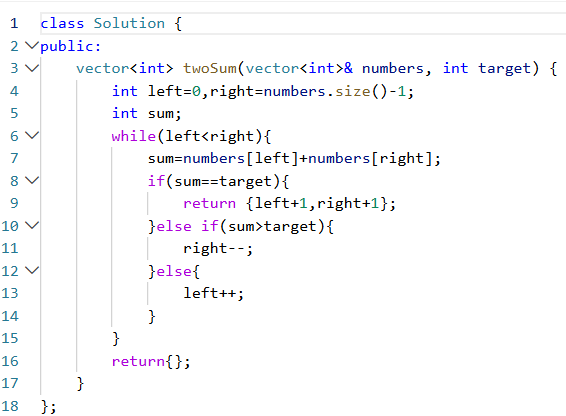
**Question12 Find smallest letter greater or same as Targeted**

****

**Question 13 Minimum Common Value**

****

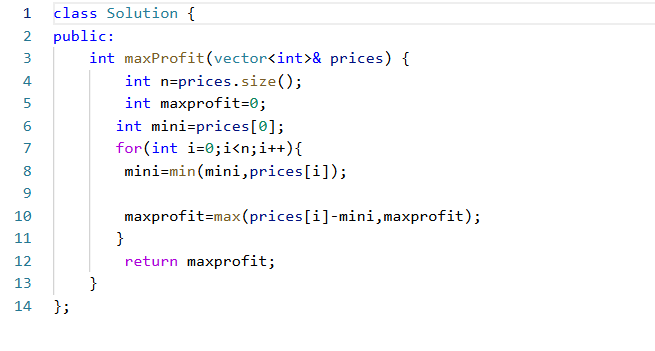
**Question 14** [**Two Sum II - Input Array Is Sorted**](https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/)

****

**Question 15: Check If N an Its Double Exist**

****

**Question 16:Best Time To Buy and Sell Stock:**

****

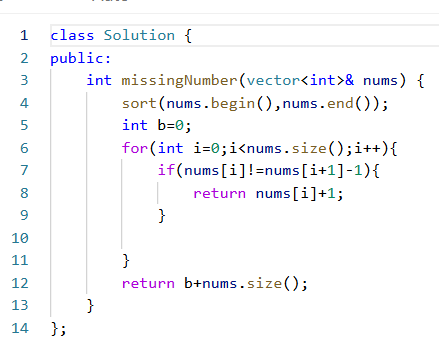
**Question 17 Next Permutations:**

****

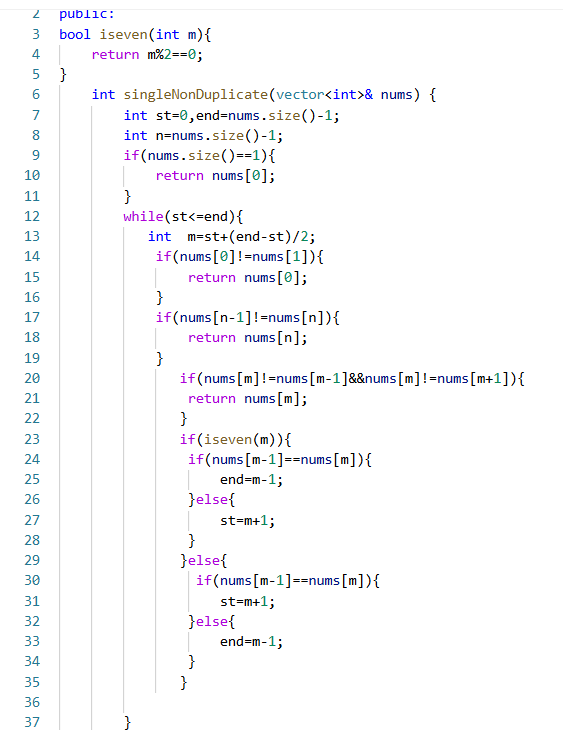
**Question 18 Distribute Candles:**

****

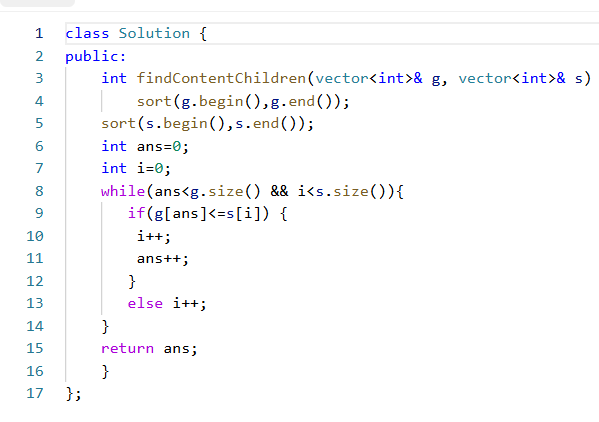
**Question 19 Missing Numbers**

****

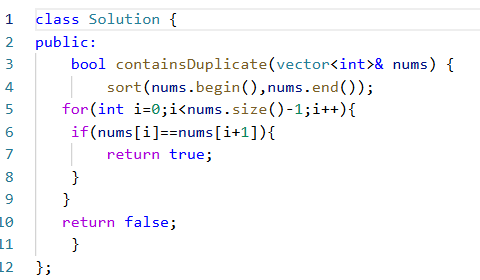
**Question 20 Single Element in A sorted array**

****

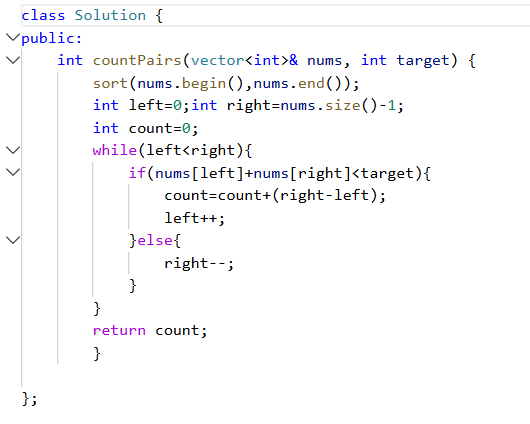
**Question 21 Assign Cookies**

****

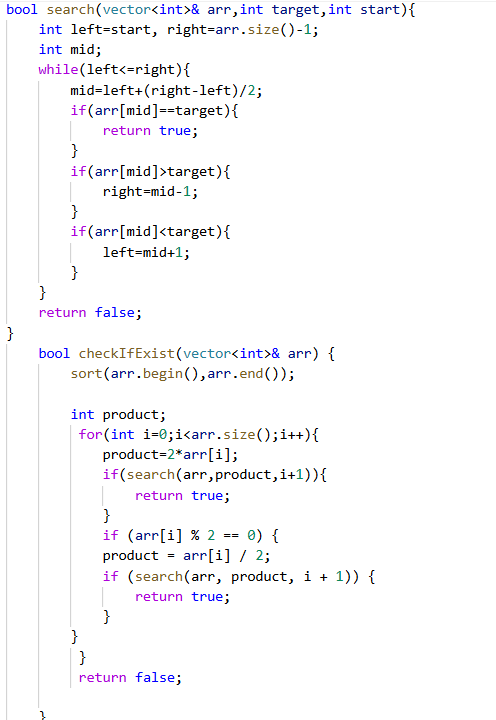
**Question 22 Contain Duplicates**

****

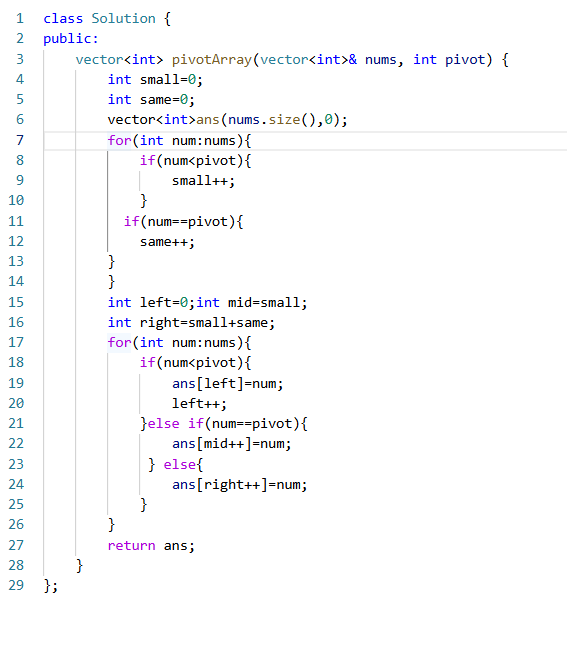
**Question 23 Count Pair sum Whose Target Sum is less than Target:**

****

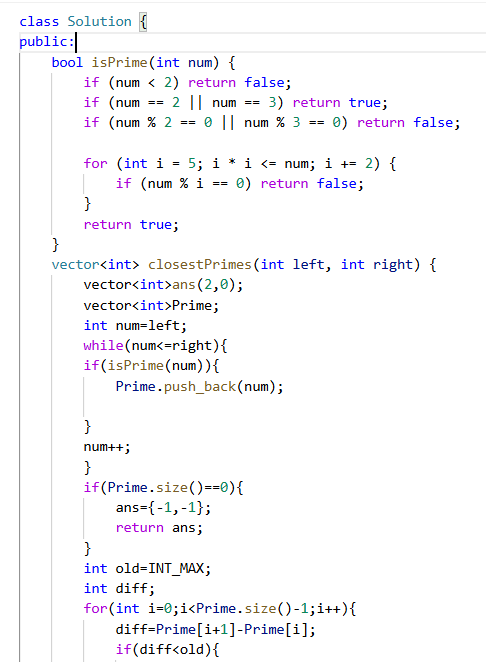
**Question 24 Check If the Number And its Double Exists**

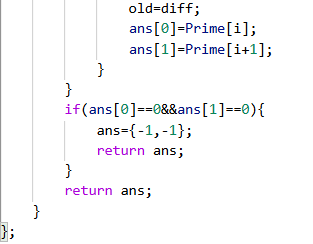
****

**Question 25** [**Partition Array According to Given Pivot**](https://leetcode.com/problems/partition-array-according-to-given-pivot/)

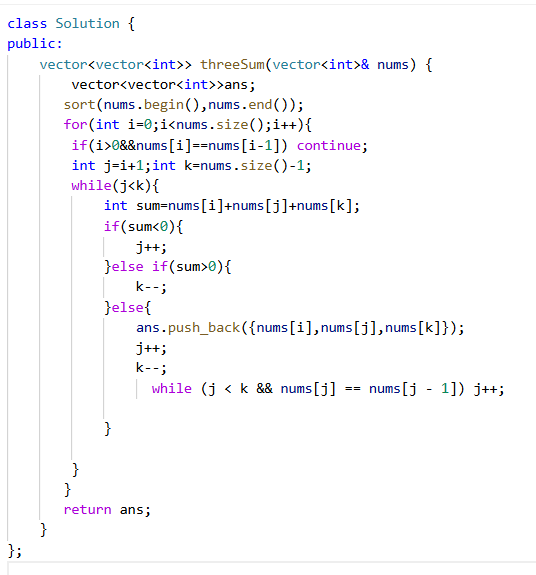
****

**Question 26** [**Closest Prime Numbers in Range**](https://leetcode.com/problems/closest-prime-numbers-in-range/)**\**

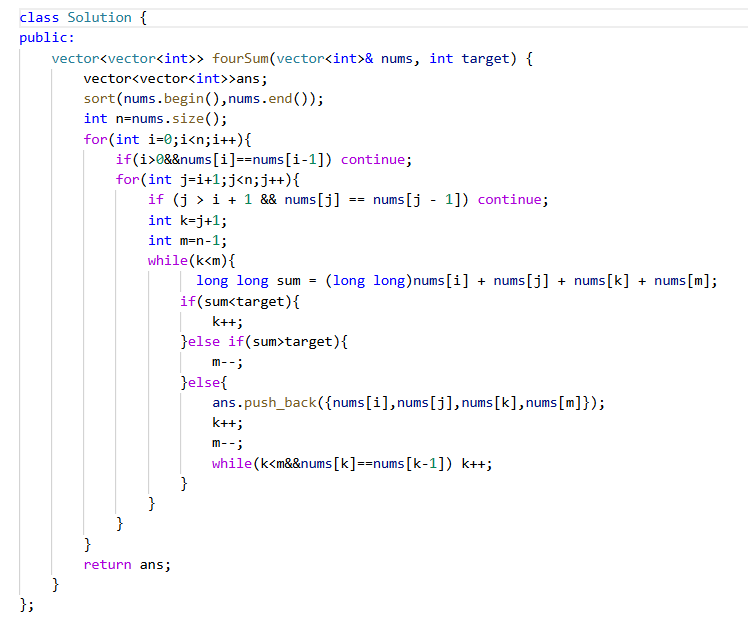
****

****

**Question 27 Three Sum:**

****

**Question 28 Four Sum:**

****

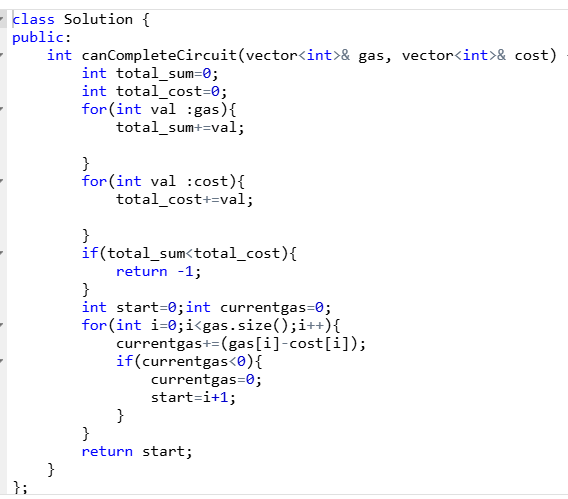
**Question 29 Intersection of Two array:**

****

**Question 30 Valid Perfect Square:**

****

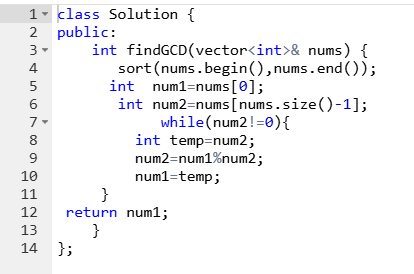
**Question 31 Gas Station using Greedy Approach:**

****

**Question 32 Intersection of Two Array -2**

****

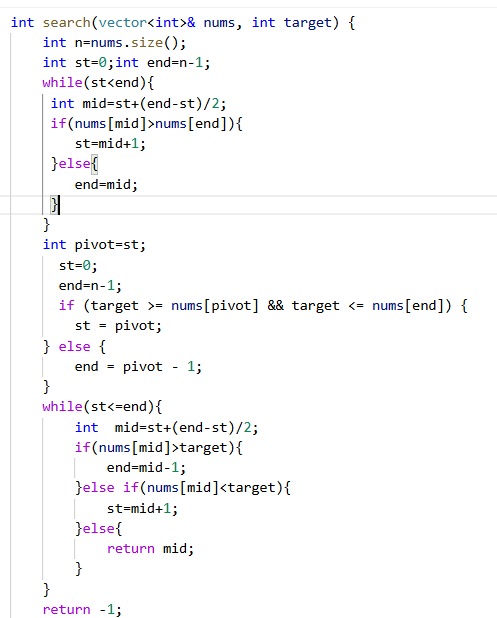
**Question 33 Find Greatest COmmon Divisior in array:**

****

**Question 34 Median of two merged Sorted array:**

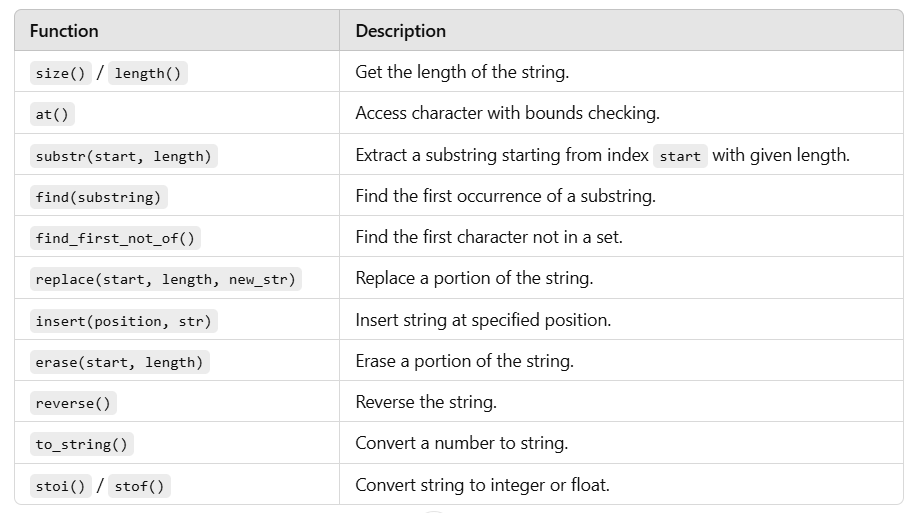
****

**Question Search In a Rotated Array**

****

**Strings**

In Data Structures and Algorithms (DSA), strings are a fundamental data type. They are sequences of characters, and many algorithms and data structures use strings as inputs or store them. In C++, strings are represented using the std::string class from the Standard Library, which provides many built-in functions for manipulation.



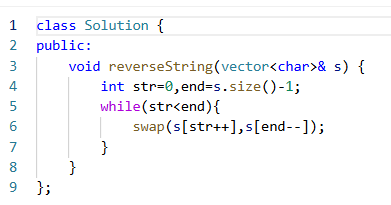
Strings in C++ are extremely versatile and efficient. Understanding string operations and common algorithms is critical for solving a wide range of problems in DSA. The std::string class in C++ provides various methods to manipulate, compare, and process strings efficiently, with most operations running in linear time.

**Question 1: Valid Palindrome**

****

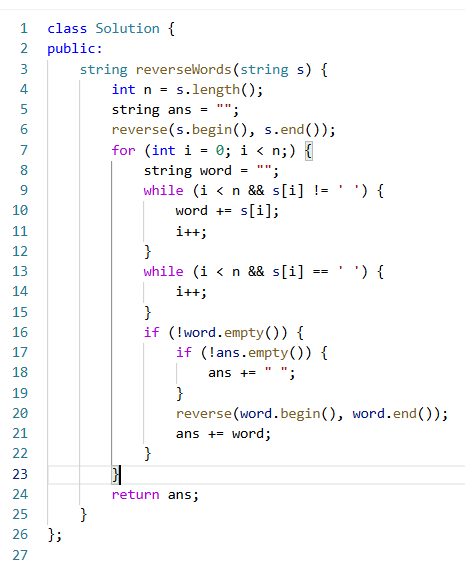
**\**

**Question 2 Reverse string**

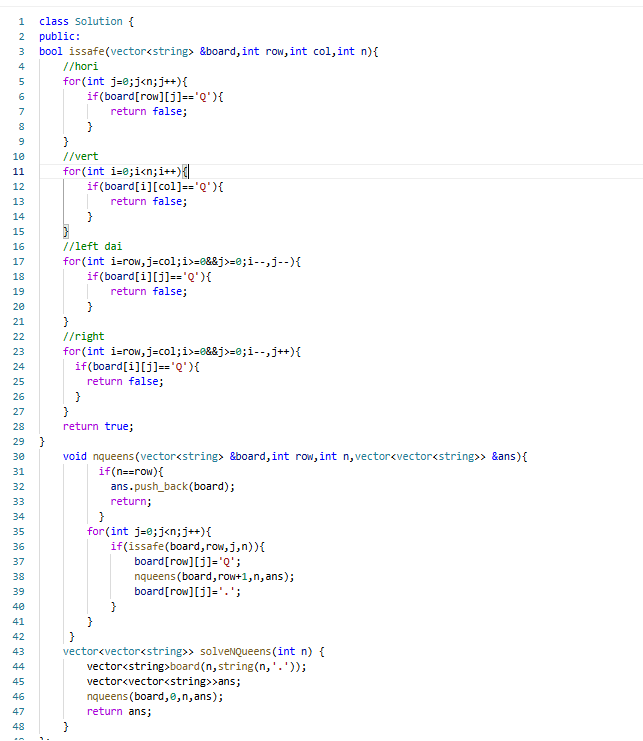
****

**Question Reverse Words In A string**

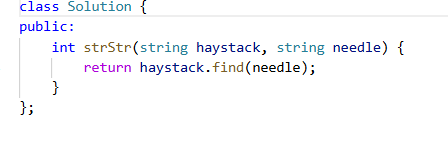
**:**

****

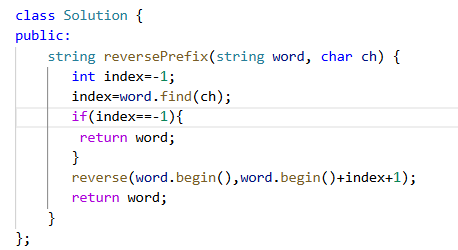
**Question N Queens:**

****

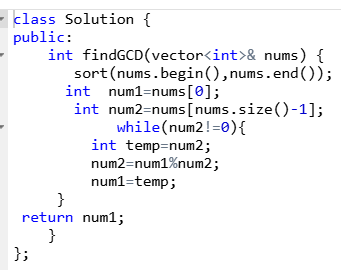
**Question 5 Find the index of the First Occurence Of string:**

****

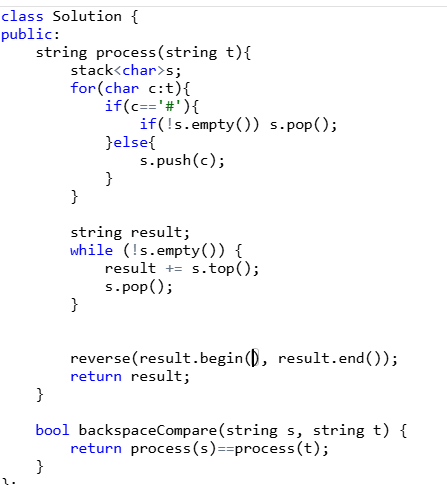
**Question 6 Reverse Prefix of a word:**

****

**Question length of last word**

****

**Question Backspace String Compare**

****

**LINKED LIST**

**BASICS**

A **linked list** in C++ is a data structure used for organizing items in a sequence. Unlike an array, the elements in a linked list are not stored in contiguous memory locations. Instead, each element (called a **node**) contains:

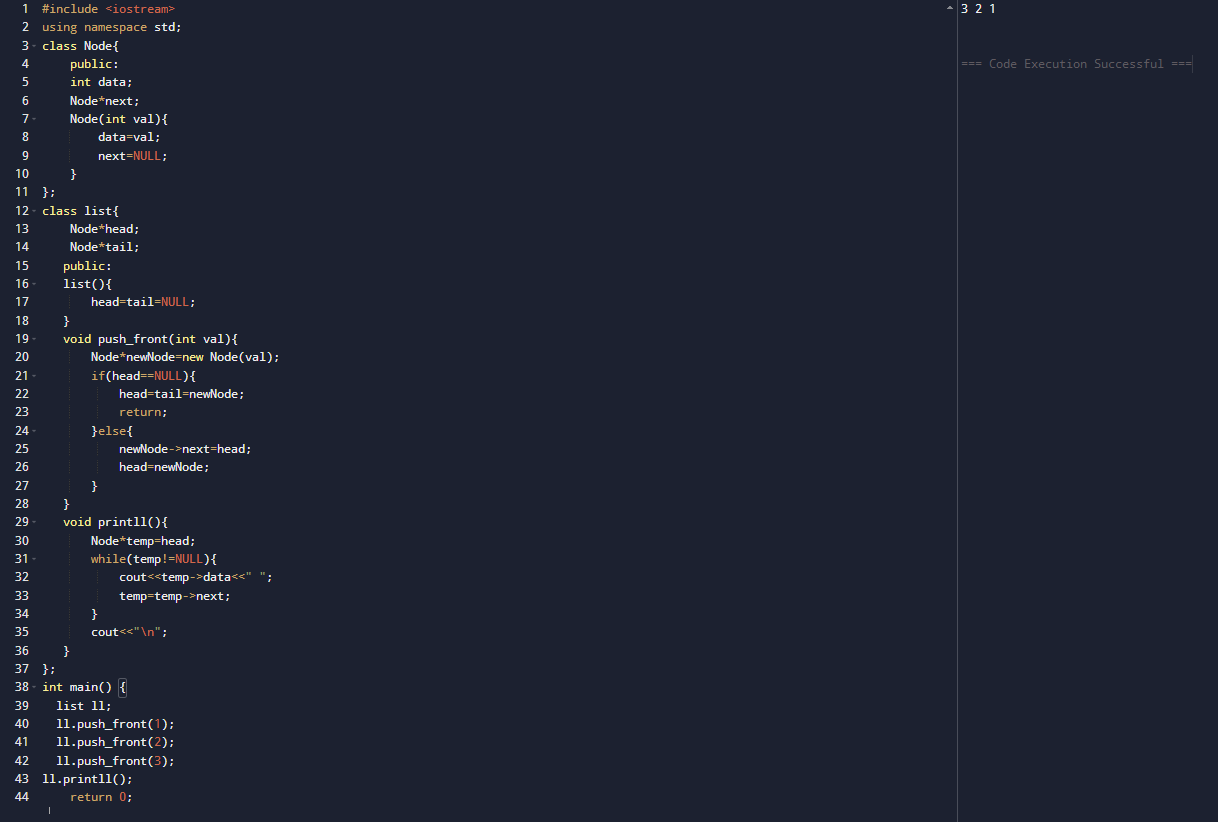
1.**Data**: The actual value or content stored in the node.

**2.Pointer**: A reference (or address) to the next node in the sequence.

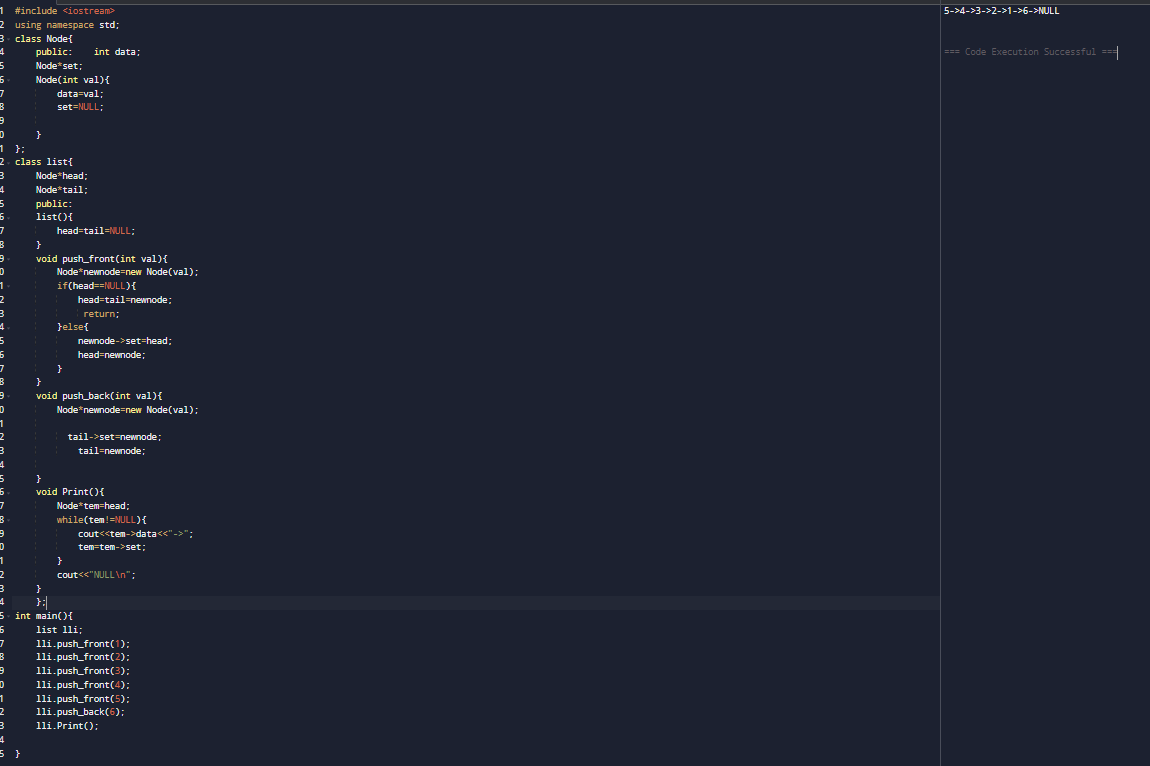
### **Types of Linked Lists:**

1. **Singly Linked List**: Each node has a pointer to the next node in the list. The last node has a nullptr (null pointer) indicating the end of the list.
2. **Doubly Linked List**: Each node has two pointers: one to the next node and one to the previous node.
3. **Circular Linked List**: The last node points back to the first node, forming a circular structure

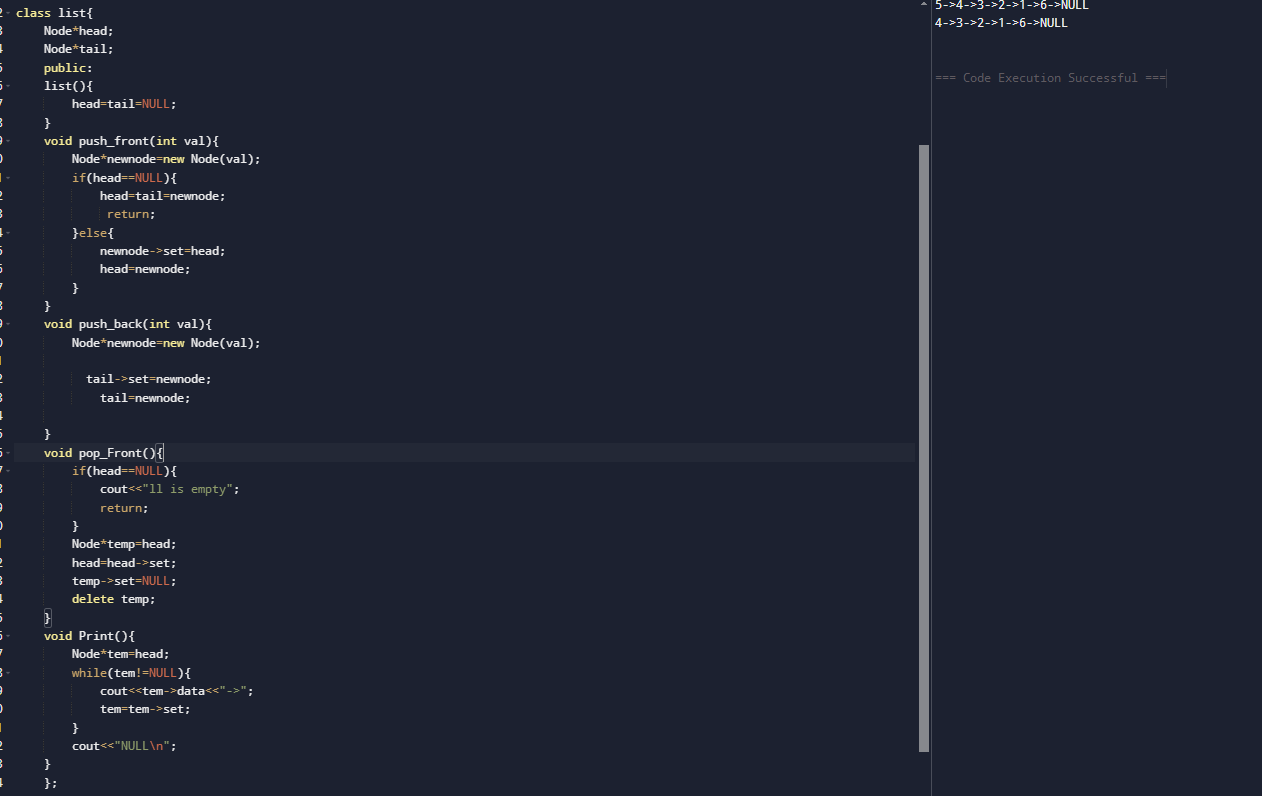
**Question Making of A linked List:**

****

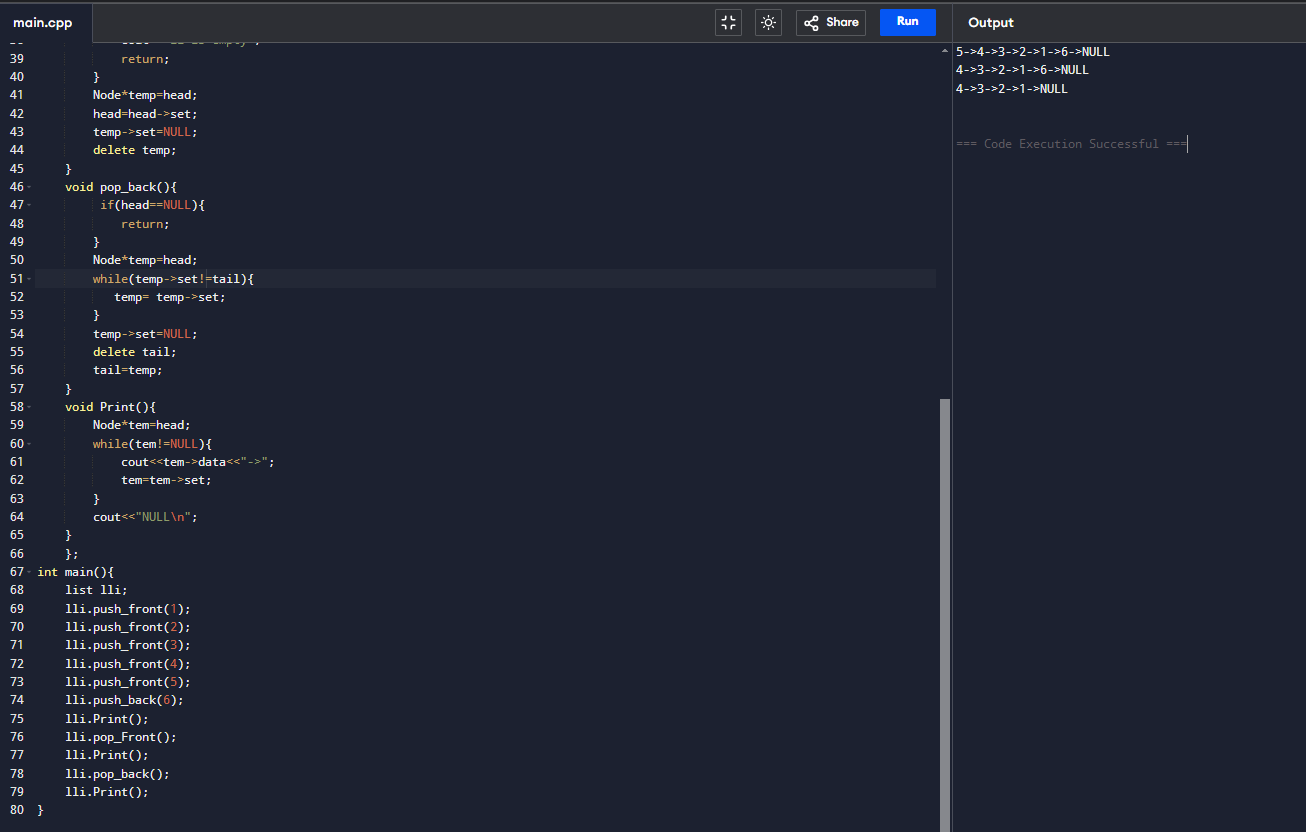
**USING PUSH BACK:**

****

**POP FRONT0(1)**

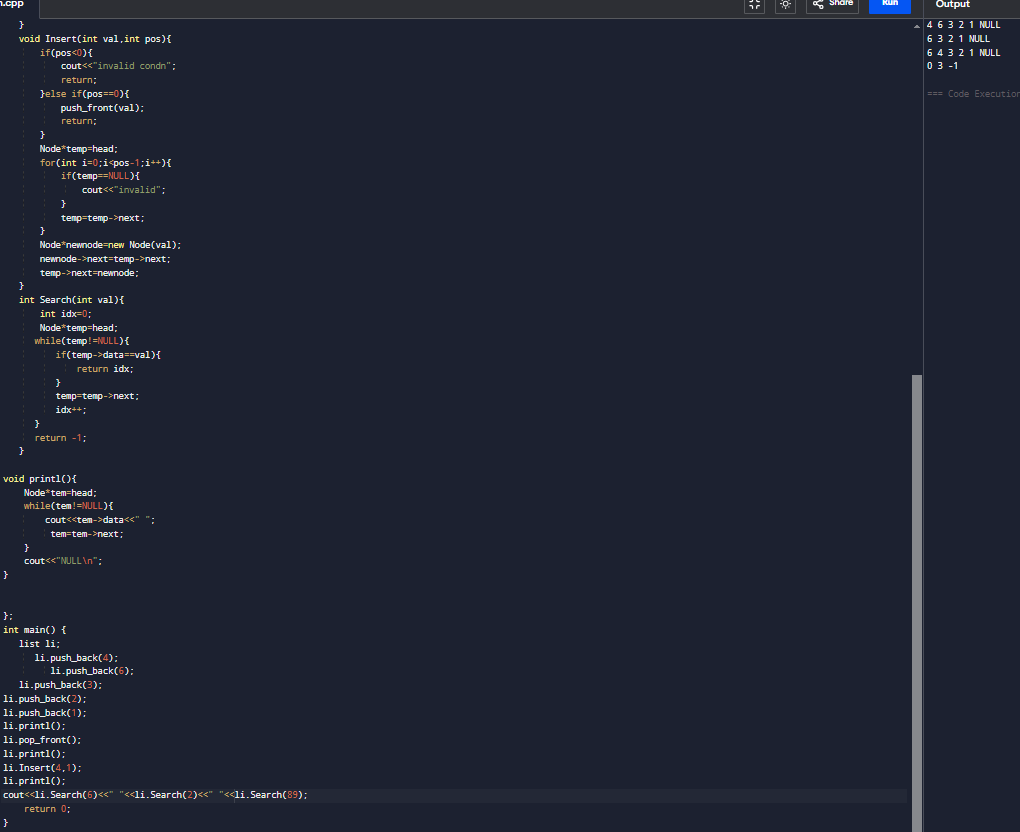


**POP BACK//0(n)**

****

**.**

**Insert And Search In Linked List//0(n)**

****

**Question Reversing a linked list:**

**class {**

**public:**

**ListNode\* reverseList(ListNode\* head) {**

**ListNode\*prev=NULL;**

**ListNode\*curr=head;**

**ListNode\*net=NULL;**

**while(curr!=NULL){**

**net=curr->next;**

**curr->next=prev;**

**prev=curr;**

**curr=net;**

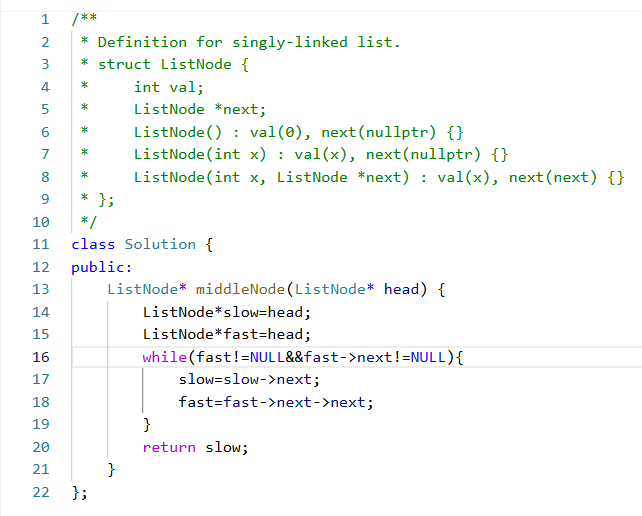
**}**

**return prev;**

**}**

**};**

**Question Middle Of A Linked List**

****

**Question Remove Duplicates From Sorted Array**

****

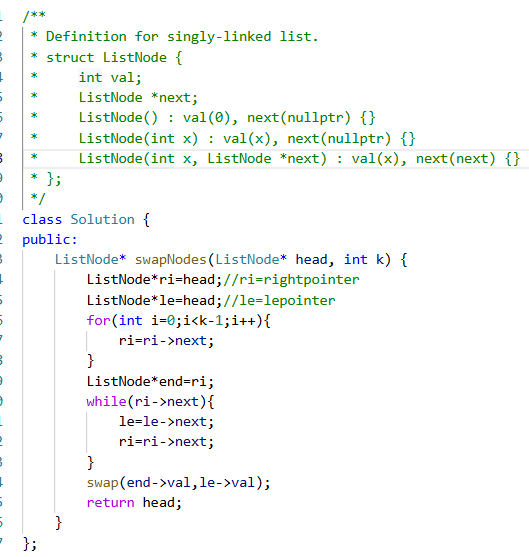
**Question Linked List Cycle**

****

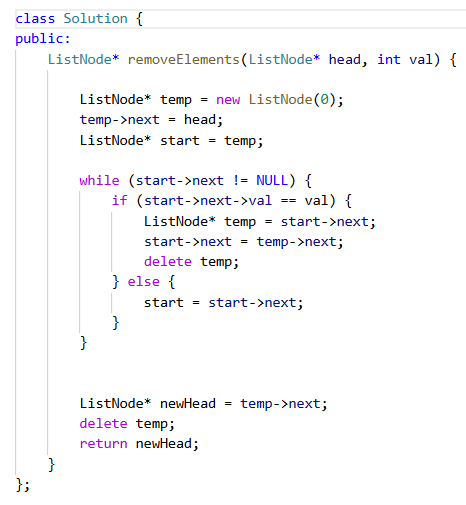
**Question Reverse Nodes in K group**

****

**Question:**[**Swapping Nodes in a Linked List**](https://leetcode.com/problems/swapping-nodes-in-a-linked-list/)

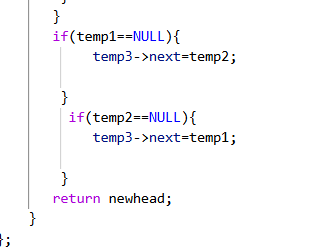
****

**Question Remove Linked List Elements**

****

**Question merge two sorted linked list:**

****

****

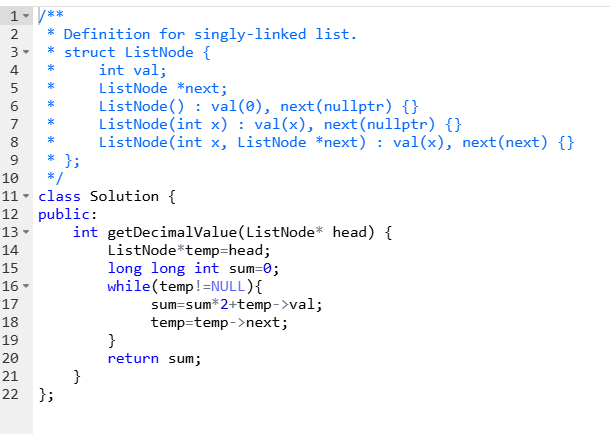
**Question Next Greater Elementin a linked list:**

****

**Question Linked List Cycle 2**

****

**Question**[**Convert Binary Number in a Linked List to Integer**](https://leetcode.com/problems/convert-binary-number-in-a-linked-list-to-integer/)

****

**STACK**

In C++, a stack is a data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Stacks are typically used in problems where you need to reverse things, such as in recursive calls or backtracking algorithms.

### **Stack Operations:**

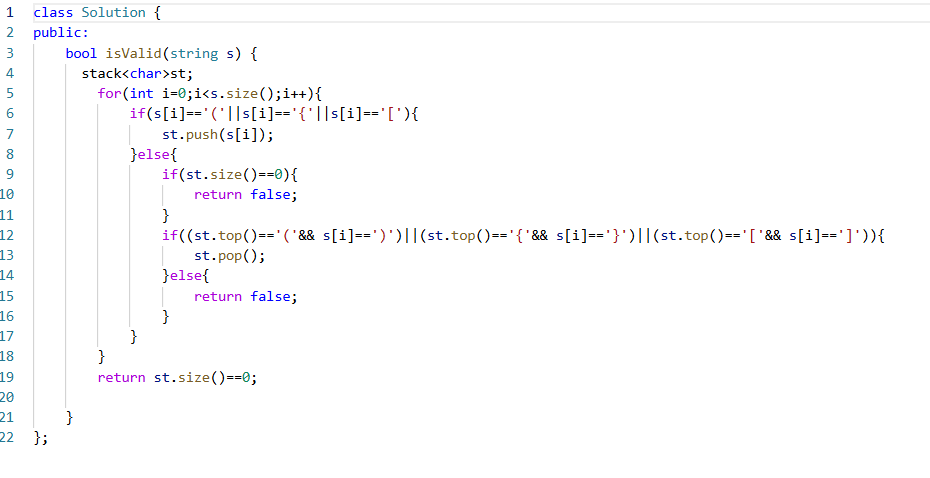
1. push(): Adds an element to the top of the stack.
2. pop(): Removes the top element from the stack.
3. top(): Returns the top element of the stack without removing it.
4. empty(): Checks whether the stack is empty.
5. size(): Returns the number of elements in the stack.

### **Underlying Container**

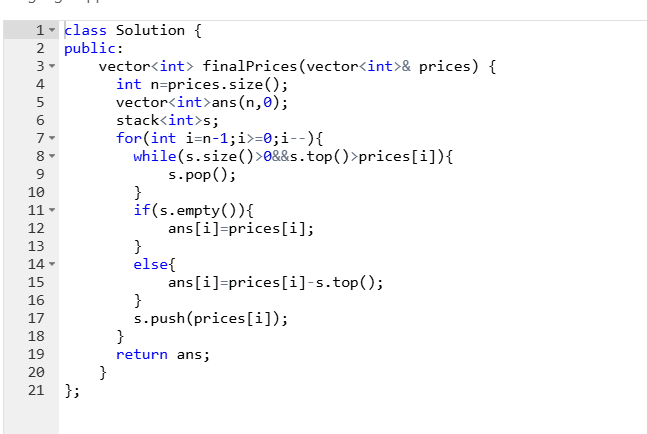
In C++, the **std::stack** container adapter uses an underlying container to store elements. By default, std::deque (double-ended queue) is used, but you can also specify other containers, such as std::vector or std::list.

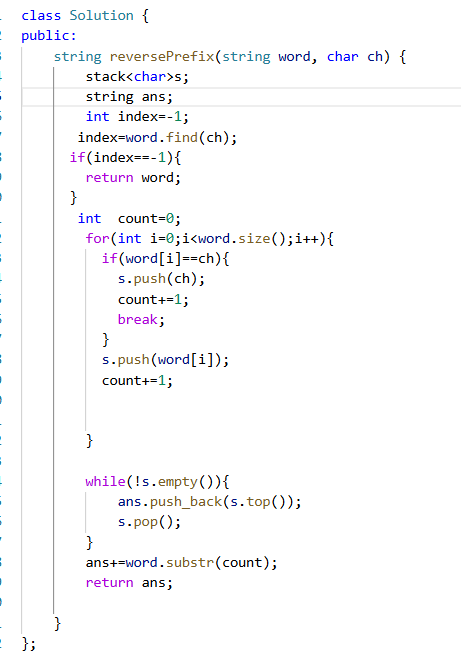
* **std::deque**: A **double-ended queue** provides fast insertion and deletion at both ends, which works well for stack operations.
* **std::vector**: This allows dynamic resizing but can be less efficient than std::deque for stack operations since resizing can be costly.

**Question 1 Valid Parenthesis**

****

**Question 2** [**Final Prices With a Special Discount in a Shop**](https://leetcode.com/problems/final-prices-with-a-special-discount-in-a-shop/)

**Question 3 Reverse Prefix in a word:**

****

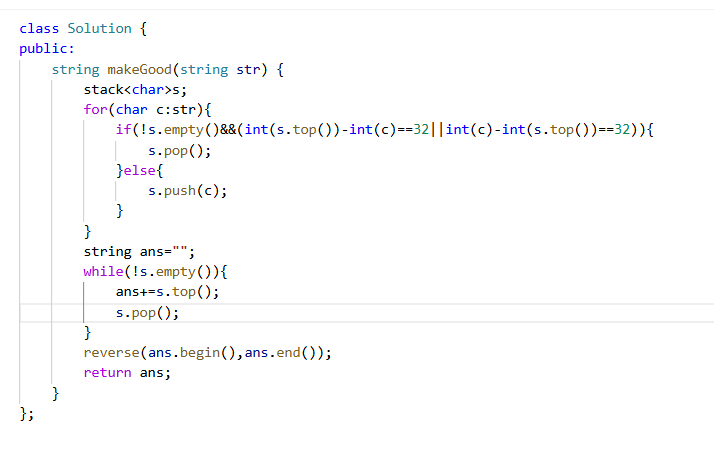
**Question4:Next Greater Element-1**

****

**Question 5:Clear Digits**

****

**Question Make String Looks Great**

****

**Question Baseball Game:**

****

**Question Crawler Log Folder**

****

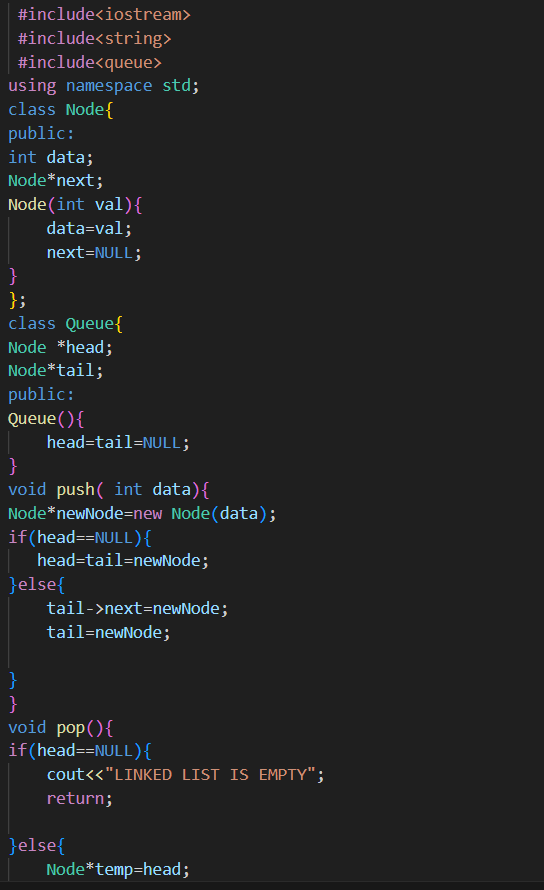
Queue

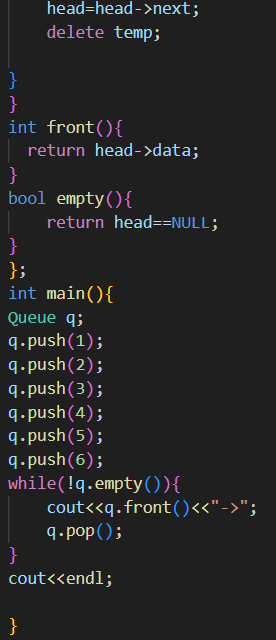
A queue in C++ is a linear data structure that follows the FIFO (First In, First Out) principle. This means that the element inserted first will be the first one to be removed.

#### **Key Operations of a Queue**

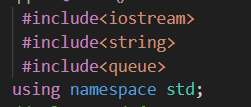
1. enqueue(x) → Adds an element x to the back of the queue.
2. dequeue() → Removes the front element from the queue.
3. front() → Returns the front element of the queue without removing it.
4. back() → Returns the last element of the queue without removing it.
5. empty() → Checks if the queue is empty.
6. size() → Returns the number of elements in the queue.

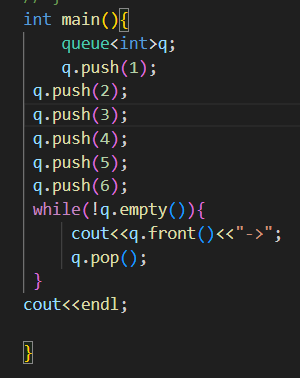
**BASIC IMPLEMENTATION FROM SCRATCH:**

****

****

**Basic Implementation Using STL:**

****

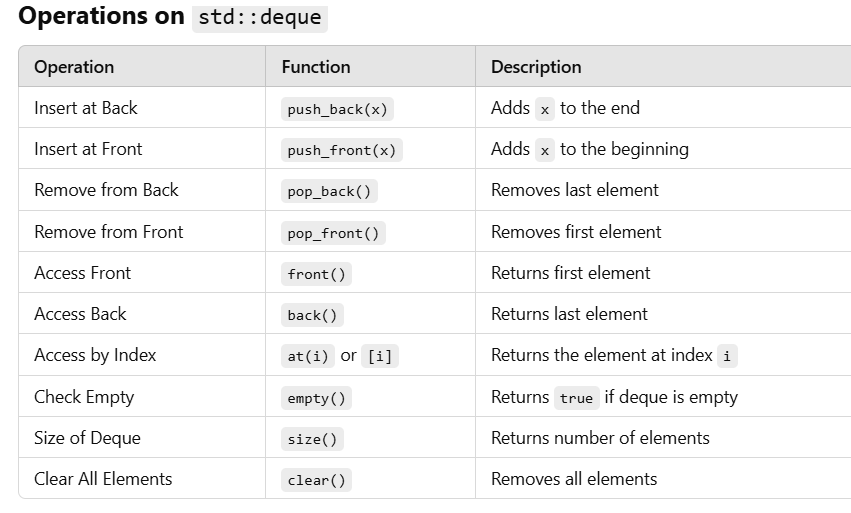
****

**DEQUE:**

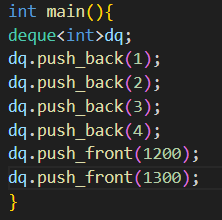
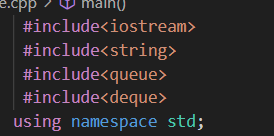
A deque (double-ended queue) is a dynamic data structure that allows insertion and deletion of elements from both front and back in constant time (O(1) on average). It is implemented in C++ using the std::deque container from the Standard Template Library (STL).

### Key Features of std::deque

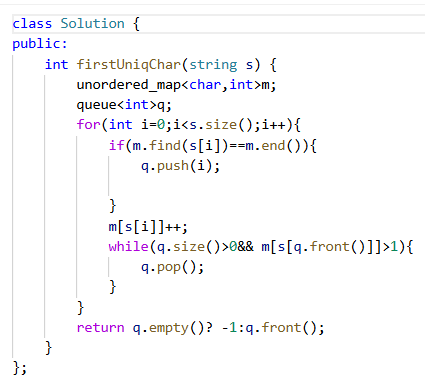
1. Dynamic Size: Unlike arrays, it resizes dynamically.
2. Efficient Operations:
   * O(1) insertion/deletion at both ends.
   * O(1) access to elements using indexing.
3. Better than vector for Frequent Insertions at the Front:
   * std::vector has O(n) complexity for front insertions.
   * std::deque maintains O(1) for both front and back insertion



**IMPLEMENTATION:**

****

**Question 1:First unique character in a string:**

****

**Binary Trees**

A **binary tree** is a tree data structure in which each node has at most two children referred to as the **left child** and the **right child**. Binary trees are fundamental structures used in computer science for tasks such as searching, sorting, and traversing hierarchical data.

### **Components of a Binary Tree**

A binary tree is composed of several key components:

1. **Node**: A node is the basic unit of a binary tree. It contains:  
   * **Data**: The value or key stored in the node.
   * **Left Child**: A pointer/reference to the left child node (or nullptr if no left child exists).
   * **Right Child**: A pointer/reference to the right child node (or nullptr if no right child exists).
2. **Root**: The topmost node in the binary tree. Every binary tree has a single root node, which is the entry point to the entire tree.
3. **Leaf Node**: A node that has no children, meaning both its left and right children are nullptr.
4. **Subtree**: A tree consisting of a node and all its descendants (left and right children, and their children).
5. **Parent Node**: A node that has children, i.e., the node to which other nodes (child nodes) are connected.
6. **Height of a Node**: The number of edges on the longest path from that node to a leaf. The height of the root node is the height of the tree.
7. **Depth of a Node**: The number of edges from the root to the node.
8. **Level of a Node**: The number of edges from the root to that node, plus one (i.e., level starts from 1).

### **Properties of Binary Trees**

1. **Maximum Number of Nodes at Level L**: The maximum number of nodes that can exist at level L (0-based index) is 2^L.
2. **Maximum Number of Nodes in a Binary Tree of Height H**: The maximum number of nodes a binary tree can have with height H is 2^(H+1) - 1.
3. **Balanced Binary Tree**: A binary tree is balanced if the height of the left and right subtrees of every node differs by no more than 1.
4. **Perfect Binary Tree**: A binary tree is perfect if all internal nodes have two children and all leaf nodes are at the same level.
5. **Complete Binary Tree**: A binary tree is complete if all levels are fully filled except possibly for the last level, which is filled from left to right.
6. **Full Binary Tree**: A binary tree is full if every node has either 0 or exactly 2 children.

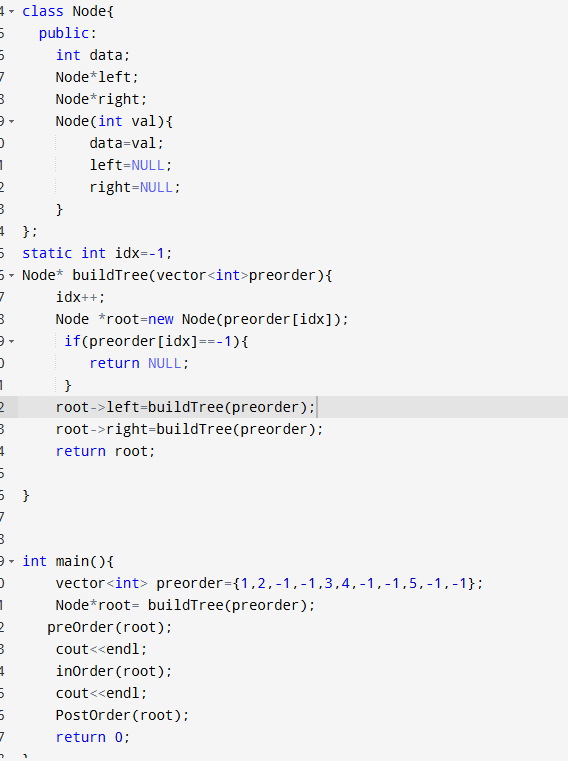
### **Types of Binary Trees**

1. **Full Binary Tree**: As mentioned, every node in a full binary tree has either 0 or 2 children.
2. **Complete Binary Tree**: A binary tree where all levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
3. **Perfect Binary Tree**: A tree in which all the internal nodes have exactly two children, and all leaf nodes are at the same level.
4. **Balanced Binary Tree**: A binary tree in which the left and right subtrees of every node differ in height by at most 1.
5. **Degenerate (or Pathological) Tree**: A tree where each parent node has only one child, essentially making the tree behave like a linked list.

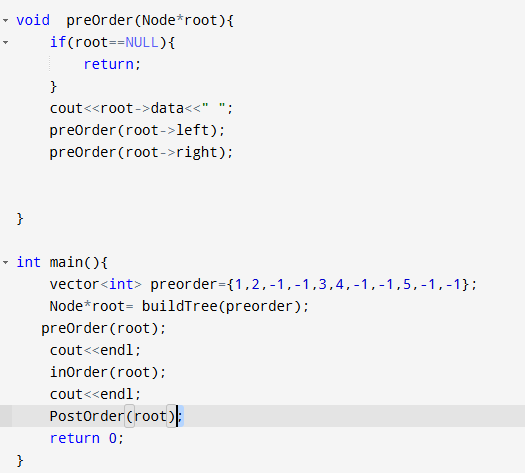
### **Operations on Binary Trees**

1. **Insertion**: Insert a node into the tree. The position of insertion depends on the type of binary tree:  
   * In a **complete** binary tree, insertion typically occurs at the leftmost available position at the last level.
   * In a **binary search tree**, insertion depends on whether the value is smaller or larger than the current node, and is inserted to the left or right subtree accordingly.
2. **Traversal**: Traversing a binary tree refers to visiting each node exactly once. There are three common ways to traverse a binary tree:  
   * **In-order Traversal**: Visit the left subtree, then the node, and then the right subtree.
   * **Pre-order Traversal**: Visit the node first, then the left subtree, and then the right subtree.
   * **Post-order Traversal**: Visit the left subtree, then the right subtree, and finally the node.
   * **Level-order Traversal**: Visit nodes level by level, starting from the root.
3. **Deletion**: Deleting a node from a binary tree involves finding the node and removing it. In a binary search tree, this might require replacing the deleted node with its in-order successor or predecessor.
4. **Searching**: In a **binary search tree (BST)**, you can search for a value by starting from the root and moving left or right based on the comparison of the target value and the current node’s value.

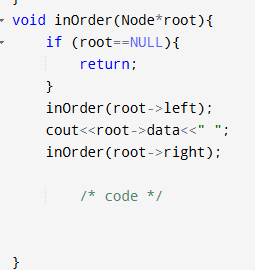
**\*Creation Of Binary Trees**

****

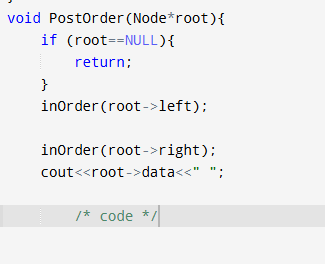
**\*PreOrder Traversal:**

****

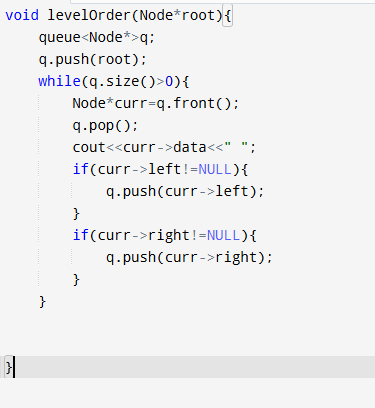
**\*Inorder**

****

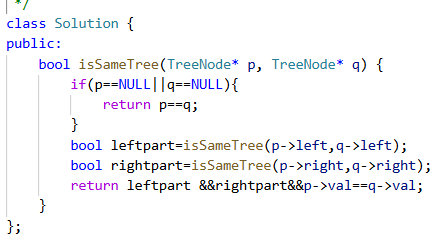
**PostOrder**

****

**LevelOrder**

****

**Question Same Tree**

****

**Question Subtree in Another Tree**

****

**Question Diameter of A Binary Tree**

****

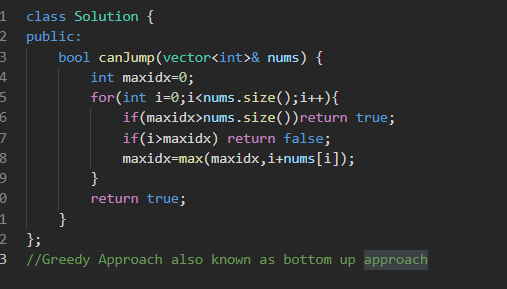
**Greedy Approach**

T The Greedy Approach is an algorithmic paradigm where you build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit (greedy choice).

## **How to Analyze if Greedy Works?**

1. **Define the problem clearly.**
2. **Look for sorting opportunities** – most greedy solutions involve sorting.
3. **Find a candidate greedy strategy.**
4. **Prove** that the local optimal leads to global optimal using:  
   * **Mathematical proof**
   * **Counterexamples (when it doesn’t work)**
5. **Compare to other approaches** (e.g., DP or backtracking)

**Q1) Jump Game :**

****