88. Merge Sorted Array

Easy

4377426Add to ListShare

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

Output: [1,2,2,3,5,6]

Explanation: The arrays we are merging are [1,2,3] and [2,5,6].

The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Example 2:

Input: nums1 = [1], m = 1, nums2 = [], n = 0

Output: [1]

Explanation: The arrays we are merging are [1] and [].

The result of the merge is [1].

Example 3:

Input: nums1 = [0], m = 0, nums2 = [1], n = 1

Output: [1]

Explanation: The arrays we are merging are [] and [1].

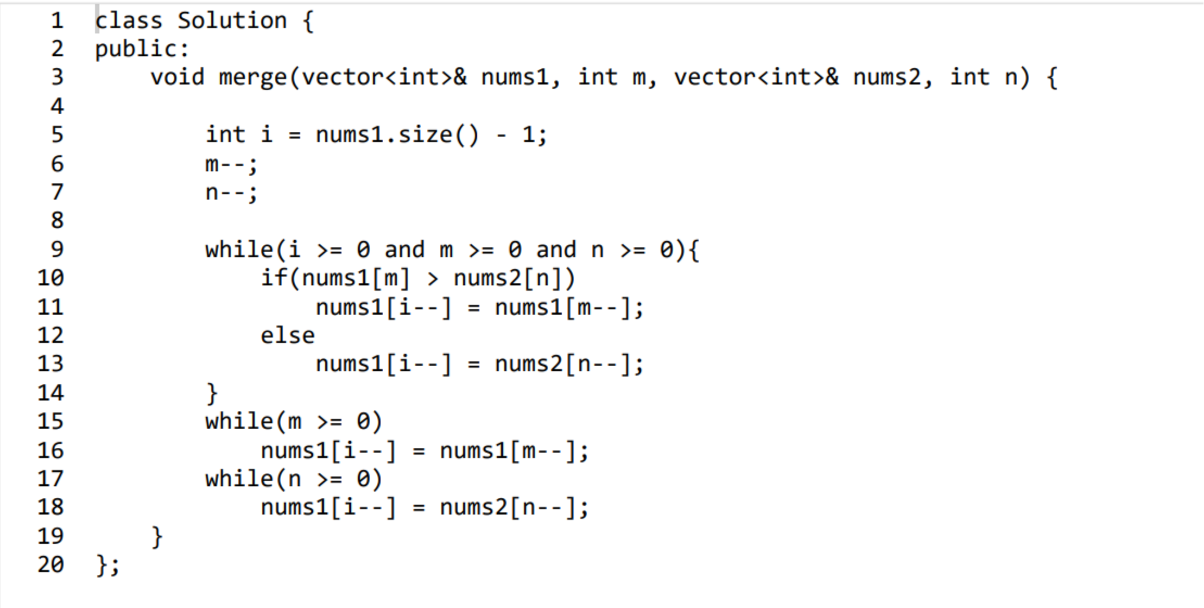
The result of the merge is [1].

Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

Constraints:

- nums1.length == m + n
- nums2.length == n
- 0 <= m, n <= 200
- 1 <= m + n <= 200
- -10^9 <= nums1[i], nums2[j] <= 10^9

Follow up: Can you come up with an algorithm that runs in O(m + n) time?



```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

        int i = nums1.size() - 1;
        m--;
        n--;

        while(i >= 0 and m >= 0 and n >= 0){
            if(nums1[m] > nums2[n])
                nums1[i--] = nums1[m--];
            else
                nums1[i--] = nums2[n--];
        }
        while(m >= 0)
            nums1[i--] = nums1[m--];
        while(n >= 0)
            nums1[i--] = nums2[n--];
    }
};
```

73. Set Matrix Zeroes

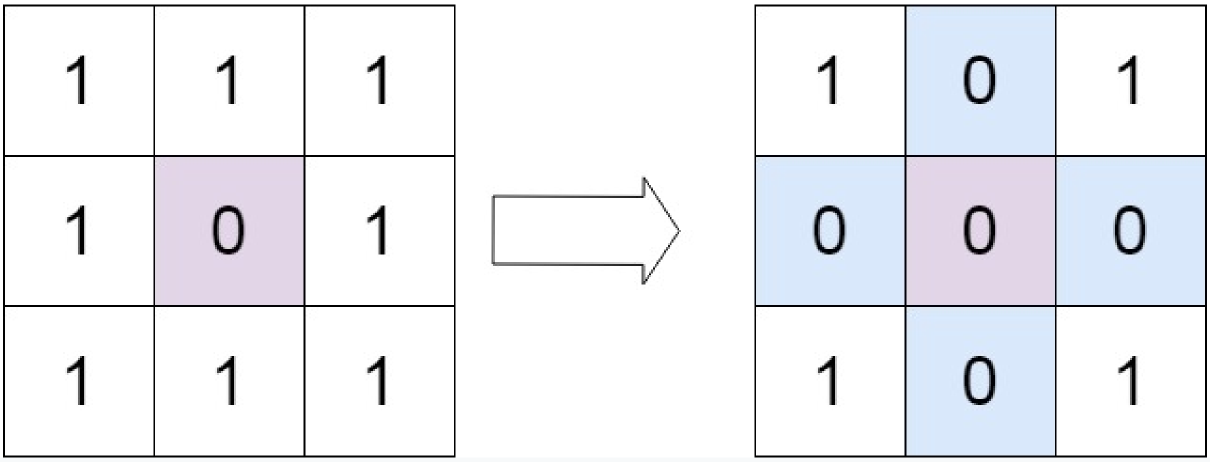
Medium

6825485Add to ListShare

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

You must do it in place.

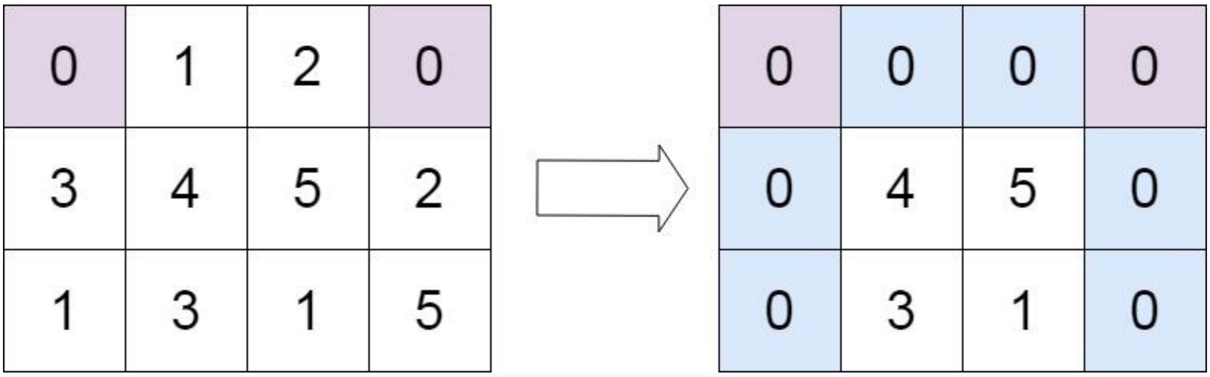
Example 1:



Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]

Output: [[1,0,1],[0,0,0],[1,0,1]]

Example 2:



Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]

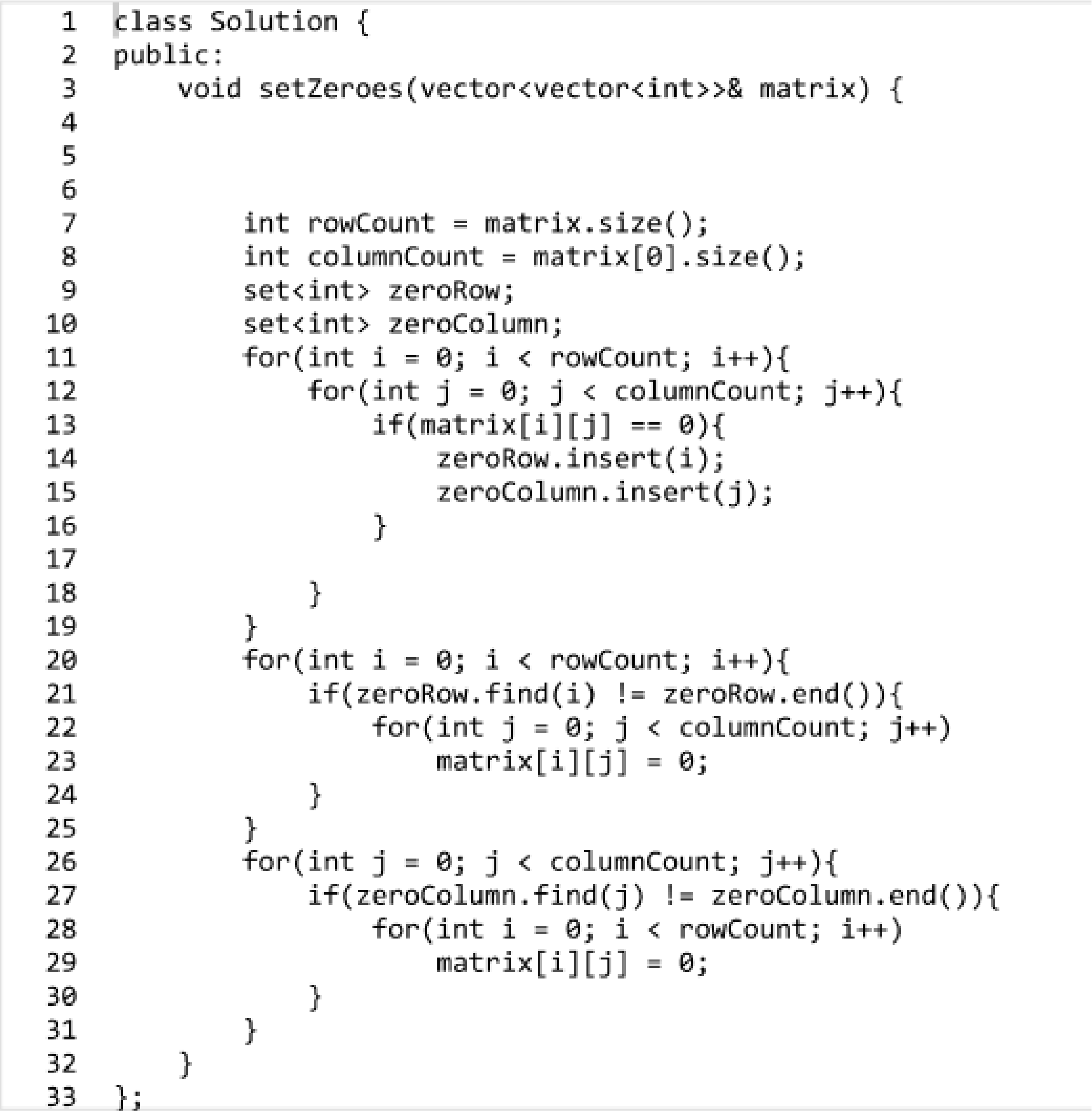
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

Constraints:

- m == matrix.length
- n == matrix[0].length
- 1 <= m, n <= 200
- -2^{31} <= matrix[i][j] <= 2^{31} - 1

Follow up:

- A straightforward solution using O(mn) space is probably a bad idea.
- A simple improvement uses O(m + n) space, but still not the best solution.
- Could you devise a constant space solution?



```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {



        int rowCount = matrix.size();
        int columnCount = matrix[0].size();
        set<int> zeroRow;
        set<int> zeroColumn;
        for(int i = 0; i < rowCount; i++){
            for(int j = 0; j < columnCount; j++){
                if(matrix[i][j] == 0){
                    zeroRow.insert(i);
                    zeroColumn.insert(j);
                }

            }
        }
        for(int i = 0; i < rowCount; i++){
            if(zeroRow.find(i) != zeroRow.end()){
                for(int j = 0; j < columnCount; j++)
                    matrix[i][j] = 0;
            }
        }
        for(int j = 0; j < columnCount; j++){
            if(zeroColumn.find(j) != zeroColumn.end()){
                for(int i = 0; i < rowCount; i++)
                    matrix[i][j] = 0;
            }
        }
    }
};
```

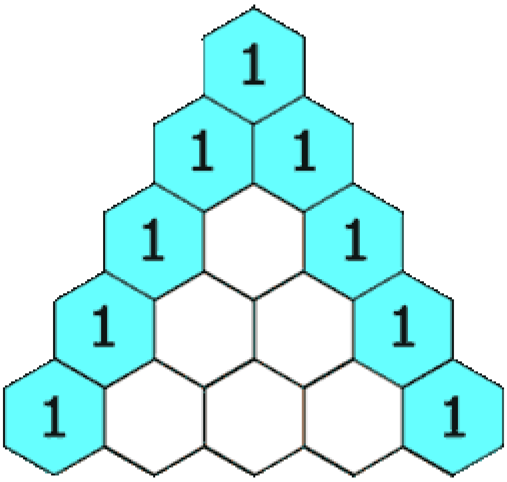
118. Pascal's Triangle

Easy

5609207Add to ListShare

Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



Example 1:

Input: numRows = 5

Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

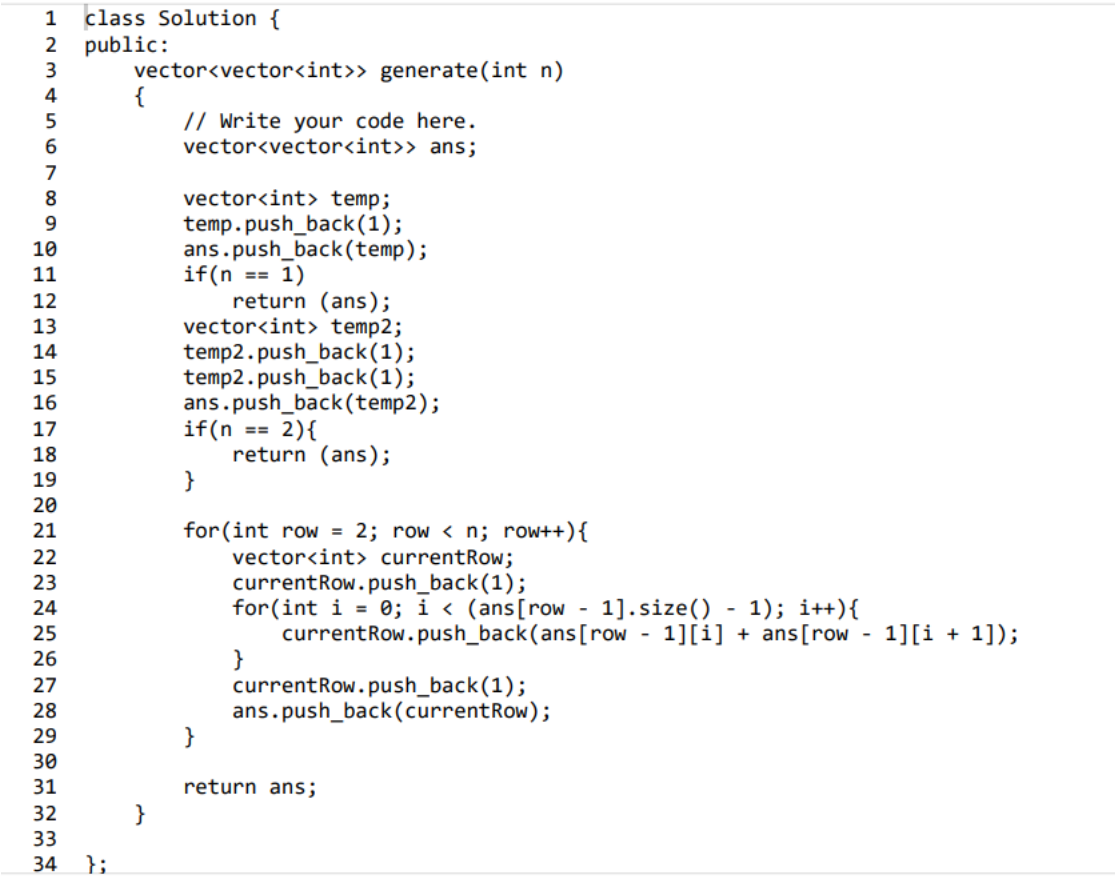
Example 2:

Input: numRows = 1

Output: [[1]]

Constraints:

- 1 <= numRows <= 30



```
class Solution {
public:
    vector<vector<int>> generate(int n)
    {
        // Write your code here.
        vector<vector<int>> ans;

        vector<int> temp;
        temp.push_back(1);
        ans.push_back(temp);
        if(n == 1)
            return (ans);
        vector<int> temp2;
        temp2.push_back(1);
        temp2.push_back(1);
        ans.push_back(temp2);
        if(n == 2){
            return (ans);
        }

        for(int row = 2; row < n; row++){
            vector<int> currentRow;
            currentRow.push_back(1);
            for(int i = 0; i < (ans[row - 1].size() - 1); i++){
                currentRow.push_back(ans[row - 1][i] + ans[row - 1][i + 1]);
            }
            currentRow.push_back(1);
            ans.push_back(currentRow);
        }

        return ans;
    }

};
```

287. Find the Duplicate Number

Medium

135291561Add to ListShare

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

You must solve the problem **without** modifying the array nums and uses only constant extra space.

Example 1:

Input: nums = [1,3,4,2,2]

Output: 2

Example 2:

Input: nums = [3,1,3,4,2]

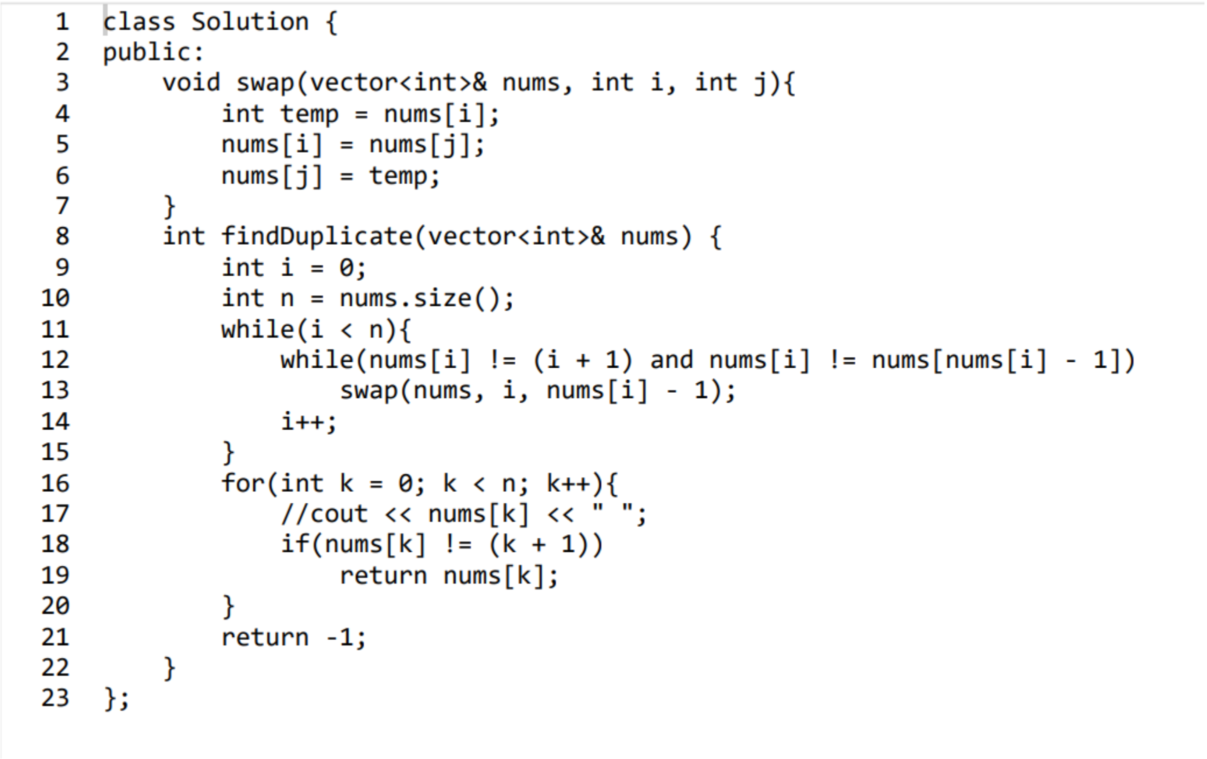
Output: 3

Constraints:

- 1 <= n <= 10^5
- nums.length == n + 1
- 1 <= nums[i] <= n
- All the integers in nums appear only **once** except for **precisely one integer** which appears **two or more** times.

Follow up:

- How can we prove that at least one duplicate number must exist in nums?
- Can you solve the problem in linear runtime complexity?



```
class Solution {
public:
    void swap(vector<int>& nums, int i, int j){
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
    int findDuplicate(vector<int>& nums) {
        int i = 0;
        int n = nums.size();
        while(i < n){
            while(nums[i] != (i + 1) and nums[i] != nums[nums[i] - 1])
                swap(nums, i, nums[i] - 1);
            i++;
        }
        for(int k = 0; k < n; k++){
            //cout << nums[k] << " ";
            if(nums[k] != (k + 1))
                return nums[k];
        }
        return -1;
    }
};
```

344. Reverse String

Easy

4934941Add to ListShare

Write a function that reverses a string. The input string is given as an array of characters s.

You must do this by modifying the input array in-place with O(1) extra memory.

Example 1:

Input: s = ["h","e","l","l","o"]

Output: ["o","l","l","e","h"]

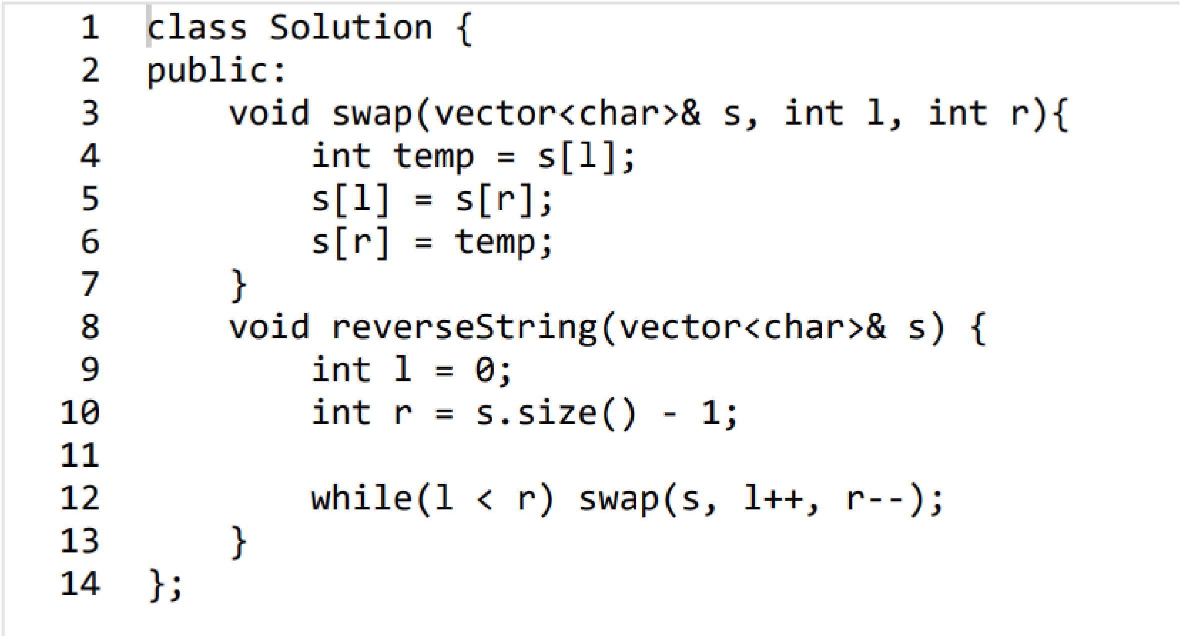
Example 2:

Input: s = ["H","a","n","n","a","h"]

Output: ["h","a","n","n","a","H"]

Constraints:

- 1 <= s.length <= 10^5
- s[i] is a printable ascii character.



```
class Solution {
public:
    void swap(vector<char>& s, int l, int r){
        int temp = s[l];
        s[l] = s[r];
        s[r] = temp;
    }
    void reverseString(vector<char>& s) {
        int l = 0;
        int r = s.size() - 1;

        while(l < r) swap(s, l++, r--);
    }
};
```

75. Sort Colors

Medium

9623411Add to ListShare

Given an array nums with n objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

Input: nums = [2,0,2,1,1,0]

Output: [0,0,1,1,2,2]

Example 2:

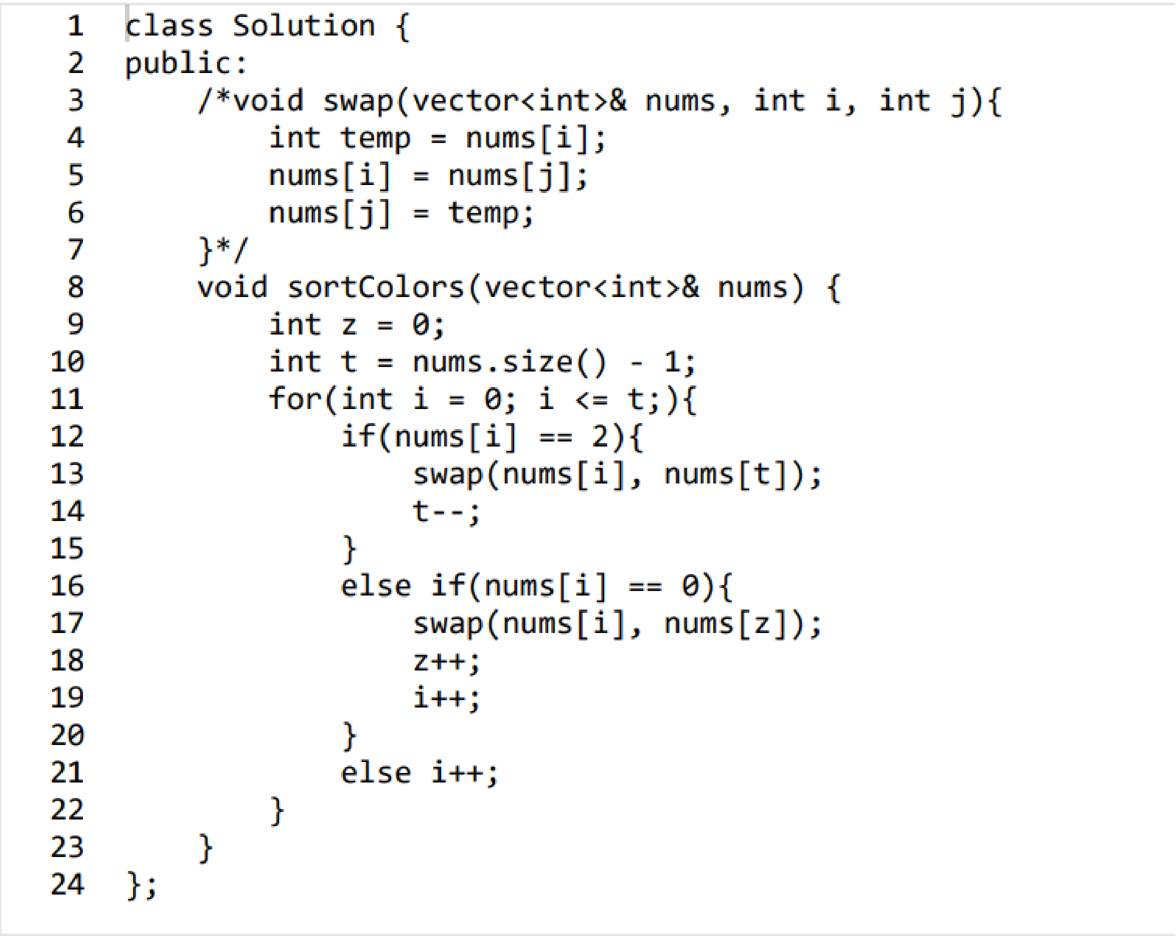
Input: nums = [2,0,1]

Output: [0,1,2]

Constraints:

- n == nums.length
- 1 <= n <= 300
- nums[i] is either 0, 1, or 2.

Follow up: Could you come up with a one-pass algorithm using only constant extra space?



```
class Solution {
public:
    /*void swap(vector<int>& nums, int i, int j){
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }*/
    void sortColors(vector<int>& nums) {
        int z = 0;
        int t = nums.size() - 1;
        for(int i = 0; i <= t;){
            if(nums[i] == 2){
                swap(nums[i], nums[t]);
                t--;
            }
            else if(nums[i] == 0){
                swap(nums[i], nums[z]);
                z++;
                i++;
            }
            else i++;
        }
    }
};
```

189. Rotate Array

Medium

92511281Add to ListShare

Given an array, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3

Output: [5,6,7,1,2,3,4]

Explanation:

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

Example 2:

Input: nums = [-1,-100,3,99], k = 2

Output: [3,99,-1,-100]

Explanation:

rotate 1 steps to the right: [99,-1,-100,3]

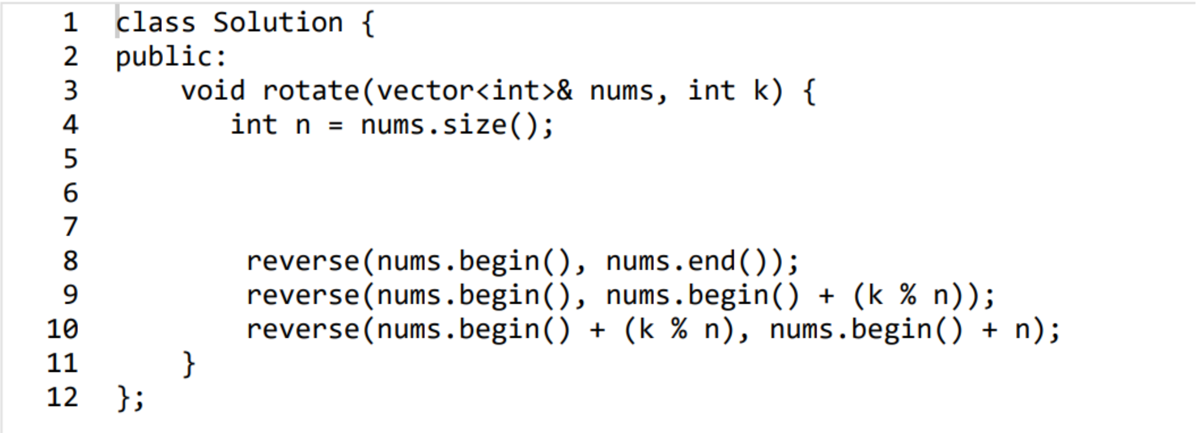
rotate 2 steps to the right: [3,99,-1,-100]

Constraints:

- 1 <= nums.length <= 10^5
- -2^{31} <= nums[i] <= $2^{31} - 1$
- 0 <= k <= 10^5

Follow up:

- Try to come up with as many solutions as you can. There are at least **three** different ways to solve this problem.
- Could you do it in-place with O(1) extra space?



```
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
       int n = nums.size();



        reverse(nums.begin(), nums.end());
        reverse(nums.begin(), nums.begin() + (k % n));
        reverse(nums.begin() + (k % n), nums.begin() + n);
    }
};
```

45. Jump Game II

Medium

7869293Add to ListShare

Given an array of non-negative integers nums, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

Example 1:

Input: nums = [2,3,1,1,4]

Output: 2

Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

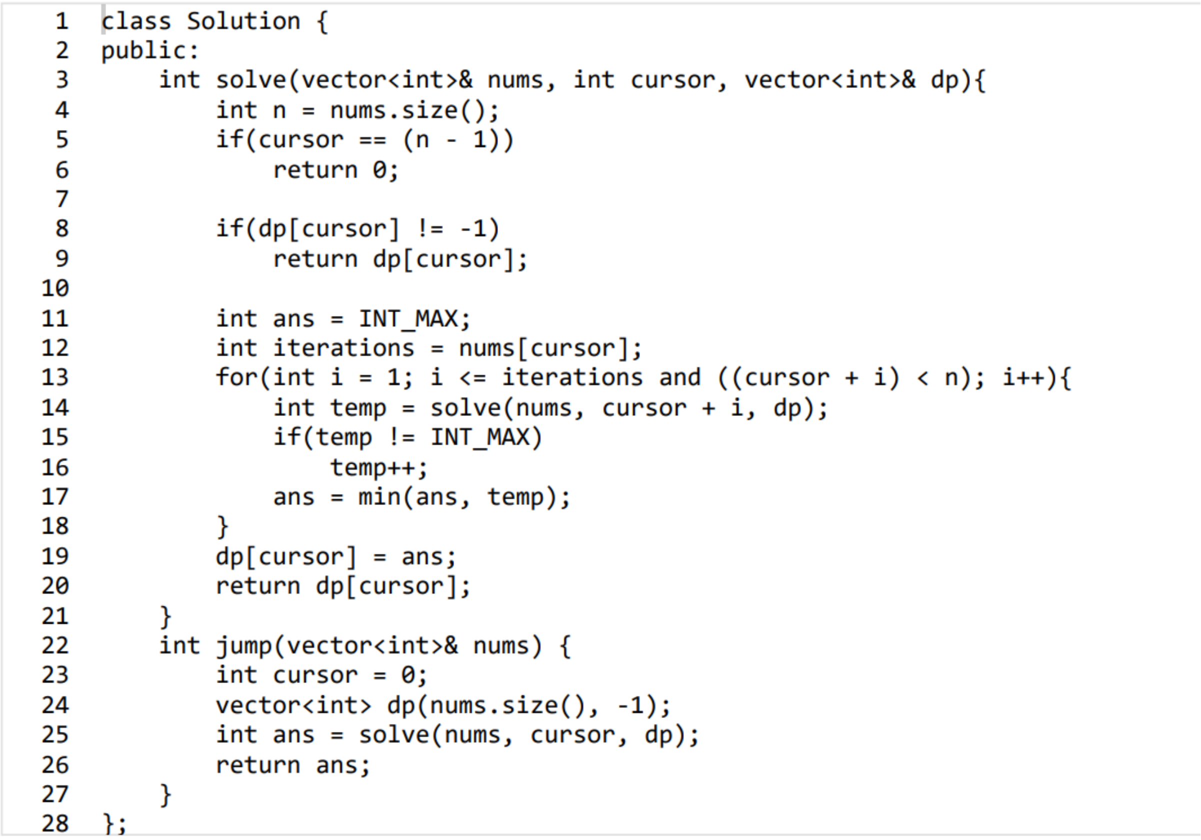
Example 2:

Input: nums = [2,3,0,1,4]

Output: 2

Constraints:

- 1 <= nums.length <= 10^4
- 0 <= nums[i] <= 1000



```
class Solution {
public:
    int solve(vector<int>& nums, int cursor, vector<int>& dp){
        int n = nums.size();
        if(cursor == (n - 1))
            return 0;

        if(dp[cursor] != -1)
            return dp[cursor];

        int ans = INT_MAX;
        int iterations = nums[cursor];
        for(int i = 1; i <= iterations and ((cursor + i) < n); i++){
            int temp = solve(nums, cursor + i, dp);
            if(temp != INT_MAX)
                temp++;
            ans = min(ans, temp);
        }
        dp[cursor] = ans;
        return dp[cursor];
    }
    int jump(vector<int>& nums) {
        int cursor = 0;
        vector<int> dp(nums.size(), -1);
        int ans = solve(nums, cursor, dp);
        return ans;
    }
};
```

31. Next Permutation

Medium

105033395Add to ListShare

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are considered permutations of arr: [1,2,3], [1,3,2], [3,1,2], [2,3,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be **in place** and use only constant extra memory.

Example 1:

Input: nums = [1,2,3]

Output: [1,3,2]

Example 2:

Input: nums = [3,2,1]

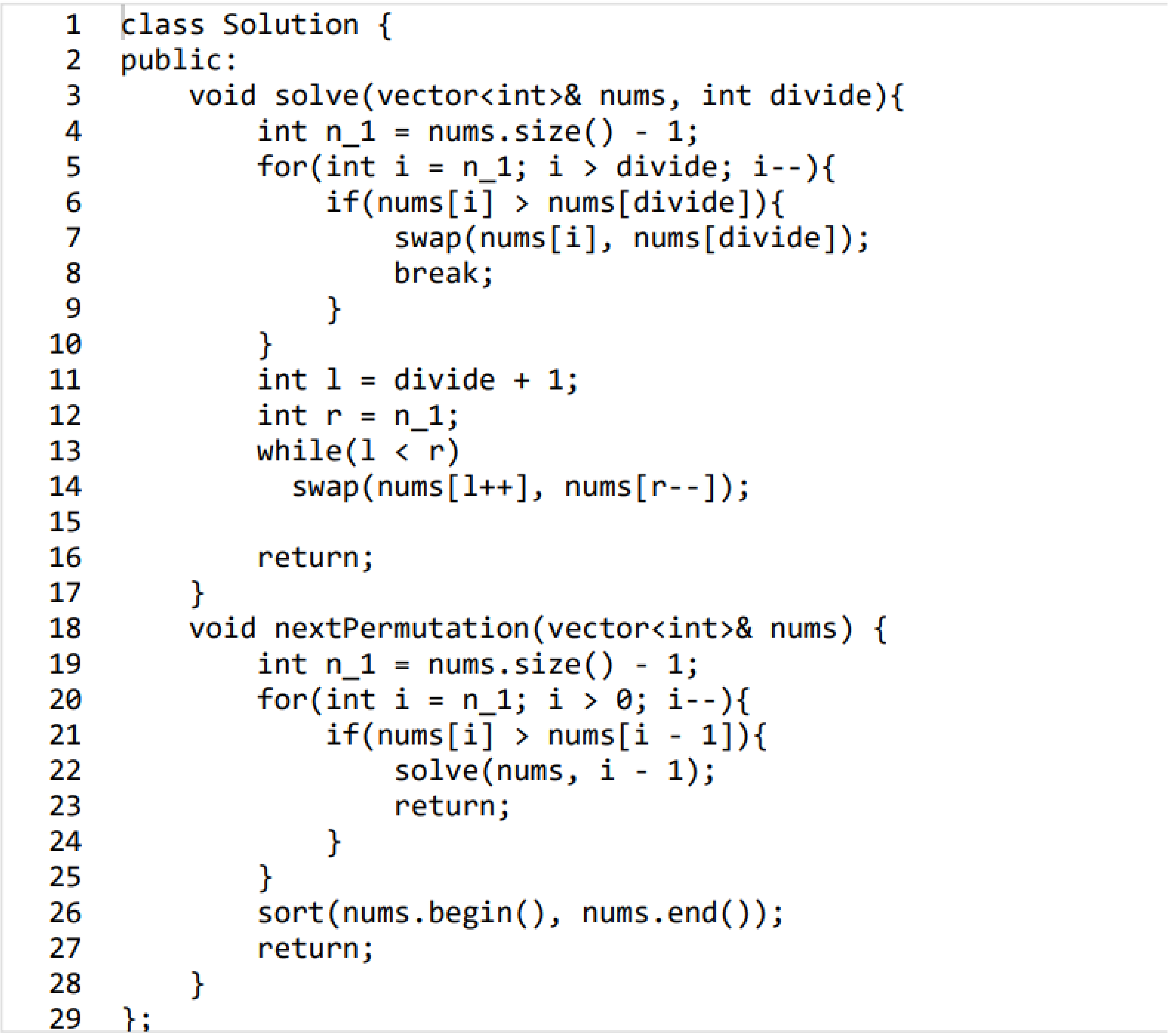
Output: [1,2,3]

Example 3:

Input: nums = [1,1,5]

Output: [1,5,1]

Constraints: `1 <= nums.length <= 100`, `0 <= nums[i] <= 100`



```
class Solution {
public:
    void solve(vector<int>& nums, int divide){
        int n_1 = nums.size() - 1;
        for(int i = n_1; i > divide; i--){
            if(nums[i] > nums[divide]){
                swap(nums[i], nums[divide]);
                break;
            }
        }
        int l = divide + 1;
        int r = n_1;
        while(l < r)
          swap(nums[l++], nums[r--]);

        return;
    }
    void nextPermutation(vector<int>& nums) {
        int n_1 = nums.size() - 1;
        for(int i = n_1; i > 0; i--){
            if(nums[i] > nums[i - 1]){
                solve(nums, i - 1);
                return;
            }
        }
        sort(nums.begin(), nums.end());
        return;
    }
};
```

543. Diameter of Binary Tree

Easy

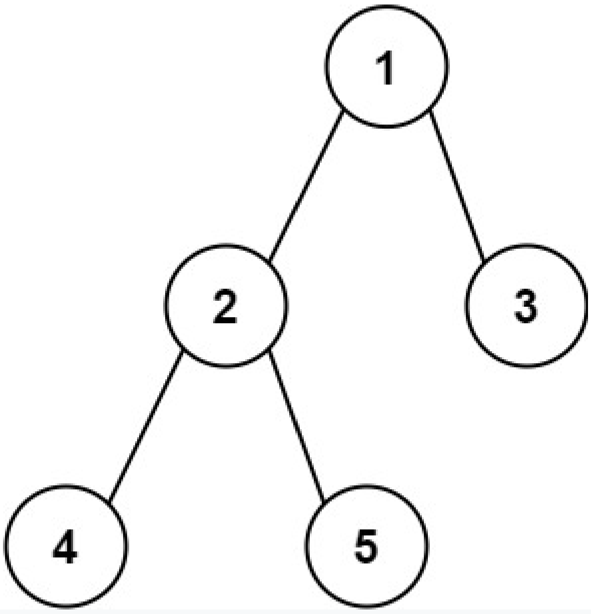
7719489Add to ListShare

Given the root of a binary tree, return *the length of the* ***diameter*** *of the tree.*

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

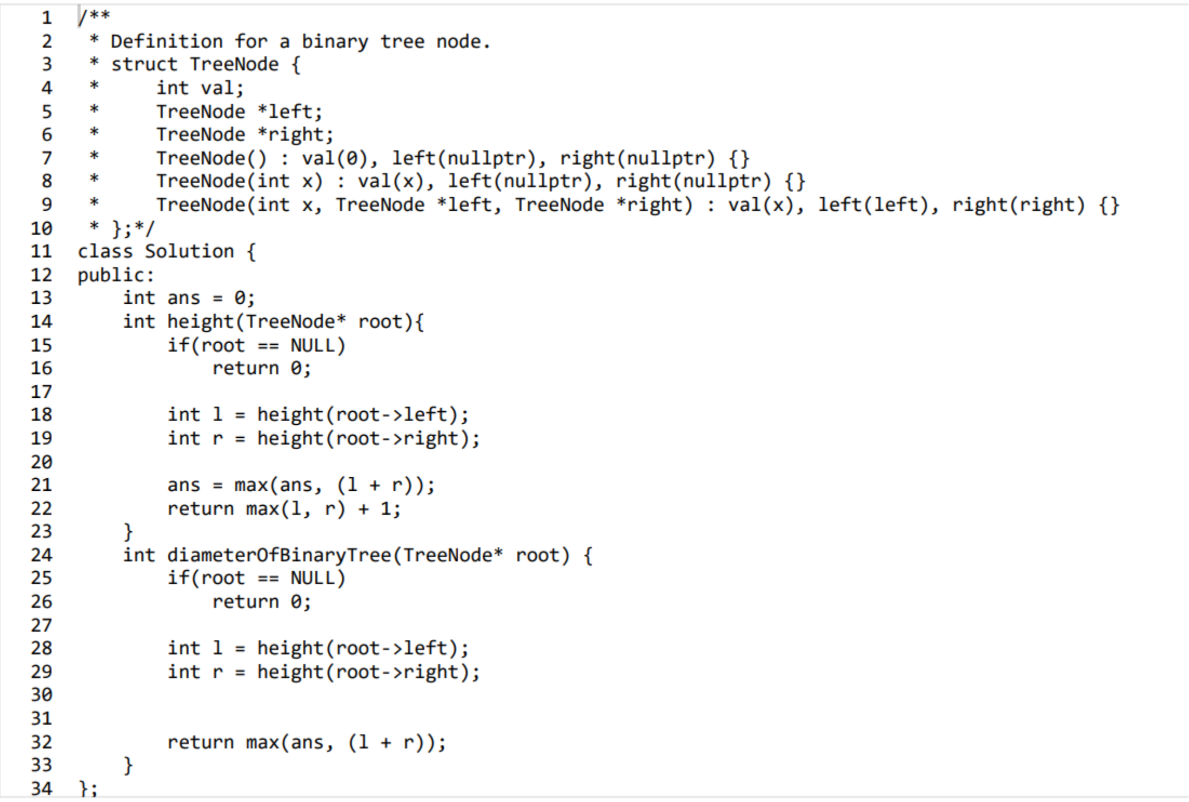
Example 2:

Input: root = [1,2]

Output: 1

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- -100 <= Node.val <= 100



```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };*/
class Solution {
public:
    int ans = 0;
    int height(TreeNode* root){
        if(root == NULL)
            return 0;

        int l = height(root->left);
        int r = height(root->right);

        ans = max(ans, (l + r));
        return max(l, r) + 1;
    }
    int diameterOfBinaryTree(TreeNode* root) {
        if(root == NULL)
            return 0;

        int l = height(root->left);
        int r = height(root->right);


        return max(ans, (l + r));
    }
};
```

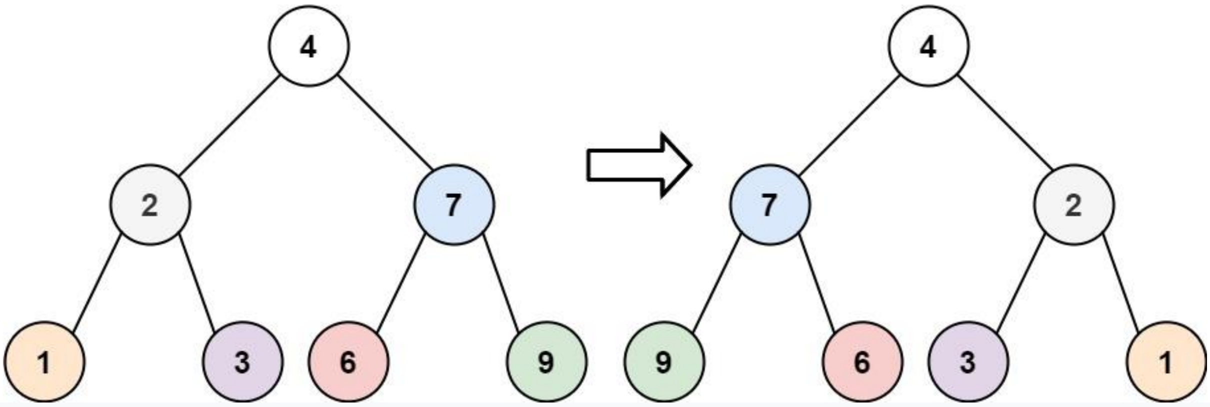
226. Invert Binary Tree

Easy

8206110Add to ListShare

Given the root of a binary tree, invert the tree, and return *its root*.

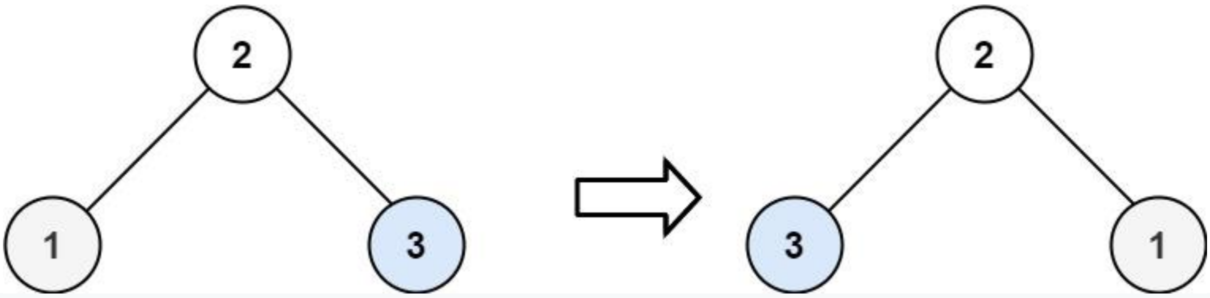
Example 1:



Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

Example 2:



Input: root = [2,1,3]

Output: [2,3,1]

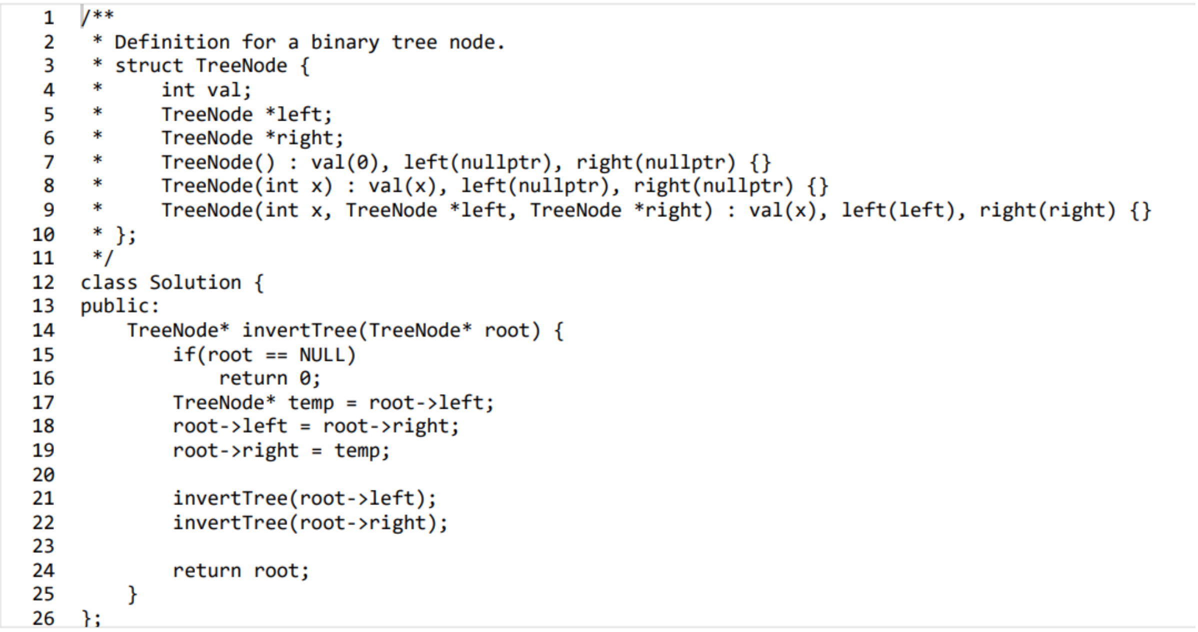
Example 3:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100



```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root == NULL)
            return 0;
        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;

        invertTree(root->left);
        invertTree(root->right);

        return root;
    }
};
```

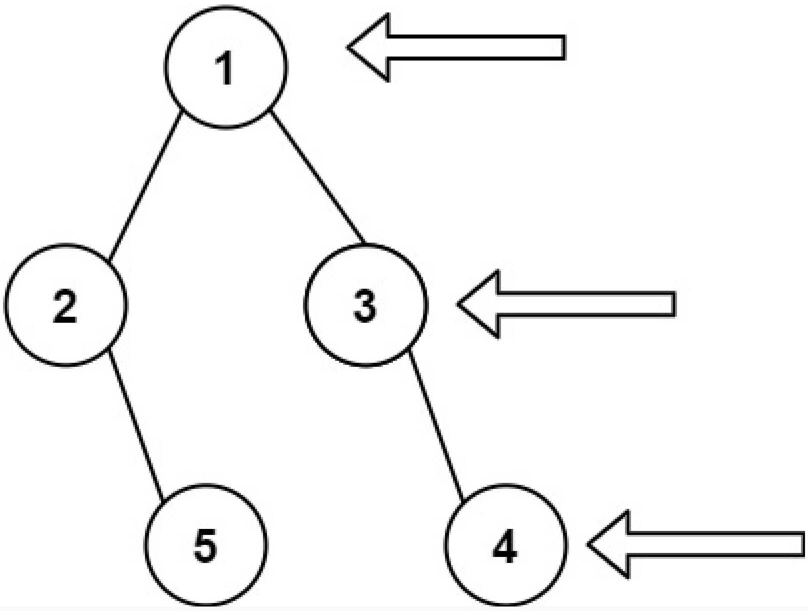
199. Binary Tree Right Side View

Medium

6383347Add to ListShare

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

Example 1:



Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

Example 2:

Input: root = [1,null,3]

Output: [1,3]

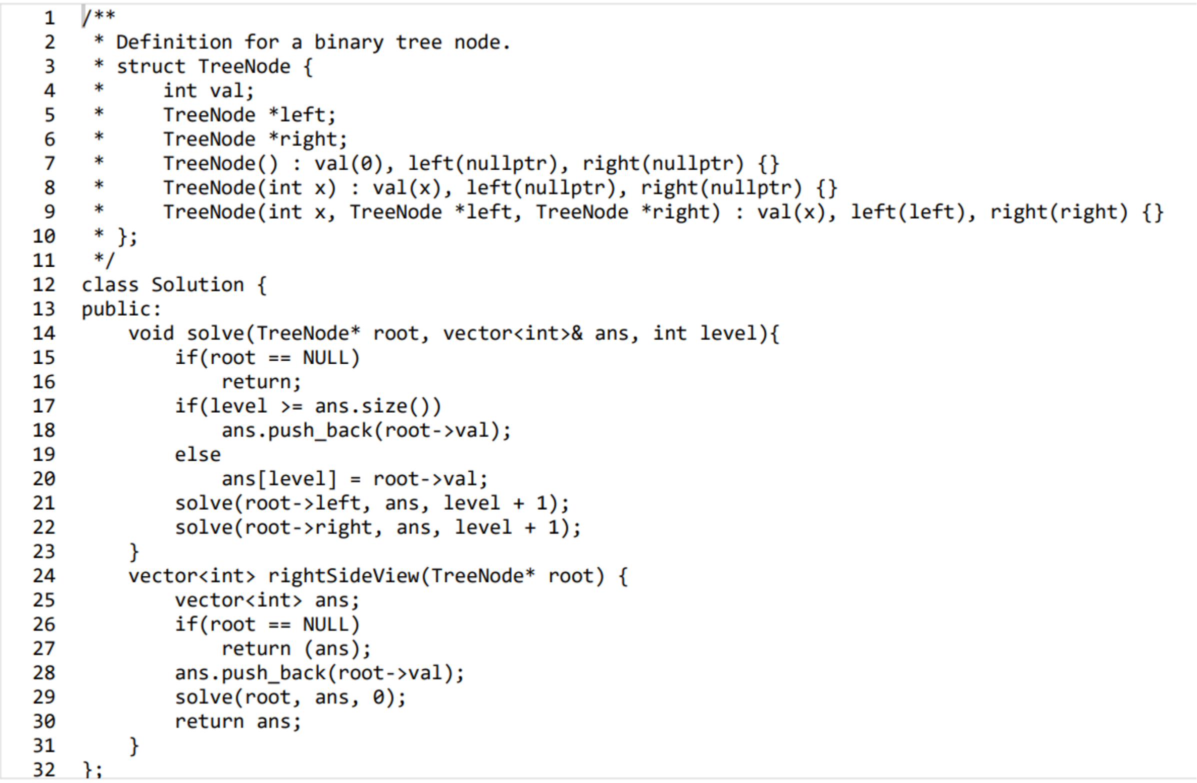
Example 3:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100



```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    void solve(TreeNode* root, vector<int>& ans, int level){
        if(root == NULL)
            return;
        if(level >= ans.size())
            ans.push_back(root->val);
        else
            ans[level] = root->val;
        solve(root->left, ans, level + 1);
        solve(root->right, ans, level + 1);
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> ans;
        if(root == NULL)
            return (ans);
        ans.push_back(root->val);
        solve(root, ans, 0);
        return ans;
    }
};
```

Left View of Binary Tree

Easy Accuracy: 37.86% Submissions: 100k+ Points: 2

Given a Binary Tree, print Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Left view of following tree is 1 2 4 8.

```
      1
    /   \
   2      3
 /   \   /   \
4    5  6    7
 \
  8
```

Example 1:

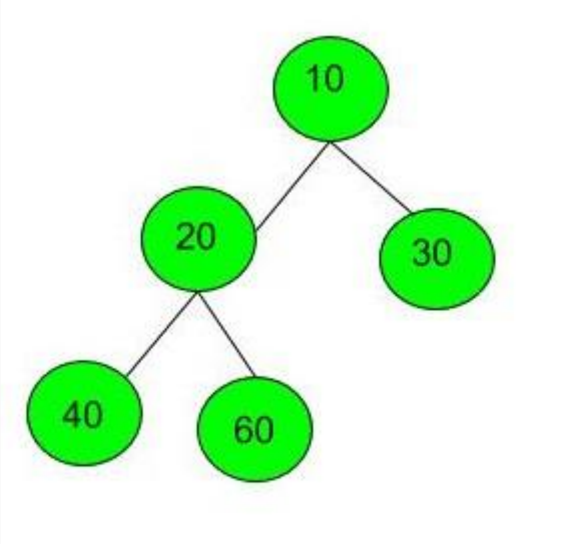
Input:

```
  1
 / \
3   2
```

Output: 1 3

Example 2:

Input:

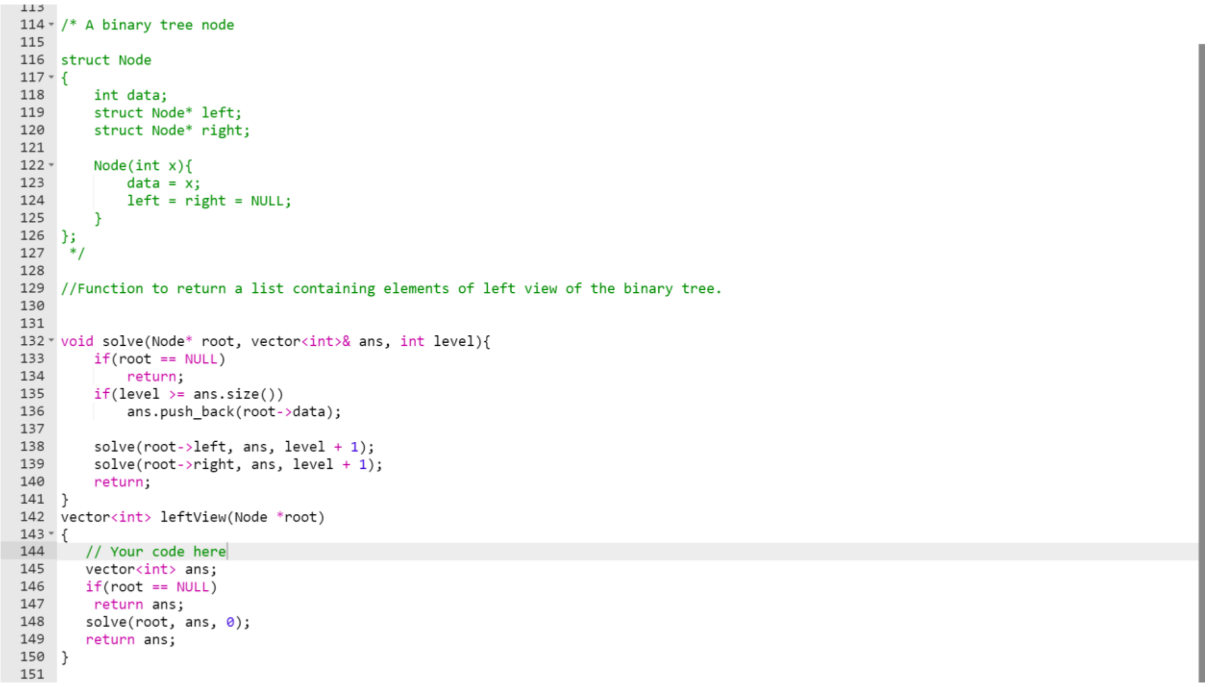


Output: 10 20 40

Your Task:
You just have to **complete** the function **leftView()** that prints the left view. The

newline is automatically appended by the driver code.
Expected Time Complexity: O(N).
Expected Auxiliary Space: O(Height of the Tree).

Constraints:
0 <= Number of nodes <= 100
1 <= Data of a node <= 1000



```cpp
/* A binary tree node

struct Node
{
    int data;
    struct Node* left;
    struct Node* right;

    Node(int x){
        data = x;
        left = right = NULL;
    }
};
 */

//Function to return a list containing elements of left view of the binary tree.


void solve(Node* root, vector<int>& ans, int level){
    if(root == NULL)
        return;
    if(level >= ans.size())
        ans.push_back(root->data);

    solve(root->left, ans, level + 1);
    solve(root->right, ans, level + 1);
    return;
}
vector<int> leftView(Node *root)
{
   // Your code here
   vector<int> ans;
   if(root == NULL)
    return ans;
   solve(root, ans, 0);
   return ans;
}

```

Top View of Binary Tree

Medium Accuracy: 32.3% Submissions: 100k+ Points: 4

Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree

```
      1
   /     \
  2       3
 /  \    /   \
4    5  6    7
```

Top view will be: 4 2 1 3 7
Note: Return nodes from **leftmost** node to **rightmost** node.

Example 1:

Input:

```
     1
   /   \
  2     3
```

Output: 2 1 3

Example 2:

Input:

```
       10
    /     \
  20       30
 /  \    /   \
40   60  90    100
```

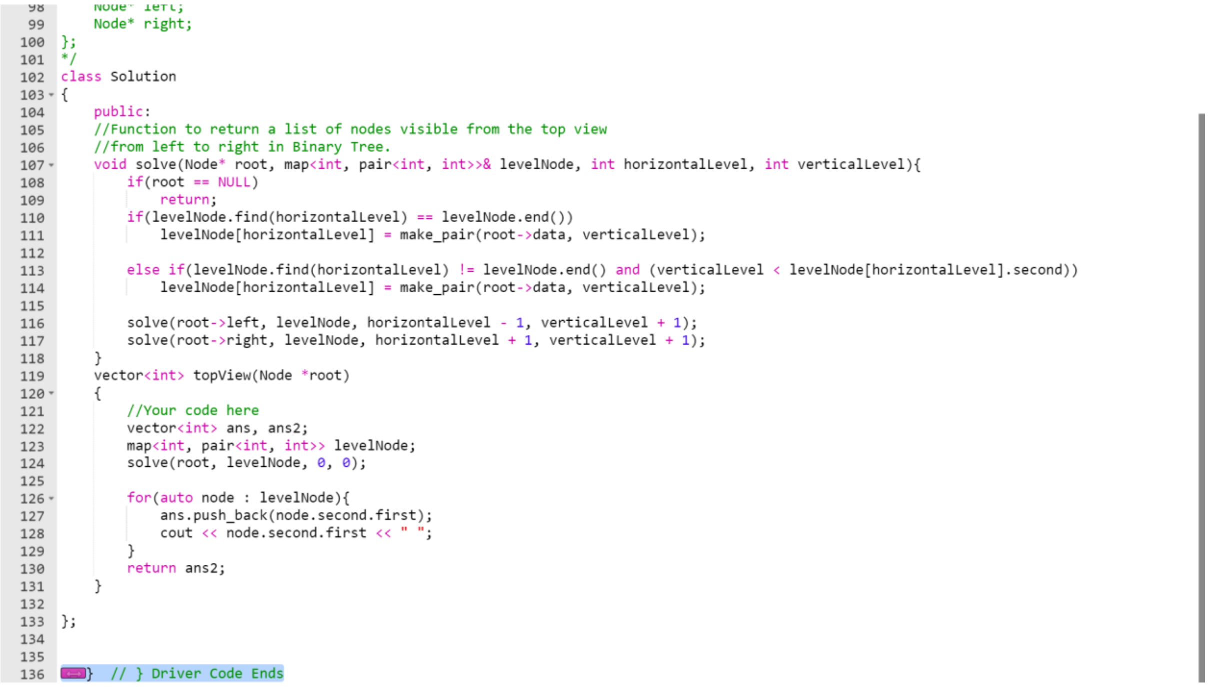
Output: 40 20 10 30 100

Your Task:
Since this is a function problem. You don't have to take input. Just complete the function **topView()** that takes **root node** as parameter and returns a list of nodes visible from the top view from left to right.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N).

Constraints:
$1 \le N \le 10^5$
$1 \le$ Node Data $\le 10^5$

Note: The **Input/Ouput** format and **Example** given are used for system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from stdin/console. The task is to complete the function specified, and not to write the full code.



```
    Node* left;
    Node* right;
};
*/
class Solution
{
    public:
    //Function to return a list of nodes visible from the top view
    //from left to right in Binary Tree.
    void solve(Node* root, map<int, pair<int, int>>& levelNode, int horizontalLevel, int verticalLevel){
        if(root == NULL)
            return;
        if(levelNode.find(horizontalLevel) == levelNode.end())
            levelNode[horizontalLevel] = make_pair(root->data, verticalLevel);

        else if(levelNode.find(horizontalLevel) != levelNode.end() and (verticalLevel < levelNode[horizontalLevel].second))
            levelNode[horizontalLevel] = make_pair(root->data, verticalLevel);

        solve(root->left, levelNode, horizontalLevel - 1, verticalLevel + 1);
        solve(root->right, levelNode, horizontalLevel + 1, verticalLevel + 1);
    }
    vector<int> topView(Node *root)
    {
        //Your code here
        vector<int> ans, ans2;
        map<int, pair<int, int>> levelNode;
        solve(root, levelNode, 0, 0);

        for(auto node : levelNode){
            ans.push_back(node.second.first);
            cout << node.second.first << " ";
        }
        return ans2;
    }

};


} // } Driver Code Ends
```

Bottom View of Binary Tree

Medium Accuracy: 45.32% Submissions: 100k+ Points: 4

Given a binary tree, print the bottom view from left to right.
A node is included in bottom view if it can be seen when we look at the tree from bottom.

```
                      20
                    /    \
                  8       22
                /   \        \
              5      3       25
                    /   \
                  10    14
```

For the above tree, the bottom view is 5 10 3 14 25.
If there are **multiple** bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.

```
                      20
                    /    \
      8      22
    /  \    /  \
  5      3 4     25
        /   \
     10      14
```

For the above tree the output should be 5 10 4 14 25.

Example 1:

Input:

```
      1
    /   \
  3      2
```

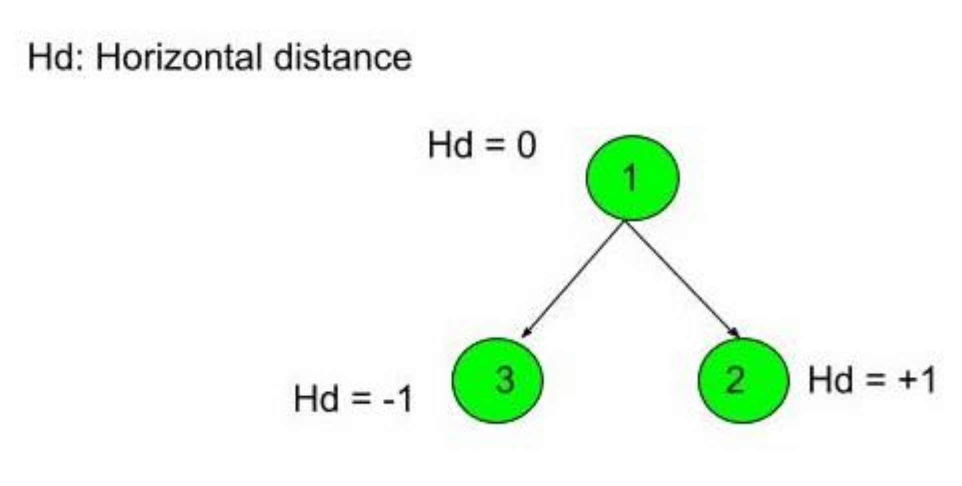
Output: 3 1 2

Explanation:

First case represents a tree with 3 nodes

and 2 edges where root is 1, left child of

1 is 3 and right child of 1 is 2.



Thus nodes of the binary tree will be

printed as such 3 1 2.

Example 2:

Input:

```
        10
      /    \
    20   30
   / \
  40   60
```

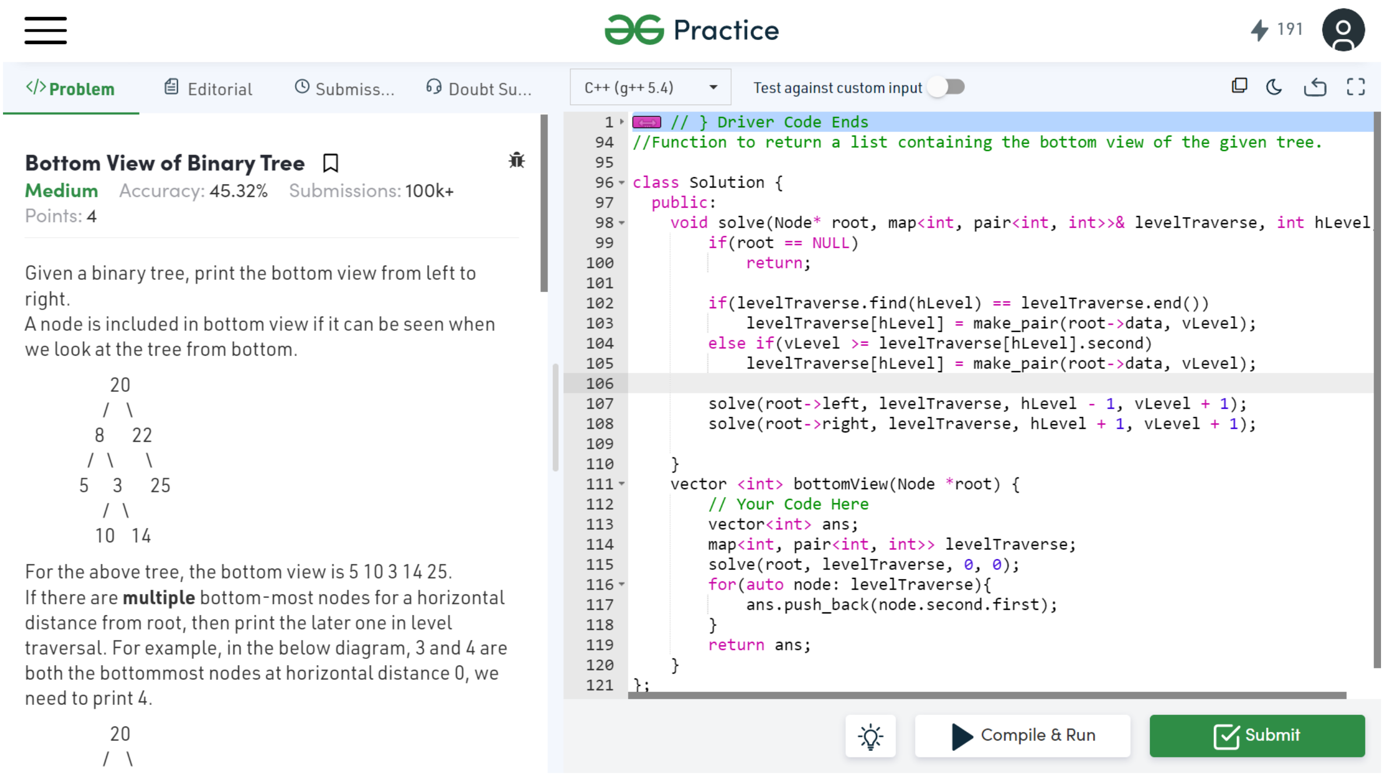
Output: 40 20 60 30

Your Task:
This is a functional problem, you **don't** need to care about input, just complete the function **bottomView**() which takes the root node of the tree as input and returns an array containing the bottom view of the given tree.

Expected Time Complexity: O(N).
Expected Auxiliary Space: O(N).

Constraints:
1 <= Number of nodes <= 10^5
1 <= Data of a node <= 10^5

Note: The **Input/Output** format and **Example** given are used for the system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from the stdin/console. The task is to complete the function specified, and not to write the full code.



Bottom View of Binary Tree
Medium Accuracy: 45.32% Submissions: 100k+
Points: 4

Given a binary tree, print the bottom view from left to right.
A node is included in bottom view if it can be seen when we look at the tree from bottom.

```
          20
        /    \
      8       22
    /   \      \
  5      3      25
        / \
      10    14
```

For the above tree, the bottom view is 5 10 3 14 25.
If there are **multiple** bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.

```
          20
        /    \
```

```
// } Driver Code Ends
//Function to return a list containing the bottom view of the given tree.

class Solution {
  public:
    void solve(Node* root, map<int, pair<int, int>>& levelTraverse, int hLevel
        if(root == NULL)
            return;

        if(levelTraverse.find(hLevel) == levelTraverse.end())
            levelTraverse[hLevel] = make_pair(root->data, vLevel);
        else if(vLevel >= levelTraverse[hLevel].second)
            levelTraverse[hLevel] = make_pair(root->data, vLevel);

        solve(root->left, levelTraverse, hLevel - 1, vLevel + 1);
        solve(root->right, levelTraverse, hLevel + 1, vLevel + 1);

    }
    vector <int> bottomView(Node *root) {
        // Your Code Here
        vector<int> ans;
        map<int, pair<int, int>> levelTraverse;
        solve(root, levelTraverse, 0, 0);
        for(auto node: levelTraverse){
            ans.push_back(node.second.first);
        }
        return ans;
    }
};
```

ZigZag Tree Traversal
Easy Accuracy: 49.78% Submissions: 62361 Points: 2

Given a Binary Tree. Find the Zig-Zag Level Order Traversal of the Binary Tree.

Example 1:

Input:

```
      3
    /   \
   2     1
```

Output:

```
3 1 2
```

Example 2:

Input:

```
          7
        /   \
       9     7
     /  \   /
    8    8 6
   /  \
  10   9
```

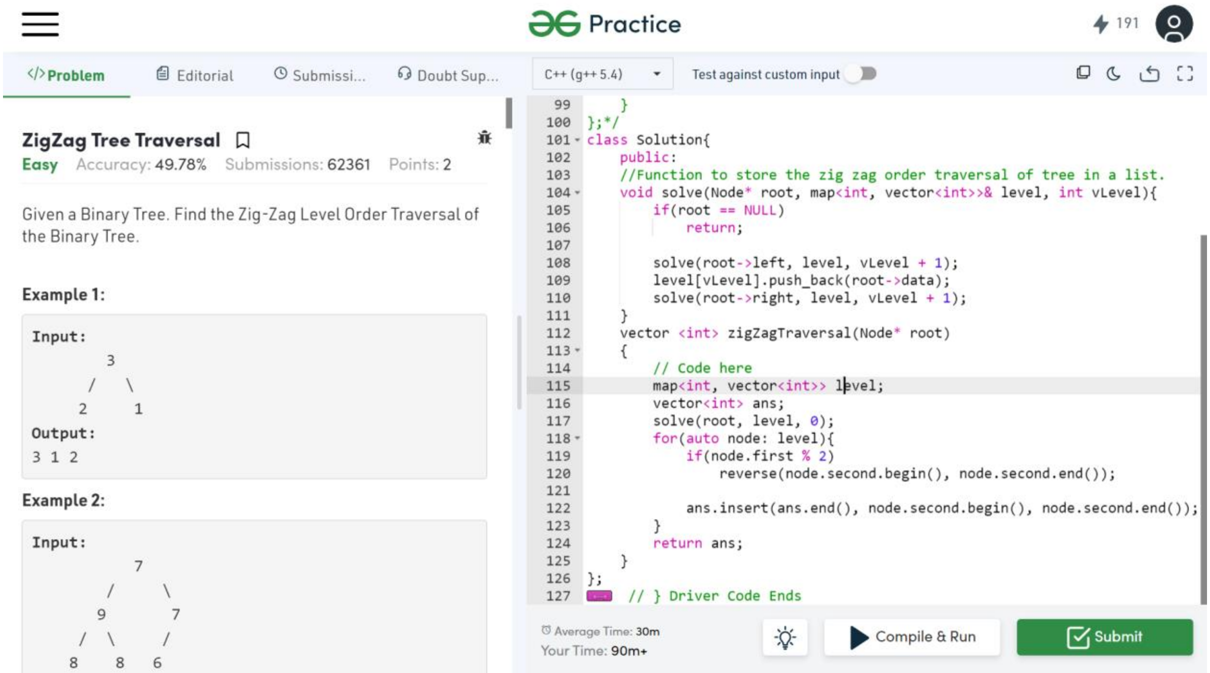
Output:

```
7 7 9 8 8 6 9 10
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **zigZagTraversal()** which takes the root node of the Binary Tree as its input and returns a list containing the node values as they appear in the Zig-Zag Level-Order Traversal of the Tree.

Expected Time Complexity: O(N).
Expected Auxiliary Space: O(N).

Constraints:
1 <= N <= 10^4



ZigZag Tree Traversal

Easy Accuracy: 49.78% Submissions: 62361 Points: 2

Given a Binary Tree. Find the Zig-Zag Level Order Traversal of the Binary Tree.

Example 1:

```
Input:
         3
       /   \
      2     1
Output:
3 1 2
```

Example 2:

```
Input:
           7
        /     \
       9       7
     /  \     /
    8    8   6
```

```
    }
};*/
class Solution{
    public:
    //Function to store the zig zag order traversal of tree in a list.
    void solve(Node* root, map<int, vector<int>>& level, int vLevel){
        if(root == NULL)
            return;

        solve(root->left, level, vLevel + 1);
        level[vLevel].push_back(root->data);
        solve(root->right, level, vLevel + 1);
    }
    vector <int> zigZagTraversal(Node* root)
    {
        // Code here
        map<int, vector<int>> level;
        vector<int> ans;
        solve(root, level, 0);
        for(auto node: level){
            if(node.first % 2)
                reverse(node.second.begin(), node.second.end());

            ans.insert(ans.end(), node.second.begin(), node.second.end());
        }
        return ans;
    }
};
 // } Driver Code Ends
```

Check for Balanced Tree

Easy Accuracy: 50.11% Submissions: 100k+ Points: 2

Given a binary tree, find if it is height balanced or not.
A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

A height balanced tree

```
        1
     /     \
   10      39
  /
5
```

An unbalanced tree

```
        1
     /
   10
  /
5
```

Example 1:

Input:

```
    1
   /
  2
   \
    3
```

Output: 0

Explanation: The max difference in height

of left subtree and right subtree is 2,

which is greater than 1. Hence unbalanced

Example 2:

Input:

```
      10
     /   \
    20   30
  /   \
 40   60
```

Output: 1

Explanation: The max difference in height

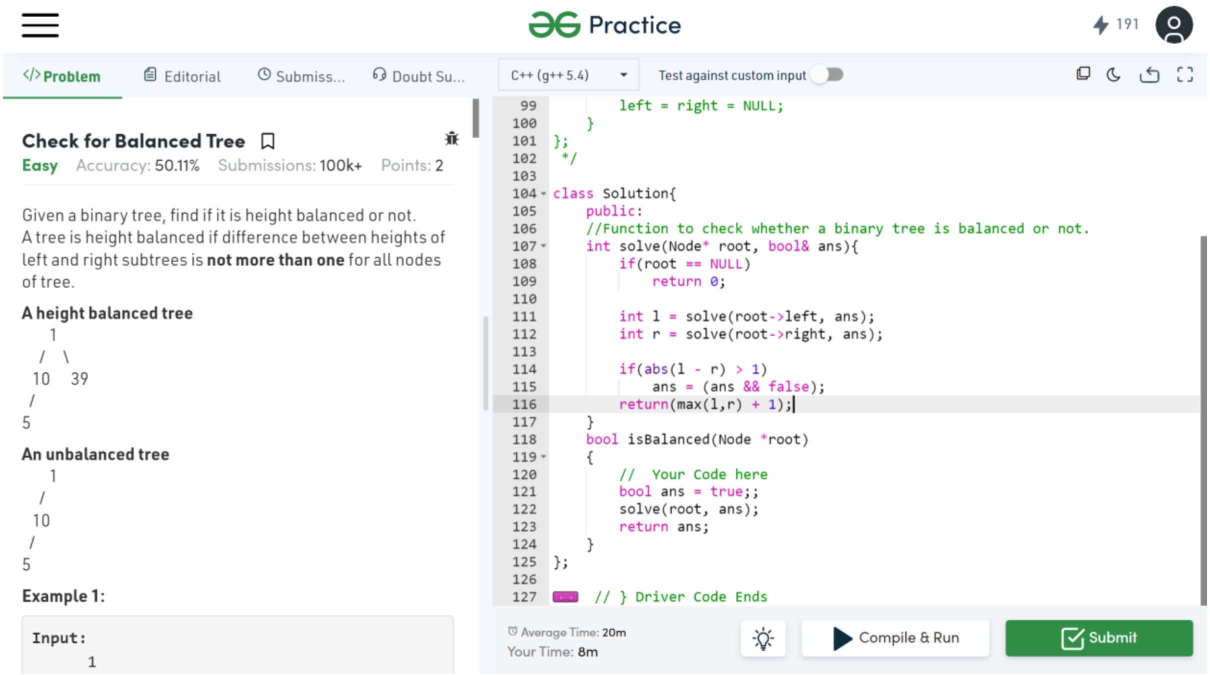
of left subtree and right subtree is 1.

Hence balanced.

Your Task:
You don't need to take input. Just complete the function **isBalanced()** that takes root **node** as parameter and returns **true,** if the tree is balanced else returns **false**.

Constraints:
1 <= Number of nodes <= 10^5
0 <= Data of a node <= 10^6

Expected time complexity: O(N)
Expected auxiliary space: O(h) , where h = height of tree



Check for Balanced Tree

Easy Accuracy: 50.11% Submissions: 100k+ Points: 2

Given a binary tree, find if it is height balanced or not. A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

A height balanced tree

```
     1
   /   \
  10   39
 /
5
```

An unbalanced tree

```
     1
   /
  10
 /
5
```

Example 1:

```
Input:
      1
```

```
            left = right = NULL;
        }
};
 */

class Solution{
    public:
    //Function to check whether a binary tree is balanced or not.
    int solve(Node* root, bool& ans){
        if(root == NULL)
            return 0;

        int l = solve(root->left, ans);
        int r = solve(root->right, ans);

        if(abs(l - r) > 1)
            ans = (ans && false);
        return(max(l,r) + 1);
    }
    bool isBalanced(Node *root)
    {
        //  Your Code here
        bool ans = true;;
        solve(root, ans);
        return ans;
    }
};

 // } Driver Code Ends
```

Average Time: 20m
Your Time: 8m

509. Fibonacci Number

Easy

3351262Add to ListShare

The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

```
F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given n, calculate F(n).

Example 1:

```
Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Example 2:

Input: n = 3

Output: 2

Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Example 3:

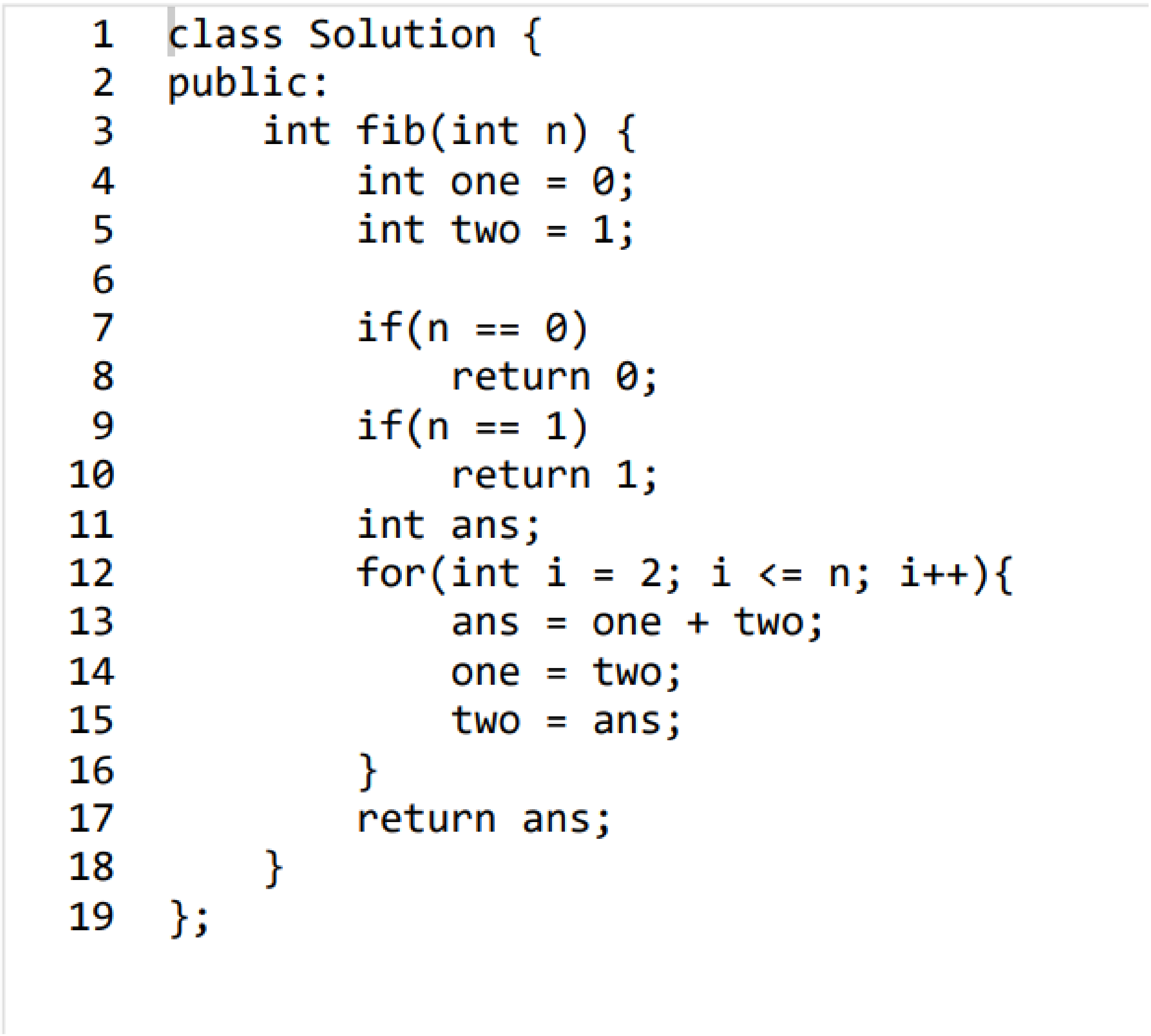
Input: n = 4

Output: 3

Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.

Constraints:

- `0 <= n <= 30`



```
class Solution {
public:
    int fib(int n) {
        int one = 0;
        int two = 1;

        if(n == 0)
            return 0;
        if(n == 1)
            return 1;
        int ans;
        for(int i = 2; i <= n; i++){
            ans = one + two;
            one = two;
            two = ans;
        }
        return ans;
    }
};
```

70. Climbing Stairs

Easy

11502359Add to ListShare

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

Input: n = 2

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

Example 2:

Input: n = 3

Output: 3

Explanation: There are three ways to climb to the top.

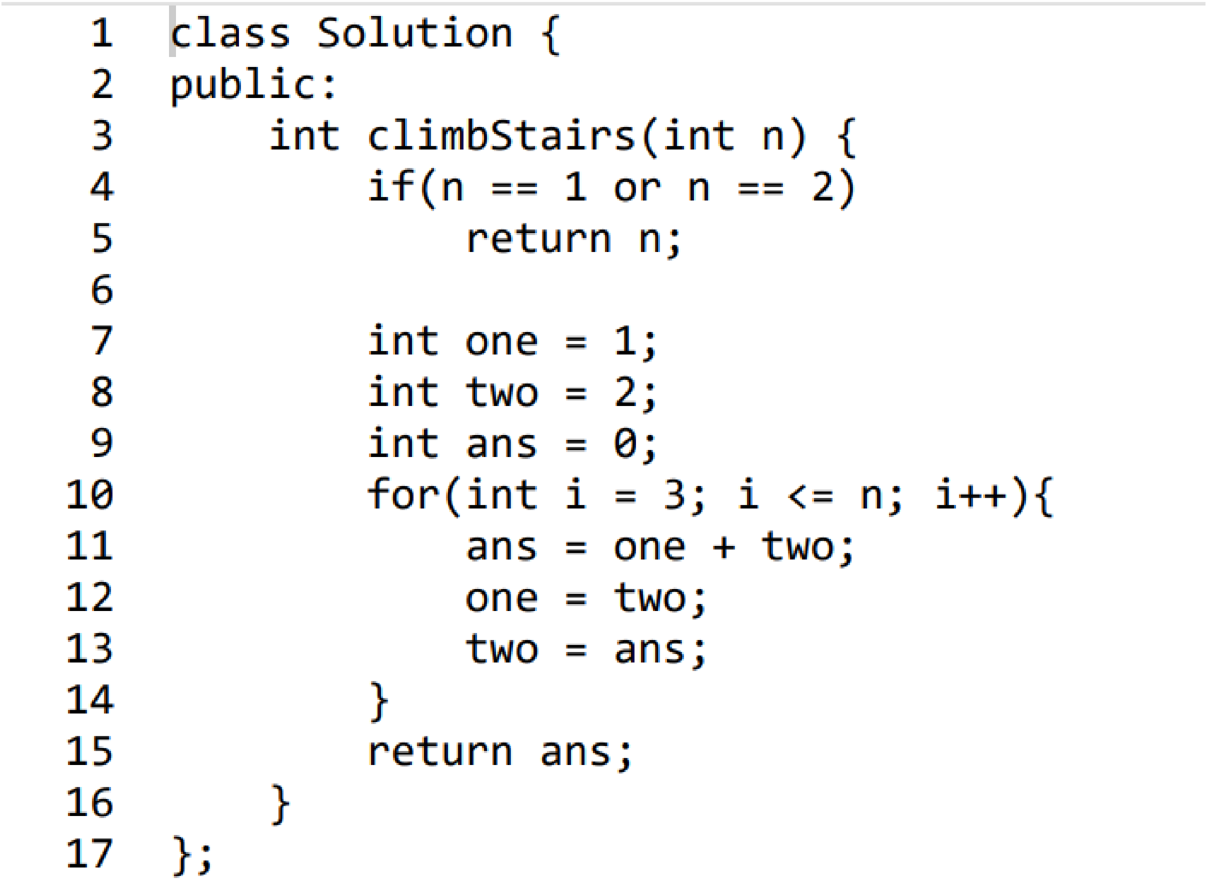
1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

Constraints:

- `1 <= n <= 45`



```
class Solution {
public:
    int climbStairs(int n) {
        if(n == 1 or n == 2)
            return n;

        int one = 1;
        int two = 2;
        int ans = 0;
        for(int i = 3; i <= n; i++){
            ans = one + two;
            one = two;
            two = ans;
        }
        return ans;
    }
};
```

652. Find Duplicate Subtrees

Medium

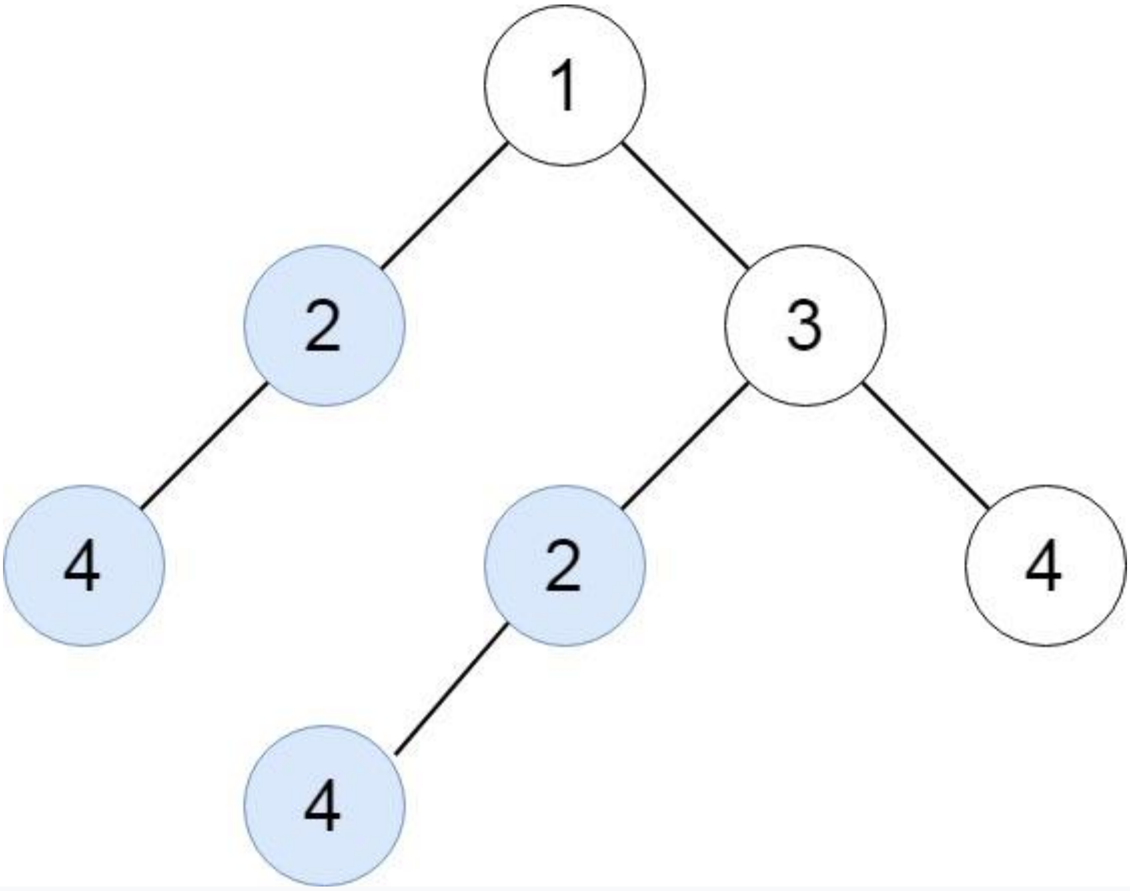
3189294Add to ListShare

Given the root of a binary tree, return all **duplicate subtrees**.

For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

Two trees are **duplicate** if they have the **same structure** with the **same node values**.

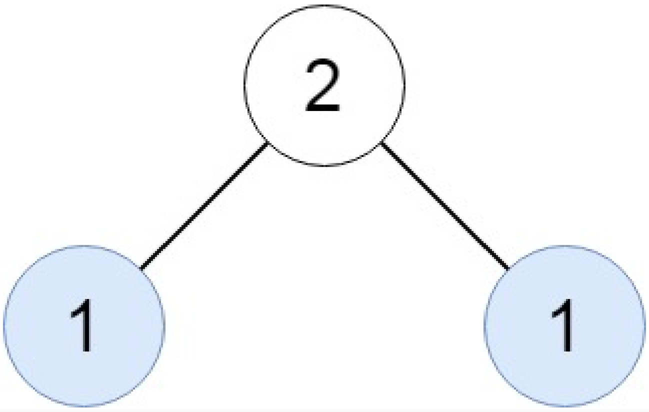
Example 1:



Input: root = [1,2,3,4,null,2,4,null,null,4]

Output: [[2,4],[4]]

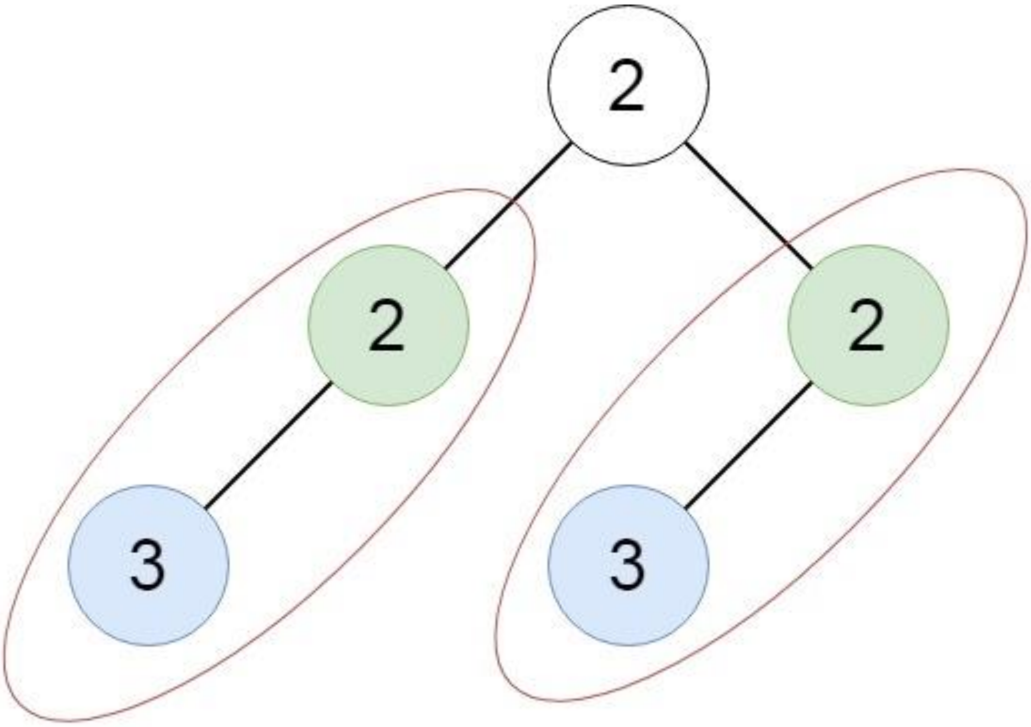
Example 2:



Input: root = [2,1,1]

Output: [[1]]

Example 3:

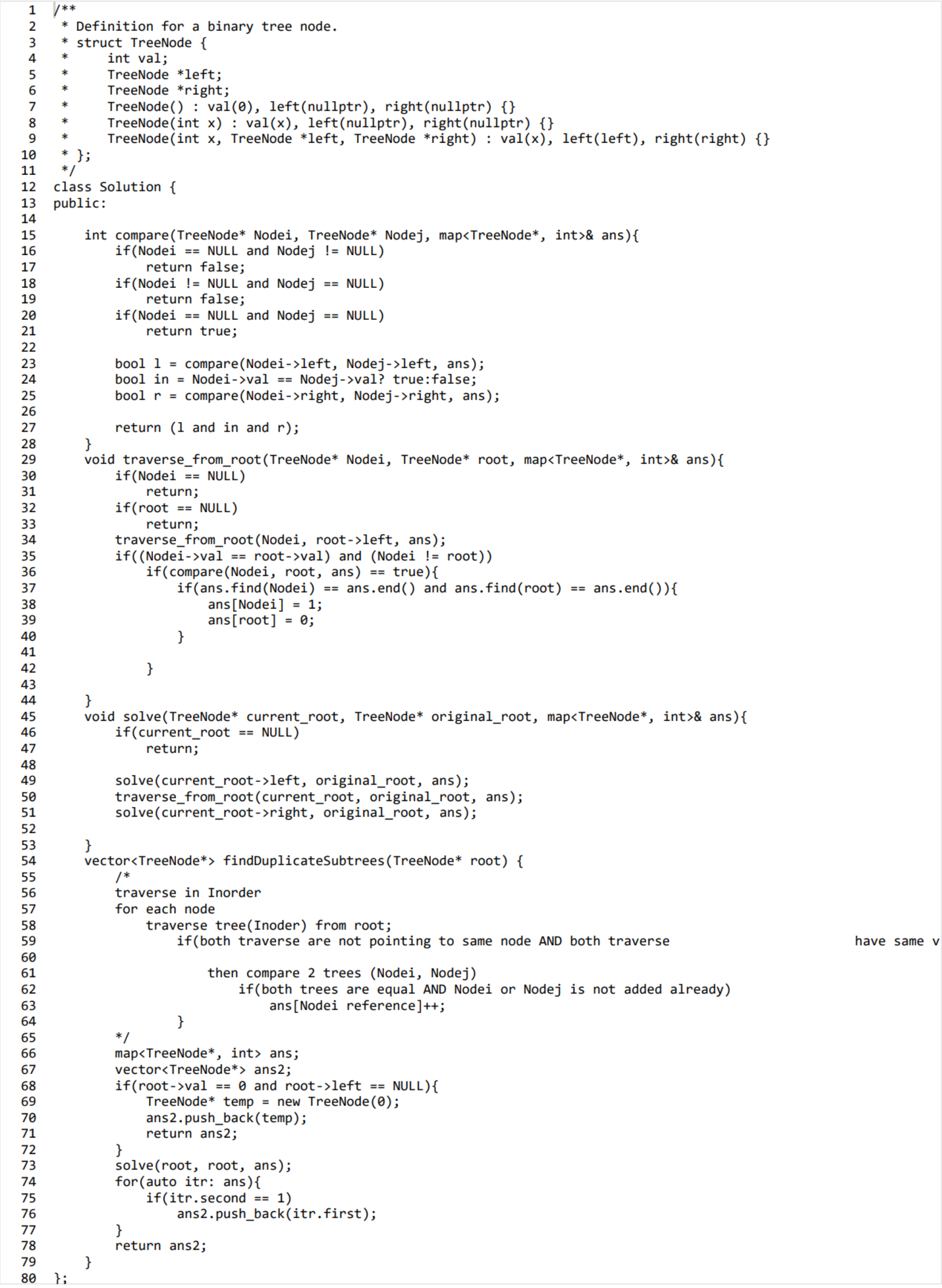


Input: root = [2,2,2,3,null,3,null]

Output: [[2,3],[3]]

Constraints:

- The number of the nodes in the tree will be in the range [1, 10^4]
- -200 <= Node.val <= 200



```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:

    int compare(TreeNode* Nodei, TreeNode* Nodej, map<TreeNode*, int>& ans){
        if(Nodei == NULL and Nodej != NULL)
            return false;
        if(Nodei != NULL and Nodej == NULL)
            return false;
        if(Nodei == NULL and Nodej == NULL)
            return true;

        bool l = compare(Nodei->left, Nodej->left, ans);
        bool in = Nodei->val == Nodej->val? true:false;
        bool r = compare(Nodei->right, Nodej->right, ans);

        return (l and in and r);
    }
    void traverse_from_root(TreeNode* Nodei, TreeNode* root, map<TreeNode*, int>& ans){
        if(Nodei == NULL)
            return;
        if(root == NULL)
            return;
        traverse_from_root(Nodei, root->left, ans);
        if((Nodei->val == root->val) and (Nodei != root))
            if(compare(Nodei, root, ans) == true){
                if(ans.find(Nodei) == ans.end() and ans.find(root) == ans.end()){
                    ans[Nodei] = 1;
                    ans[root] = 0;
                }

            }

    }
    void solve(TreeNode* current_root, TreeNode* original_root, map<TreeNode*, int>& ans){
        if(current_root == NULL)
            return;

        solve(current_root->left, original_root, ans);
        traverse_from_root(current_root, original_root, ans);
        solve(current_root->right, original_root, ans);

    }
    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        /*
        traverse in Inorder
        for each node
            traverse tree(Inoder) from root;
                if(both traverse are not pointing to same node AND both traverse                        have same v

                    then compare 2 trees (Nodei, Nodej)
                        if(both trees are equal AND Nodei or Nodej is not added already)
                            ans[Nodei reference]++;
                }
        */
        map<TreeNode*, int> ans;
        vector<TreeNode*> ans2;
        if(root->val == 0 and root->left == NULL){
            TreeNode* temp = new TreeNode(0);
            ans2.push_back(temp);
            return ans2;
        }
        solve(root, root, ans);
        for(auto itr: ans){
            if(itr.second == 1)
                ans2.push_back(itr.first);
        }
        return ans2;
    }
};
```

236. Lowest Common Ancestor of a Binary Tree

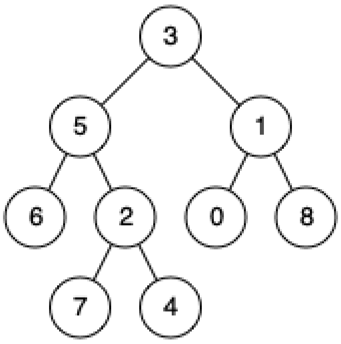
Medium

9556274Add to ListShare

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:

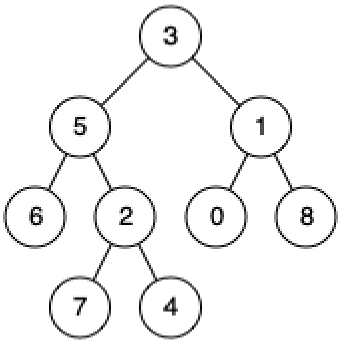


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

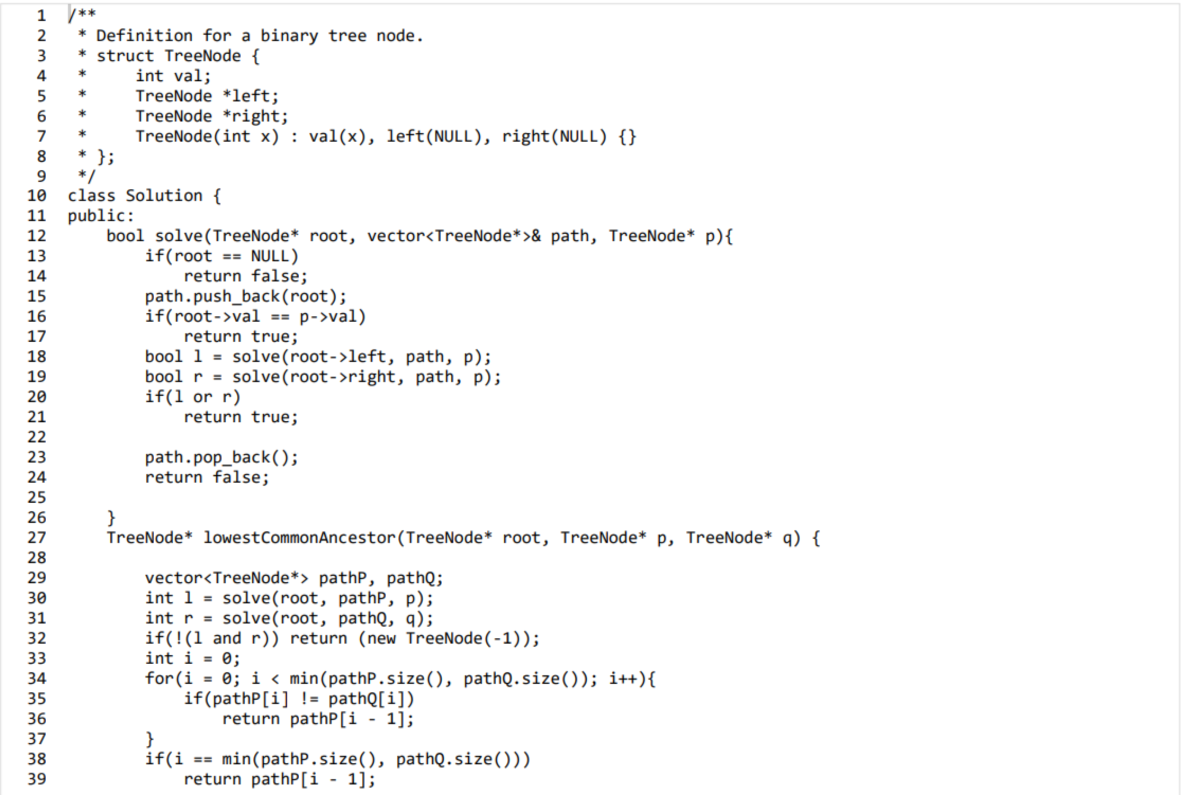
Example 3:

Input: root = [1,2], p = 1, q = 2

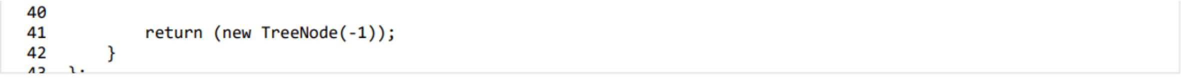
Output: 1

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- -10^9 <= Node.val <= 10^9
- All Node.val are **unique**.
- p != q
- p and q will exist in the tree.



```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool solve(TreeNode* root, vector<TreeNode*>& path, TreeNode* p){
        if(root == NULL)
            return false;
        path.push_back(root);
        if(root->val == p->val)
            return true;
        bool l = solve(root->left, path, p);
        bool r = solve(root->right, path, p);
        if(l or r)
            return true;

        path.pop_back();
        return false;

    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

        vector<TreeNode*> pathP, pathQ;
        int l = solve(root, pathP, p);
        int r = solve(root, pathQ, q);
        if(!(l and r)) return (new TreeNode(-1));
        int i = 0;
        for(i = 0; i < min(pathP.size(), pathQ.size()); i++){
            if(pathP[i] != pathQ[i])
                return pathP[i - 1];
        }
        if(i == min(pathP.size(), pathQ.size()))
            return pathP[i - 1];
```



```

        return (new TreeNode(-1));
    }
};
```

Leaf at same level

Easy Accuracy: 49.76% Submissions: 43820 Points: 2

Given a Binary Tree, check if all leaves are at same level or not.

Example 1:

```
Input:
            1
          /   \
         2     3

Output: 1

Explanation:
Leaves 2 and 3 are at same level.
```

Example 2:

```
Input:
            10
          /    \
        20      30
       /  \
     10    15

Output: 0
```

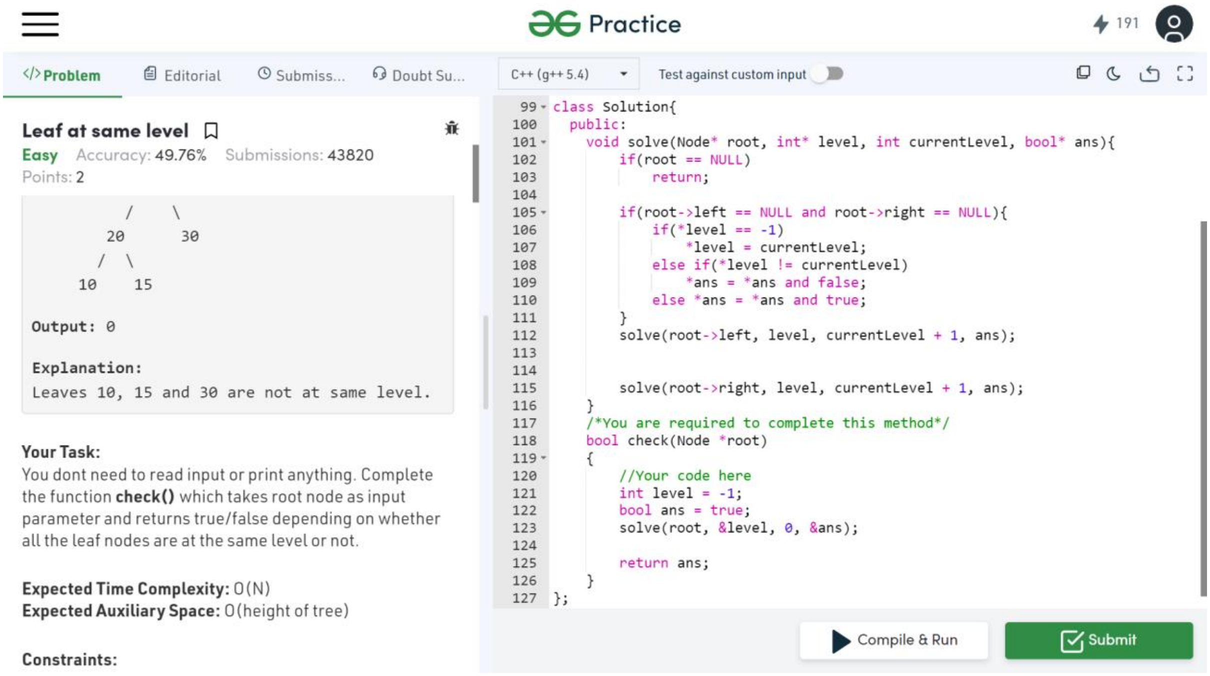
Explanation:

Leaves 10, 15 and 30 are not at same level.

Your Task:
You dont need to read input or print anything. Complete the function **check()** which takes root node as input parameter and returns true/false depending on whether all the leaf nodes are at the same level or not.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(height of tree)

Constraints:
1 ≤ N ≤ 10^3



Leaf at same level
Easy Accuracy: 49.76% Submissions: 43820
Points: 2

```
         /     \
       20       30
      /  \
    10    15

Output: 0

Explanation:
Leaves 10, 15 and 30 are not at same level.
```

Your Task:
You dont need to read input or print anything. Complete the function **check()** which takes root node as input parameter and returns true/false depending on whether all the leaf nodes are at the same level or not.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(height of tree)

Constraints:

```cpp
class Solution{
  public:
    void solve(Node* root, int* level, int currentLevel, bool* ans){
        if(root == NULL)
            return;

        if(root->left == NULL and root->right == NULL){
            if(*level == -1)
                *level = currentLevel;
            else if(*level != currentLevel)
                *ans = *ans and false;
            else *ans = *ans and true;
        }
        solve(root->left, level, currentLevel + 1, ans);


        solve(root->right, level, currentLevel + 1, ans);
    }
    /*You are required to complete this method*/
    bool check(Node *root)
    {
        //Your code here
        int level = -1;
        bool ans = true;
        solve(root, &level, 0, &ans);

        return ans;
    }
};
```

Sum Tree

Medium Accuracy: 33.33% Submissions: 100k+ Points: 4

Given a Binary Tree. Return **true** if, for every node **X** in the tree other than the leaves, its value is equal to the sum of its left subtree's value and its right subtree's value. Else return **false**.

An empty tree is also a Sum Tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a Sum Tree.

Example 1:

Input:

```
   3
 /   \
1     2
```

Output: 1

Explanation:

The sum of left subtree and right subtree is

1 + 2 = 3, which is the value of the root node.

Therefore,the given binary tree is a **sum tree**.

Example 2:

Input:

```
        10
      /    \
    20      30
  /   \
 10    10
```

Output: 0

Explanation:

The given tree is not a sum tree.

For the root node, sum of elements

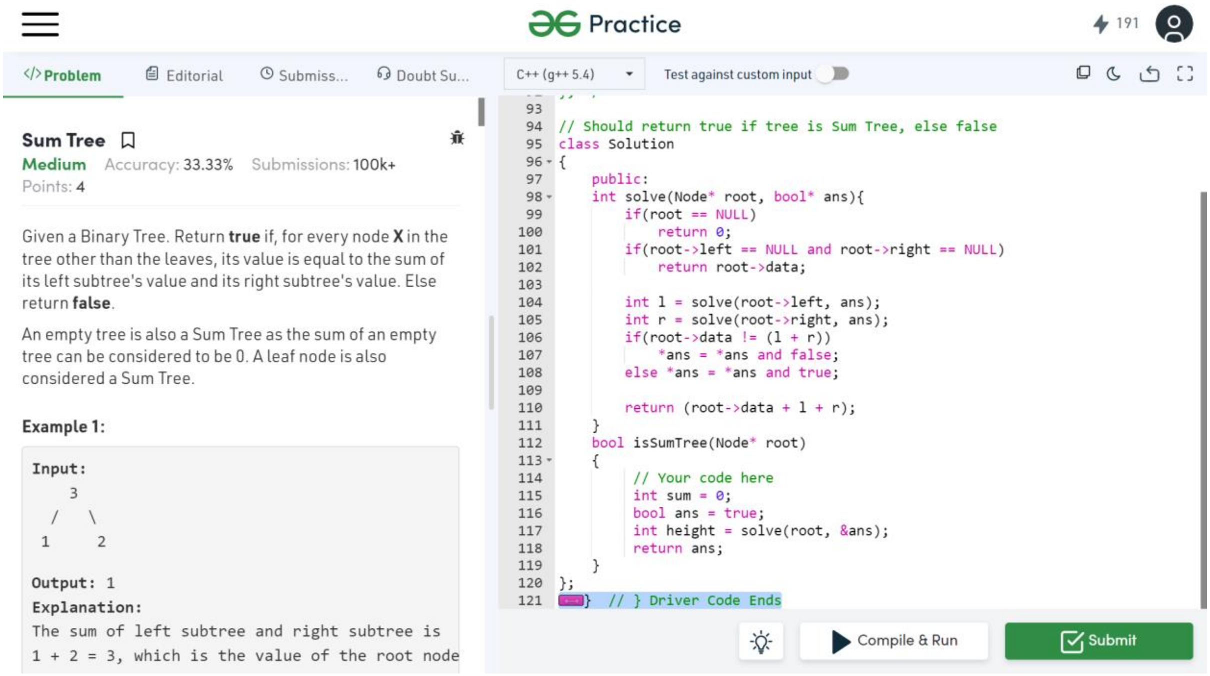
in left subtree is 40 and sum of elements

```
in right subtree is 30. Root element = 10

which is not equal to 30+40.
```

Your Task:
You don't need to read input or print anything. Complete the function **isSumTree()** which takes **root** node as input parameter and returns true if the tree is a SumTree else it returns false.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(Height of the Tree)

Constraints:
1 ≤ number of nodes ≤ 10^4



Sum Tree

Medium Accuracy: 33.33% Submissions: 100k+ Points: 4

Given a Binary Tree. Return **true** if, for every node **X** in the tree other than the leaves, its value is equal to the sum of its left subtree's value and its right subtree's value. Else return **false**.

An empty tree is also a Sum Tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a Sum Tree.

Example 1:

```
Input:
    3
  /   \
 1     2

Output: 1
Explanation:
The sum of left subtree and right subtree is
1 + 2 = 3, which is the value of the root node
```

```cpp

// Should return true if tree is Sum Tree, else false
class Solution
{
    public:
    int solve(Node* root, bool* ans){
        if(root == NULL)
            return 0;
        if(root->left == NULL and root->right == NULL)
            return root->data;

        int l = solve(root->left, ans);
        int r = solve(root->right, ans);
        if(root->data != (l + r))
            *ans = *ans and false;
        else *ans = *ans and true;

        return (root->data + l + r);
    }
    bool isSumTree(Node* root)
    {
         // Your code here
         int sum = 0;
         bool ans = true;
         int height = solve(root, &ans);
         return ans;
    }
};
} // } Driver Code Ends
```

Reverse Level Order Traversal

Easy Accuracy: 47.34% Submissions: 71039 Points: 2

Given a binary tree of size N, find its reverse level order traversal. ie- the traversal must begin from the last level.

Example 1:

Input :

```
      1
    /   \
   3     2
```

Output: 3 2 1

Explanation:

Traversing level 1 : 3 2

Traversing level 0 : 1

Example 2:

Input :

```
      10
    /  \
   20   30
  / \
 40  60
```

Output: 40 60 20 30 10

Explanation:

Traversing level 2 : 40 60

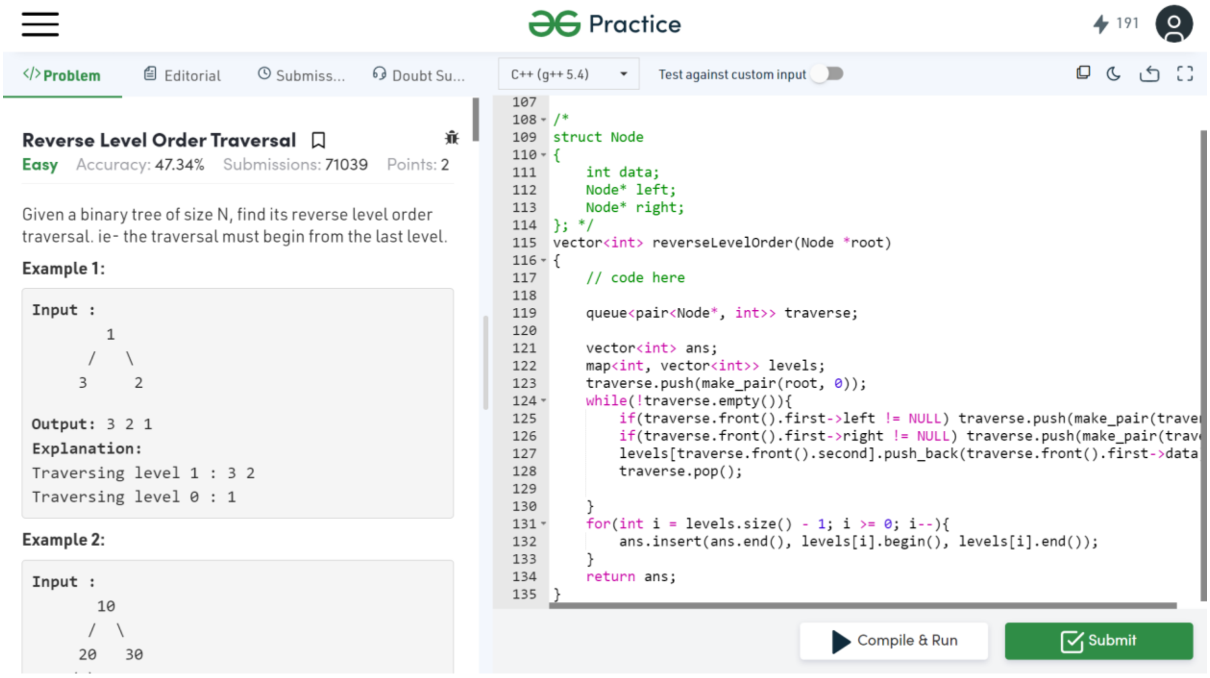
Traversing level 1 : 20 30

Traversing level 0 : 10

Your Task:
You dont need to read input or print anything. Complete the function **reverseLevelOrder()** which takes the root of the tree as input parameter and returns a list containing the reverse level order traversal of the given tree.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

Constraints:
1 ≤ N ≤ 10^4



Reverse Level Order Traversal

Easy Accuracy: 47.34% Submissions: 71039 Points: 2

Given a binary tree of size N, find its reverse level order traversal. ie- the traversal must begin from the last level.

Example 1:

```
Input :
        1
      /   \
     3     2

Output: 3 2 1
Explanation:
Traversing level 1 : 3 2
Traversing level 0 : 1
```

Example 2:

```
Input :
       10
      /  \
     20   30
```

```
/*
struct Node
{
    int data;
    Node* left;
    Node* right;
}; */
vector<int> reverseLevelOrder(Node *root)
{
    // code here

    queue<pair<Node*, int>> traverse;

    vector<int> ans;
    map<int, vector<int>> levels;
    traverse.push(make_pair(root, 0));
    while(!traverse.empty()){
        if(traverse.front().first->left != NULL) traverse.push(make_pair(trave
        if(traverse.front().first->right != NULL) traverse.push(make_pair(trav
        levels[traverse.front().second].push_back(traverse.front().first->data
        traverse.pop();

    }
    for(int i = levels.size() - 1; i >= 0; i--){
        ans.insert(ans.end(), levels[i].begin(), levels[i].end());
    }
    return ans;
}
```

First and last occurrences of x

Basic Accuracy: 53.04% Submissions: 62293 Points: 1

Given a sorted array **arr** containing **n** elements with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element **x** in the given array.

Example 1:

Input:

n=9, x=5

arr[] = { 1, 3, 5, 5, 5, 5, 67, 123, 125 }

Output: 2 5

Explanation: First occurrence of 5 is at index 2 and last

occurrence of 5 is at index 5.

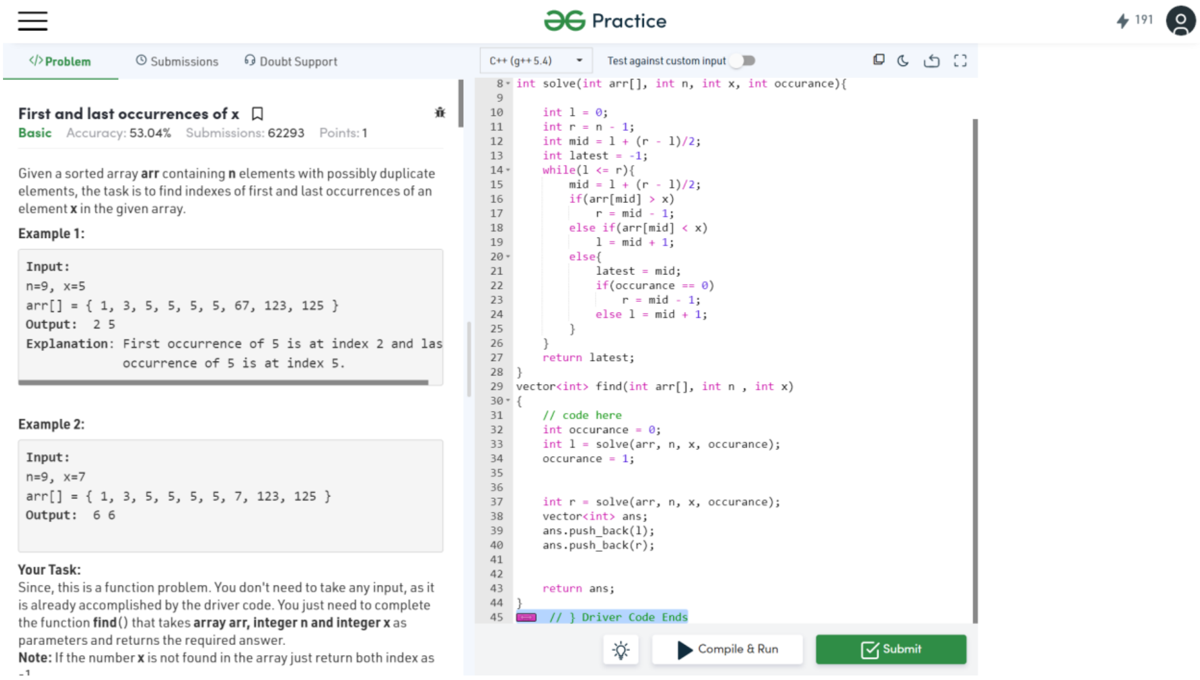
Example 2:

```
Input:
n=9, x=7
arr[] = { 1, 3, 5, 5, 5, 5, 7, 123, 125 }
Output:  6 6
```

Your Task:
Since, this is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function **find**() that takes **array arr, integer n and integer x** as parameters and returns the required answer.
Note: If the number **x** is not found in the array just return both index as -1.

Expected Time Complexity: O(logN)
Expected Auxiliary Space: O(1).

Constraints:
$1 \le N \le 10^7$



First and last occurrences of x
Basic Accuracy: 53.04% Submissions: 62293 Points: 1

Given a sorted array **arr** containing **n** elements with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element **x** in the given array.

Example 1:

```
Input:
n=9, x=5
arr[] = { 1, 3, 5, 5, 5, 5, 67, 123, 125 }
Output:  2 5
Explanation: First occurrence of 5 is at index 2 and las
             occurrence of 5 is at index 5.
```

Example 2:

```
Input:
n=9, x=7
arr[] = { 1, 3, 5, 5, 5, 5, 7, 123, 125 }
Output:  6 6
```

Your Task:
Since, this is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function **find**() that takes **array arr, integer n and integer x** as parameters and returns the required answer.
Note: If the number **x** is not found in the array just return both index as -1

```cpp
int solve(int arr[], int n, int x, int occurance){

    int l = 0;
    int r = n - 1;
    int mid = l + (r - l)/2;
    int latest = -1;
    while(l <= r){
        mid = l + (r - l)/2;
        if(arr[mid] > x)
            r = mid - 1;
        else if(arr[mid] < x)
            l = mid + 1;
        else{
            latest = mid;
            if(occurance == 0)
                r = mid - 1;
            else l = mid + 1;
        }
    }
    return latest;
}
vector<int> find(int arr[], int n , int x)
{
    // code here
    int occurance = 0;
    int l = solve(arr, n, x, occurance);
    occurance = 1;


    int r = solve(arr, n, x, occurance);
    vector<int> ans;
    ans.push_back(l);
    ans.push_back(r);


    return ans;
}
  // } Driver Code Ends
```

Search a node in BST
Basic Accuracy: 55.04% Submissions: 39391 Points: 1

Given a **Binary Search Tree** and a node value X, find if the node with value X is present in the BST or not.

Example 1:

```
Input:       2
            \
             81
           /   \
         42     87
          \      \
           66     90
          /
        45
X = 87
Output: 1
Explanation: As 87 is present in the
given nodes , so the output will be
1.
```

Example 2:

```
Input:     6
         \
          8
         / \
        7   9
```

X = 11

Output: 0

Explanation: As 11 is not present in

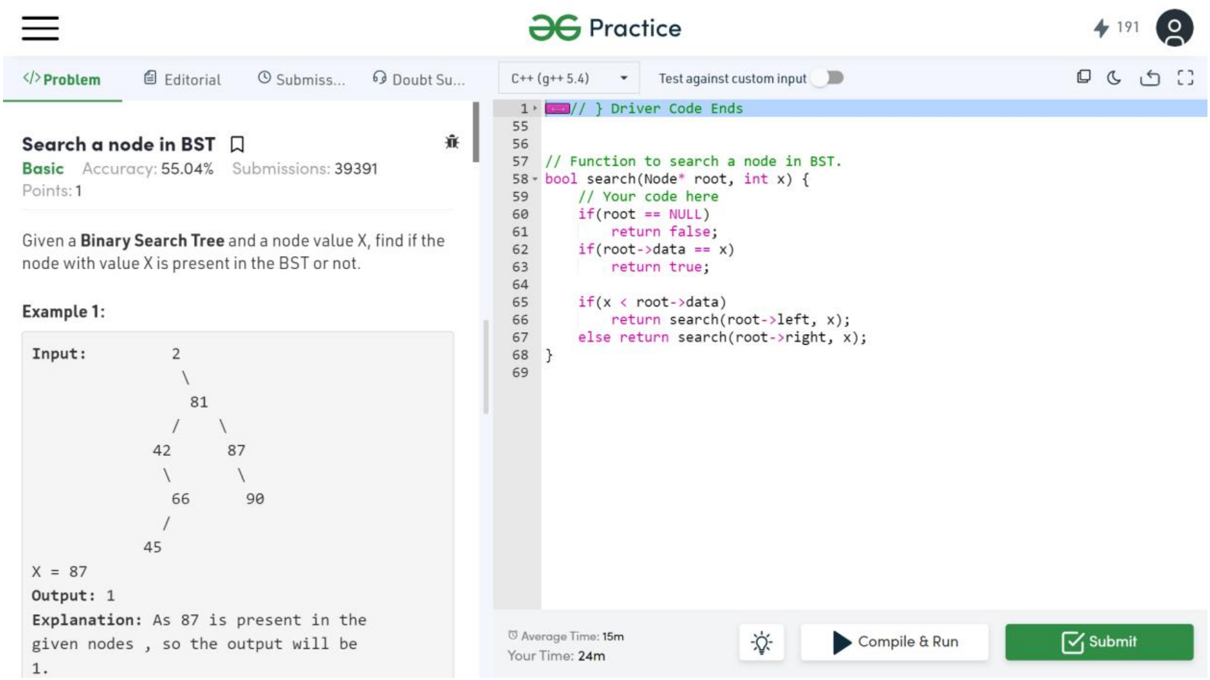
the given nodes , so the output will

be 0.

Your Task:
You don't need to read input or print anything. Complete the function **search()**which returns **true** if the node with **value x** is **present** in the BST**else returns false**.

Expected Time Complexity: O(Height of the BST)
Expected Auxiliary Space: O(1).

Constraints:
1 <= Number of nodes <= 10^5



Search a node in BST

Basic Accuracy: 55.04% Submissions: 39391
Points: 1

Given a **Binary Search Tree** and a node value X, find if the node with value X is present in the BST or not.

Example 1:

```
Input:          2
                 \
                  81
                /    \
              42      87
               \        \
                66       90
               /
             45
X = 87
Output: 1
Explanation: As 87 is present in the
given nodes , so the output will be
1.
```

```cpp
// } Driver Code Ends


// Function to search a node in BST.
bool search(Node* root, int x) {
    // Your code here
    if(root == NULL)
        return false;
    if(root->data == x)
        return true;

    if(x < root->data)
        return search(root->left, x);
    else return search(root->right, x);
}

```

Average Time: 15m
Your Time: 24m

1137. N-th Tribonacci Number

Easy
1743104Add to ListShare
The Tribonacci sequence T_n is defined as follows: $T_0 = 0$, $T_1 = 1$, $T_2 = 1$, and $T_{n+3} = T_n + T_{n+1} + T_{n+2}$ for n >= 0.

Given n, return the value of T_n.

Example 1:

Input: n = 4 **Output:** 4

Explanation:

T_3 = 0 + 1 + 1 = 2

T_4 = 1 + 1 + 2 = 4

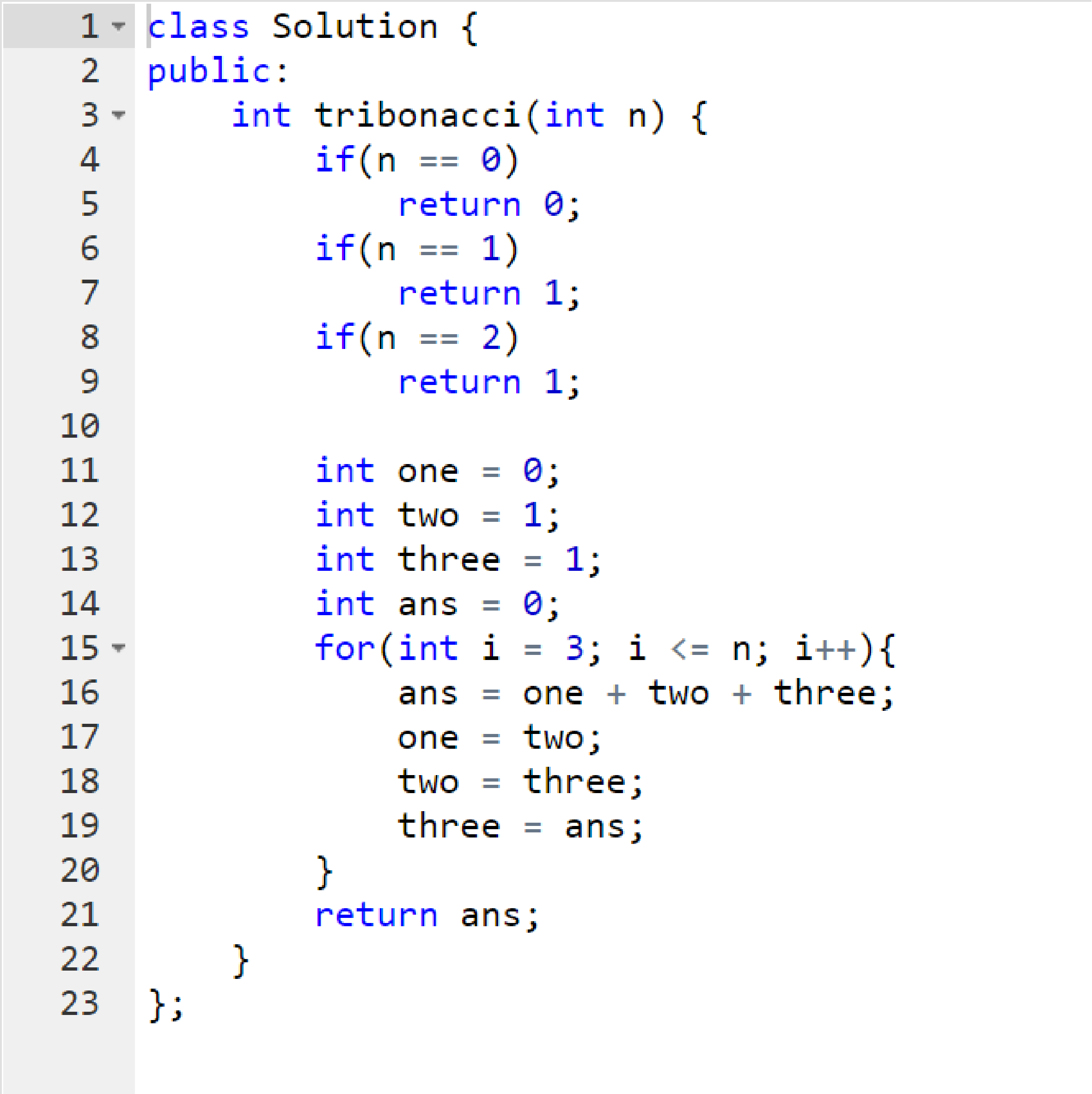
Example 2:

Input: n = 25

Output: 1389537

Constraints: 0 <= n <= 37

The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1.



```
class Solution {
public:
    int tribonacci(int n) {
        if(n == 0)
            return 0;
        if(n == 1)
            return 1;
        if(n == 2)
            return 1;

        int one = 0;
        int two = 1;
        int three = 1;
        int ans = 0;
        for(int i = 3; i <= n; i++){
            ans = one + two + three;
            one = two;
            two = three;
            three = ans;
        }
        return ans;
    }
};
```

Lowest Common Ancestor in a Binary Tree
Medium Accuracy: 39.75% Submissions: 98099 Points: 4

Given a Binary Tree with all **unique** values and two nodes value, **n1** and **n2**. The task is to find the **lowest common ancestor** of the given two nodes. We may assume that either both n1 and n2 are present in the tree or none of them are present.

Example 1:

Input:

n1 = 2 , n2 = 3

    1

```
   / \
  2   3
```

Output: 1

Explanation:

LCA of 2 and 3 is 1.

Example 2:

Input:

n1 = 3 , n2 = 4

```
        5
       /
      2
     / \
    3   4
```

Output: 2

Explanation:

LCA of 3 and 4 is 2.

Your Task:
You don't have to read, input, or print anything. Your task is to complete the function **lca()** that takes nodes, **n1, and n2** as parameters and returns the **LCA** node as output.

Expected Time Complexity:O(N).
Expected Auxiliary Space:O(Height of Tree).

Constraints:
1 ≤ Number of nodes ≤ 10^5
1 ≤ Data of a node ≤ 10^5

```
class Solution
{
    public:
    bool solve(Node* root, vector<Node*>& path, int n){
        if(root == NULL)
            return false;

        path.push_back(root);
        if(root->data == n)
```

```
            return true;

        bool l = solve(root->left, path, n);
        bool r = solve(root->right, path, n);

        if(l or r)
            return true;
        path.pop_back();

        return false;
    }
    //Function to return the lowest common ancestor in a Binary Tree.
    Node* lca(Node* root ,int n1 ,int n2 )
    {
       //Your code here
       vector<Node*> path1, path2;
       bool o = solve(root, path1, n1);
       bool t = solve(root, path2, n2);

       if(!(o and t)) return (new Node(-1));
       int i = 0;
       for(i = 0; i < min(path1.size(), path2.size()); i++){
           if(path1[i] != path2[i])
                return path1[i - 1];
       }
       return path1[i - 1];
    }
};
```

Lowest Common Ancestor in a BST

Easy Accuracy: 50.22% Submissions: 86321 Points: 2

Given a Binary Search Tree (with all values unique) and two node values. Find the Lowest Common Ancestors of the two nodes in the BST.

Example 1:

Input:

```
           5
         /   \
       4      6
      /        \
     3          7
                 \
                  8
```

n1 = 7, n2 = 8

Output: 7

Example 2:

Input:

```
    2
  /   \
 1     3
```

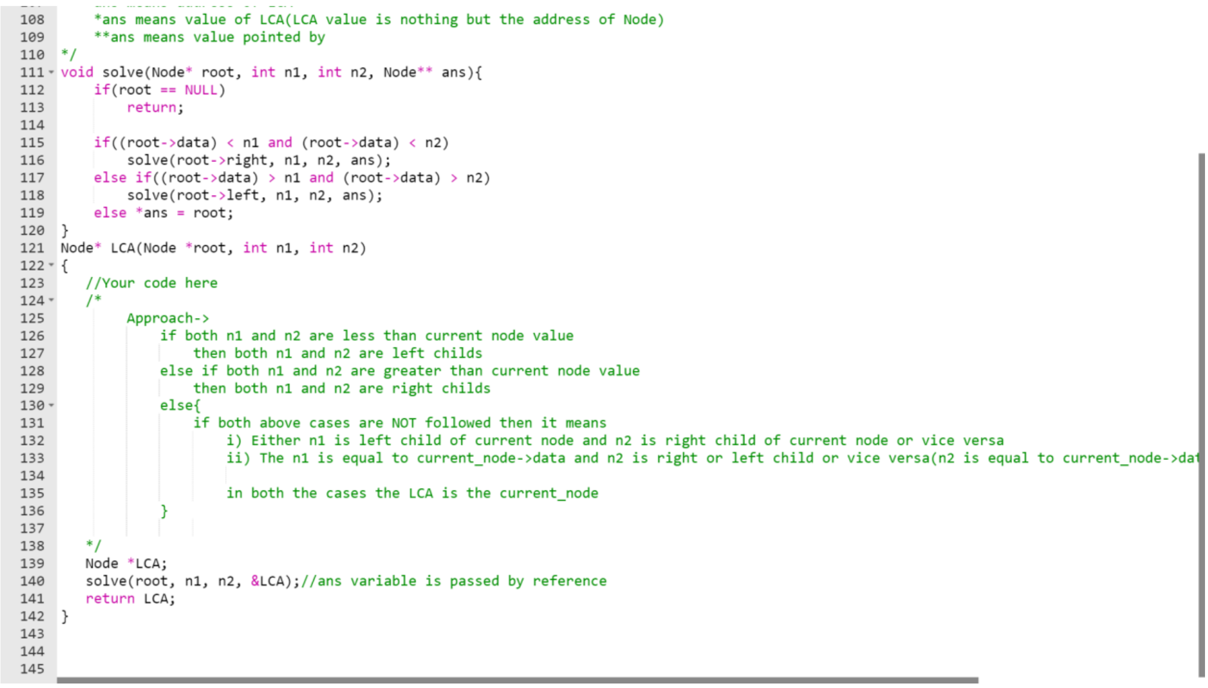
n1 = 1, n2 = 3

Output: 2

Your Task:
You don't need to read input or print anything. Your task is to complete the function **LCA()** which takes the root Node of the BST and two integer values n1 and n2 as inputs and returns the Lowest Common Ancestor of the Nodes with values n1 and n2 in the given BST.

Expected Time Complexity: O(Height of the BST).
Expected Auxiliary Space: O(Height of the BST).

Constraints:
1 <= N <= 10^4



```
    *ans means value of LCA(LCA value is nothing but the address of Node)
    **ans means value pointed by
*/
void solve(Node* root, int n1, int n2, Node** ans){
    if(root == NULL)
        return;

    if((root->data) < n1 and (root->data) < n2)
        solve(root->right, n1, n2, ans);
    else if((root->data) > n1 and (root->data) > n2)
        solve(root->left, n1, n2, ans);
    else *ans = root;
}
Node* LCA(Node *root, int n1, int n2)
{
   //Your code here
   /*
        Approach->
            if both n1 and n2 are less than current node value
                then both n1 and n2 are left childs
            else if both n1 and n2 are greater than current node value
                then both n1 and n2 are right childs
            else{
                if both above cases are NOT followed then it means
                    i) Either n1 is left child of current node and n2 is right child of current node or vice versa
                    ii) The n1 is equal to current_node->data and n2 is right or left child or vice versa(n2 is equal to current_node->dat

                    in both the cases the LCA is the current_node
            }

   */
   Node *LCA;
   solve(root, n1, n2, &LCA);//ans variable is passed by reference
   return LCA;
}
```

Row with max 1s

Medium Accuracy: 42.51% Submissions: 88996 Points: 4

Given a boolean 2D array of n x m dimensions where each row is sorted. Find the 0-based index of the first row that has the maximum number of **1's**.

Example 1:

```
Input:
N = 4 , M = 4
Arr[][] = {{0, 1, 1, 1},
           {0, 0, 1, 1},
           {1, 1, 1, 1},
           {0, 0, 0, 0}}
Output: 2
Explanation: Row 2 contains 4 1's (0-based
indexing).
```

Example 2:

Input:

N = 2, M = 2

Arr[][] = {{0, 0}, {1, 1}}

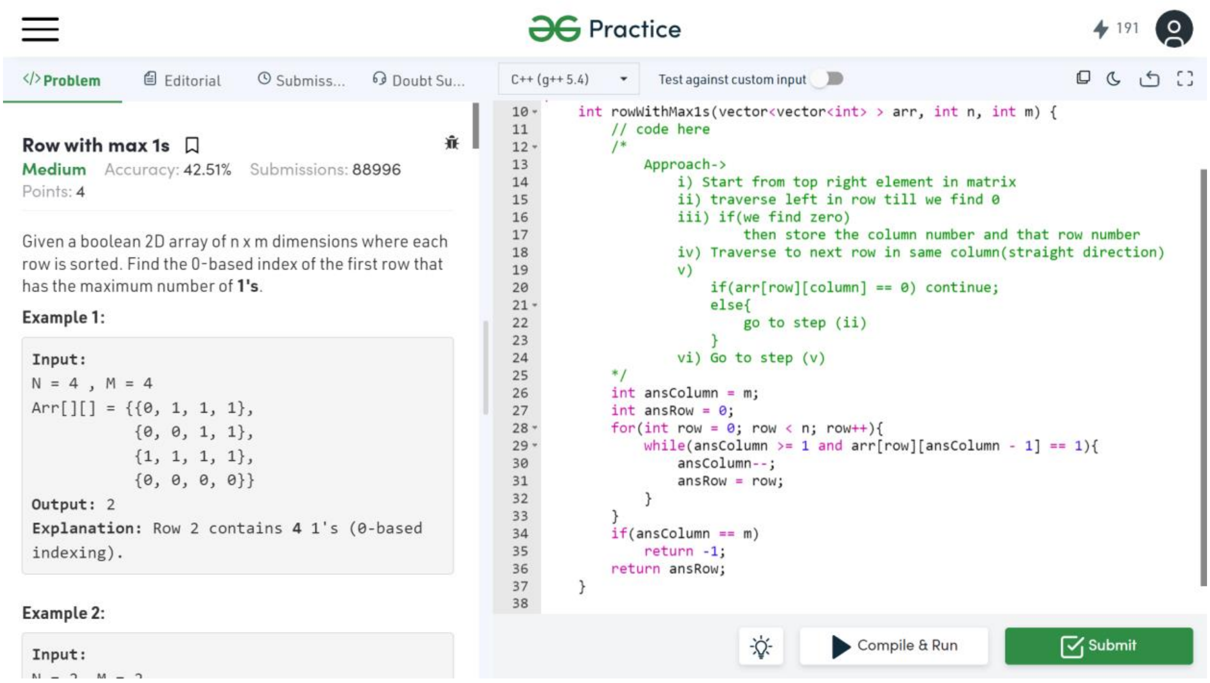
Output: 1

Explanation: Row 1 contains **2** 1's (0-based indexing).

Your Task:
You don't need to read input or print anything. Your task is to complete the function **rowWithMax1s()** which takes the array of booleans **arr[][], n** and **m** as input parameters and returns the 0-based index of the first row that has the most number of 1s. If no such row exists, return -1.

Expected Time Complexity: O(N+M)
Expected Auxiliary Space: O(1)

Constraints:
$1 \leq N, M \leq 10^3$
$0 \leq Arr[i][j] \leq 1$



Row with max 1s

Medium Accuracy: 42.51% Submissions: 88996
Points: 4

Given a boolean 2D array of n x m dimensions where each row is sorted. Find the 0-based index of the first row that has the maximum number of **1's**.

Example 1:

```
Input:
N = 4 , M = 4
Arr[][] = {{0, 1, 1, 1},
           {0, 0, 1, 1},
           {1, 1, 1, 1},
           {0, 0, 0, 0}}
Output: 2
Explanation: Row 2 contains 4 1's (0-based
indexing).
```

Example 2:

```
Input:
N = 2 , M = 2
```

```
    int rowWithMax1s(vector<vector<int> > arr, int n, int m) {
        // code here
        /*
            Approach->
                i) Start from top right element in matrix
                ii) traverse left in row till we find 0
                iii) if(we find zero)
                        then store the column number and that row number
                iv) Traverse to next row in same column(straight direction)
                v)
                    if(arr[row][column] == 0) continue;
                    else{
                        go to step (ii)
                    }
                vi) Go to step (v)
        */
        int ansColumn = m;
        int ansRow = 0;
        for(int row = 0; row < n; row++){
            while(ansColumn >= 1 and arr[row][ansColumn - 1] == 1){
                ansColumn--;
                ansRow = row;
            }
        }
        if(ansColumn == m)
            return -1;
        return ansRow;
    }
```

74. Search a 2D Matrix

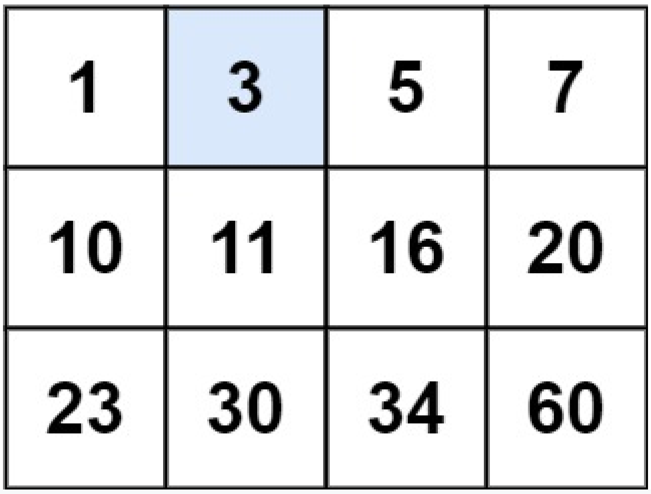
Medium

7388264Add to ListShare

Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

Example 1:

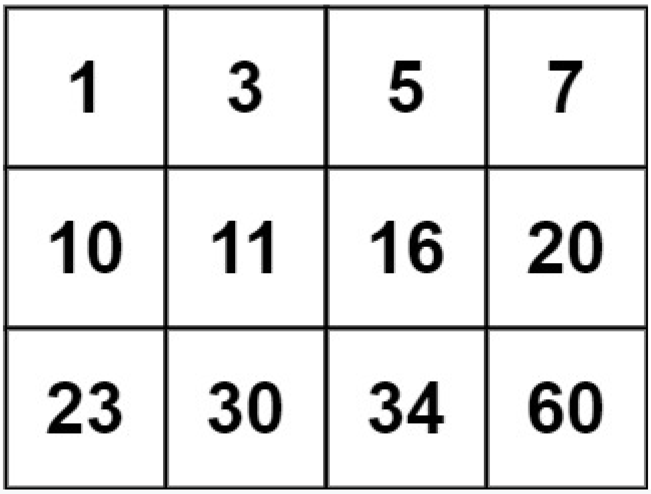


1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

Output: true

Example 2:



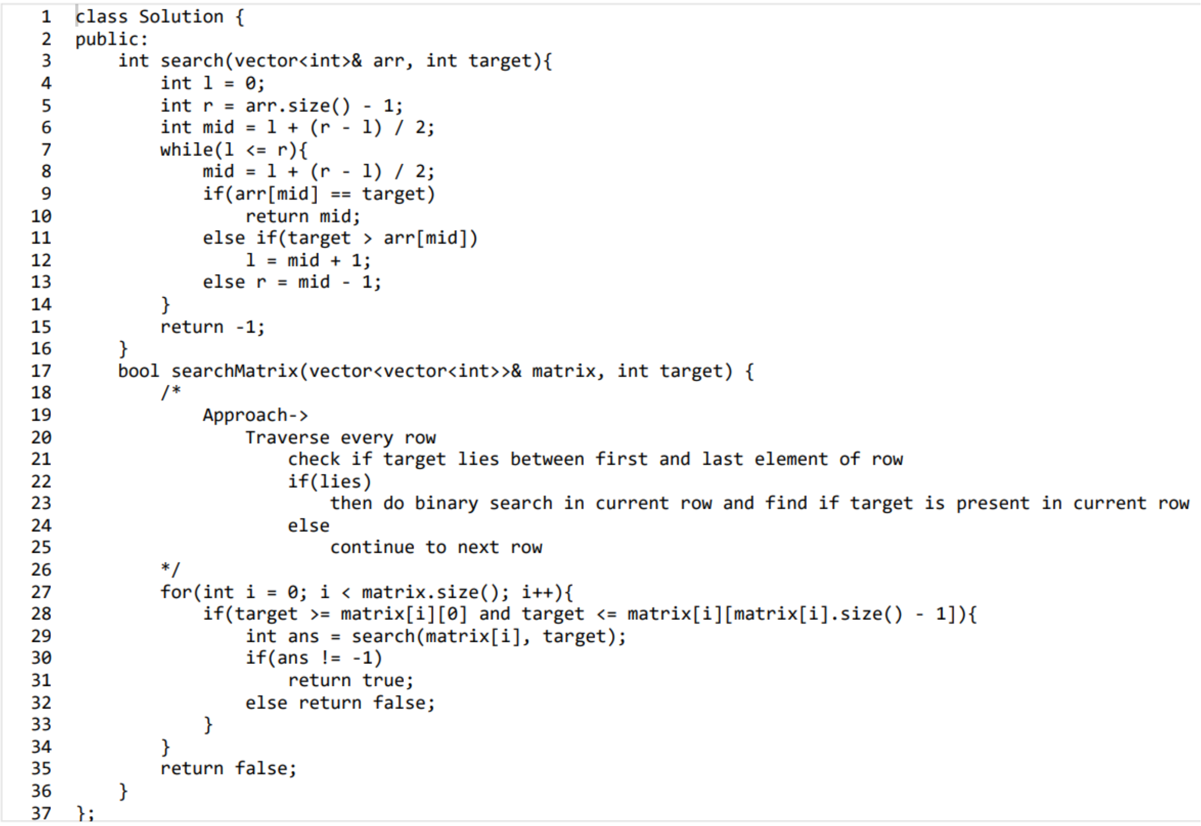
1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

Output: false

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 100
- -10^4 <= matrix[i][j], target <= 10^4



```
class Solution {
public:
    int search(vector<int>& arr, int target){
        int l = 0;
        int r = arr.size() - 1;
        int mid = l + (r - l) / 2;
        while(l <= r){
            mid = l + (r - l) / 2;
            if(arr[mid] == target)
                return mid;
            else if(target > arr[mid])
                l = mid + 1;
            else r = mid - 1;
        }
        return -1;
    }
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        /*
            Approach->
                Traverse every row
                    check if target lies between first and last element of row
                    if(lies)
                        then do binary search in current row and find if target is present in current row
                    else
                        continue to next row
        */
        for(int i = 0; i < matrix.size(); i++){
            if(target >= matrix[i][0] and target <= matrix[i][matrix[i].size() - 1]){
                int ans = search(matrix[i], target);
                if(ans != -1)
                    return true;
                else return false;
            }
        }
        return false;
    }
};
```

Boundary Traversal of binary tree

Medium Accuracy: 26.78% Submissions: 100k+ Points: 4

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

1. **Left boundary nodes:** defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree.
2. **Leaf nodes:** All the leaf nodes except for the ones that are part of left or right boundary.

3. **Reverse right boundary nodes:** defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

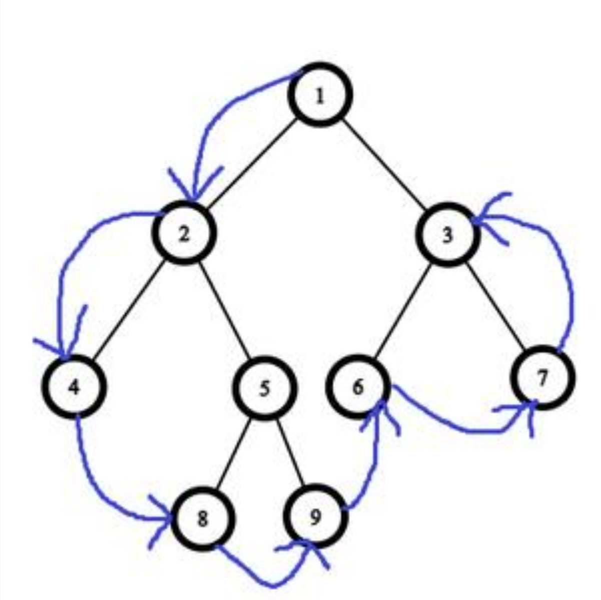
Example 1:

Input:

```
      1
   /  \
  2    3
   / \  / \
 4   5 6   7
   / \
  8   9
```

Output: 1 2 4 8 9 6 7 3

Explanation:



Example 2:

Input:

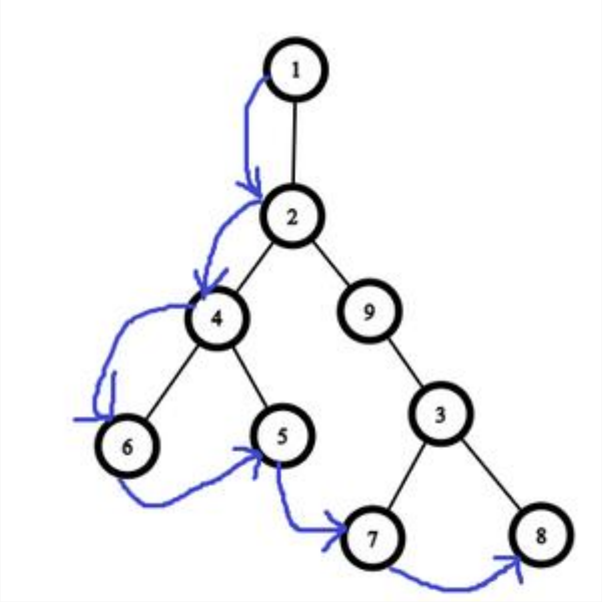
```
        1
       /
      2
    /  \
   4    9
 /  \    \
6    5    3
         /  \
        7    8
```

Output: 1 2 4 6 5 7 8

Explanation:

As you can see we have not taken right

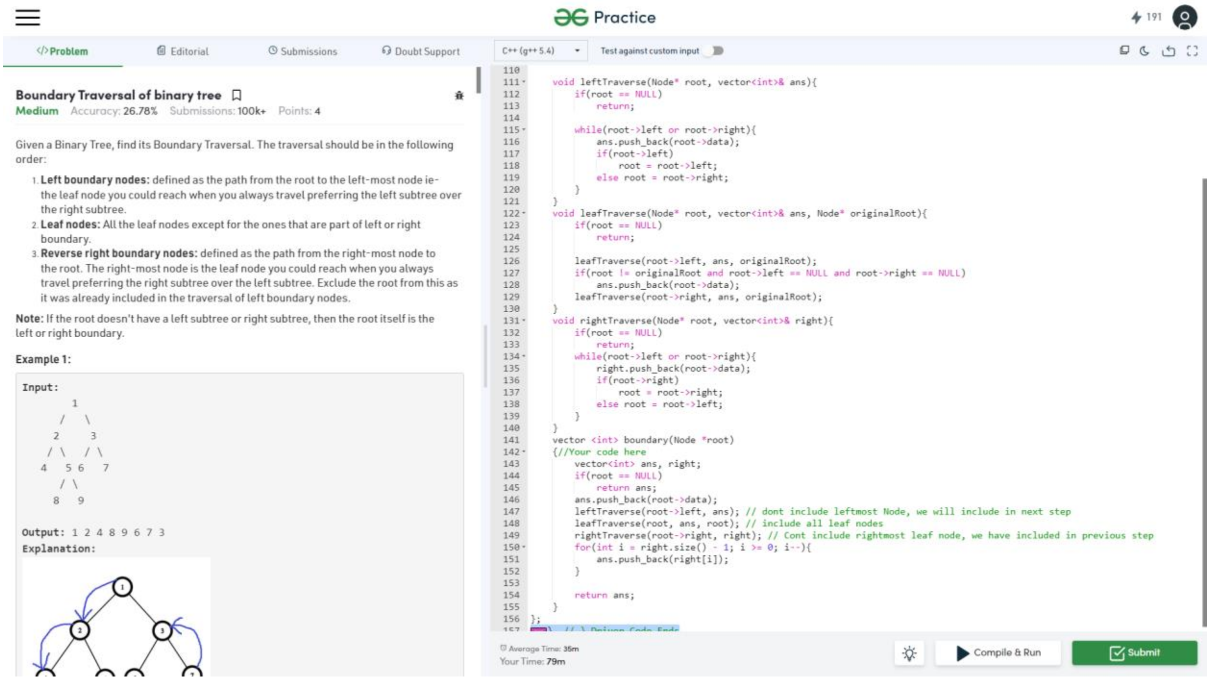
subtree. See **Note**



Your Task:
This is a function problem. You don't have to take input. Just complete the **function boundary()** that takes the root node as input and returns an array containing the boundary values in anti-clockwise.

Expected Time Complexity: O(N).
Expected Auxiliary Space: O(Height of the Tree).

Constraints:
1 ≤ Number of nodes ≤ 10^5
1 ≤ Data of a node ≤ 10^5



Boundary Traversal of binary tree
Medium Accuracy: 26.78% Submissions: 100k+ Points: 4

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

1. **Left boundary nodes:** defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree.
2. **Leaf nodes:** All the leaf nodes except for the ones that are part of left or right boundary.
3. **Reverse right boundary nodes:** defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

Example 1:

```
Input:
        1
      /   \
     2     3
    / \   / \
   4   5 6   7
      / \
     8   9

Output: 1 2 4 8 9 6 7 3
Explanation:
```

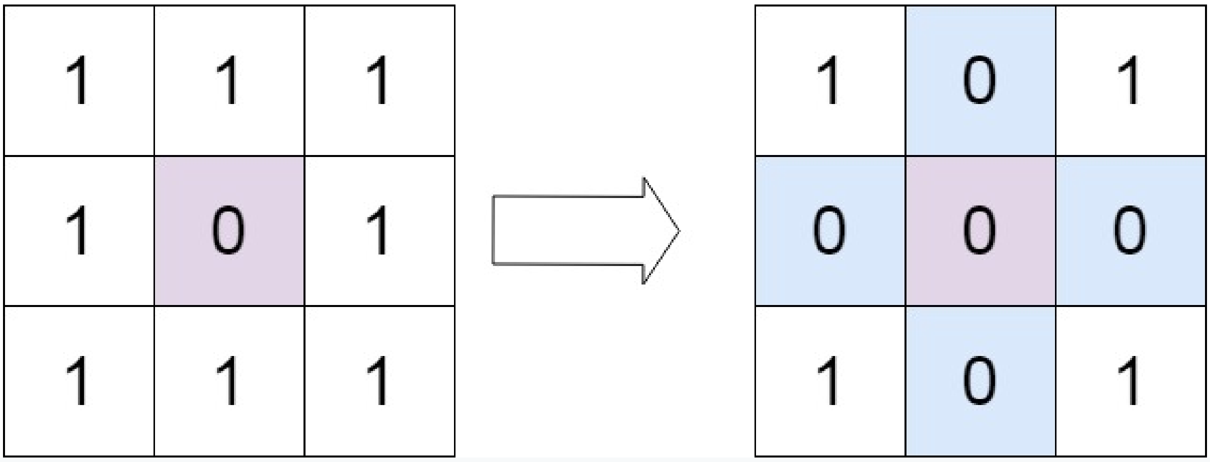
```cpp

    void leftTraverse(Node* root, vector<int>& ans){
        if(root == NULL)
            return;

        while(root->left or root->right){
            ans.push_back(root->data);
            if(root->left)
                root = root->left;
            else root = root->right;
        }
    }
    void leafTraverse(Node* root, vector<int>& ans, Node* originalRoot){
        if(root == NULL)
            return;

        leafTraverse(root->left, ans, originalRoot);
        if(root != originalRoot and root->left == NULL and root->right == NULL)
            ans.push_back(root->data);
        leafTraverse(root->right, ans, originalRoot);
    }
    void rightTraverse(Node* root, vector<int>& right){
        if(root == NULL)
            return;
        while(root->left or root->right){
            right.push_back(root->data);
            if(root->right)
                root = root->right;
            else root = root->left;
        }
    }
    vector <int> boundary(Node *root)
    {//Your code here
        vector<int> ans, right;
        if(root == NULL)
            return ans;
        ans.push_back(root->data);
        leftTraverse(root->left, ans); // dont include leftmost Node, we will include in next step
        leafTraverse(root, ans, root); // include all leaf nodes
        rightTraverse(root->right, right); // Cont include rightmost leaf node, we have included in previous step
        for(int i = right.size() - 1; i >= 0; i--){
            ans.push_back(right[i]);
        }

        return ans;
    }
};
}  // } Driver Code Ends
```

73. Set Matrix Zeroes

Medium
6826485Add to ListShare
Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

You must do it in place.

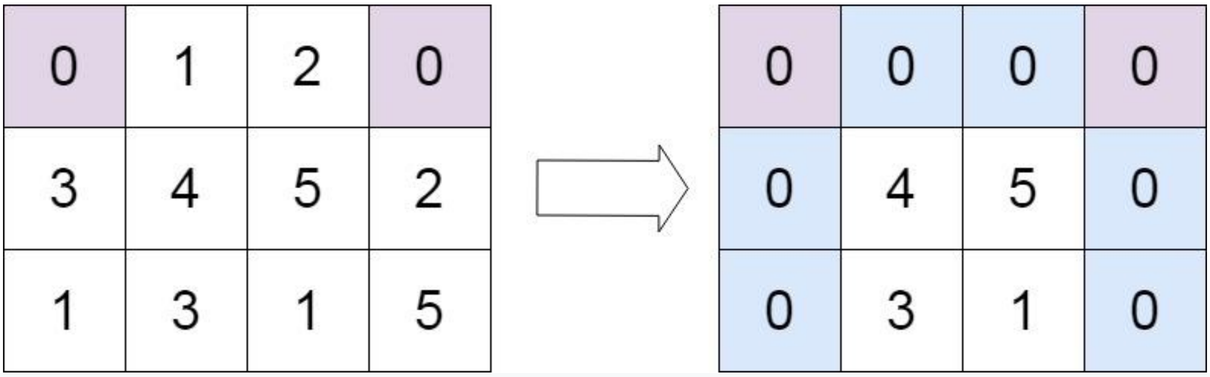
Example 1:



Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]

Output: [[1,0,1],[0,0,0],[1,0,1]]

Example 2:



Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]

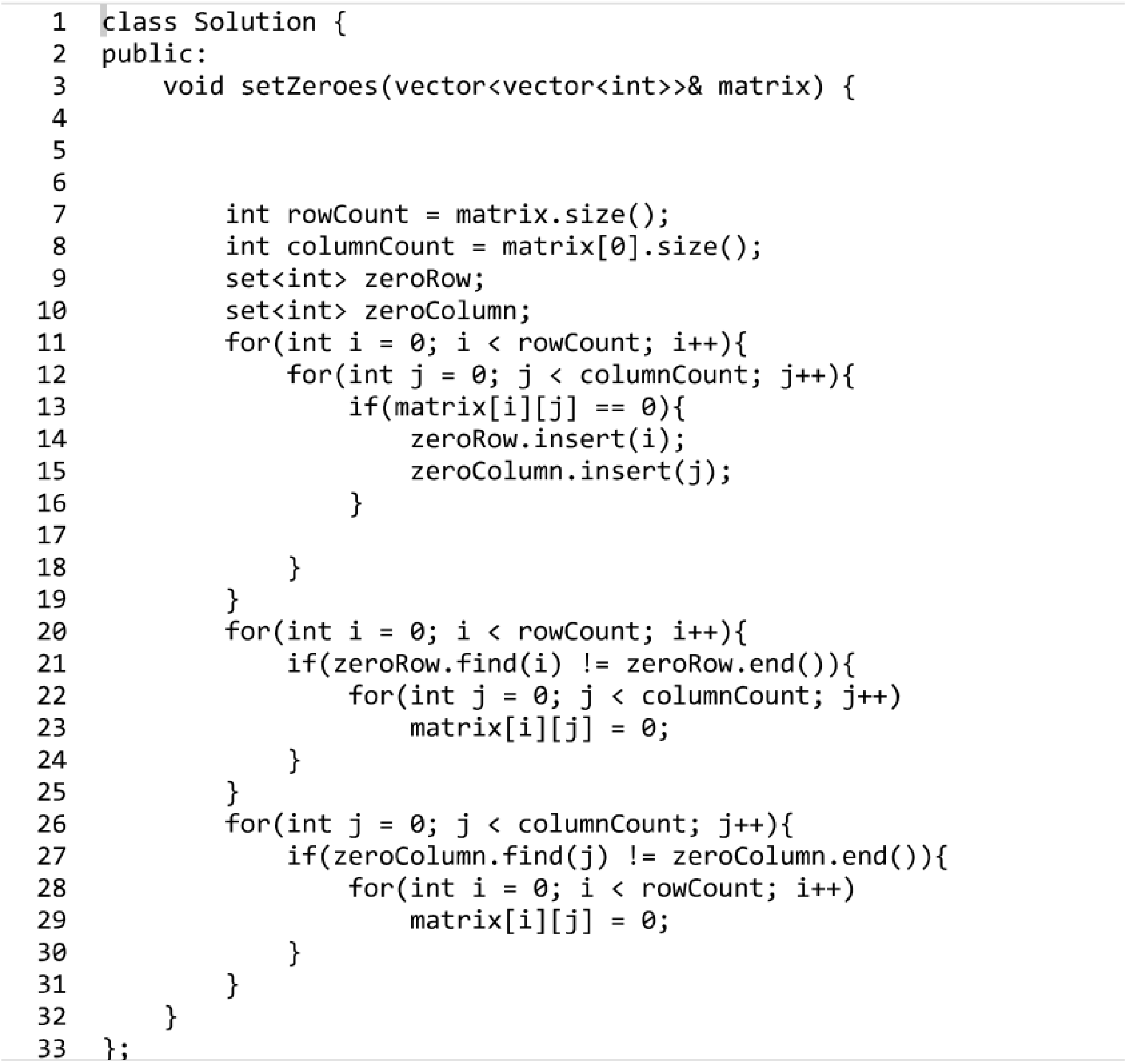
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

Constraints:

- m == matrix.length
- n == matrix[0].length
- 1 <= m, n <= 200
- -2^{31} <= matrix[i][j] <= 2^{31} - 1

Follow up:

- A straightforward solution using O(mn) space is probably a bad idea.
- A simple improvement uses O(m + n) space, but still not the best solution.
- Could you devise a constant space solution?



```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {



        int rowCount = matrix.size();
        int columnCount = matrix[0].size();
        set<int> zeroRow;
        set<int> zeroColumn;
        for(int i = 0; i < rowCount; i++){
            for(int j = 0; j < columnCount; j++){
                if(matrix[i][j] == 0){
                    zeroRow.insert(i);
                    zeroColumn.insert(j);
                }

            }
        }
        for(int i = 0; i < rowCount; i++){
            if(zeroRow.find(i) != zeroRow.end()){
                for(int j = 0; j < columnCount; j++)
                    matrix[i][j] = 0;
            }
        }
        for(int j = 0; j < columnCount; j++){
            if(zeroColumn.find(j) != zeroColumn.end()){
                for(int i = 0; i < rowCount; i++)
                    matrix[i][j] = 0;
            }
        }
    }
};
```