WEEK2

Contents

Binary Search

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9 Output: 4 Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2 Output: -1 Explanation: 2 does not exist in nums so return -1
```

Constraints:

- `1 <= nums.length <=` 10^4
- -10^4 `< nums[i], target <` 10^4
- All the integers in `nums` are **unique**.
- `nums` is sorted in ascending order.

class Solution { public: int search(vector& nums, int target) { } };

Find Smallest Letter Greater Than Target

You are given an array of characters `letters` that is sorted in **non-decreasing order**, and a character `target`. There are **at least two different** characters in `letters`.

Return *the smallest character in* `letters` *that is lexicographically greater than* `target`. If such a character does not exist, return the first character in `letters`.

Example 1:

Input: letters = ["c","f","j"], target = "a" **Output:** "c" **Explanation:** The smallest character that is lexicographically greater than 'a' in letters is 'c'.

Example 2:

Input: letters = ["c","f","j"], target = "c" **Output:** "f" **Explanation:** The smallest character that is lexicographically greater than 'c' in letters is 'f'.

Example 3:

Input: letters = ["x","x","y","y"], target = "z" **Output:** "x" **Explanation:** There are no characters in letters that is lexicographically greater than 'z' so we return letters[0].

Constraints:

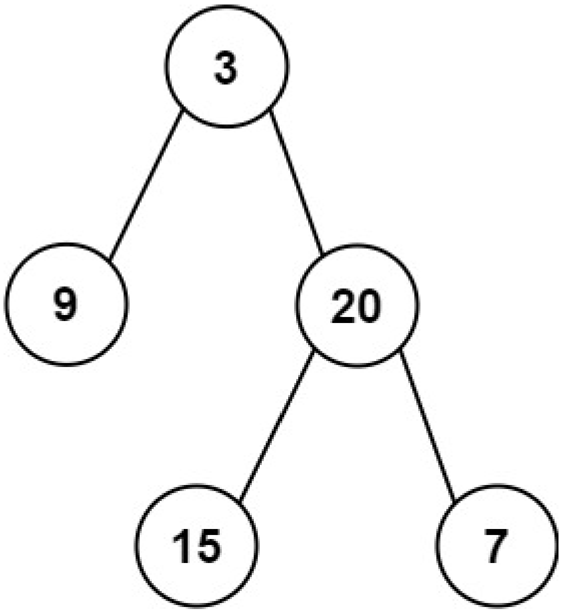
- $2 <= letters.length <= 10^4$
- `letters[i]` is a lowercase English letter.
- `letters` is sorted in **non-decreasing** order.
- `letters` contains at least two different characters.
- `target` is a lowercase English letter.

class Solution { public: char nextGreatestLetter(vector& letters, char target) { } };

Average of Levels in Binary Tree

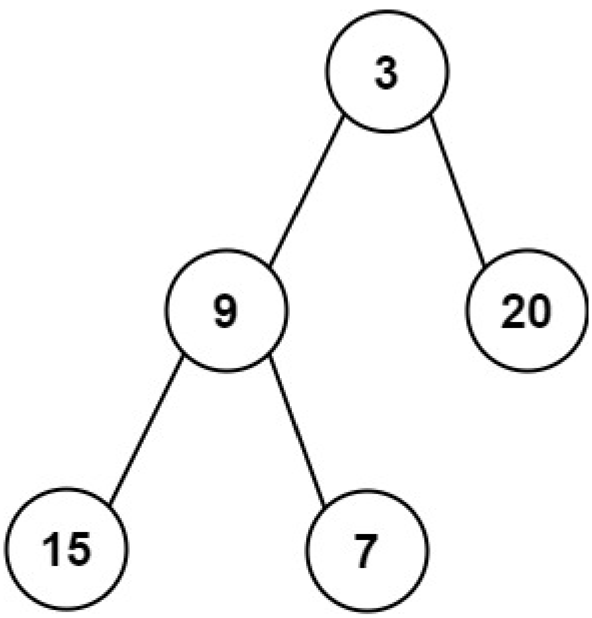
Given the `root` of a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:



`Input: root = [3,9,20,null,null,15,7] Output: [3.00000,14.50000,11.00000] Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11. Hence return [3, 14.5, 11].`

Example 2:



`Input: root = [3,9,20,15,7] Output: [3.00000,14.50000,11.00000]`

Constraints:

- The number of nodes in the tree is in the range `[1, 10^4]`.
- -2^{31} `<= Node.val <=` 2^{31} `- 1`

/** * 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: vector averageOfLevels(TreeNode* root) { } };

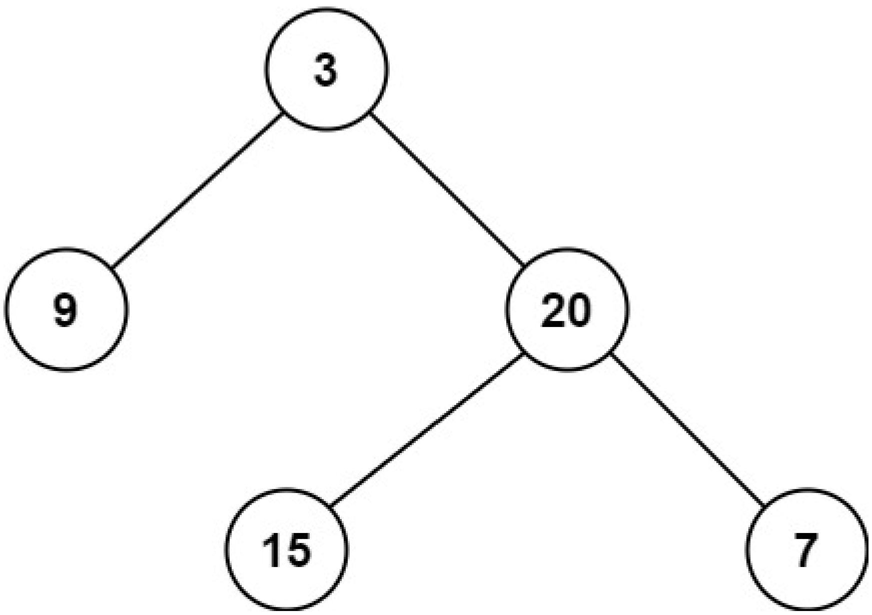
Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

Example 1:



Input: `root = [3,9,20,null,null,15,7]` **Output:** `2`

Example 2:

Input: `root = [2,null,3,null,4,null,5,null,6]` **Output:** `5`

Constraints:

- The number of nodes in the tree is in the range $[0, 10^5]$.
- `-1000 <= Node.val <= 1000`

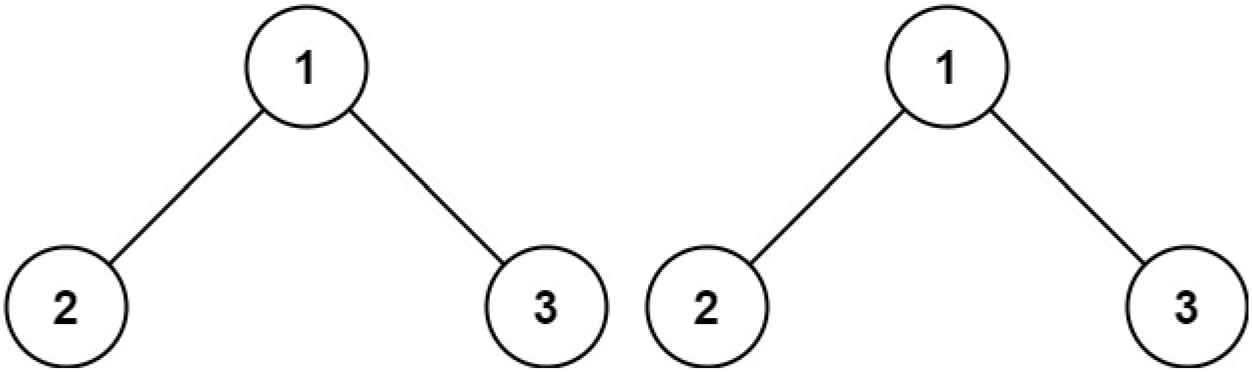
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right)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

Same Tree

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

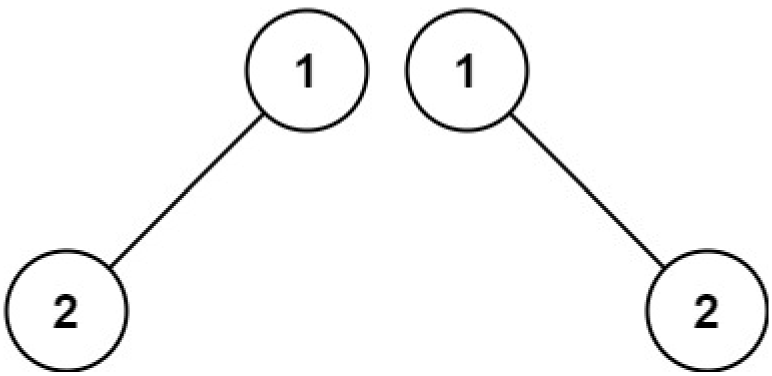
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



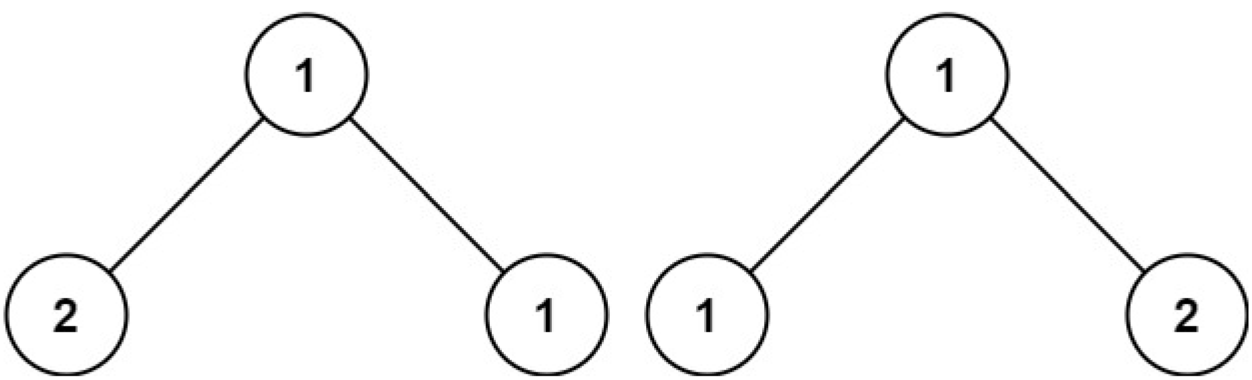
Input: `p = [1,2,3], q = [1,2,3]` **Output:** `true`

Example 2:



Input: `p = [1,2], q = [1,null,2]` **Output:** `false`

Example 3:



Input: `p = [1,2,1], q = [1,1,2]` **Output:** `false`

Constraints:

- The number of nodes in both trees is in the range `[0, 100]`.
- -10^4 `<= Node.val <=` 10^4

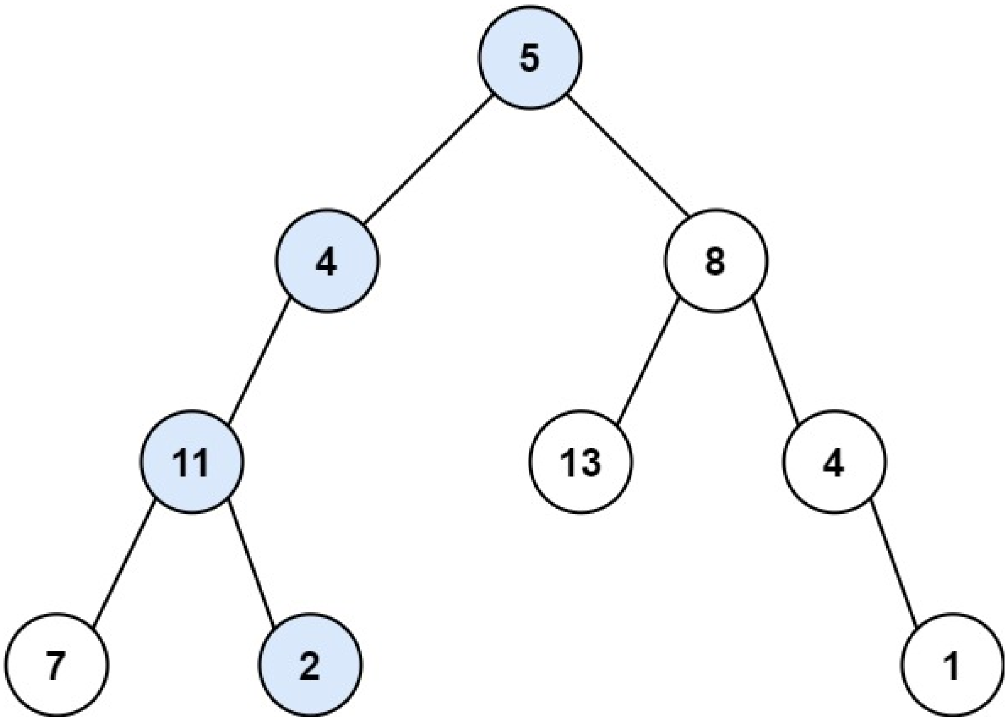
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right)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

Path Sum

Given the `root` of a binary tree and an integer `targetSum`, return `true` if the tree has a **root-to-leaf** path such that adding up all the values along the path equals `targetSum`.

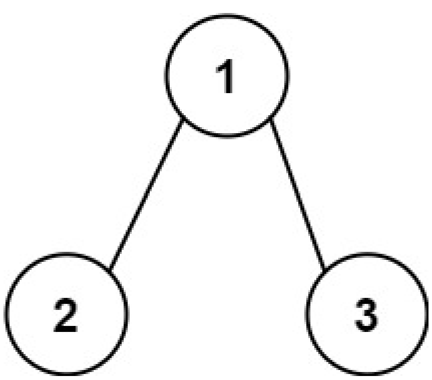
A **leaf** is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22 **Output:** true **Explanation:** The root-to-leaf path with the target sum is shown.

Example 2:



Input: root = [1,2,3], targetSum = 5 **Output:** false **Explanation:** There two root-to-leaf paths in the tree: (1 --> 2): The sum is 3. (1 --> 3): The sum is 4. There is no root-to-leaf path with sum = 5.

Example 3:

Input: root = [], targetSum = 0 **Output:** false **Explanation:** Since the tree is empty, there are no root-to-leaf paths.

Constraints:

- The number of nodes in the tree is in the range `[0, 5000]`.
- `-1000 <= Node.val <= 1000`
- `-1000 <= targetSum <= 1000`

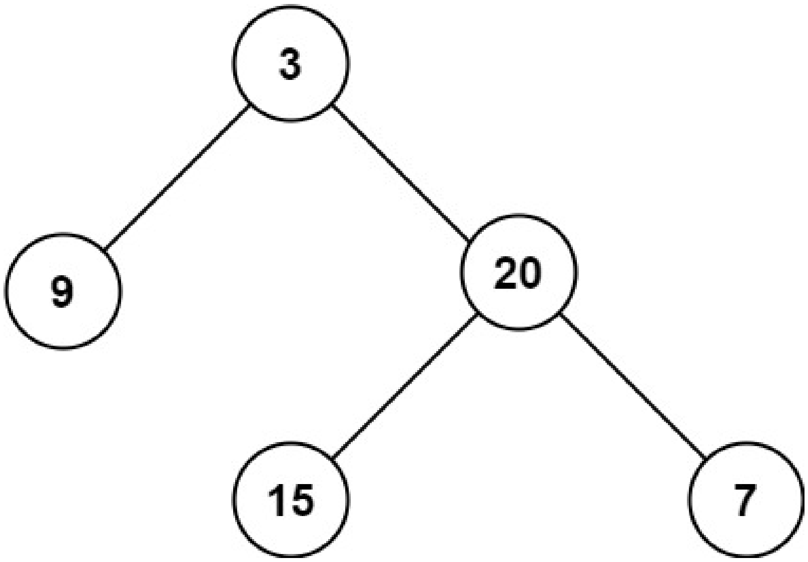
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right)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

Maximum Depth of Binary Tree

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



Input: `root = [3,9,20,null,null,15,7]` **Output:** `3`

Example 2:

Input: `root = [1,null,2]` **Output:** `2`

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- `-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 maxDepth(TreeNode* root) { } };

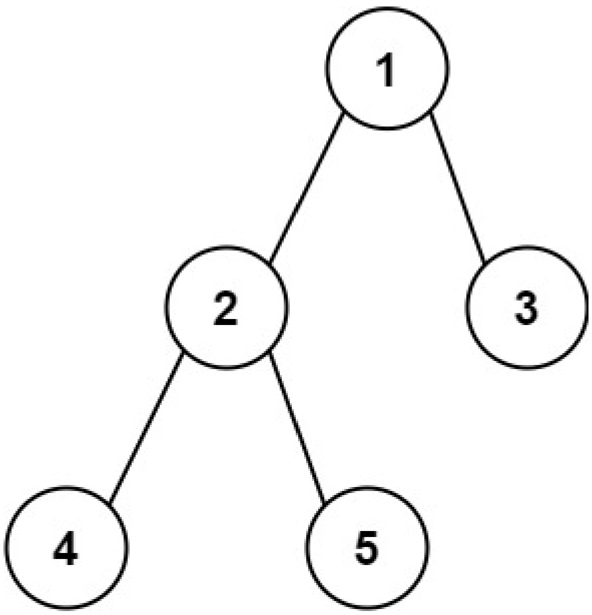
Diameter of Binary Tree

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 diameterOfBinaryTree(TreeNode* root) { } };

Merge Two Binary Trees

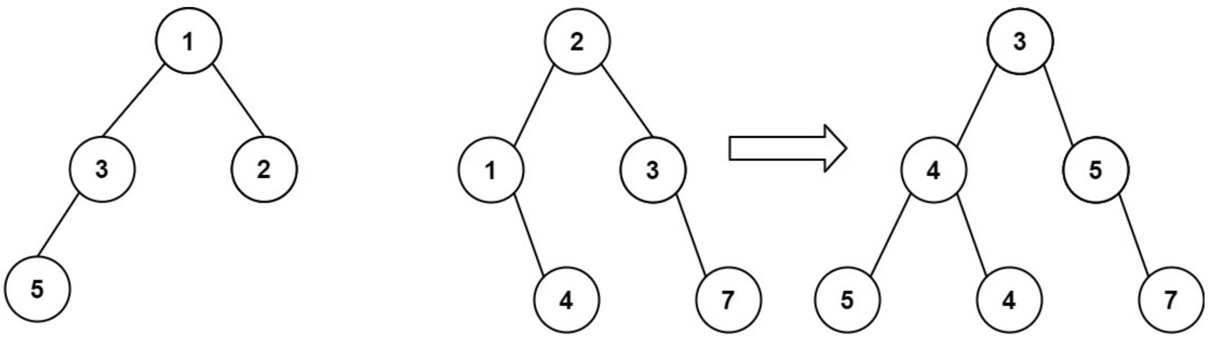
You are given two binary trees `root1` and `root2`.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

Note: The merging process must start from the root nodes of both trees.

Example 1:



Input: `root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]` **Output:** `[3,4,5,5,4,null,7]`

Example 2:

Input: `root1 = [1], root2 = [1,2]` **Output:** `[2,2]`

Constraints:

- The number of nodes in both trees is in the range `[0, 2000]`.
- -10^4 <= `Node.val` <= 10^4

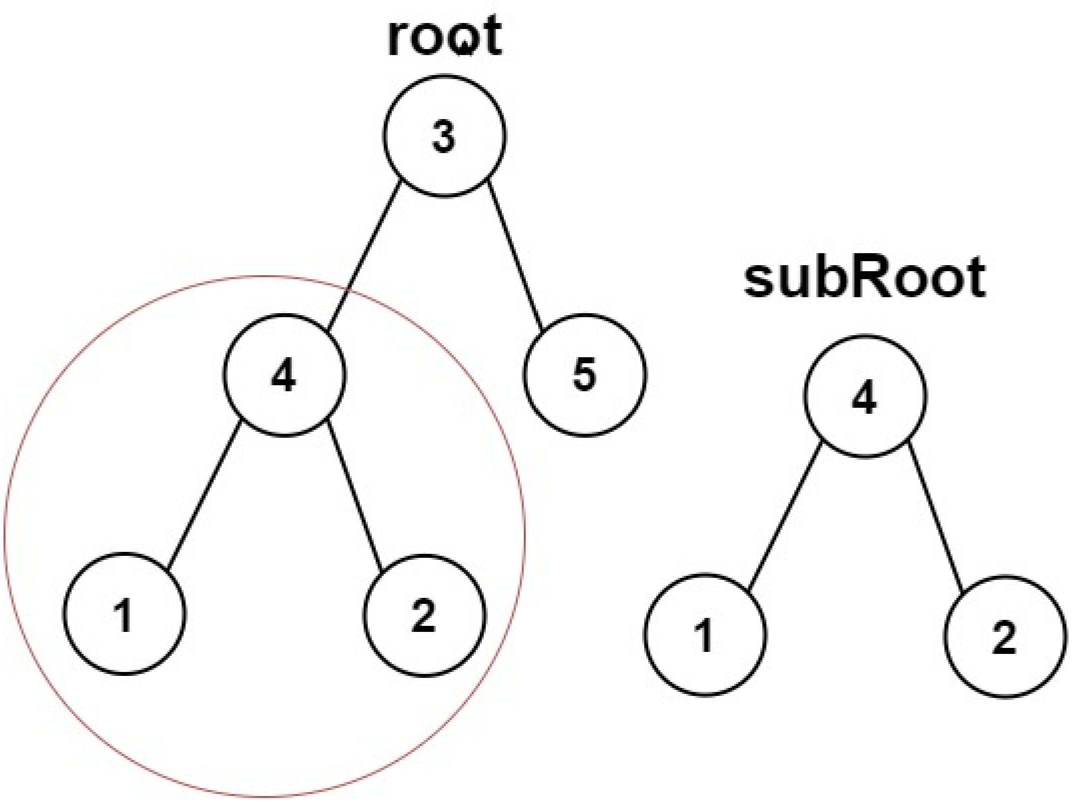
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} * }; */ class Solution { public: TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) { } };

Subtree of Another Tree

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

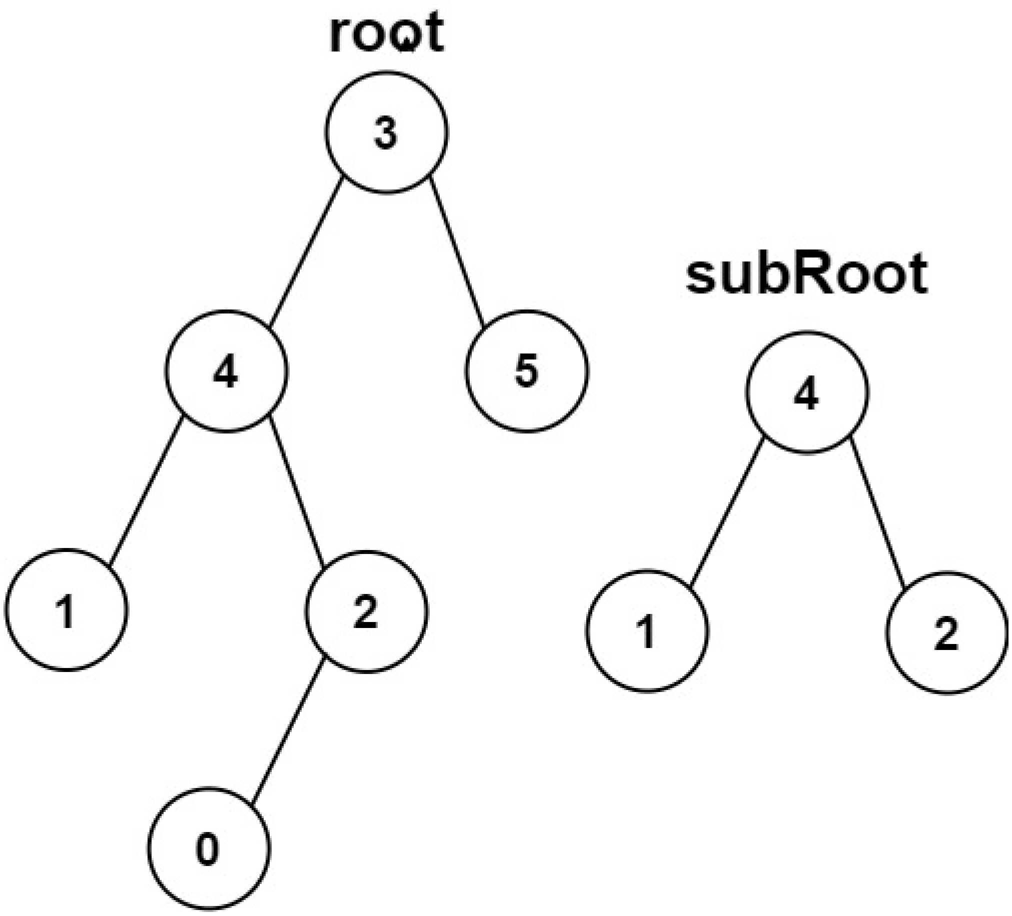
A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Example 1:



Input: `root = [3,4,5,1,2], subRoot = [4,1,2]` **Output:** `true`

Example 2:



Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2] **Output:** false

Constraints:

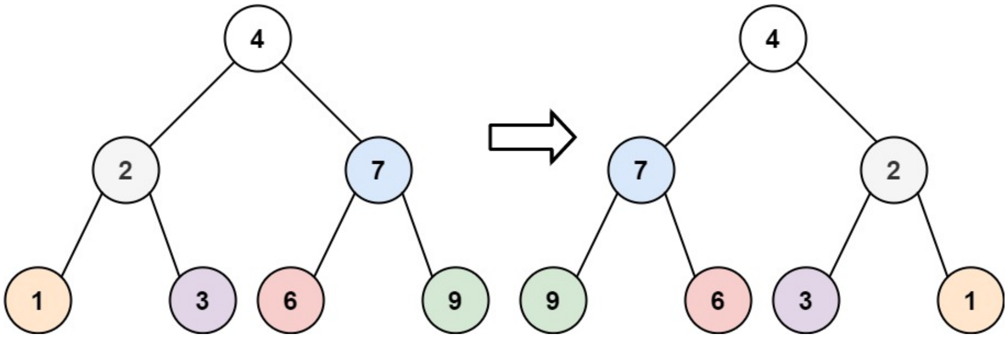
- The number of nodes in the `root` tree is in the range `[1, 2000]`.
- The number of nodes in the `subRoot` tree is in the range `[1, 1000]`.
- -10^4 <= root.val <= 10^4
- -10^4 <= subRoot.val <= 10^4

/** * 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: bool isSubtree(TreeNode* root, TreeNode* subRoot) { } };

Invert Binary Tree

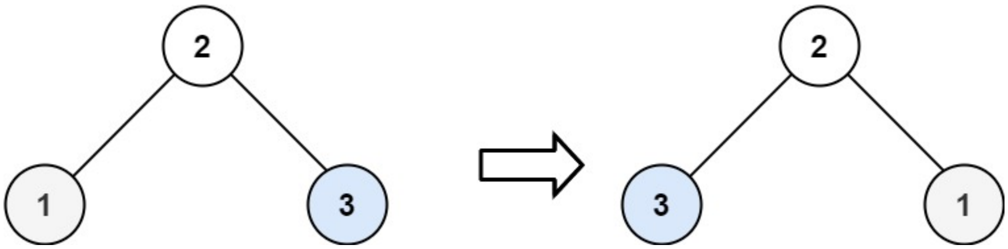
Given the `root` of a binary tree, invert the tree, and return *its root*.

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) { } };

Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to `target`*.

You may assume that each input would have ***exactly*** **one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

`**Input:** nums = [2,7,11,15], target = 9 **Output:** [0,1] **Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].`

Example 2:

`**Input:** nums = [3,2,4], target = 6 **Output:** [1,2]`

Example 3:

`**Input:** nums = [3,3], target = 6 **Output:** [0,1]`

Constraints:

- 2 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?
class Solution { public: vector twoSum(vector& nums, int target) { } };

Squares of a Sorted Array

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of* ***the squares of each number*** *sorted in non-decreasing order*.

Example 1:

```
Input: nums = [-4,-1,0,3,10] Output: [0,1,9,16,100] Explanation: After squaring, the array becomes [16,1,0,9,100]. After sorting, it becomes [0,1,9,16,100].
```

Example 2:

```
Input: nums = [-7,-3,2,3,11] Output: [4,9,9,49,121]
```

Constraints:

- `1 <= nums.length <=` 10^4
- -10^4 `<= nums[i] <=` 10^4
- `nums` is sorted in **non-decreasing** order.

Follow up: Squaring each element and sorting the new array is very trivial, could you find an `O(n)` solution using a different approach?

class Solution { public: vector sortedSquares(vector& nums) { } };

Backspace String Compare

Given two strings `s` and `t`, return `true` *if they are equal when both are typed into empty text editors*. `'#'` means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1:

`Input:` `s = "ab#c", t = "ad#c"` **`Output:`** `true` **`Explanation:`** `Both s and t become "ac".`

Example 2:

`Input:` `s = "ab##", t = "c#d#"` **`Output:`** `true` **`Explanation:`** `Both s and t become "".`

Example 3:

`Input:` `s = "a#c", t = "b"` **`Output:`** `false` **`Explanation:`** `s becomes "c" while t becomes "b".`

Constraints:

- `1 <= s.length, t.length <= 200`
- `s` and `t` only contain lowercase letters and `'#'` characters.

Follow up: Can you solve it in `O(n)` time and `O(1)` space?

class Solution { public: bool backspaceCompare(string s, string t) { } };

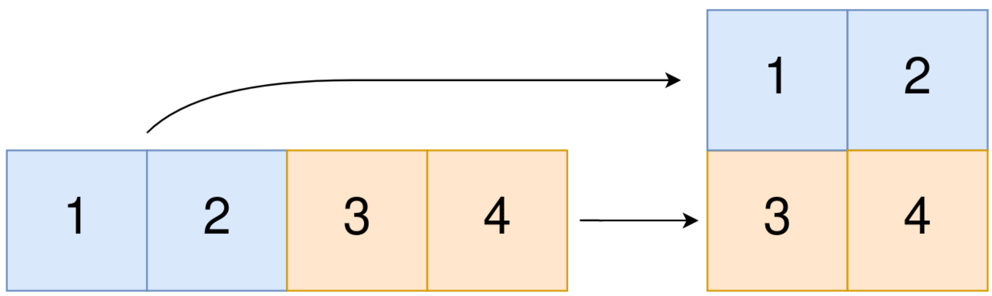
Convert 1D Array Into 2D Array

You are given a **0-indexed** 1-dimensional (1D) integer array `original`, and two integers, `m` and `n`. You are tasked with creating a 2-dimensional (2D) array with `m` rows and `n` columns using **all** the elements from `original`.

The elements from indices `0` to `n - 1` (**inclusive**) of `original` should form the first row of the constructed 2D array, the elements from indices `n` to `2 * n - 1` (**inclusive**) should form the second row of the constructed 2D array, and so on.

Return *an* `m x n` *2D array constructed according to the above procedure, or an empty 2D array if it is impossible*.

Example 1:



Input: original = [1,2,3,4], m = 2, n = 2 **Output:** [[1,2],[3,4]] **Explanation:** The constructed 2D array should contain 2 rows and 2 columns. The first group of n=2 elements in original, [1,2], becomes the first row in the constructed 2D array. The second group of n=2 elements in original, [3,4], becomes the second row in the constructed 2D array.

Example 2:

Input: original = [1,2,3], m = 1, n = 3 **Output:** [[1,2,3]] **Explanation:** The constructed 2D array should contain 1 row and 3 columns. Put all three elements in original into the first row of the constructed 2D array.

Example 3:

Input: original = [1,2], m = 1, n = 1 **Output:** [] **Explanation:** There are 2 elements in original. It is impossible to fit 2 elements in a 1x1 2D array, so return an empty 2D array.

Constraints:

- `1 <= original.length <= 5 * 10`^{4}
- `1 <= original[i] <= 10`^{5}
- `1 <= m, n <= 4 * 10`^{4}

class Solution { public: vector> construct2DArray(vector& original, int m, int n) { } };

Move Zeroes

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: nums = [0,1,0,3,12] **Output:** [1,3,12,0,0]

Example 2:

Input: nums = [0] **Output:** [0]

Constraints:

- $1 <= nums.length <= 10^4$
- $-2^{31} <= nums[i] <= 2^{31} - 1$

Follow up: Could you minimize the total number of operations done?

class Solution { public: void moveZeroes(vector& nums) { } };

Is Subsequence

Given two strings `s` and `t`, return `true` *if* `s` *is a* ***subsequence*** *of* `t`*, or* `false` *otherwise*.

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., `"ace"` is a subsequence of `"abcde"` while `"aec"` is not).

Example 1:

Input: `s = "abc", t = "ahbgdc"` **Output:** `true`

Example 2:

Input: `s = "axc", t = "ahbgdc"` **Output:** `false`

Constraints:

- `0 <= s.length <= 100`
- `0 <= t.length <=` 10^4
- `s` and `t` consist only of lowercase English letters.

Follow up: Suppose there are lots of incoming `s`, say $s_1, s_2, ..., s_k$ where $k >= 10^9$, and you want to check one by one to see if `t` has its subsequence. In this scenario, how would you change your code?

class Solution { public: bool isSubsequence(string s, string t) { } };