**BINARY TREE**

1. <https://leetcode.com/problems/binary-tree-inorder-traversal/description/>

**Intuition**

**In an in-order traversal, we start from the root node and recursively visit the left subtree, then process the current node, and finally visit the right subtree. This ensures that nodes are visited in ascending order for a binary search tree, but the approach is general for any binary tree.**

**Approach**

**Define the Helper Function: Created a helper function traverse that will perform the recursive traversal. This function takes a node and a list to store the result.**

**Recursive Traversal:**

**If the node is None, return immediately as there is nothing to process.**

**Recursively traverse the left subtree of the current node.**

**Adding the current node's value to the result list.**

**Recursively traverse the right subtree of the current node.**

**Initialize and Call the Helper Function:**

**In the main method, in order Traversal, initialize an empty list to store the result.**

**Call the traverse function starting with the root node.**

**Return the result list which contains the values of nodes in in-order.**

**Complexity**

**Time Complexity:**

**O(n), where n is the number of nodes in the binary tree. Each node is visited exactly once during the traversal.**

**Space Complexity:**

**O(h), where h is the height of the binary tree. This space is used for the call stack during recursion. In the worst case (unbalanced tree), the space complexity can be O(n), but in a balanced tree, it is O(log n).**

**CODE:**

A screenshot of a computer program

Description automatically generated

**2.**[**https://leetcode.com/problems/same-tree/description/**](https://leetcode.com/problems/same-tree/description/)

**Intuition:**

**The intuition behind the solution is to recursively check if two binary trees are identical. If both trees are empty (null), they are considered identical. If only one tree is empty or the values of the current nodes are different, the trees are not identical. Otherwise, we recursively check if the left and right subtrees of both trees are identical.**

**Approach:**

**Checking the base case: if both trees are null, return true.**

**Checking if only one tree is null or the values of the current nodes are different, return false.**

**Recursively checking if the left subtrees of both trees are identical.**

**Recursively check if the right subtrees of both trees are identical.**

**Return the logical AND of the results from step 3 and 4.**

**Complexity:**

**Time complexity: The time complexity of the solution is *O*(*min*(*N*,*M*)), where N and M are the number of nodes in the two trees, respectively. This is because we need to visit each node once in order to compare their values. In the worst case, where both trees have the same number of nodes, the time complexity would be O(N).**

**Space complexity:  
The space complexity of the solution is*O*(*min*(*H*1,*H*2)), where H1 and H2 are the heights of the two trees, respectively. This is because the space used by the recursive stack is determined by the height of the smaller tree. In the worst case, where one tree is significantly larger than the other, the space complexity would be closer to O(N) or O(M), depending on which tree is larger.**

**Code:**

**A screenshot of a computer program

Description automatically generated**

**3.**[**https://leetcode.com/problems/symmetric-tree/description/**](https://leetcode.com/problems/symmetric-tree/description/)

**Intuition:**

**To check if a binary tree is symmetric, we need to compare its left subtree and right subtree. To do this, we can traverse the tree recursively and compare the left and right subtrees at each level. If they are symmetric, we continue the traversal. Otherwise, we can immediately return false.**

**Approach:**

**I have defined a recursive helper function that takes two nodes as input, one from the left subtree and one from the right subtree. The helper function returns true if both nodes are null, or if their values are equal and their subtrees are symmetric.**

**Complexity:**

**Time complexity: The time complexity of the algorithm is *O*(*n*), where n is the number of nodes in the binary tree. We need to visit each node once to check if the tree is symmetric.**

**Space complexity: The space complexity of the algorithm is *O*(*h*), where h is the height of the binary tree. In the worst case, the tree can be completely unbalanced, and the recursion stack can go as deep as the height of the tree.**

Code:

A screenshot of a computer program

Description automatically generated

**4.**[**https://leetcode.com/problems/maximum-depth-of-binary-tree/**](https://leetcode.com/problems/maximum-depth-of-binary-tree/)

**Approach:**

**Base Case: If the current node (root) is None, the depth is 0 since we've reached the end of a path.**

**Recursive Case: For a non-null node, we:**

**Recursively calculate the depth of the left subtree.**

**Recursively calculate the depth of the right subtree.**

**The depth of the current node is the maximum depth between the left and right subtrees, plus one (for the current node).**

**The final depth is determined by the maximum of these two depths at each node as we backtrack through the recursion.**

**Complexity**

**Time Complexity: O(n), where n is the number of nodes in the tree. Each node is visited exactly once.**

**Space Complexity: O(h), where h is the height of the tree. This is the space required for the recursive call stack, which in the worst case (a skewed tree) could be as deep as the number of nodes, but in a balanced tree would be O(log n).**

**Code:**

A screenshot of a computer program

Description automatically generated

5. <https://leetcode.com/problems/balanced-binary-tree/>

**Intuition:**

**A height-balanced tree ensures that the tree is as compact as possible, which makes operations like search, insert, and delete more efficient. The key observation is that a tree is balanced if and only if the left and right subtrees of every node are balanced and the difference in their heights is no more than 1.**

**Approach:**

**Post-order Traversal**: Using a recursive function to perform a post-order traversal of the tree. This means that for each node, the function will first calculate the height of its left and right subtrees before checking if the current node is balanced.

**Height Calculation**: For each node, calculating the height of the left and right subtrees. The height of a node is the maximum of the heights of its left and right subtrees, plus one.

**Balance Check**: If the absolute difference in heights between the left and right subtrees is greater than 1, mark the tree as unbalanced.

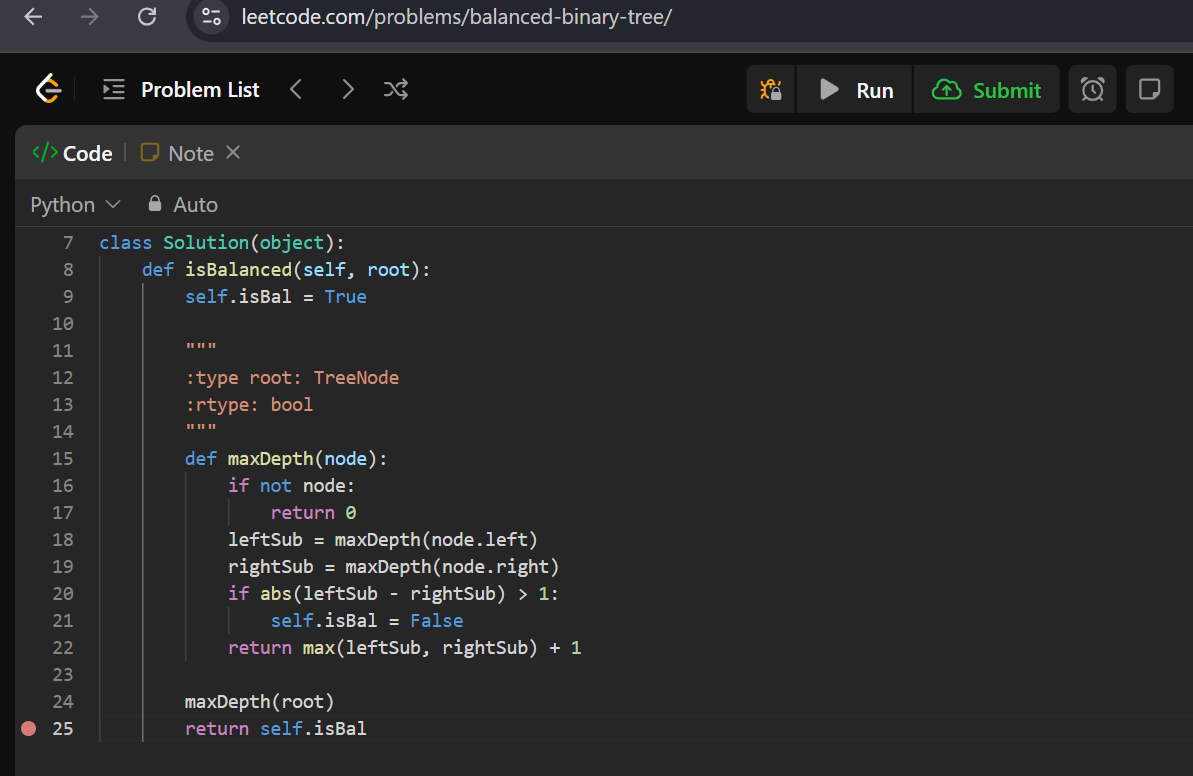
**Return the Result**: After the entire tree has been traversed, return the result indicating whether the tree is balanced or not.

**Complexity:**

**Time Complexity:** The time complexity is (O(n)), where n is the number of nodes in the tree. This is because each node is visited once during the traversal.

**Space Complexity:** The space complexity is (O(h)), where h is the height of the tree. This is due to the recursion stack, which at most will be as deep as the height of the tree.

**Code:**



**BINARY SEARCH TREE**

1. [**https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/**](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/)

**Approach**

**Base Case**: If the start index (s) exceeds the end index (e), the subarray is invalid, and we return None, indicating the absence of a subtree.

**Recursive Case**:

Calculate the middle index mid of the current subarray (s to e).

Created a TreeNode with the value nums[mid], which becomes the root of the current subtree.

Recursively builded the left subtree using the elements before mid.

Recursively builded the right subtree using the elements after mid.

Return the current TreeNode, which now represents the root of the subtree formed by the current subarray.

The initial call to the helper function is made with the entire array, ensuring that the entire tree is constructed in a balanced manner.

**Complexity**

**Time Complexity**: O(n), where n is the number of elements in the array. Every element in the array is visited exactly once to construct the tree nodes.

**Space Complexity**: O(log n) for the recursive stack in the best case (balanced tree) and O(n) in the worst case (completely skewed tree). However, since the problem guarantees a balanced BST, the stack space is typically O(log n).

**CODE: ZOOM IT for better visibility**

**A screenshot of a computer

Description automatically generated**

**2.**[**https://leetcode.com/problems/find-mode-in-binary-search-tree/description/**](https://leetcode.com/problems/find-mode-in-binary-search-tree/description/)

**Approach**

**Initialization: The class `Solution` initializes three attributes:**

**`self.dict`: A dictionary that stores the frequency of each unique value encountered in the binary tree.**

**`self.ans`: A list that stores the mode(s) (the most frequently occurring value(s)) in the tree.**

**`self.mx`: An integer that keeps track of the maximum frequency encountered during the traversal of the tree.**

**Depth-First Search (DFS) Traversal:**

**The `solve` method is a recursive function that performs a depth-first search traversal of the binary tree.**

**If the current node (`root`) is `None`, the function returns, indicating that there is no further subtree to process.**

**For each non-empty node, the method:**

**Updates `self.dict` to increase the frequency count for the current node's value.**

**Compares the frequency of the current value with `self.mx`:**

**If the frequency equals `self.mx`, it appends the value to `self.ans`, indicating that this value is also a mode.**

**If the frequency exceeds `self.mx`, it updates `self.mx` to this new frequency and resets `self.ans` to contain only the current value.**

**The method then recursively calls itself on the left and right subtrees of the current node, continuing the DFS traversal of the entire tree.**

**Finding the Mode:**

**The `findMode` method is the entry point of the solution. It initiates the DFS traversal by calling the `solve` method with the root of the tree.**

**After the traversal is complete, `self.ans` contains all the mode(s) of the tree, which is then returned.**

**Usage:**

**To find the mode(s) in a binary search tree, an instance of the `Solution` class is created, and the `findMode` method is called with the root of the tree as an argument.**

**Complexity**

**Time Complexity: O(n) where `n` is the number of nodes in the tree. This is because the algorithm must visit each node exactly once to count the frequencies.**

**Space Complexity: O(n) for the dictionary (`self.dict`) that stores the frequencies of each node's value. Additionally, the recursion stack might take up to `O(h)` space, where `h` is the height of the tree, but the overall space complexity is dominated by the dictionary, making it `O(n)`.**

**CODE:**

**A screenshot of a computer program

Description automatically generated**

**3.**[**https://leetcode.com/problems/minimum-absolute-difference-in-bst/description/**](https://leetcode.com/problems/minimum-absolute-difference-in-bst/description/)

**Intuition:**

**The key intuition behind solving this problem is that an in-order traversal of a BST yields node values in ascending order. Thus, the minimum difference will always be found between two consecutive values in this sorted order.**

**Approach**

**In-Order Traversal: Performed an in-order traversal of the BST to collect the node values in a list. This ensures the values are in ascending order.**

**Compute Minimum Difference:**

**Iterate through the sorted list of values and compute the differences between consecutive elements.**

**Track the minimum difference encountered during this iteration.**

**Edge Cases:**

**Handle the case where the tree is empty.**

**Ensure the solution works for trees with only one node or two nodes.**

**Complexity**

**Time Complexity: O(n), where n is the number of nodes in the BST. This is because we perform an in-order traversal of all nodes and then a single pass to compute the minimum difference.**

**Space Complexity: O(n) for storing the values of the nodes in a list during the in-order traversal.**

**CODE:**

**A screenshot of a computer program

Description automatically generated**

**DYNAMIC PROGRAMMING**

1. [**https://leetcode.com/problems/climbing-stairs/**](https://leetcode.com/problems/climbing-stairs/)

**Intuition: This problem was thought to find the number of ways to reach the top of a staircase with `n` steps, where we can have either take 1 or 2 steps at a time. This problem is related to the Fibonacci sequence. Specifically, the number of ways to reach the nth step is the sum of the number of ways to reach the (n-1)th step and the (n-2)th step. This is because from step (n-1), you can take a single step to reach n, and from step (n-2), you can take two steps to reach n.**

**Approach:**

**Base Case: If `n` is 1, there's only one way to climb the stairs (taking one step).**

**Dynamic Programming Transition:**

**Using two variables, `one\_before` and `two\_before`, to keep track of the number of ways to reach the previous step (`n-1`) and the step before that (`n-2`), respectively.**

**For each step from 2 to `n`, calculated the total number of ways to reach that step by summing `one\_before` and `two\_before`.**

**Update `two\_before` to the value of `one\_before`, and `one\_before` to the new total, effectively moving up the staircase.**

**Return the Result: After the loop completes, `total` will contain the number of ways to reach the top (the nth step).**

**Complexity:**

**Time Complexity: O(n) The loop runs `n-1` times, so the time complexity is linear in relation to the number of steps.**

**Space Complexity: O (1) Only a constant amount of extra space is used (for the `one\_before`, `two\_before`, and `total` variables), so the space complexity is constant.**

**CODE:**

**A screenshot of a computer program

Description automatically generated**

1. [**https://leetcode.com/problems/counting-bits/description/**](https://leetcode.com/problems/counting-bits/description/)

**Dynamic Programming with Bit Manipulation**

**Intuition and Logic Behind the Solution**

**The idea is to leverage the relationship between the number of set bits (1's) in the binary representation of a number `i` and the number of set bits in `i >> 1` (which is the number `i` divided by 2). The bitwise operation `i >> 1` shifts the bits of `i` one place to the right, effectively removing the least significant bit (the last bit). By adding this last bit back, we can compute the number of 1's in `i`.**

**Approach:**

**Initialization: Create an array `ans` of length `n + 1`, initialized with zeros. This array will store the number of 1's for each integer from `0` to `n`.**

**Main Algorithm:**

**Iterate from `1` to `n`. For each `i`, calculate the number of set bits as:**

**ans[i] = ans[i >> 1] + (i & 1)**

**ans[i >> 1] gives the number of set bits in the integer `i // 2`.**

**(i & 1) checks if the last bit of `i` is 1. If it is, it adds 1 to the count.**

**Return the Result: After the loop, `ans` will contain the number of set bits for every integer from `0` to `n`, which is then returned.**

**Complexity Analysis**

**Time Complexity: `O(n)` — The algorithm iterates through the array once, performing constant-time operations during each iteration.**

**Space Complexity: `O(n)` — We use an array of size `n + 1` to store the results.**

**CODE:**

**A screenshot of a computer program

Description automatically generated**