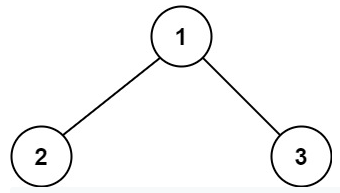
<https://leetcode.com/problems/binary-tree-maximum-path-sum/>

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

**Example 1:**

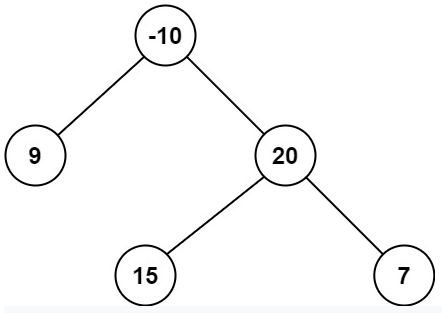


Input: root = [1,2,3]

Output: 6

Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

**Example 2:**



Input: root = [-10,9,20,null,null,15,7]

Output: 42

Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

**Constraints:**

The number of nodes in the tree is in the range [1, 3 \* 10^4].

-1000 <= Node.val <= 1000

**Attempt 1: 2022-12-06**

**Solution 1:  Recursive traversal (120 min)**

**The solution style is exactly same as L543.Diameter of Binary Tree**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    int maxValue;

    public int maxPathSum(TreeNode root) {

        maxValue = Integer.MIN\_VALUE;

        helper(root);

        return maxValue;

    }

    private int helper(TreeNode root) {

        if(root == null) {

            return 0;

        }

        int leftMax = Math.max(0, helper(root.left));

        int rightMax = Math.max(0, helper(root.right));

        maxValue = Math.max(maxValue, root.val + leftMax + rightMax);

        return root.val + Math.max(leftMax, rightMax);

    }

}

Time Complexity : O(N)

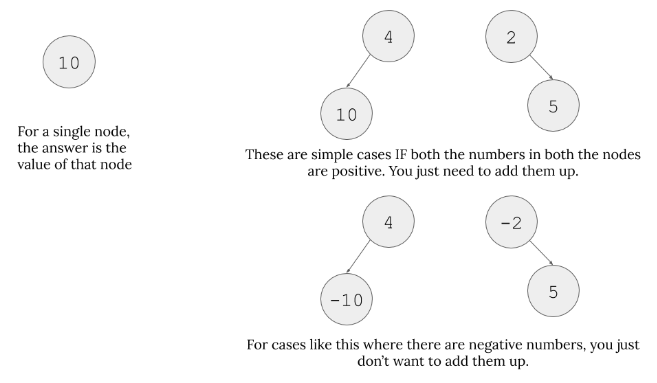
Space Complexity: O(N)

**Refer to**

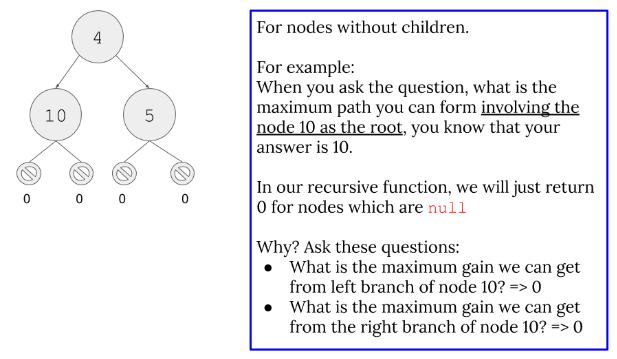
<https://leetcode.com/problems/binary-tree-maximum-path-sum/solutions/603423/python-recursion-stack-thinking-process-diagram/>

This problem requires quite a bit of quirky thinking steps. Take it slow until you fully grasp it.

**Basics**



**Base cases**

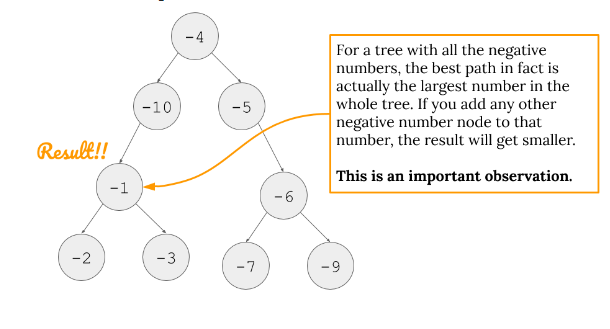


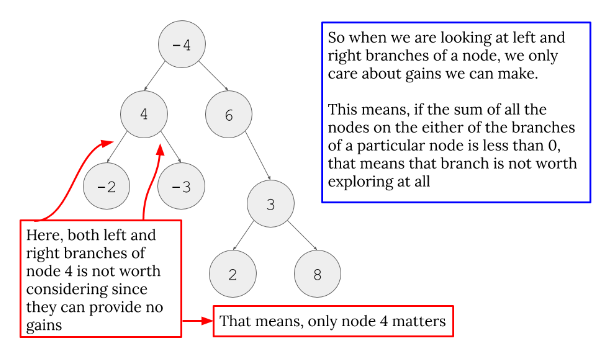
**Important Observations**

These important observations are very important to understand Line 9 and Line 10 in the code.

For example, in the code (Line 9), we do something like max(get\_max\_gain(node.left), 0). The important part is: why do we take maximum value between 0 and maximum gain we can get from left branch? Why 0?

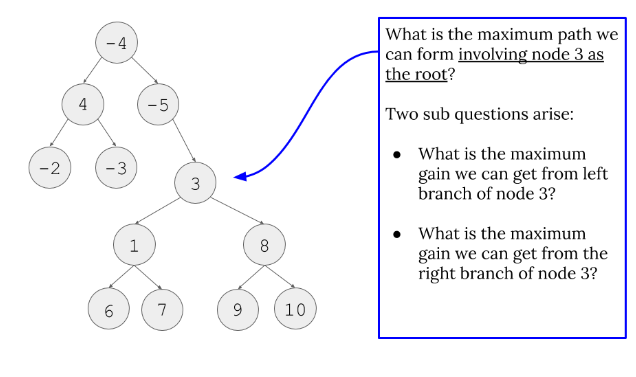
Check the two images below first.

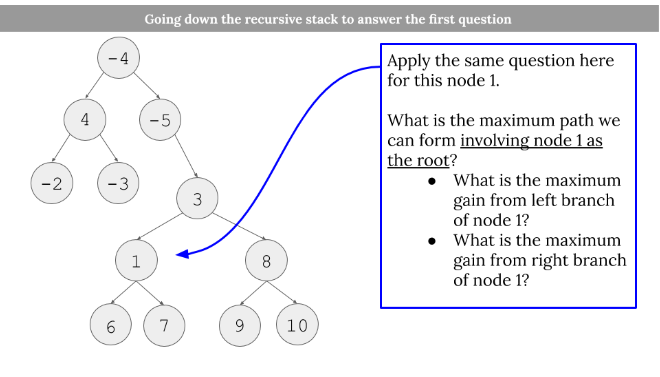


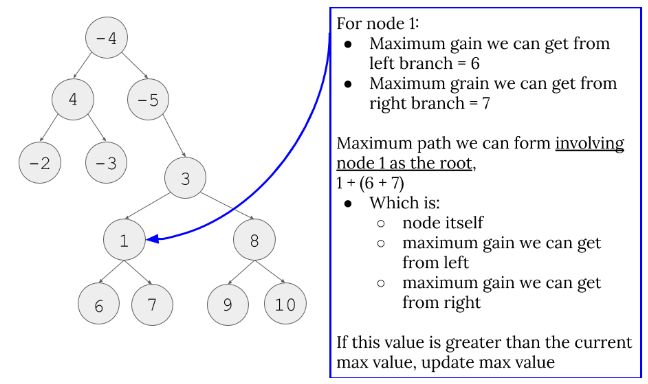


The important thing is "We can only get any sort of gain IF our branches are not below zero. If they are below zero, why do we even bother considering them? Just pick 0 in that case. Therefore, we do max(<some gain we might get or not>, 0).

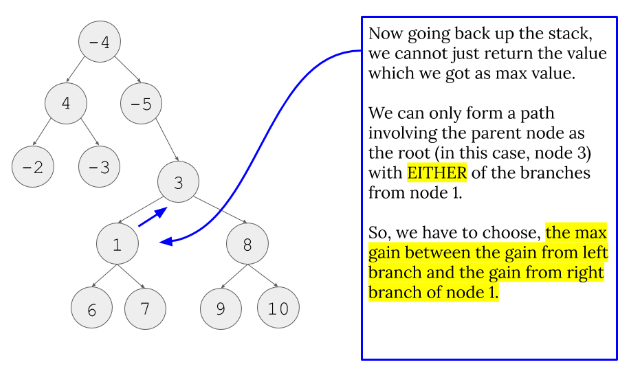
**Going down the recursion stack for one example**







Because of this, we do Line 12 and Line 13. It is important to understand the different between looking for the maximum path INVOLVING the current node in process and what we return for the node which starts the recursion stack. Line 12 and Line 13 takes care of the former issue and Line 15 (and the image below) takes care of the latter issue.



Because of this fact, we have to return like Line 15. For our example, for node 1, which is the recursion call that node 3 does for max(get\_max\_gain(node.left), 0), node 1 cannot include both node 6 and node 7 for a path to include node 3. Therefore, we can only pick the max gain from left path or right path of node 1.

**Python**

1. class Solution:

2.    def maxPathSum(self, root: TreeNode) -> int:

3. max\_path = float("-inf") # placeholder to be updated

4. def get\_max\_gain(node):

5. nonlocal max\_path # This tells that max\_path is not a local variable

6. if node is None:

7. return 0

8.

9. gain\_on\_left = max(get\_max\_gain(node.left), 0) # Read the part important observations

10. gain\_on\_right = max(get\_max\_gain(node.right), 0)  # Read the part important observations

11.

12. current\_max\_path = node.val + gain\_on\_left + gain\_on\_right # Read first three images of going down the recursion stack

13. max\_path = max(max\_path, current\_max\_path) # Read first three images of going down the recursion stack

14.

15. return node.val + max(gain\_on\_left, gain\_on\_right) # Read the last image of going down the recursion stack

16.

17.

18. get\_max\_gain(root) # Starts the recursion chain

19. return max\_path

**Refer to**

<https://algo.monster/liteproblems/124>

**Problem Description**

In this problem, we are given the root of a binary [tree](https://algo.monster/problems/tree_intro) and need to find the maximum path sum from any sequence of nodes in the tree. Here, a path is defined as a sequence of nodes where consecutive nodes are connected by an edge, and a node can only be used once within a single path. The path sum is the total of the values of all nodes in that path. Notably, the path does not have to go through the root of the tree. Our goal is to calculate the maximum sum from all such possible paths in the tree.

**Intuition**

When dealing with trees, recursion is often a natural approach. Since we are looking for the maximum path sum, at every node we have a choice: include that node on a path extending left or right, or start a new path through that node. We can recursively find the maximum path sums going through the left child and the right child. However, when combining these sums at a node, we must realize that we cannot include both child paths since that would create a loop, not a valid path.

The solution's intuition hinges on realizing that for any node, the maximum sum wherein that node is the highest point (i.e., the 'root' of that path) is its value plus the maximum sums of its left and right subtrees. We only add the subtree sums if they are positive, since any path would only include a subtree if it contributed positively to the total sum.

We define the dfs function that does the following:

When the node is None, return 0 because an empty node contributes nothing to the path sum.

Recursively calculate the maximum path sum for the left and right subtrees. If the sum is negative, we reset it to 0, as described earlier.

Calculate the potential maximum path sum at the current node by adding the node's value to the maximum path sums from both subtrees. This represents the largest value that could be achieved when passing through that node and potentially including both left and right paths (but not combining them).

Update a global variable ans that tracks the overall maximum path sum found anywhere in the tree with this new potential maximum.

Finally, for the recursion to continue upwards, return the node's value plus the greater of the two maximum sums from the left or right subtree. This represents the best contribution that node can make towards a higher path.

The maxPathSum function initiates this recursive process, starting from the root, and returns the maximum path sum found.

**Solution Approach**

The solution uses a Depth-First Search (DFS) algorithm to recursively traverse the binary tree and calculate the path sums. This approach ensures that every node is visited, and that the maximum path sums for the subtrees of each node are considered. Here are the steps and ideas involved:

The core function is dfs, which is a recursive function that takes a node of the tree as an argument and returns the maximum path sum obtained by including the current node and extending to either its left or right child (not both).

In the base case, if the current node is None, the function returns 0, meaning there is no contribution to the path sum.

When dfs is called on a non-null node, it first recursively calculates the maximum path sums of the left and right subtrees with left = max(0, dfs(root.left)) and right = max(0, dfs(root.right)). The max(0, ...) pattern is used to ignore negative path sums since including a negative path would decrease the overall sum, which is not optimal.

After obtaining the maximum non-negative path sums from the left and right subtrees, it calculates the maximum path sum that includes the current node and both children: root.val + left + right. This is not the value returned by the function because a path should only extend in one direction; however, this value is crucial because it is the highest path sum for the subtree for which root is the highest point.

The global variable ans is updated with the maximum of its current value and the newly calculated sum (ans = max(ans, root.val + left + right)). This step is essential because ans keeps track of the maximum path sum found in the entire tree, including paths that do not extend to the root of the entire tree.

Finally, dfs returns root.val + max(left, right), taking into account the current node's value and the maximum contribution from either the left or right child. This allows the function to relay upward through the recursion the best path sum contribution of the current node to its parent.

The overall maximum path sum is found by setting the initial value of ans to -inf to ensure that any path sum in the tree will be larger (since the tree is non-empty), and then calling dfs(root) which triggers the recursive process. The final value of ans after the recursion completes gives us the result we are seeking – the maximum path sum of any non-empty path in the given binary tree.

**Example Walkthrough**

Let's illustrate the solution approach with a small binary tree example:

Consider the following binary tree:

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

We want to find the maximum path sum in this tree using the solution approach described.

Start with the function dfs at the root node with the value 5.

The dfs function is called recursively on the left child 4 and the function call sequence continues down to 11.

At 11, dfs goes to 7 (left child), which returns 7 because it is a leaf node.

Then dfs goes to the 2 (right child), which returns 2.

At node 11, the function now has left and right values: the dfs result from the left is 7 and the right is 2. Node 11 itself has a value of 11, so the local maximum we can generate at this node (including its value and both children) is 11 + 7 + 2 = 20. However, for the purposes of propagation up the tree, we return 11 + max(7, 2) = 11 + 7 = 18.

The global maximum ans is updated if 20 is greater than the current value of ans.

The recursion unfolds, and a similar process occurs for node 4 on the left and for 8, 13, 4, and 1 on the right side of the tree.

At node 5, the maximum path sums from left and right sides are collected (already calculated by dfs for child nodes).

The overall maximum path that can be formed including node 5 and its children is compared against the global maximum ans. The value returned is 5 + max(left, right) to propagate up the tree.

In each step, whenever a higher path sum is found at a node, the global maximum ans is updated.

The process continues until the topside of the recursion, where the dfs function was initially called with the root of the tree.

The final value of ans after recursion completes represents the maximum path sum from any sequence of nodes in the given binary tree.

For our example, let's assume that the left side (with nodes 5, 4, and 11) produces a path sum of 18, and the right side (with nodes 5, 8, 4, and 1) produces a path sum of 18 as well. The highest path sum that includes the root would be 5 + 18 + 18 = 41. But suppose there's an even higher sum path that doesn't include the root, say 13 + 8 + 4, which equals 25 (node 13 is the root of this path). Hence, the final answer for this example would depend on which sum is larger, and ans would be updated with the maximum sum found.

**Solution Implementation**

class Solution {

private int maxSum = Integer.MIN\_VALUE; // Initialize maxSum with the smallest possible integer value.

// Returns the maximum path sum of any path that goes through the nodes of the given binary tree.

public int maxPathSum(TreeNode root) {

calculateMaxPathFromNode(root);

return maxSum;

}

// A helper method that computes the maximum path sum from a given node and updates the overall maxSum.

private int calculateMaxPathFromNode(TreeNode node) {

if (node == null) {

// If the current node is null, we return 0 since null contributes nothing to the path sum.

return 0;

}

// Compute and get the maximum sum of paths from the left child;

// if the value is negative, we ignore the left child's contribution by taking 0.

int leftMaxSum = Math.max(0, calculateMaxPathFromNode(node.left));

// Compute and get the maximum sum of paths from the right child;

// if the value is negative, we ignore the right child's contribution by taking 0.

int rightMaxSum = Math.max(0, calculateMaxPathFromNode(node.right));

// Update maxSum with the greater of the current maxSum or the sum of the current node value plus leftMaxSum and rightMaxSum.

// This accounts for the scenario where the path involving the current node and its left and right children yields the max path sum.

maxSum = Math.max(maxSum, node.val + leftMaxSum + rightMaxSum);

// This call must return the maximum path sum including the currently evaluated node and one of its subtrees

// since a path cannot have branches and must be straight through the parents or children nodes.

return node.val + Math.max(leftMaxSum, rightMaxSum);

}

}

/\*\*

\* Definition for a binary tree node.

\*/

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) {

this.val = val;

}

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

**Time and Space Complexity**

The time complexity of the code is O(n) because the dfs function visits each node in the binary tree exactly once, where n is the number of nodes in the binary tree.

The space complexity of the code is O(h), where h is the height of the tree. This is due to the recursion stack during the depth-first search (DFS). In the worst case of a skewed tree, the space complexity becomes O(n), where n is the number of nodes, because the height of the tree can be n in the case of a completely unbalanced tree. For a balanced tree, the space complexity would be O(log n), because the height h would be proportional to log n.

**Refer to**

[L543.P9.6.Diameter of Binary Tree (Ref.L124)](note://85BDF2215E0649FDAF66B25CCC3FCD49)

[L257.Binary Tree Paths (Ref.L1430,L549,L124)](note://0FCBE93DB9274C62B8DB410E76CC3C7E)

[L298.Binary Tree Longest Consecutive Sequence (Ref.L257)](note://AC092AED87E046E5A52ECF94B66C5E4F)

[L549.Binary Tree Longest Consecutive Sequence II (Ref.L257,L298)](note://857146268EE34C6EAC3D963EBC1C1199)