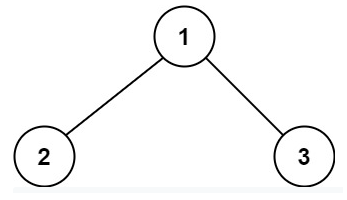
<https://leetcode.com/problems/binary-tree-maximum-path-sum/>

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:**

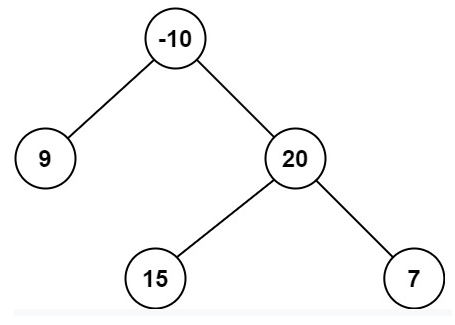


Input: root = [1,2,3]

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 \* 104].
* -1000 <= Node.val <= 1000

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

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

/\*\*

\* 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)

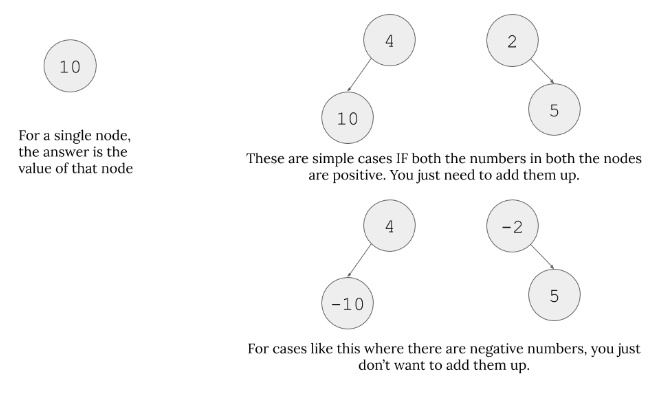
Space Complexity: O(N)

**Refer to**

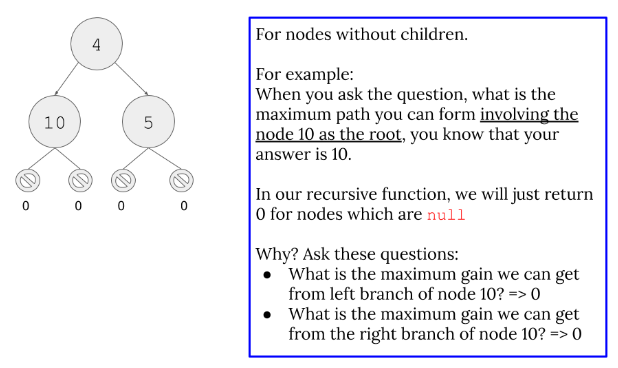
<https://leetcode.com/problems/binary-tree-maximum-path-sum/solutions/603423/python-recursion-stack-thinking-process-diagram/>

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

**Basics**

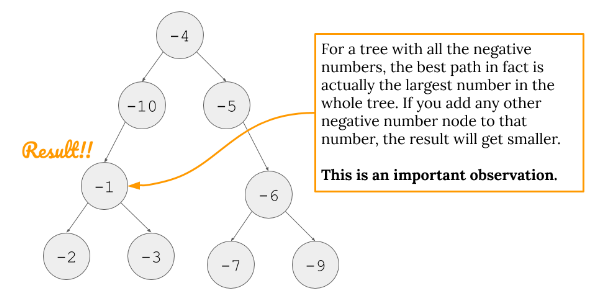


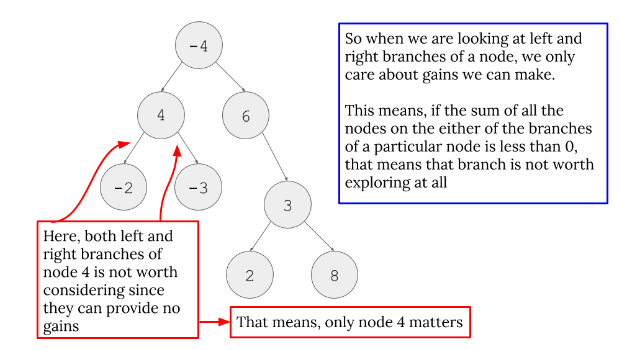
**Base cases**



**Important Observations**

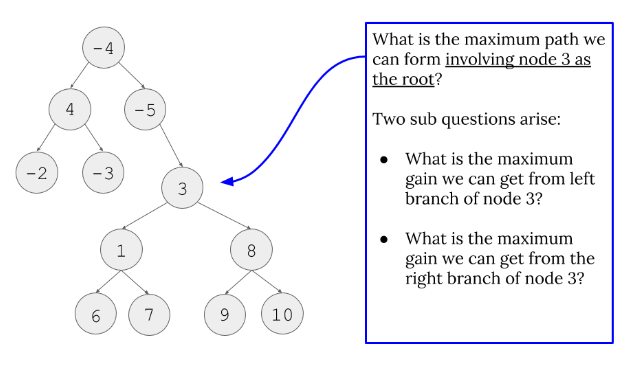
* These important observations are very important to understand Line 9 and Line 10 in the code.
  + For example, in the code (Line 9), we do something like max(get\_max\_gain(node.left), 0). The important part is: why do we take maximum value between 0 and maximum gain we can get from left branch? Why 0?
  + Check the two images below first.

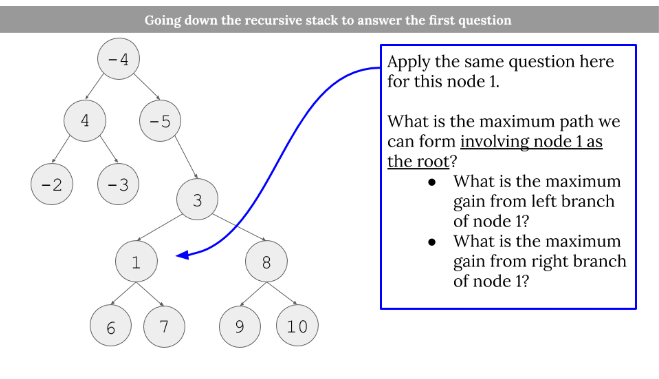


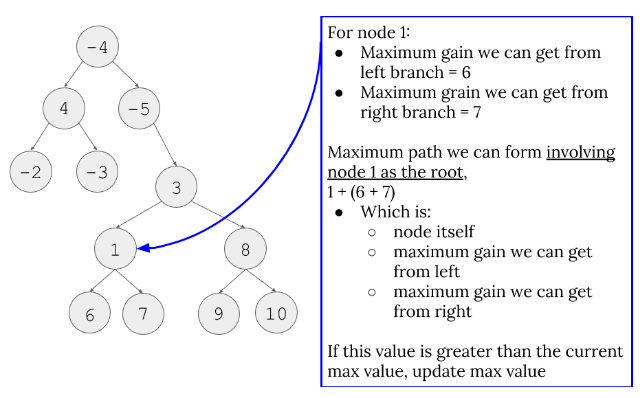


* The important thing is "We can only get any sort of gain IF our branches are not below zero. If they are below zero, why do we even bother considering them? Just pick 0 in that case. Therefore, we do max(<some gain we might get or not>, 0).

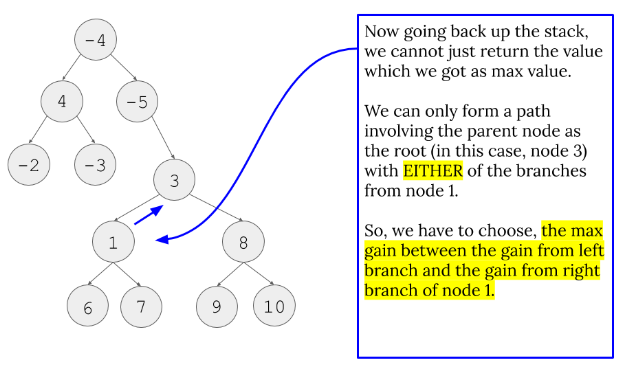
**Going down the recursion stack for one example**







* 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