**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:**

A picture containing text, clipart

Description automatically generated

**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:**

A picture containing clipart

Description automatically generated

**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

/\*\*

\* Definition for a binary tree node.

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

\* this.val = (val===undefined ? 0 : val)

\* this.left = (left===undefined ? null : left)

\* this.right = (right===undefined ? null : right)

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var maxPathSum = function(root) {

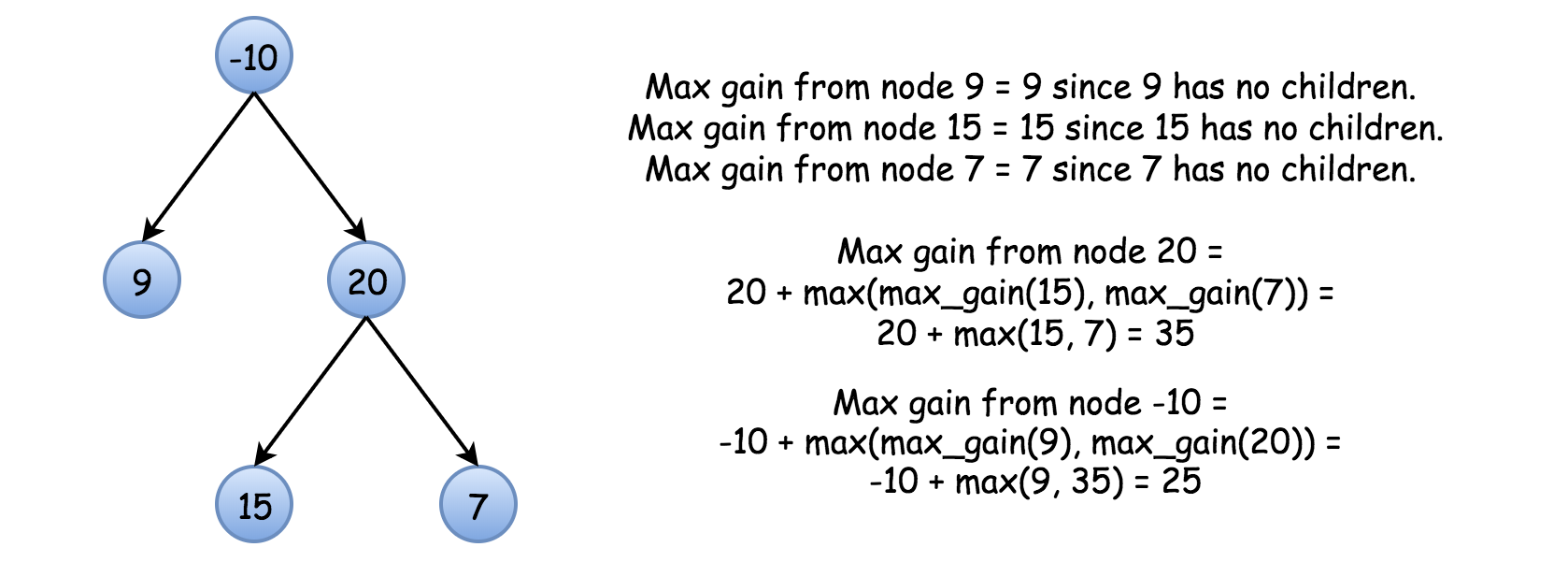
};

Approach 1: Recursion

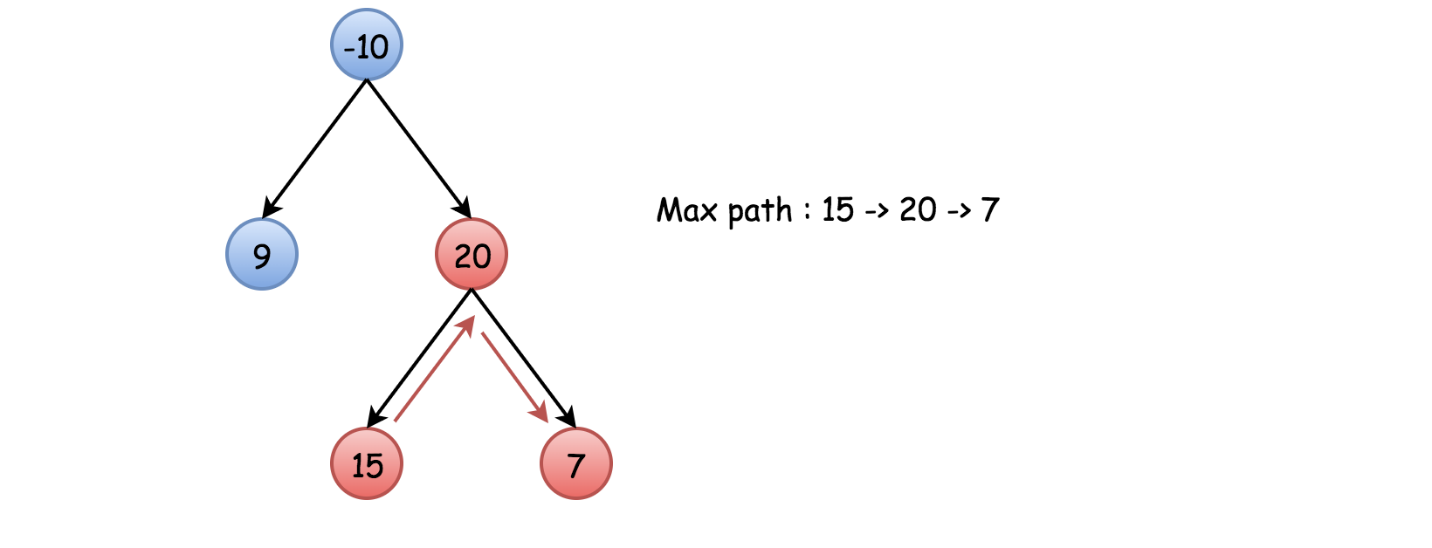
**Intuition**

First of all, let's simplify the problem and implement a function max\_gain(node) which takes a node as an argument and computes a maximum contribution that this node and one/zero of its subtrees could add.

In other words, it's a *maximum gain* one could have including the node (and maybe one of its subtrees) into the path.



Hence if one would know for sure that the max path contains root, the problem would be solved as max\_gain(root). Unfortunately, *the max path does not need to go through the root*, and here is an example of such a tree



That means one needs to modify the above function and to check at each step what is better : to continue the current path or to start a new path with the current node as a highest node in this new path.

**Algorithm**

Now everything is ready to write down an algorithm.

* Initiate max\_sum as the smallest possible integer and call max\_gain(node = root).
* Implement max\_gain(node) with a check to continue the old path/to start a new path:
  + Base case : if node is null, the max gain is 0.
  + Call max\_gain recursively for the node children to compute max gain from the left and right subtrees : left\_gain = max(max\_gain(node.left), 0) and  
    right\_gain = max(max\_gain(node.right), 0).
  + Now check to continue the old path or to start a new path. To start a new path would cost price\_newpath = node.val + left\_gain + right\_gain. Update max\_sum if it's better to start a new path.
  + For the recursion return the max gain the node and one/zero of its subtrees could add to the current path : node.val + max(left\_gain, right\_gain).

**Tree Node**

Here is the definition of the TreeNode which we would use in the following implementation.

/\* Definition for a binary tree node. \*/

public class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) {

val = x;

}

}

class Solution {

int max\_sum = Integer.MIN\_VALUE;

public int max\_gain(TreeNode node) {

if (node == null) return 0;

// max sum on the left and right sub-trees of node

int left\_gain = Math.max(max\_gain(node.left), 0);

int right\_gain = Math.max(max\_gain(node.right), 0);

// the price to start a new path where `node` is a highest node

int price\_newpath = node.val + left\_gain + right\_gain;

// update max\_sum if it's better to start a new path

max\_sum = Math.max(max\_sum, price\_newpath);

// for recursion :

// return the max gain if continue the same path

return node.val + Math.max(left\_gain, right\_gain);

}

public int maxPathSum(TreeNode root) {

max\_gain(root);

return max\_sum;

}

}

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N is number of nodes, since we visit each node not more than 2 times.
* Space complexity: \mathcal{O}(H)O(*H*), where H*H* is a tree height, to keep the recursion stack. In the average case of balanced tree, the tree height H = \log N*H*=log*N*, in the worst case of skewed tree, H = N*H*=*N*.