# Question

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

**Example 2:**



**Input:** root = [1,2,3], targetSum = 5

**Output:** false

**Example 3:**

**Input:** root = [1,2], targetSum = 0

**Output:** false

**Constraints:**

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

# Solution

#### **Binary tree definition**

First of all, 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;  }  } |

#### **Approach 1: Recursion**

The most intuitive way is to use a recursion here. One is going through the tree by considering at each step the node itself and its children. If node is not a leaf, one calls recursively hasPathSum method for its children with a sum decreased by the current node value. If node is a leaf, one checks if the the current sum is zero, i.e if the initial sum was discovered.

|  |
| --- |
| class Solution {  public boolean hasPathSum(TreeNode root, int sum) {  if (root == null)  return false;  sum -= root.val;  if ((root.left == null) && (root.right == null))  return (sum == 0);  return hasPathSum(root.left, sum) || hasPathSum(root.right, sum);  }  } |

**Complexity Analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*), where N*N* is the number of nodes.
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only one child node, the recursion call would occur N*N* times (the height of the tree), therefore the storage to keep the call stack would be \mathcal{O}(N)O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be \log(N)log(*N*). Therefore, the space complexity in this case would be \mathcal{O}(\log(N))O(log(*N*)).

#### **Approach 2: Iterations**

**Algorithm**

We could also convert the above recursion into iteration, with the help of stack. DFS would be better than BFS here since it works faster except the worst case. In the worst case the path root->leaf with the given sum is the last considered one and in this case DFS results in the same productivity as BFS.

The idea is to visit each node with the DFS strategy, while updating the remaining sum to cumulate at each visit.

So we start from a stack which contains the root node and the corresponding remaining sum which is sum - root.val. Then we proceed to the iterations: pop the current node out of the stack and return True if the remaining sum is 0 and we're on the leaf node. If the remaining sum is not zero or we're not on the leaf yet then we push the child nodes and corresponding remaining sums into stack.

|  |
| --- |
| class Solution {  public boolean hasPathSum(TreeNode root, int sum) {  if (root == null)  return false;  LinkedList<TreeNode> node\_stack = new LinkedList();  LinkedList<Integer> sum\_stack = new LinkedList();  node\_stack.add(root);  sum\_stack.add(sum - root.val);  TreeNode node;  int curr\_sum;  while ( !node\_stack.isEmpty() ) {  node = node\_stack.pollLast();  curr\_sum = sum\_stack.pollLast();  if ((node.right == null) && (node.left == null) && (curr\_sum == 0))  return true;  if (node.right != null) {  node\_stack.add(node.right);  sum\_stack.add(curr\_sum - node.right.val);  }  if (node.left != null) {  node\_stack.add(node.left);  sum\_stack.add(curr\_sum - node.left.val);  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : the same as the recursion approach \mathcal{O}(N)O(*N*).
* Space complexity : \mathcal{O}(N)O(*N*) since in the worst case, when the tree is completely unbalanced, e.g. each node has only one child node, we would keep all N*N* nodes in the stack. But in the best case (the tree is balanced), the height of the tree would be \log(N)log(*N*). Therefore, the space complexity in this case would be \mathcal{O}(\log(N))O(log(*N*)).