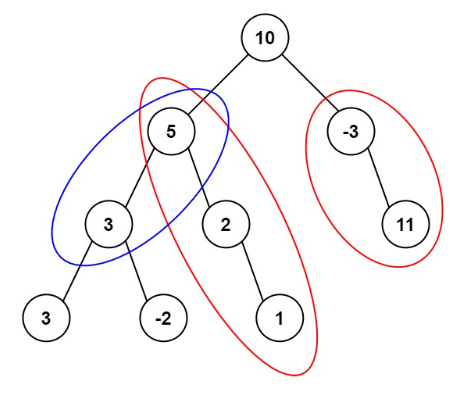
<https://leetcode.com/problems/path-sum-iii/>

Given the root of a binary tree and an integer targetSum, return *the number of paths where the sum of the values along the path equals* targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

**Example 1:**



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

Output: 3

Explanation: The paths that sum to 8 are shown.

**Example 2:**

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

Output: 3

**Constraints:**

* The number of nodes in the tree is in the range [0, 1000].
* -109 <= Node.val <= 109
* -1000 <= targetSum <= 1000

**Attempt 1: 2022-11-05**

**Solution 1:  Brute force two layer recursive traversal with global variable (10min)**

/\*\*

\* 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 {

// Have to update 'count', 'targetSum' from int to long

// New test case

// Input: [1000000000,1000000000,null,294967296,null,1000000000,null,1000000000,null,1000000000], 0

// Output: 2

// Expected: 0

//int count;

long count;

public int pathSum(TreeNode root, long targetSum) {

count = 0;

helper(root, targetSum);

return (int)count;

}

private void helper(TreeNode root , long targetSum) {

if(root == null) {

return;

}

checkForEachNode(root, targetSum);

helper(root.left, targetSum);

helper(root.right, targetSum);

}

private void checkForEachNode(TreeNode root, long targetSum) {

if(root == null) {

return;

}

if(root.val == targetSum) {

count++;

}

checkForEachNode(root.left, targetSum - root.val);

checkForEachNode(root.right, targetSum - root.val);

}

}

Time Complexity: O(nlogn) ~ O(n^2), where n is number of nodes in the Binary Tree

Space Complexity: O(logn) ~ O(n)

**Solution 2:  Brute force two layer recursive traversal as divide and conquer (10min)**

/\*\*

\* 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 {

public int pathSum(TreeNode root, int targetSum) {

// Base case

if(root == null) {

return 0;

}

// Divide

// Rquire second layer DFS to find number of paths for current node

long currentNodeNumOfPaths = helper(root, targetSum);

long allLeftSubtreeNodesNumOfPaths = pathSum(root.left, targetSum);

long allRightSubtreeNodesNumOfPaths = pathSum(root.right, targetSum);

// Conquer

return (int)(currentNodeNumOfPaths + allLeftSubtreeNodesNumOfPaths + allRightSubtreeNodesNumOfPaths);

}

private long helper(TreeNode root, long targetSum) {

long count = 0;

// Base case

if(root == null) {

return count;

}

if(root.val == targetSum) {

count++;

}

// Divide

long left = helper(root.left, targetSum - root.val);

long right = helper(root.right, targetSum - root.val);

// Conquer

return count + left + right;

}

}

Time Complexity: O(nlogn) ~ O(n^2), where n is number of nodes in the Binary Tree

Space Complexity: O(logn) ~ O(n)

**Solution 3:  Prefix Sum with backtracking (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 {

public int pathSum(TreeNode root, int targetSum) {

if(root == null) {

return 0;

}

Map<Long, Integer> map = new HashMap<Long, Integer>();

map.put(0L, 1);

return helper(root, targetSum, 0, map);

}

private int helper(TreeNode root, int targetSum, long currSum, Map<Long, Integer> map) {

if(root == null) {

return 0;

}

currSum += root.val;

int numOfPathsFromRootToCurrentNode = map.getOrDefault(currSum - targetSum, 0);

map.put(currSum, map.getOrDefault(currSum, 0) + 1);

int left = helper(root.left, targetSum, currSum, map);

int right = helper(root.right, targetSum, currSum, map);

numOfPathsFromRootToCurrentNode += left + right;

map.put(currSum, map.get(currSum) - 1);

return numOfPathsFromRootToCurrentNode;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**1. How prefix sum work with DFS traversal ?**

**Refer to**

<https://leetcode.com/problems/path-sum-iii/discuss/91878/17-ms-O(n)-java-Prefix-sum-method/552675>

I would give some brief thoughts after I struggle there for quite long time. Firstly, let's explain some usage of function and variable

* the function pathSumHelper actually returns the number of paths given the target (target sum) and currSum (sum of root to parent of current node)
* map actually contains all the possible sums of path between root node to middle nodes in the branch.

So map.getOrDefault(currSum-target, 0) will find out the number of paths that equals to currSum-target, **which means sum of root to middle node, every path of (root -> middle node == currSum-target) will construct a path from (middle node -> current node).**

In short, it is like root node -> middle node -> current node, every root -> middle == currSum-target will make sure that middle node -> current node == target since root -> current node == currSum.

[**Illustration**](https://leetcode.com/problems/path-sum-iii/discuss/91878/17-ms-O(n)-java-Prefix-sum-method/719476)

It's not very hard if we understand the elementary idea:

Given we have a route:

·················sum(a,b)···········sum(b,c)

a----------------b------------------c

we know that, sum(a,c) = sum(a,b) + sum(b,c)

if target == sum(b,c), then sum(a,c) = sum(a,b) + target

Now, we could infer that, if exists a point b, that conforming sum(a,b) = sum(a,c) - target,

the, b-c is the path we want to find.

So we use that result numOfPaths and go to its children to see if there are some other paths. **The two put statements of hash map is used for backtracking in order to make sure that possible paths from others branches (not from this root -> current node) is not counted into final result.**

**Note: map.put(0, 1) is to make sure root is also counted. Otherwise all the paths containing root will not be counted.**

Please see example given in question. Consider Target is 15. Now, 10 + 5 = 15. It is direct path from Root to One node. Now, OldPath = CurPath (15) - Target (15) = 0. This is valid result. Thus we have initiated cache as {0:1}.

**Implementation**

HashMap<Integer, Integer> map = new HashMap<>();

public int pathSum(TreeNode root, int sum) {

if(root == null) {

return 0;

}

map.put(0, 1);

return pathSumHelper(root, 0, sum);

}

private int pathSumHelper(TreeNode node, int currSum, int target) {

if(node == null) {

return 0;

}

currSum += node.val;

int numOfPaths = map.getOrDefault(currSum-target, 0);

map.put(currSum, map.getOrDefault(currSum, 0)+1);

numOfPaths += pathSumHelper(node.left, currSum, target) + pathSumHelper(node.right, currSum, target);

map.put(currSum, map.get(currSum)-1);

return numOfPaths;

}

**2. Why need backtracking ?**

**Explain 1:**

<https://leetcode.com/problems/path-sum-iii/discuss/91878/17-ms-O(n)-java-Prefix-sum-method/96417>

I just wanted to add one additional point, which took me some time to think through. I was wondering how you can keep track of a sequence of sums in a 1D hash table, when a tree can have multiple branches, how do you keep track of duplicate sums on different branches of the tree? The answer is that this method only keeps track of 1 branch at a time. Because we're doing a depth first search, we will iterate all the way to the end of a single branch before coming back up. However, as we're coming back up, we're removing the nodes at the bottom of the branch from our hash table, using line map.put(sum, map.get(sum) - 1);, before ending the function.

To summarize, the hash table is only keeping track of a portion of a single branch at any given time, from the root node to the current node only.

**Explain 2:**

<https://leetcode.com/problems/path-sum-iii/discuss/141424/Python-step-by-step-walk-through.-Easy-to-understand.-Two-solutions-comparison.-:-)/1612879>

Someone mentioned that it is similar to backtracking; I agree with this. The variable "currPathSum" can be thought of as a "sum from root to the current node". **The cache's job is to store all possible sum from root to the current node only.**

So say that we have a very simple case with target 5

10

/ \

5 2

/

3

When you are at root, your cache will be {10:1}

When you are at left child, your cache will be {10: 1, 15: 1}.

Now since we finish left path traversal and switch to right path, we will remove count for 15 in the cache as {10: 1, 15: 0}

When you are at right child (2), your cache will be {10: 1, 15:0, 12:1}. This makes sense, as we see that there is no sum from root to node (2) that can add up to 15, all possible sum from root to (2) are stored in cache.

When you are at right child (3), your cache will be {10:1, 15:1, 12:1}.

By doing this, we make sure that we don't double count total sum "15" when we are in the right side of the tree.

**3. Why need to combine as "numOfPathsFromRootToCurrentNode += left + right" ?**

<https://leetcode.com/problems/path-sum-iii/discuss/91889/Simple-Java-DFS/364604>

For people who are confused why calling pathSumFrom(root, sum) + pathSum(root.left, sum) + pathSum(root.right, sum);

Think in this way: because the search could start from any node in the tree with sum, we should see every node as root and do a DFS. Thus the O(n^2) time complexity

**Another full explain**

<https://leetcode.com/problems/path-sum-iii/discuss/141424/Python-step-by-step-walk-through.-Easy-to-understand.-Two-solutions-comparison.-%3A>-)

**1. Brute Force: O(nlogn) ~ O(n^2)**

**1.1 High level walk-through:**

1. (Define return) Define a global var: self.numOfPaths in the main function.
2. (1st layer DFS) Use recursive traverse to go through each node (can be **any order**: pre, in, post all fine).
3. (2nd layer DFS) For each node, walk all paths. If a path sum equals to the target: self.numOfPaths += 1
4. Return result: return self.numOfPaths

**1.2 Complexity analysis**

**1.2.1 Space**

1. Space complexity is O(1), due to there is no extra cache. However, for **any recursive** question, we need to think about **stack overflow**, namely the recursion should not go too deep.
2. Assume we have **n** TreeNodes in total, the tree height will vary from **O(n)** (single sided tree) to **O(logn)**(balanced tree).
3. The two DFS will go as deep as the tree height.

**1.2.2 Time**

1. Time complexity depends on the two DFS.
2. 1st layer DFS will always take **O(n)**, due to here we will take each node out, there are in total **n** TreeNodes
3. 2nd layer DFS will take range from **O(n)** (single sided tree) to **O(logn)**(balanced tree). This is due to here we are get all the paths from a given node. The length of path is proportional to the tree height.
4. Therefore, the total time complexity is **O(nlogn)** to **O(n^2)**.

**1.3 Code**

class Solution(object):

def pathSum(self, root, target):

"""

:type root: TreeNode

:type sum: int

:rtype: int

"""

# define global return var

self.numOfPaths = 0

# 1st layer DFS to go through each node

self.dfs(root, target)

# return result

return self.numOfPaths

# define: traverse through the tree, at each treenode, call another DFS to test if a path sum include the answer

def dfs(self, node, target):

# exit condition

if node is None:

return

# dfs break down

self.test(node, target) # you can move the line to any order, here is pre-order

self.dfs(node.left, target)

self.dfs(node.right, target)

# define: for a given node, DFS to find any path that sum == target, if find self.numOfPaths += 1

def test(self, node, target):

# exit condition

if node is None:

return

if node.val == target:

self.numOfPaths += 1

# test break down

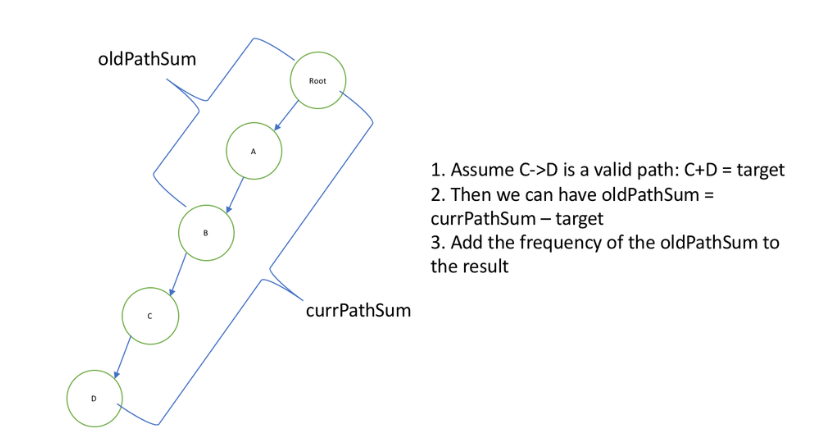
self.test(node.left, target-node.val)

self.test(node.right, target-node.val)

**2. Memorization of path sum: O(n)**

**2.1 High level walk through**

1. In order to optimize from the brutal force solution, we will have to think of a clear way to memorize the intermediate result. Namely in the brutal force solution, we did a lot repeated calculation. For example 1->3->5, we calculated: 1, 1+3, 1+3+5, 3, 3+5, 5.
2. This is a classical 'space and time tradeoff': we can create a dictionary (named cache) which saves all the path sum (from root to current node) and their frequency.
3. Again, we traverse through the tree, at each node, we can get the currPathSum (from root to current node). If within this path, there is a valid solution, then there must be a oldPathSum such that currPathSum - oldPathSum = target.
4. We just need to add the frequency of the oldPathSum to the result.
5. During the DFS break down, we need to -1 in cache[currPathSum], because this path is not available in later traverse.
6. Check the graph below for easy visualization.



**2.2 Complexity analysis:**

**2.2.1 Space complexity**

O(n) extra space

**2.2.1 Time complexity**

O(n) as we just traverse once

**2.3 Code:**

class Solution(object):

def pathSum(self, root, target):

# define global result and path

self.result = 0

cache = {0:1}

# recursive to get result

self.dfs(root, target, 0, cache)

# return result

return self.result

def dfs(self, root, target, currPathSum, cache):

# exit condition

if root is None:

return

# calculate currPathSum and required oldPathSum

currPathSum += root.val

oldPathSum = currPathSum - target

# update result and cache

self.result += cache.get(oldPathSum, 0)

cache[currPathSum] = cache.get(currPathSum, 0) + 1

# dfs breakdown

self.dfs(root.left, target, currPathSum, cache)

self.dfs(root.right, target, currPathSum, cache)

# when move to a different branch, the currPathSum is no longer available, hence remove one.

cache[currPathSum] -= 1