Path Sum III

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

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

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

Example:

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

Return 3. The paths that sum to 8 are:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

Solution 1

So the idea is similar as Two sum, using HashMap to store (key : the prefix sum, value : how many ways get to this prefix sum) , and whenever reach a node, we check if prefix sum - target exists in hashmap or not, if it does, we added up the ways of prefix sum - target into res.
For instance : in one path we have 1,2,-1,-1,2, then the prefix sum will be: 1, 3, 2, 1, 3, let's say we want to find target sum is 2, then we will have {2}, {1, 2, -1, -1, 2}, { 2, -1, -1, 2} ways.

I used global variable count, but obviously we can avoid global variable by passing the count from bottom up. The time complexity is O(n). This is my first post in discuss, open to any improvement or criticism. :)

```
    public int pathSum(TreeNode root, int sum) {
        HashMap<Integer, Integer> preSum = new HashMap();
        preSum.put(0,1);
        helper(root, 0, sum, preSum);
        return count;
    }
    int count = 0;
    public void helper(TreeNode root, int sum, int target, HashMap<Integer, Integ
er> preSum) {
        if (root == null) {
            return;
        }

        sum += root.val;

        if (preSum.containsKey(sum - target)) {
            count += preSum.get(sum - target);
        }

        if (!preSum.containsKey(sum)) {
            preSum.put(sum, 1);
        } else {
            preSum.put(sum, preSum.get(sum)+1);
        }

        helper(root.left, sum, target, preSum);
        helper(root.right, sum, target, preSum);
        preSum.put(sum, preSum.get(sum) - 1);
    }
```

Thanks for your advice, @StefanPochmann . Here is the modified version, concise and shorter:

```
    public int pathSum(TreeNode root, int sum) {
        HashMap<Integer, Integer> preSum = new HashMap();
        preSum.put(0,1);
        return helper(root, 0, sum, preSum);
    }

    public int helper(TreeNode root, int sum, int target, HashMap<Integer, Intege
r> preSum) {
        if (root == null) {
            return 0;
        }

        sum += root.val;
        int res = preSum.getOrDefault(sum - target, 0);
        preSum.put(sum, preSum.getOrDefault(sum, 0) + 1);

        res += helper(root.left, sum, target, preSum) + helper(root.right, sum, t
arget, preSum);
        preSum.put(sum, preSum.get(sum) - 1);
        return res;
    }
```

written by tankztc original link here

Solution 2

Each time find all the path start from current node
Then move start node to the child and repeat.
Time Complexity should be O(N^2) for the worst case and O(NlogN) for balanced binary Tree.

```
public class Solution {
    public int pathSum(TreeNode root, int sum) {
        if(root == null)
            return 0;
        return findPath(root, sum) + pathSum(root.left, sum) + pathSum(root.right
, sum);
    }

    public int findPath(TreeNode root, int sum){
        int res = 0;
        if(root == null)
            return res;
        if(sum == root.val)
            res++;
        res += findPath(root.left, sum - root.val);
        res += findPath(root.right, sum - root.val);
        return res;
    }

}
```

A better solution is suggested in 17ms O(n) java prefix sum by tankztc. It use a hash map to store all the prefix sum and each time check if the any subarray sum to the target, add with some comments:

```
    public int pathSum(TreeNode root, int sum) {
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);  //Default sum = 0 has one count
        return backtrack(root, 0, sum, map);
    }
    //BackTrack one pass
    public int backtrack(TreeNode root, int sum, int target, Map<Integer, Integer
> map){
        if(root == null)
            return 0;
        sum += root.val;
        int res = map.getOrDefault(sum - target, 0);    //See if there is a subar
ray sum equals to target
        map.put(sum, map.getOrDefault(sum, 0)+1);
        //Extend to left and right child
        res += backtrack(root.left, sum, target, map) + backtrack(root.right, sum
, target, map);
        map.put(sum, map.get(sum)-1);   //Remove the current node so it wont affe
ct other path
        return res;
    }
```

written by star1993 original link here

Solution 3

For tree structure problems. recursion is usually intuitive and easy to write. lol

```
class Solution {
public:
    int pathSum(TreeNode* root, int sum) {
        if(!root) return 0;
        return sumUp(root, 0, sum) + pathSum(root->left, sum) + pathSum(root->rig
ht, sum);
    }
private:
    int sumUp(TreeNode* root, int pre, int& sum){
        if(!root) return 0;
        int current = pre + root->val;
        return (current == sum) + sumUp(root->left, current, sum) + sumUp(root->r
ight, current, sum);
    }
};
```

written by Undo original link here