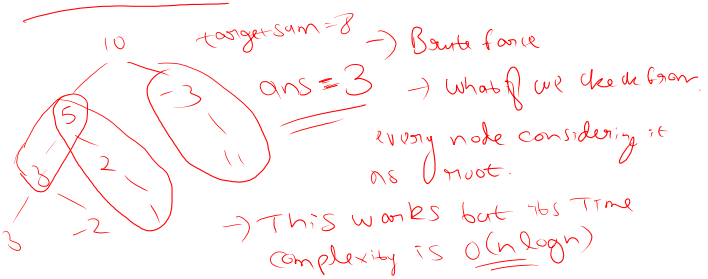
PATH SUM



targetSum = 8

→ Brute force

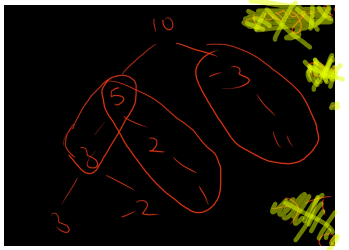
ans = 3

→ What if we check from every node considering it as root.

→ This works but its Time complexity is O(nlogn)

Can we find more optimal soln?

→ What if we keep an array of all possible paths excluding target sum, because in the previous approach we are visiting them again & again.



10
10 5
10 5 3
10 5 3 3
10 5 3 -2
10 5 2
10 5 2 1
10 -3
10 -3 11

This is an excellent idea and took me some time to figure out the logic behind.
Hope my comment below could help some folks better understand this solution.

1. The prefix stores the sum from the root to the current node in the recursion
2. The map stores <prefix sum, frequency> pairs before getting to the current node. We can imagine a path from the root to the current node. The sum from any node in the middle of the path to the current node = the difference between the sum from the root to the current node and the prefix sum of the node in the middle.
3. We are looking for some consecutive nodes that sum up to the given target value, which means the difference discussed in 2. should equal to the target value. In addition, we need to know how many differences are equal to the target value.
4. Here comes the map. The map stores the frequency of all possible sum in the path to the current node. If the difference between the current sum and the target value exists in the map, there must exist a node in the middle of the path, such that from this node to the current node, the sum is equal to the target value.
5. Note that there might be multiple nodes in the middle that satisfy what is discussed in 4. The frequency in the map is used to help with that.
6. Therefore, in each recursion, the map stores all information we need to calculate the number of ranges that sum up to target. Note that each range starts from a middle node, ended by the current node.
7. To get the total number of path count, we add up the number of valid paths ended by EACH node in the tree.
8. Each recursion returns the total count of valid paths in the subtree rooted at the current node. And this sum can be divided into three parts:
   - the total number of valid paths in the subtree rooted at the current node's left child
   - the total number of valid paths in the subtree rooted at the current node's right child
   - the number of valid paths ended by the current node

The interesting part of this solution is that the prefix is counted from the top(root) to the bottom(leaves), and the result of total count is calculated from the bottom to the top :D

```
public int pathSumRec(TreeNode root, int currentSum,HashMap<Integer,Integer> ma
    if(root == null){
        return 0;
    }

    currentSum += root.data;

    int numberOfPaths = map.getOrDefault(currentSum - targetSum,0);

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

    int result = pathSumRec(root.left,currentSum,map,targetSum)+pathSumRec(root

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

    return result+numberOfPaths;


}
```