


 pinglu85 / algoExpert Public[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) main ▾

...

[algoExpert](#) / [Easy](#) / branch-sums.md

pinglu85 update content

 History 1 contributor

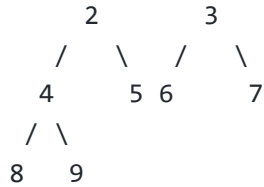
Branch Sums

Understanding the problem

Given a Binary Tree, we are asked to compute all of the branch sums of the tree and return them in an array, ordered from leftmost branch sum to rightmost branch sum. In a tree, a branch is a path that starts at the root node and ends at one of the leaf nodes. A branch sum means the sum of all values in a branch.

Sample Input:

```
tree =    1
        /  \
```



The output should be:

[15, 16, 8, 10, 11]

Approach 1: Iterative DFS

We traverse the binary tree using iterative Depth-First Search. Since we need to compute the branch sum, we are going to need to not only store every node in the stack, but also store the running sum till each node. When we get to a leaf node, we push the running sum to the output array. Because a stack stores items in an Last-In/First-Out manner, if we first push the left child node onto the stack then the right child node, at next iteration, the right child node will come off the stack first. Since the branch sums should be ordered from leftmost branch sum to rightmost branch sum, we need to push the right child node onto the stack first.

Implementation

JavaScript

```

// This is the class of the input root.
// Do not edit it.
class BinaryTree {
  constructor(value) {
    this.value = value;
    this.left = null;

```

```
    this.right = null;
  }
}

function branchSums(root) {
  const sums = [];
  const stack = [{ node: root, runningSum: 0 }];

  while (stack.length > 0) {
    const { node, runningSum } = stack.pop();

    const newRunningSum = runningSum + node.value;

    if (!node.left && !node.right) {
      sums.push(newRunningSum);
      continue;
    }

    if (node.right) {
      stack.push({ node: node.right, runningSum: newRunningSum });
    }

    if (node.left) {
      stack.push({ node: node.left, runningSum: newRunningSum });
    }
  }

  return sums;
}
```

Go:

```
package main

// This is the struct of the input root. Do not edit it.
type BinaryTree struct {
```

```
Value int
Left  *BinaryTree
Right *BinaryTree
}

type unvisited struct {
    node      *BinaryTree
    runningSum int
}

func BranchSums(root *BinaryTree) []int {
    sums := []int{}
    stack := []unvisited{unvisited{root, 0}}

    for len(stack) > 0 {
        lastIdx := len(stack) - 1
        node, runningSum := stack[lastIdx].node, stack[lastIdx].runningSum
        stack = stack[:lastIdx]

        runningSum += node.Value

        if node.Left == nil && node.Right == nil {
            sums = append(sums, runningSum)
            continue
        }

        if node.Right != nil {
            stack = append(stack, unvisited{node.Right, runningSum})
        }

        if node.Left != nil {
            stack = append(stack, unvisited{node.Left, runningSum})
        }
    }

    return sums
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the number of nodes in the Binary Tree.
- Space Complexity: $O(N)$.

☰ 204 lines (151 sloc) | 5.56 KB

...

Approach 2: Recursive DFS

Instead of using a stack to remember the next node to be visited and the running sum till that node, we can also use the call stack to track these info. We are going to define a recursive function `calculateBranchSums` and call it on each node, starting from the root node. At each recursive call, we will pass down the next node to visit, the sum of values from nodes above the node we are going to visit, and an array of branch sums which starts out empty. Whenever we get to a leaf node, we push the sum of all values in the current branch into the array of branch sums.

Implementation

JavaScript:

```
// This is the class of the input root.
// Do not edit it.
class BinaryTree {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

function branchSums(root) {
  const sums = [];
  calculateBranchSums(root, 0, sums);
}
```

```
    return sums;
}

function calculateBranchSums(node, runningSum, sums) {
    if (!node) return;

    const newRunningSum = runningSum + node.value;
    if (!node.left && !node.right) {
        sums.push(newRunningSum);
    }

    calculateBranchSums(node.left, newRunningSum, sums);
    calculateBranchSums(node.right, newRunningSum, sums);
}
```

Go:

```
package main

// This is the struct of the input root. Do not edit it.
type BinaryTree struct {
    Value int
    Left  *BinaryTree
    Right *BinaryTree
}

func BranchSums(root *BinaryTree) []int {
    sums := []int{}
    calculateBranchSums(root, 0, &sums)
    return sums
}

func calculateBranchSums(node *BinaryTree, runningSum int, sums *[]int) {
    if node == nil {
        return
    }
}
```

```
    runningSum += node.Value
    if node.Left == nil && node.Right == nil {
        *sums = append(*sums, runningSum)
        return
    }

    calculateBranchSums(node.Left, runningSum, sums)
    calculateBranchSums(node.Right, runningSum, sums)
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the number of nodes in the Binary Tree.
- Space Complexity: $O(N)$, where N is the number of nodes in the Binary Tree. Each recursive call to the `calculateBranchSums` function adds a new frame on the call stack. On average we will never have more than $\log(N)$ recursive calls on the call stack, since we eliminate half the nodes in the remaining tree at each recursive call. In the worst case, when the input tree is very imbalanced, we would have $O(N)$ space from the recursive calls, since we would have N recursive calls on the call stack at once. Besides the space utilized by the recursive calls, we also return an array of branch sums. The size of the array is same as the number of branches in the Binary Tree, which is the number of leaf nodes in the Binary Tree. There are roughly half of N leaf nodes in the Binary Tree and half of N is equal to $O(N)$ in the space time complexity analysis.