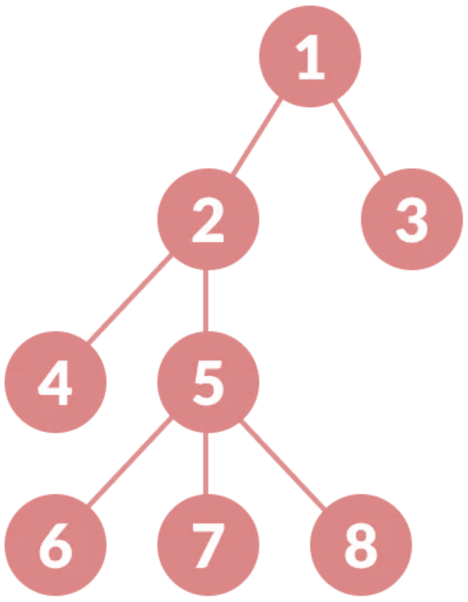
Number of Nodes in the Subtree

Problem Statement: Given a tree with N nodes, rooted at node 1, find the number of nodes in subtree of each node.

Note: The subtree of a node u is defined as the set of all nodes v such that u is an ancestor of v, including u itself.



Example: Consider the tree above. The number of nodes in subtree of each of the nodes is as follows:

1 -> 8
2 -> 6
3 -> 1
4 -> 1
5 -> 4
6 -> 1
7 -> 1
8 -> 1

Naive Algorithm: For each node u in the tree, we can start a dfs starting at that node and visiting only the nodes in it's subtree. So our answer for a particular node will be the total number of nodes visited in that call.
The code for the above approach is as follows :

Pseudocode:

```
/*
        recursive dfs function that takes a node and current root
        and increases count of the subtree nodes of root
*/

function dfs(node, root, sub)

        //   increase the count of sub[root]
        sub[root] += 1

        for v in children[node]
                //   recursively call dfs for child
                dfs(v, root, sub)
        return

/*
        a function that takes in graph G and
        returns the subtree count for each node from 1 to n
*/

function subtree(G)

        sub = array[n]
        for i from 1 to n
                /*
                        with node i as root call the dfs function to
                        compute answer for this node
                */
                dfs(i, i, sub)
        return sub
```

Time Complexity: The time complexity of the above algorithm is **$O(n^2)$.** Since we are making n dfs calls and each dfs call can visit O(n) nodes in the worst case (a bamboo tree), we get the quadratic complexity.

Space Complexity: The space complexity of the above way is **O(n)** since we are making dfs call and the maximum possible recursion depth can become O(n) in the worst case.

Dynamic Programming Approach: Let us try to improve the complexity using dynamic programming on trees. Let dp_u be the total number of nodes in subtree of node u. We observe that except u itself, each of the nodes will be included in the subtree of exactly one of its children. Also since each of the subtrees of children is disjoint, we can simply add them and then add 1 to account for node u.
So out final recurrence looks like as follows :

$$dp\ (node) = 1\ + dp(v_1) + dp(v_2) \ldots . + dpf(v_k)$$

We can solve this in a bottom-up fashion by first computing this function for children and then merging it.

The code for the above approach is as follows:-

```
/*
        recursive dfs function that takes a node and array dp
        and calculates the subtree value for this node using tree dp
*/

function dfs(node , dp)

        //   increase the count of sub[node]
        dp[node] = 1

        for v in children[node]
                //   recursively call dfs for child
                dfs(v, dp)

                //   add the dp contribution of child
                dp[node] += dp[v]

        return

/*
        a function that takes in graph G and
        returns the subtree count for each node from 1 to n
*/

function subtree(G)

        sub = array[n]
        dfs(1, sub)

        return sub
```

Time Complexity: The time complexity of the above algorithm is **O(n)**. Since we are making dfs calls in such a way that each node is visited exactly once and inside each visit a constant amount of work is done.

Space Complexity: The space complexity of the above way is **O(n)** since we are making a dfs call and the maximum possible recursion depth can become O(n) in the worst-case. Also, we are maintaining a dp array of size n so the total space will be O(n) + O(n) = O(n).

Additional Exercise: Another variation of the above problem is that each node has some value associated with it. We are interested in finding the sum of values in the subtree of each node. Can you solve this?