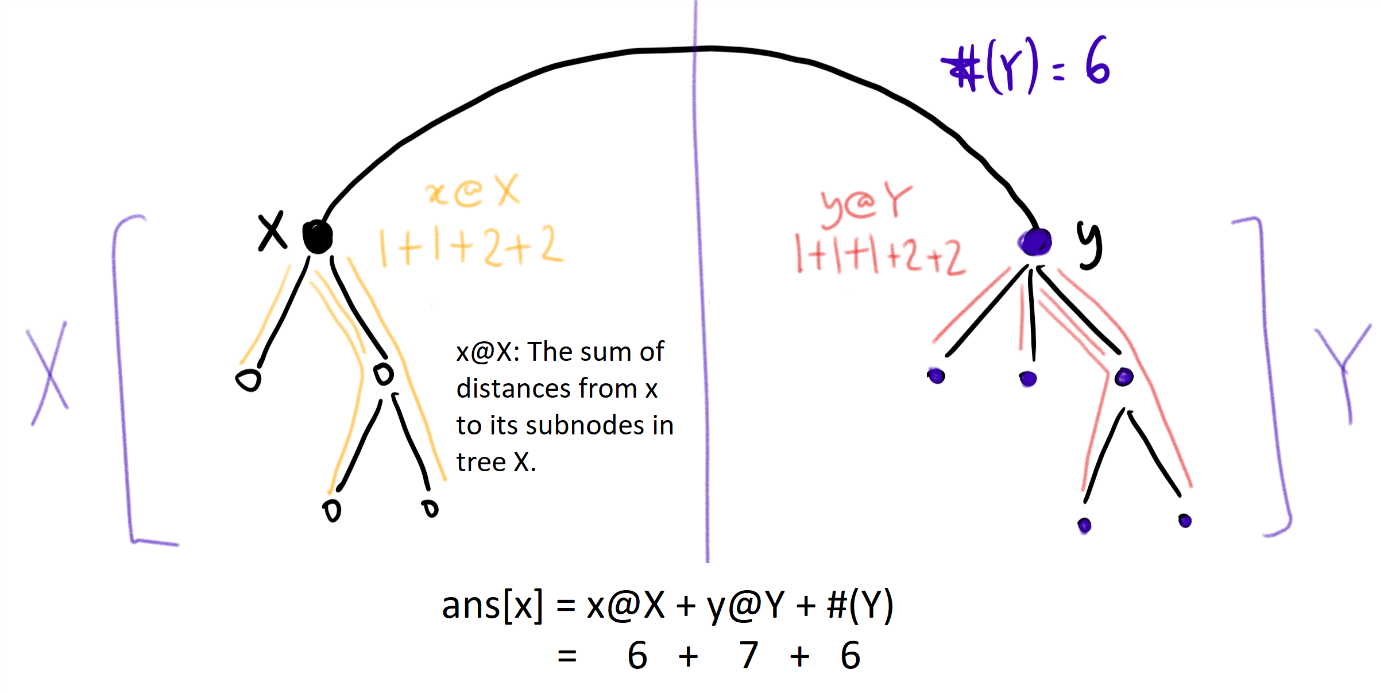
Approach #1: Subtree Sum and Count [Accepted]

**Intuition**

Let ans be the returned answer, so that in particular ans[x] be the answer for node x.

Naively, finding each ans[x] would take O(N)*O*(*N*) time (where N*N* is the number of nodes in the graph), which is too slow. This is the motivation to find out how ans[x] and ans[y] are related, so that we cut down on repeated work.

Let's investigate the answers of neighboring nodes x*x* and y*y*. In particular, say xy*xy* is an edge of the graph, that if cut would form two trees X*X* (containing x*x*) and Y*Y* (containing y*y*).



Then, as illustrated in the diagram, the answer for x*x* in the entire tree, is the answer of x*x* on X*X* "x@X", plus the answer of y*y* on Y*Y* "y@Y", plus the number of nodes in Y*Y* "#(Y)". The last part "#(Y)" is specifically because for any node z in Y, dist(x, z) = dist(y, z) + 1.

By similar reasoning, the answer for y*y* in the entire tree is ans[y] = x@X + y@Y + #(X). Hence, for neighboring nodes x*x* and y*y*, ans[x] - ans[y] = #(Y) - #(X).

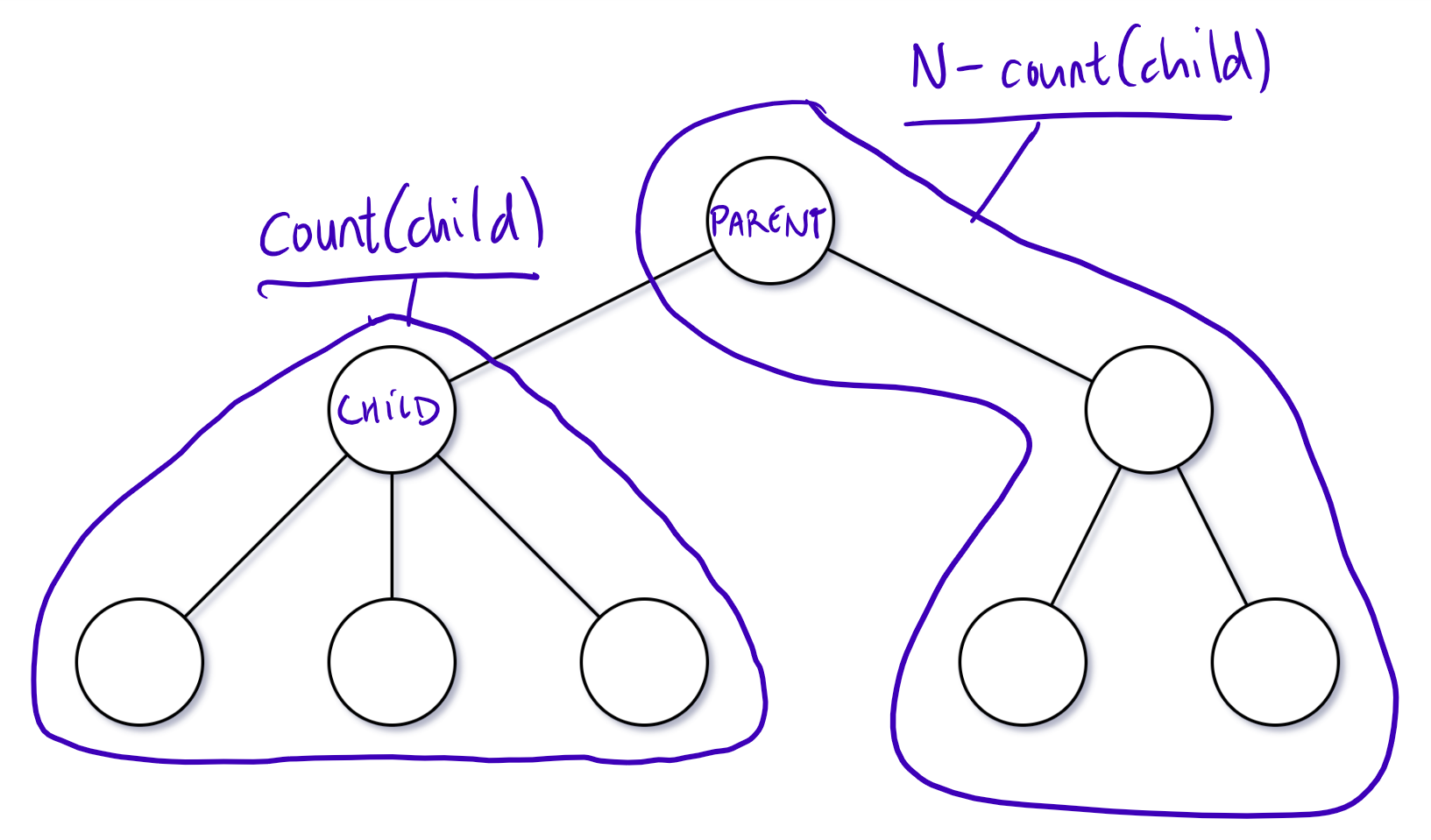
**Algorithm**

Root the tree. For each node, consider the subtree S\_{\text{node}}*S*node​ of that node plus all descendants. Let count[node] be the number of nodes in S\_{\text{node}}*S*node​, and stsum[node] ("subtree sum") be the sum of the distances from node to the nodes in S\_{\text{node}}*S*node​.

We can calculate count and stsum using a post-order traversal, where on exiting some node, the count and stsum of all descendants of this node is correct, and we now calculate count[node] += count[child] and stsum[node] += stsum[child] + count[child].

This will give us the right answer for the root: ans[root] = stsum[root].

Now, to use the insight explained previously: if we have a node parent and it's child child, then these are neighboring nodes, and so ans[child] = ans[parent] - count[child] + (N - count[child]). This is because there are count[child] nodes that are 1 easier to get to from child than parent, and N-count[child] nodes that are 1 harder to get to from child than parent.



Using a second, pre-order traversal, we can update our answer in linear time for all of our nodes.

class Solution {

    int[] ans, count;

    List<Set<Integer>> graph;

    int N;

    public int[] sumOfDistancesInTree(int N, int[][] edges) {

        this.N = N;

        graph = new ArrayList<Set<Integer>>();

        ans = new int[N];

        count = new int[N];

        Arrays.fill(count, 1);

        for (int i = 0; i < N; ++i)

            graph.add(new HashSet<Integer>());

        for (int[] edge: edges) {

            graph.get(edge[0]).add(edge[1]);

            graph.get(edge[1]).add(edge[0]);

        }

        dfs(0, -1);

        dfs2(0, -1);

        return ans;

    }

    public void dfs(int node, int parent) {

        for (int child: graph.get(node))

            if (child != parent) {

                dfs(child, node);

                count[node] += count[child];

                ans[node] += ans[child] + count[child];

            }

    }

    public void dfs2(int node, int parent) {

        for (int child: graph.get(node))

            if (child != parent) {

                ans[child] = ans[node] - count[child] + N - count[child];

                dfs2(child, node);

            }

    }

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the graph.
* Space Complexity: O(N)*O*(*N*).