Problem Description: Counting Independent Sets in a Tree

In this algorithm project, our primary objective is to develop an efficient algorithm that works in linear time to count independent sets within a tree structure. Independent sets, in the context of graph theory, refer to subsets of vertices in which no two vertices are adjacent. In our case, we focus on trees, which are a special type of graph with no cycles.

Solution Description:

We utilized the depth fisrt search method. Traverse the nodes from the leaves to the root(bottom-up). For each node, consider a subtree, whose root is that node, to get the IS of that subtree, denoted as S, we divide it two subsets U and C:

1. IS without picking that node. denote as U (unchosen), the count number of it, it's the cardinality of the set, denoted as $|U|$, the recursive procedure to get it is combinationsWithoutCurrent,

2. IS with the node, denote as C (chosen), the count of it is $|C|$, the procedure is combinationWithCurrent .

3. the all IS, denoted as S , count number of all IS is $|S|$, the procedure is getAllSetsCount. Intuitively, the count number of the IS has formula as below:
$$|S| = |C| + |U|$$

We will calculate above three states for each node recursively, until we reach the root, the number of all IS count of the root, $|S_{root}|$, is exactly the output expected. Pseudo code in python is as below:

```
def combinationsWithoutCurrent(current):
    num = 1
    for child in current:
        num *= child.cachedWithoutValue + child.
           cachedWithValue
    return num


def combinationsWithCurrent(current):
    num = 1
    for child in current:
        num *= child.cachedWithoutValue
    return num


def getAllSetsCount(arrayOfNodes):
    for node in arrayOfNodes:
        with_current = combinationsWithCurrent(node)
        without_current = combinationsWithoutCurrent(
           node)
        node.cachedWithValue = with_current
        node.cachedWithoutValue = without_current
    # return root node calculated value [-1] is the
       last element which is root in this case
    return arrayOfNodes[-1].cachedWithValue +
       arrayOfNodes[-1].cachedWithoutValue
```

The tree and its node definitions are as below:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.cachedWithValue = 0
        self.cachedWithoutValue = 0
        self.children = []

class Tree:
    def __init__(self, root_data):
        self.root = TreeNode(root_data)
```

Tree is nothing but a root tree node where we can end and all IS count of the tree is exactly S_{root}.

Proof of correctness and time complexity analysis

We traverse the tree node exactly once, we denote the number of all nodes in the tree as N. So the time complexity of the algorithm with Big O notation is $O(N)$

Proof is as in the below, we define some notations: recall three states from the above, we denote sets U_L, C_L, S_L, as each set for the node L, we consider the bottom up recursion, using the induction method:

- Base case: Consider the case, a leaf node, denoted as L, a leaf is a node who has no children, the IS of L without picking it, is $\emptyset$, $|U_L = 1|$ the IS of L including it, is the node itself$L$, $C_L = 1$, all IS set count is: $|S_L| = |U_L| + |C_L| = 2$
- Inductive Hypothesis: Because we traverse from the bottom to the top, let's move one level up, consider the subtree starting from the parent of the node, let's denote the parent node as P. To calculate all IS Count of the subtree roots from P, let's divide into two cases:

  1. IS without P
  2. IS with P.

  In case of 1, think about each child node, denoted as $D_{1\ldots n}$, meaning Nth child node of the parent node P, from 1st to Nth, let's consider all Independent sets of each substree roots from the child node,

  $$S_D = U_D \cup C_D \qquad (*)$$

  Because there is no P in the any set of S_P, recall the definition of independent set, we have no limit in combine all the sets inside each child IS set. We consider Cartesion product of each S_{D_n}, we denote a_{D_n} as any element in S_{D_n}:

  $$U_P = S_{D_1} \times S_{D_2} \times \ldots \times S_{D_n} = \{(a_{D_1}, a_{D_2}, \ldots, a_{D_n}) \mid a_{D_1} \in S_{D_1}, a_{D_2} \in S_{D_2}, \ldots, a_{D_n} \in S_{D_n}\}$$

  Here, the each pair does not need be ordered.

  The cardinality of the Cartesian product $S_{D_1} \times S_{D_2} \times \ldots \times S_{D_n}$ is the product of the cardinalities of the individual sets:

  $$|U_P| = |S_{D_1} \times S_{D_2} \times \ldots \times S_{D_n}| = |S_{D_1}| \cdot |S_{D_2}| \cdot \ldots \cdot |S_{D_n}|$$

  In case of 2, due to the fact the parent node is included in each set, similarily, the result will be the combinations of IS of each child without the child node inside (set of U), hence:

  $$C_P = U_{D_1} \times U_{D_2} \times \ldots \times U_{D_n} = \{(a_{D_1}, a_{D_2}, \ldots, a_{D_n}) \mid a_{D_1} \in U_{D_1}, a_{D_2} \in U_{D_2}, \ldots, a_{D_n} \in U_{D_n}\}$$

  $$|C_P| = |U_{D_1} \times U_{D_2} \times \ldots \times U_{D_n}| = |U_{D_1}| \cdot |U_{D_2}| \cdot \ldots \cdot |U_{D_n}|$$

recall equation $*$, in the case one, to get S_D, we already get U_D first, due to the cache, we don't need compute it again.

hence, the case two does not add extra computaltional complexity.

Above two cases are exactly what described in the pseudo code.

The final step:

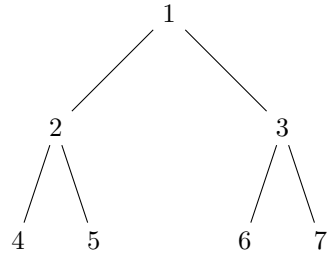
$$S_P = U_P \cup C_P$$

$$|S_P| = |U_P| + |C_P|$$

- Inductive Step: We level up until reach the root. The stable set of the root is the result expected.

Input and output

Input: a text file, each line represents the tree in a string format, for example: 1 [2 [4, 5], 3 [6, 7]], representing a tree as below:



The expected output is 41, a number represents the count of the independent sets in the tree.

Input file will be input.txt under the current directory while output file will be output.txt.

Each line of the input is a tree, the corresponding line in the output file will be IS count.