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 with two different process, both of linear time complexity. For each node, to count the IS in the subtree which roots from that node, we consider three states:

1. Count the number of ISs without picking the node. we name it combinationWithoutCurrent, denote as $\boldsymbol{U}$
2. Count the number of ISs with the node, we name it combinationWithCurrent denote as $\boldsymbol{C}$.
3. the whole IS count rooting from the node, we name it stableSet, denoted as $\boldsymbol{S}$, which is intuitively calculated as $\boldsymbol{S} = \boldsymbol{C} + \boldsymbol{U}$

For each node, to calculate $\boldsymbol{C}$ and $\boldsymbol{U}$ we will need do combinations of its children states correspondingly. Pseudo code in python is as below (Note: to avoid redundant search, we utilized the cache):

```
def combinationsWithoutCurrent(current):
    num = 1
    for child in current:
        num *= child.cachedWithoutValue + child.
            cachedWithValue
    print("calculated for " + str(current.data))
    return num


def combinationsWithCurrent(current):
    num = 1
    for child in current:
        num *= child.cachedWithoutValue
    print("calculated for " + str(current.data))
    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
```

For the recursion and sum up, We can consider two methods:

1. Top down from the root
2. Bottom up from leaves

Proof of correctness and time complexity analysis

We traverse the tree node exactly once, with cache, for topdown, we can avoid redundant search. hence, $O(N)$

Proof, here we only consider the bottom up recursion, using the induction method:

- Base case: leaf node N, who has no children, according to the algorithm $U = 1, C = 1, S = 2$, which is correct, IS set count is: $\emptyset + N = 2$
- Inductive Hypothesis: The parent of the node N, denoted P. To calculate IS Count roots from P, let's divide into two cases:

  1. IS without P
  2. IS with P.

  In case of 1, the result is combinations of IS of each child. $\prod_S(C)$. We denote $S(C)$ as the stable set of the node C, the below follows this convention.

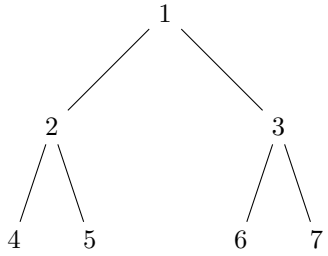
  In case of 2, the result will be the combinations of IS of each child without the child node inside, $\prod_U(C)$. However when we were calculating the $S(C)$, recall, $S = C + U$, we got and memorized the result of U, so this step does not cost any computational complexity.

  Above two cases are exactly what described in the pseudo code. The stable set of P is sum of two cases

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

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.