Good leaf node pairs:

Given the root of a binary tree and an integer distance. A pair of two different **leaf** nodes of a binary tree is said to be good if the length of **the shortest path** between them is less than or equal to distance. Return the number of good leaf node pairs in the tree.

Overview:

The problem states that we have to return a count of the pairs of leaf nodes such that the shortest path between them is less than or equal to the given distance. The function signature is public int **countPairs(TreeNode root, int distance)**. The input is a reference variable root of type **TreeNode** and distance of type **int.**

Let us now carefully look at the constraints given in the problem.

1) The number of nodes in the tree is in the range of [1, 2^10] – This means with a full binary tree, maximum number of nodes is 2^10. This implies there are a maximum of 9 levels in this tree. This follows from the properties of full binary trees i.e., if the number of levels in a full binary tree is l, the total number of nodes is 2^(l+1)
2) Each node's value is between 1 and 100. Implying duplicates are allowed.
3)The value of distance to determine good pairs is `1 <= distance <= 10`

Whenever the word "shortest path" is used in a graph problem, it usually hints at using breadth first search(bfs). A tree being a non-cyclic graph, we can do a bfs on it and this is the intuition to the first approach.

Approach 1: Breadth First Search
The tree can be cloned with a new parent pointer in each node(using any of the depth first traversals). In the process of cloning, the leaves can be collected in a separate data structure possibly a Set. A breadth-first search (bfs) can then be initiated from each of the leaves in this Set. If another leaf is reached within the "distance" we count that as a good pair. Due to double counting, we return half of the count.

Link to code: https://github.com/supriyavidur/sangraha/blob/main/GoodLeafNodePairsBFS.java

Time Complexity:
1) Cloning the tree entails visiting each node – O(n).
2) During bfs, the maximum levels explored is equal to the "good leaf distance" and at each level, we are exploring a maximum of 3 nodes(either children or parent). We do bfs for each leaf. (In a full binary tree, the maximum leaf nodes would be 1 more than half the internal nodes $(n/2)+1 \simeq (n/2)$)
The time complexity for bfs is $O(n/2)(distance*3) \simeq O(n*distance)$
Combining 1 and 2 $O(n)+O(n*distance) \simeq O(n*distance)$

Space Complexity:
1) Cloning the tree - O(n)
2) Collecting the leaves in a separate data structure -O(n/2)

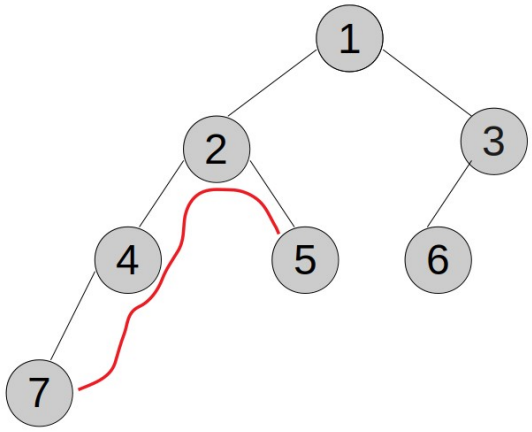
3) The level with the maximum number of nodes determines the maximum size of the queue during bfs. Assuming a full tree, the maximum size of the queue will be the number of nodes in the last level – $2^{distance}$

4) At each level of the bfs, we are exploring 3 nodes but visiting utmost 2 of them. The total number of nodes added to the visited set during bfs is $2^{(distance+1)}$. Since we can reuse the queue and visited set, we count them once in our calculation

combining 1, 2, 3 and 4, $O(n)+O(n/2)+O(2^{distance})+O(2^{distance+1})\approx O(n)$
(If distance >= depth of the tree, $2^{(distance+1)}\simeq O(n)$)

Instead of cloning the tree as above, it could also be transformed into a graph and followed by a dfs/bfs to determine the good leaf node pairs. The complexity calculations will be on similar lines.

Approach 2: Recursive Top Down



Distance between leaves 7 and 5 = Depth of 7 – Depth of 2 + Depth of 5 – Depth of 2
= 3 – 1 + 2 – 1 = 3

Another way to look at this problem is to think in terms of the least common ancestor (LCA) of the leaves under consideration. The sum of the distances to the least common ancestor is the shortest distance between the two leaves. In the top down approach, we compute the depth of each node starting from the root(top down). When we reach the leaf, we return its depth to the parent node in an array. At the parent node, we use the left and the right arrays thus returned along with the node's depth to compute the distance between the leaves and hence calculate the good node pairs.

Link to code: https://github.com/supriyavidur/sangraha/blob/main/GLNRecursiveTopDown.java

Time Complexity:
1) The dfs takes O(n)
2) We calculate the good leaf node pairs for each internal node. i.e. for each of its leaves in the left sub tree, we check the distance with each of its leaves in the right subtree. Assuming it is a full tree, we have n/2 leaves and we make half the number of comparisons i.e. n/4. At every step above the comparisons get doubled.

$n/4 + n/4 * 2 + n/4 * 2^2 + n/4 * 2^3 + ... + n/4 * 2^{depth} \approx n/4 \{1 + 2 + 2^2 ..... + 2^{depth}\}$

Applying sum of geometric progression, this would be

$n/4 \{2^{depth} - 1\} \simeq n/4 \{2^{\log_2(n)} - 1\} \simeq n/4 \{n - 1\} \simeq O(n^2)$

The depth for a full tree is $\log_2(n+1) - 1 = \log_2(^{(n+1)}/_2) \simeq O(\log_2(n))$ This is how time complexity is computed as per the code.

Another approach to arriving at the same answer would be to think of it logically. Every leaf is compared to every other leaf to see if it is at "good distance". Considering a full tree, the total number of leaves is $2^{depth} = {}^{(n+1)}/_2$ Total comparisons would be $O(n^2)$

3) The "for" loop combining the leaves to return to the next parent node takes - $^{(n+1)}/_2 * depth$

Combining 1, 2 and 3 , time complexity is $O(n^2)$

Space Complexity: At every node, we return the all the leaves below it. So at every level the total size of the array or the List would be equal to the total number of leaves in the tree. For a full tree, it will be $n+1/2 \simeq O(n)$

How do you think this can be improved? We need to check where the redundancy is coming in.

If sum of left leaf + right leaf <= "good distance" then either one can be utmost "good distance" from its parent/ancestor. We can easily figure out that instead of calculating good leaf node distance for each and every pair of leaves, we could only check it for leaves which are less than or equal to the "good distance" from their parents/ancestors. This way we can avoid unnecessary computations and significantly improve the run time and space complexities. The run time and space complexities will be determined by either the number of leaves within good distance($2^{(distance)}$)or total nodes(n) in the tree

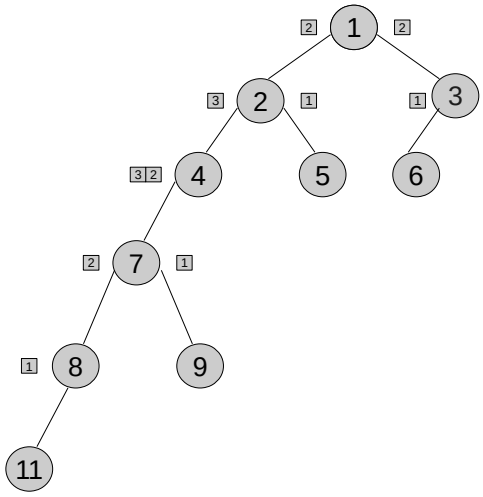
Time Complexity:

For a full tree, the number of leaves at good distance is $2 * 2^{(distance)} = 2^{(distance+1)}$ (for each of the left and right subtree). Since every leaf is compared to every other leaf, the complexity is approximately $O((2^{(distance)})^2)$ or $O(n)$ which ever is greater.

Space Complexity: $O(2^{(distance+1)})$

Approach 3: Recursive Bottom-Up

We could implement the whole of approach 2 from the Bottom- up. The run time and space complexities calculations would be similar to approach 2.

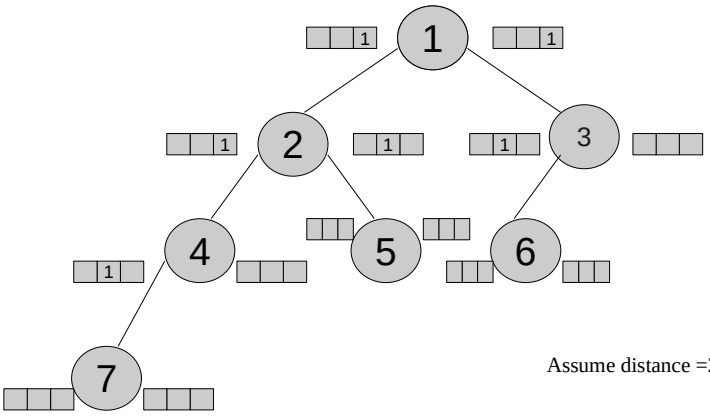


Assume distance =3

Link to code:
https://github.com/supriyavidur/sangraha/blob/main/GLNRecursiveBottomUpOptimized.java

Approach 4: Recursive Bottom Up with array size equal to distance

Can we reduce the time and space complexity even further? This will need us to look closely at the logic which is contributing to a $O\left(\left(2^{(distance)}\right)^2\right)$ and see if we can simplify this. What exactly are we doing here? We are comparing every leaf to every other leaf to see if it is at good distance. Do we really need to do this? We are not concerned about the value in each leaf rather we are interested in the "good distance" implying that individually comparing the leaves is redundant. Instead can we then "collect" all the leaves which are at a given distance from the parent/ancestor node and put it in one bucket and then use this data to obtain the good leaf pairs. For instance at every internal node in the tree, we could count all the leaves at distance 1 and store it, count all leaves at distance 2 and store it and so on until we count all the leaves at "good distance" and store it. Let us say we use an array to store this data. The index determines the distance. The maximum length of the array is "good distance + 1". At every internal node, we end up with two such arrays one for the left leaves and other for the right leaves. As we sum the indices from both the arrays and make sure they are within the "good distance", we multiply the values in the array to get the "good leaf pairs". This revises our time and space complexities as follows



Assume distance =2

Link to code: https://github.com/supriyavidur/sangraha/blob/main/GLNWithDistanceArray.java

Time complexity: $O\left(distance^2\right)$ or O(n) which ever is larger
Space complexity: O(distance+1)