

508. Most Frequent Subtree Sum



```
def findFrequentTreeSum(self, root: Optional[TreeNode]) -> List[int]:
    mapp = defaultdict(int)
    ans= []
    def dfs(root):
        nonlocal mapp
        if not root: return 0

        val = root.val+dfs(root.left)+dfs(root.right)

        mapp[val]+=1
        return val

    dfs(root)
    maxxval = max(mapp.values())
    for i,j in mapp.items():
        if maxxval == j:
            ans.append(i)
    return ans
```

543. Diameter of Binary Tree



Diameter of binary tree - Important

```
import sys
class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        def solve(root):
            nonlocal res
            if root == None:
                return 0
            left=solve(root.left)
            right=solve(root.right)

            temp=max(left,right)+1
            ans=max(temp,left+right+1)
            res=max(res,ans)
            return temp

        res=-float("inf")
        solve(root)
        return res-1
```

563. Binary Tree Tilt



```
def findTilt(self, root: Optional[TreeNode]) -> int:
    ans = 0
    if not root: return 0
    def dfs(node):
        nonlocal ans
        L = 0
        R = 0
        if node.left:
            L = dfs(node.left)
        if node.right:
            R = dfs(node.right)
        if not node.right and not node.left: return node.val
        else: ans += abs(L-R)
        return L+R+ node.val
    dfs(root)
    return ans
```

662. Maximum Width of Binary Tree



Concepts to be known: For a binary tree index of left child of a node $\rightarrow 2 * \text{parent node level} + 1$ index of right child of a node $\rightarrow 2 * \text{parent node level} + 2$ width = right_most node index - left_most node index + 1 (for a single level) Approach One queue for keeping track for all nodes One list (named: nodes) to store all nodes' indexes for a level Ans = max(Ans , current level's width)

```
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        queue = [(0,root)]
        ans = 0
        while queue:
            n =len(queue)
            nodes = []
            for _ in range(n):
                idx,node = queue.pop(0)
                nodes.append(idx)
                if node.left:
                    queue.append((2*idx+1,node.left))
                if node.right:
                    queue.append((2*idx+2,node.right))
            ans = max(ans,max(nodes)-min(nodes)+1)
        return ans
```

684. Redundant Connection



Important : UnionFind

```
class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        parent = [i for i in range(len(edges) + 1)] # Parent array to track disjoint sets
        rank = [1] * (len(edges) + 1) # Rank array for union by rank optimization

        # Find function with path compression
        def find(n):
            while n != parent[n]:
                parent[n] = parent[parent[n]] # Path compression
                n = parent[n]
            return n

        # Union function with union by rank
        def union(n1, n2):
            par1, par2 = find(n1), find(n2)
            if par1 == par2: # If they already share the same parent, this edge is redundant
                return False
            # Union by rank
            if rank[par1] > rank[par2]:
                parent[par2] = par1
                rank[par1] += rank[par2]
            else:
                parent[par1] = par2
                rank[par2] += rank[par1]
            return True

        # Process each edge
        for i, j in edges:
            if not union(i, j): # If union fails, the edge is redundant
                return [i, j]
```

863. All Nodes Distance K in Binary Tree



Very Important

```
# Definition for a binary tree node.
```

```
class Solution:
```

```
    def distanceK(self, node: TreeNode, target: TreeNode, k: int) -> List[int]:
```

```
        mapp = defaultdict(int)
```

```
        def preorder(root, parent):
```

```
            nonlocal mapp
```

```
            if not root: return
```

```
            mapp[root] = parent
```

```
            preorder(root.left, root)
```

```
            preorder(root.right, root)
```

```
        preorder(node, None)
```

```
        queue = [target]
```

```
        visited = set()
```

```
        level = 0
```

```
        while queue:
```

```
            n = len(queue)
```

```
            if level == k:
```

```
                break
```

```
            level += 1
```

```
            for _ in range(n):
```

```
                Node = queue.pop(0)
```

```
                #print(Node.val)
```

```
                visited.add(Node)
```

```
                if mapp[Node] != None and mapp[Node] not in visited:
```

```
                    queue.append(mapp[Node])
```

```
                if Node.left and Node.left not in visited:
```

```
                    queue.append(Node.left)
```

```
                if Node.right and Node.right not in visited:
```

```
                    queue.append(Node.right)
```

```
        for i in range(len(queue)):
```

```
            queue[i] = queue[i].val
```

```
        return queue
```

993. Cousins in Binary Tree



```
class Solution:
    def isCousins(self, root: Optional[TreeNode], x: int, y: int) -> bool:
        queue = [(0,root)]
        ans= []
        level = 0
        while queue:
            level+=1
            n = len(queue)
            for _ in range(n):
                idx,node = queue.pop(0)
                if node.val in [x,y]:
                    ans.append([idx,level])
                    if len(ans)==2:
                        break
                if node.left:
                    queue.append([2*idx+1,node.left])
                if node.right:
                    queue.append([2*idx+2,node.right])
            if ans[0][1] == ans[1][1]:
                idx1 =(ans[0][0]-1)//2
                idx2 =(ans[1][0]-1)//2
                return True if idx2 != idx1 else False
        return False
```

998. Maximum Binary Tree II



Maximum binary tree

```
class Solution:
    def insertIntoMaxTree(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        def dfs(root):
            temp = [root.val]
            if root.left:
                temp = dfs(root.left) + temp
            if root.right:
                temp += dfs(root.right)
            return temp
        newarr = dfs(root)
        newarr.append(val)
        def construct(nums):
            if not nums: return None
            root = max(nums)
            root_ind = nums.index(root)
            node = TreeNode(root)
            node.left = construct(nums[:root_ind])
            node.right = construct(nums[root_ind+1:])
            return node
        return construct(newarr)
```

1373. Maximum Sum BST in Binary Tree



#Important Hard

```

def maxSumBST(self, root):
    max_sum = 0
    def kunction(node):
        nonlocal max_sum
        if node is None:
            return 0, float("inf"), float("-inf")

        l_sum, l_min, l_max = kunction(node.left)
        r_sum, r_min, r_max = kunction(node.right)

        if l_max < node.val < r_min:
            max_sum = max(max_sum, node.val + l_sum + r_sum)
            return node.val + l_sum + r_sum, min(l_min, node.val), max(r_max, node.val)

        return 0, float("-inf"), float("inf")

    kunction(root)

    return max_sum

```

1376. Time Needed to Inform All Employees



```

class Solution:
    def numOfMinutes(self, n: int, headID: int, manager: List[int], informTime: List[int]) -> int:
        ans = 0
        mapp = defaultdict(list)
        for i, j in enumerate(manager):
            mapp[j].append(i)
        def dfs(node):
            if node not in mapp:
                return 0
            max_time = 0
            for i in mapp[node]:
                max_time = max(max_time, informTime[node] + dfs(i))
            return max_time
        return dfs(headID)

```

1443. Minimum Time to Collect All Apples in a Tree


```
def minTime(self, n: int, edges: List[List[int]], hasApple: List[bool]) -> int:
    mapp = {i:[] for i in range(n)}
    for k,v in edges:
        mapp[k].append(v)
        mapp[v].append(k)
    ans = 0
    visited = set()
    def dfs(node):
        nonlocal visited,mapp,ans
        have_child = False
        visited.add(node)
        for i in mapp[node]:
            if i not in visited:
                have_child = dfs(i) or have_child
        if hasApple[node] or have_child:
            ans += 2
        return hasApple[node] or have_child
    dfs(0)
    return ans if ans != 0 else ans-2
```

2421. Number of Good Paths

UnionFind

```
class UnionFind:
    def __init__(self,n):
        self.par = list(range(n))
        self.rank = [0]*n

    def find(self,n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self, a, b):
        aRoot = self.find(a)
        bRoot = self.find(b)
        if aRoot == bRoot:
            return False
        elif self.rank[aRoot] < self.rank[bRoot]:
            self.rank[bRoot] += self.rank[aRoot]
            self.par[aRoot] = bRoot
        else:
            self.rank[aRoot] += self.rank[bRoot]
            self.par[bRoot] = aRoot
        return True

class Solution:
    def numberOfGoodPaths(self, vals: List[int], edges: List[List[int]]) -> int:
        adj = defaultdict(list)
        for a,b in edges:
            adj[a].append(b)
            adj[b].append(a)

        valToIndex = defaultdict(list)
        for ind,val in enumerate(vals):
            valToIndex[val].append(ind)

        res = 0
        uf = UnionFind(len(vals))
        for val in sorted(valToIndex.keys()):
            for node in valToIndex[val]:
                for nei in adj[node]:
                    if vals[nei] <= vals[node]:
                        uf.union(nei,node)

        count = defaultdict(int)
        for i in valToIndex[val]:
            root = uf.find(i)
```

```

        count[root] += 1
        res += count[root]
    return res

```

2471. Minimum Number of Operations to Sort a Binary Tree by Level

```

def minimumOperations(self, root: Optional[TreeNode]) -> int:
    ans = 0
    queue = [root]
    def swap(arr:list,n:int):
        nonlocal ans
        correct = sorted(arr)
        hashmap = {vall:ind for ind,vall in enumerate(arr)}
        temp = 0
        for i in range(n):
            if arr[i] != correct[i]:
                ans+=1
                temp = arr[i]
                arr[i],arr[hashmap[arr[i]]] = correct[i],arr[i]
                hashmap[temp] = hashmap[correct[i]]
                hashmap[correct[i]] = i
    while queue:
        n = len(queue)
        for _ in range(n):
            node = queue.pop(0)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        arr = [i.val for i in queue]
        swap(arr,len(arr))

    return ans

```

2476. Closest Nodes Queries in a Binary Search Tree

[1, 4, 14, 15, 16] [1, 4, 14, 15, 16] 2 1 4 1 6 2 14 2 9 2 14 2 10 2 14 2

```
[[14,0],[14,0],[14,0],[14,0]]
```

2641. Cousins in Binary Tree II



```
class Solution:
    def replaceValueInTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        queue = [[0,root]]
        hashmap = defaultdict(int)
        while queue:
            n = len(queue)
            total = 0
            for _ in range(n):
                idx,node = queue.pop(0)
                if node.left:
                    queue.append([idx*2+1,node.left])
                    hashmap[idx]+=node.left.val
                    total+=node.left.val
                if node.right:
                    queue.append([idx*2+2,node.right])
                    hashmap[idx]+=node.right.val
                    total+=node.right.val
            for i in range(len(queue)):
                if len(hashmap)<=1:
                    queue[i][1].val = 0
                    continue
                idx = (queue[i][0]-1)//2
                queue[i][1].val = total - hashmap[idx]
            hashmap.clear()
        root.val = 0
        return root
```

3067. Count Pairs of Connectable Servers in a Weighted Tree Network



```

def dfs(self,node,visited,distance,signalspeed,graph):
    num = 0
    if distance%signalspeed ==0:num += 1
    visited.add(node)
    for child,weight in graph[node]:
        if child not in visited:
            child_nums = self.dfs(child,visited,distance+weight,signalspeed,graph)
            num += child_nums
    visited.remove(node)
    return num

def countPairsOfConnectableServers(self, edges: List[List[int]], signalSpeed: int)
-> List[int]:
    n = len(edges)+1
    graph = defaultdict(list)
    for a,b,w in edges:
        graph[a].append([b,w])
        graph[b].append([a,w])
    ans = [0]*n
    for node in range(n):
        num = []
        for child,weight in graph[node]:
            child_nums = self.dfs(child,set([node]),weight,signalSpeed,graph)
            num.append(child_nums)
        s = sum(num)
        for i in range(len(num)):
            ans[node] += (s-num[i]) * num[i]
        ans[node] = ans[node]//2
    return ans

```

3319. K-th Largest Perfect Subtree Size in Binary Tree



```

def kthLargestPerfectSubtree(self, root: Optional[TreeNode], k: int) -> int:
    if not root:
        return -1
    ans = []
    def dfs(root):
        nonlocal ans
        l = 0
        r = 0
        if root.left:
            l = dfs(root.left)
        if root.right:
            r = dfs(root.right)
        if not root.left and not root.right:
            ans.append(1)
            return 1
        elif l == r and l > 0:
            ans.append(l+r+1)
            return l+r+1
        return 0
    dfs(root)
    ans.sort(reverse = True)
    return ans[k-1] if len(ans) > k-1 else -1

```

3331. Find Subtree Sizes After Changes



```

class Solution:
    def findSubtreeSizes(self, p: List[int], s: str) -> List[int]:
        n, clds = len(s), defaultdict(set)
        for i in range(n):
            clds[p[i]].add(i)
        res = [0] * n
        def dfs(i, anc={}):
            prv, anc[s[i]] = anc.get(s[i]), i
            res[i] = 1
            for c in clds[i]:
                dfs(c, anc)
                res[p[c]] += res[c]
            anc[s[i]] = prv
            if prv is not None:
                p[i] = prv
        dfs(0)
        return res

```

3372. Maximize the Number of Target Nodes After Connecting Trees I




```
class Solution:
    def makegraph(self,mapp,edges):
        for i,j in edges:
            mapp[i].append(j)
            mapp[j].append(i)
        return mapp
    def do_bfs_traversal(self,mapp,k):
        reachCounts = [0]*len(mapp)
        for startnode in range(len(mapp)):
            queue = [startnode]
            visited = set()
            visited.add(startnode)
            level = 0
            while queue and level < k :
                level+=1
                n = len(queue)
                for _ in range(n):
                    node = queue.pop(0)
                    for j in mapp[node]:
                        if j not in visited:
                            visited.add(j)
                            queue.append(j)
                reachCounts[startnode] += len(visited)
        return reachCounts

    def maxTargetNodes(self, edges1: List[List[int]], edges2: List[List[int]], k: int) -> List[int]:
        if k == 0:
            return [1] * (len(edges1) + 1)
        tree1 = self.makegraph(defaultdict(list),edges1)
        tree2 = self.makegraph(defaultdict(list),edges2)
        reachableCount1 = self.do_bfs_traversal(tree1,k)
        reachableCount2 = self.do_bfs_traversal(tree2,k-1)
        maxReachableInTree2 = max(reachableCount2)
        for i in range(len(reachableCount1)):
            reachableCount1[i] += maxReachableInTree2
        return reachableCount1

    ....
```

3373. Maximize the Number of Target Nodes After Connecting Trees II



graph colouring very important


```
class Solution:
    def makegraph(self,mapp,arr):
        for i,j in arr:mapp[i].append(j);mapp[j].append(i)
        return mapp
    def color(self,tree):
        queue = [0]
        visited = {0:"b"}
        level = True
        black,white = 1,0
        while queue:
            level = not level
            n = len(queue)
            for _ in range(n):
                node = queue.pop(0)
                for i in tree[node]:
                    if i not in visited:
                        queue.append(i)
                        if level :
                            black+=1
                            visited[i] = "b"
                        else:
                            white+=1
                            visited[i] = "w"
            return black,white,visited

    def maxTargetNodes(self, edges1: List[List[int]], edges2: List[List[int]]) -> List[int]:
        tree1 = self.makegraph(defaultdict(list),edges1)
        tree2 = self.makegraph(defaultdict(list),edges2)
        black1,white1,vis1 = self.color(tree1)
        black2,white2,vis2 = self.color(tree2)
        maxcol = max(black2,white2)
        ans = []
        for i in range(len(tree1)):
            col = maxcol
            if vis1[i] == "b":col+= black1
            else:col+= white1
            ans.append(col)
        return ans
```