**TREE QUESTIONS**

Table of Contents

[LEVEL 1: **Armature** 1](#_Toc198205275)

[1. Best Time to buy and sell stock 1](#_Toc198205276)

[LEVEL 2: **Pro** 2](#_Toc198205277)

[LEVEL 3: **Legend** 3](#_Toc198205278)

[**SOLUTIONS:** 4](#_Toc198205279)

[**LEVEL 1:** 4](#_Toc198205280)

[**LEVEL 2:** 11](#_Toc198205281)

The sliding window technique is ideal for problems involving contiguous sequences, such as substrings or subarrays.

### 1. **Fixed-Size Window**

These problems involve a window of a predetermined size moving across the data structure.

* [Maximum Average Subarray I](https://leetcode.com/problems/maximum-average-subarray-i/)
* [Sliding Window Maximum](https://leetcode.com/problems/sliding-window-maximum/)
* [Number of Subarrays of Size K and Average ≥ Threshold](https://leetcode.com/problems/number-of-sub-arrays-of-size-k-and-average-greater-than-or-equal-to-threshold/)
* [Longest Subarray of 1's After Deleting One Element](https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/)[LeetCode](https://leetcode.com/problems/sliding-window-maximum/?utm_source=chatgpt.com)

### 2. **Variable-Size Window**

These problems require dynamically adjusting the window size based on certain conditions.

* [Minimum Size Subarray Sum](https://leetcode.com/problems/minimum-size-subarray-sum/)
* [Longest Repeating Character Replacement](https://leetcode.com/problems/longest-repeating-character-replacement/)
* [Permutation in String](https://leetcode.com/problems/permutation-in-string/)
* [Longest Substring with At Most K Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/)

### 3. **Substring/Anagram Matching**

These problems focus on finding substrings or anagrams within a larger string.

* [Minimum Window Substring](https://leetcode.com/problems/minimum-window-substring/)
* [Find All Anagrams in a String](https://leetcode.com/problems/find-all-anagrams-in-a-string/)
* [Substring with Concatenation of All Words](https://leetcode.com/problems/substring-with-concatenation-of-all-words/)
* [Check Inclusions](https://leetcode.com/problems/permutation-in-string/)
* [Longest Substring with At Most Two Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/)[LeetCode](https://leetcode.com/problems/minimum-window-substring/?utm_source=chatgpt.com)

### 4. **Advanced Sliding Window**

These problems require more complex data structures or logic within the sliding window.

* [Sliding Window Median](https://leetcode.com/problems/sliding-window-median/)
* [Count Number of Nice Subarrays](https://leetcode.com/problems/count-number-of-nice-subarrays/)
* [Max Sum of Rectangle No Larger Than K](https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/)
* [Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit](https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/)
* [Subarrays with K Different Integers](https://leetcode.com/problems/subarrays-with-k-different-integers/)[LeetCode](https://leetcode.com/problems/sliding-window-median/description/?envId=9rt1jt27&envType=list&utm_source=chatgpt.com)

# LEVEL 1: **Fixed-Size Window**

### \_\_\_\_\_\_\_\_

Link:<https://leetcode.com/problems/max-consecutive-ones-iii/description/>

# LEVEL 2: **Variable-Size Window**

### Max Consecutive Ones III

Link:<https://leetcode.com/problems/max-consecutive-ones-iii/description/>

class Solution:

    def longestOnes(self, nums: List[int], k: int) -> int:

        left=0

        count\_zero=0

        max\_len=0

        n=len(nums)

        for right in range(n):

            if nums[right] == 0:

                count\_zero+=1

            while count\_zero > k:

                if nums[left] == 0:

                    count\_zero-=1

                left+=1

            max\_len=max(max\_len,right-left+1)

        return max\_len

### Longest substring without repeating characters

Link:<https://leetcode.com/problems/longest-substring-without-repeating-characters/description/>

from collections import defaultdict

class Solution:

    def lengthOfLongestSubstring(self, s: str) -> int:

        l,r = 0,0

        n = len(s)

        mlen = 0

        #will keep index+1, so that 0 means char not present

        #also will help to move pointer to next index

        indexes = defaultdict(int)

        while(r<n):

            if(indexes[s[r]]==0 or indexes[s[r]]<l):

                mlen = max(mlen, r-l+1)

            else:

                l = indexes[s[r]]

            indexes[s[r]] = r+1

            r+=1

        return mlen

# LEVEL 3: **Legend**

# **SOLUTIONS:**

## **LEVEL 1:**

1. Sum of binary tree

*#Approach 1: using seperate inorder traversal method and using global variable*

class Solution:

    def sumBT(*self*, *root*):

*#run inorder traversal and get sum of all*

        def inorder(*root*):

            global ans

*if* *root* is not None:

                inorder(*root*.left)

                ans+=*root*.data

                inorder(*root*.right)

        global ans

        ans=0

        inorder(*root*)

*return* ans

*#Approach 2: using just plain recursion without global keyword*

class Solution:

    def sumBT(*self*, *root*):

*if* *root* is None:

*return* 0

        l=*self*.sumBT(*root*.left)

        r=*self*.sumBT(*root*.right)

*return* *root*.data + l + r

*#Appproach 3(shorter)*

class Solution:

    def sumBT(*self*, *root*):

*return* *root*.data+*self*.sumBT(*root*.left)+*self*.sumBT(*root*.right) *if* *root* *else* 0

1. Search a node in BST

class BST:

    def search(*self*, *node*, *x*):

*if* *node* is None:

*return* False

*if* *node*.data==*x*:

*return* True

*if* *node*.data>*x*:

*return* *self*.search(*node*.left,*x*)

*return* *self*.search(*node*.right,*x*)

1. Count Leaves in binary tree

class Solution:

    def countLeaves(*self*, *root*):

*if* *root* is None:

*return* 0

*#Means it is leave, so no need to iterate further subtree of it*

*if* *root*.left is None and *root*.right is None:

*return* 1

*return* *self*.countLeaves(*root*.left) + *self*.countLeaves(*root*.right)

*#Approach 2: One liner for above code*

class Solution:

    def countLeaves(*self*, *root*):

*if* *root* is None:

*return* 0

*return* 1 *if* not *root*.left and not *root*.right *else* *self*.countLeaves(*root*.left)+*self*.countLeaves(*root*.right)

1. N-Ary tree postorder traversal

class Solution:

    def postorder(*self*, *root*: 'Node') -> List[int]:

*#postorder:left right root*

        def helper(*node*):

*if* *node*:

*for* temp *in* *node*.children:

                    helper(temp)

                ans.append(*node*.val)

        global ans

        ans=[]

        helper(*root*)

*return* ans

1. Maximum depth of binary tree

class Solution:

    def maxDepth(*self*,*root*):

*if* *root* is None:

*return* 0

*if* not *root*.left and not *root*.right:

*return* 1

        l = *self*.maxDepth(*root*.left)

        r = *self*.maxDepth(*root*.right)

*return* 1+max(l,r)

1. Height of binary tree

class Solution:

    def height(*self*, *root*):

*# code here*

*if* *root* is None:

*return* 0

*#this is end condition but since if we return 1 from here, it will be count of nodes*

*#to get edge count (edge count =  node count-1) , so return 0*

*if* not *root*.left and not *root*.right:

*return* 0

        l = *self*.height(*root*.left)

        r = *self*.height(*root*.right)

*return* 1+max(l,r)

1. Convert BST to inorder tree with only right node

*# Approach : first get the inorder traversal*

*#create a new tree and add all values to right subtree*

class Solution:

    def increasingBST(*self*, *root*: TreeNode) -> TreeNode:

        def inorder(*root*):

*return* inorder(*root*.left)+[*root*.val]+inorder(*root*.right) *if* *root* *else* []

inorder\_trav = inorder(*root*)

        temp=TreeNode()

        ans=temp

*for* val *in* inorder\_trav:

            temp.right = TreeNode(val)

            temp = temp.right

*return* ans.right

1. Same tree

class Solution:

    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:

        if p is None and q is None:

            return True

        if (p is None) or (q is None):

            return False

        if p.val!=q.val:

            return False

        return self.isSameTree(p.left,q.left) and self.isSameTree(p.right,q.right)

Approach 2

class Solution:

    def isSameTree(*self*, *p*: Optional[TreeNode], *q*: Optional[TreeNode]) -> bool:

*#while q* *for i in range q, popleft current one, popleft from other one, if they arent #equal, break* *add left if left, add right. same for other one*

        q1 = deque([*p*])

        q2 = deque([*q*])

*while* q1:

*if* len(q1) != len(q2):

*return* False

*for* i *in* range(len(q1)):

                curr1 = q1.popleft()

                curr2 = q2.popleft()

*if* not curr1 and not curr2:

*continue*

*if* not curr1 or not curr2:

*return* False

*if* curr1.val != curr2.val:

*return* False

                q1.append(curr1.left)

                q1.append(curr1.right)

                q2.append(curr2.left)

                q2.append(curr2.right)

*return* True

1. Range sum of bst

[Solution](https://leetcode.com/problems/range-sum-of-bst/solutions/4558641/python-fastest-optimized-with-explanation/)

class Solution:

    def rangeSumBST(self, root: Optional[TreeNode], low: int, high: int) -> int:

        def travel(root):

            global ans

            if (root is None) :

                return

            else:

                if root.val>=low and root.val<=high:

                    ans+=root.val

                if root.val >low:

                    travel(root.left)

                if root.val <high:

                    travel(root.right)

        global ans

        ans=0

        travel(root)

        return ans

Approach 2: without global value

class Solution:

    def rangeSumBST(*self*, *root*: Optional[TreeNode], *low*: int, *high*: int) -> int:

*if* *root* is None:

*return* 0

*if* *root*.val<*low*:

*return* *self*.rangeSumBST(*root*.right,*low*,*high*)

*if* *root*.val>*high*:

*return* *self*.rangeSumBST(*root*.left,*low*,*high*)

*return* *root*.val+*self*.rangeSumBST(*root*.left,*low*,*high*)+*self*.rangeSumBST(*root*.right,*low*,*high*)

1. Convert BST to inorder tree with only right node

*# Approach : first get the inorder traversal*, *create a new tree and add all values to right subtree*

class Solution:

    def increasingBST(*self*, *root*: TreeNode) -> TreeNode:

        def inorder(*root*):

*return* inorder(*root*.left)+[*root*.val]+inorder(*root*.right) *if* *root* *else* []

inorder\_trav = inorder(*root*)

        temp=TreeNode()

        ans=temp

*for* val *in* inorder\_trav:

            temp.right = TreeNode(val)

            temp = temp.right

*return* ans.right

1. Average of levels in binary tree

*#Approach 1: using breadth level traversal*

from collections import deque

class Solution:

    def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:

        ans=[]

        queue = deque()

        queue.append(root)

        while(queue):

            temp = 0

            l = len(queue)

            for i in range(l):

                t = queue.popleft()

                temp+=t.val

                if t.left:

                    queue.append(t.left)

                if t.right:

                    queue.append(t.right)

            ans.append(temp/l)

        return ans

*#Approach: iterate normally but maintain level*

*#res[level]= [sum\_of\_all\_values\_in\_level , count\_of\_elemets\_in\_level]*

class Solution:

    def averageOfLevels(*self*, *root*: Optional[TreeNode]) -> List[float]:

        res=[]                 *#[[level\_sum,count\_of\_elements]]*

        def dfs(*node*, *lvl*=0):

*if* not *node*:

*return*

*if* len(res)<*lvl*+1: *#means for level+1, there is no res, so add*

                res.append([0,0])

            dfs(*node*.left, *lvl*+1)

            res[*lvl*][0]+=*node*.val

            res[*lvl*][1]+=1

            dfs(*node*.right, *lvl*+1)

        dfs(*root*)

*return*[i/k *for* i,k *in* res]

1. Sum of left leaves

class Solution:

    def sumOfLeftLeaves(*self*, *root*: Optional[TreeNode]) -> int:

        def leftSum(*node*,*isLeft*):

*if* *node*:

*if* *isLeft* and (not *node*.left and not *node*.right):

*self*.ans+=*node*.val

                leftSum(*node*.left,True)

                leftSum(*node*.right,False)

*#in place of global can use self also*

*self*.ans=0

        leftSum(*root*,False)   *#if single node is there, it is not consider left leaf*

*return* *self*.ans

## **LEVEL 2:**

1. Merge two binary tree

class Solution:

    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:

        if root1 is None:

            return root2

        if root2 is None:

            return  root1

        root1.val = root1.val + root2.val

        root1.left = *self*.mergeTrees(root1.left,root2.left)

        root1.right = *self*.mergeTrees(root1.right,root2.right)

        return root1

1. Sorted Array to binary Search tree

[Solution](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/solutions/4565640/python-code-fastest-solution-explanation/)

[](https://www.youtube.com/watch?v=0K0uCMYq5ng)

class Solution:

    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

        def helper(l,r):

            if l>r:

                return

            else:

                m = (l+r)//2

                root = TreeNode(nums[m])

                root.left = helper(l,m-1)

                root.right = helper(m+1,r)

                return root

        return helper(0,len(nums)-1)

1. Binary tree tilt

[Solution](https://leetcode.com/problems/binary-tree-tilt/solutions/4578246/python-2-solutions-easy-one-optimized-version-with-explanation/)

class Solution:

    def findTilt(self, root: TreeNode) -> int:

        def helper(root):

            if not root: return 0

            lv, rv = helper(root.left), helper(root.right)

*self*.ans += abs(lv - rv)

            return root.val + lv + rv   *#This gives sum of given subtree*

*self*.ans = 0

        helper(root)

        return *self*.ans

1. Diameter Of Binary tree

[Solution](https://leetcode.com/problems/diameter-of-binary-tree/solutions/1143907/python-thought-process/)

[](https://www.youtube.com/watch?v=bkxqA8Rfv04)

def diameterOfBinaryTree(self, root):

    def recurse(node):

        if not node: return 0

        left, right = recurse(node.left), recurse(node.right)

*self*.result = max(*self*.result, left+right)

        return 1 + max(left, right)

*self*.result = 0

    recurse(root)

    return *self*.result

Similar way try to code for finding height of binary tree.

1. Subtree of another tree

class Solution:

    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:

        if subRoot == None :

            return True

        if root == None :

            return False

        if *self*.same(root , subRoot):

            return True

        return *self*.isSubtree(root.left , subRoot) or *self*.isSubtree(root.right , subRoot)

*#This is code to check if one tree is equal to other*

*#check root vals and left and right subtree vals, can't do r==s(this don't work)*

    def same(self , r , s):

        if r == None and s == None :

            return True

        if r and s and r.val == s.val:

            return *self*.same(r.right , s.right) and *self*.same(r.left , s.left)

        return False