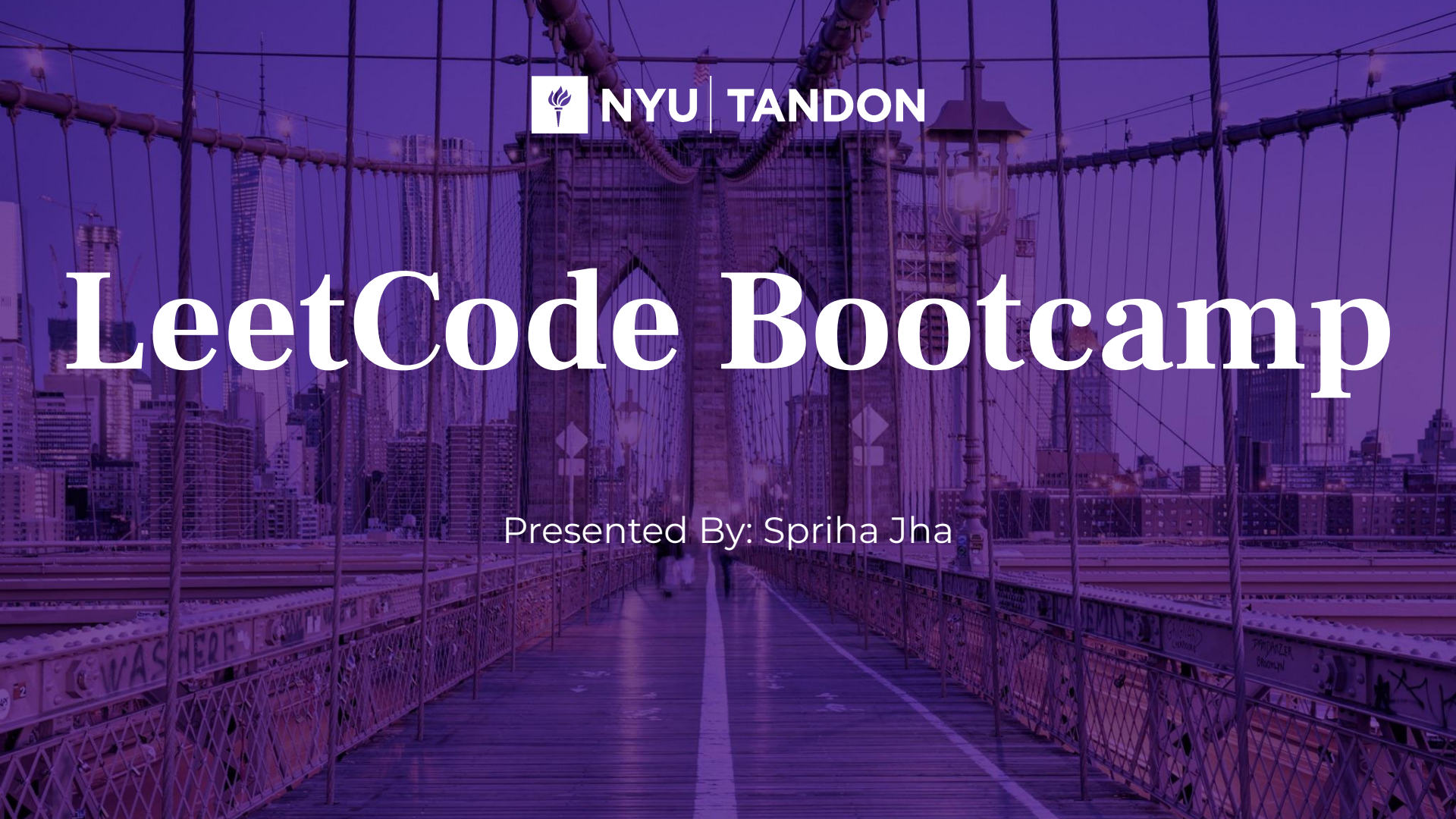

NYU TANDON
LeetCode Bootcamp
Presented By: Spriha Jha

Session Outline

01. Introduction to Binary Tree, Trie & Heap

02. Problem Sets

03. Debrief & Q/A

Binary Tree

- One of the important types of non-linear data structures is a tree.
- Trees — like linked lists — are made up of nodes and links.
- Trees are undirected and connected acyclic graph. There are no cycles or loops.
- Complete binary tree - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Balanced binary tree - A binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.
- In-order traversal - Left -> Root -> Right
- Pre-order traversal - Root -> Left -> Right
- Post-order traversal - Left -> Right -> Root

Binary Tree Terms

- **Root**: the topmost node of the tree, which never has any links or edges connecting to it
- **Neighbor** - Parent or child of a node
- **Ancestor** - A node reachable by traversing its parent chain
- **Descendant** - A node in the node's subtree
- **Degree** of a tree - Maximum degree of nodes in the tree
- **Distance** - Number of edges along the shortest path between two nodes
- **Level/Depth** - Number of edges along the unique path between a node and the root node
- **Width** - Number of nodes in a level

Binary Tree

- If a tree has n nodes, it will always have one less number of edges (n-1).
- Trees are recursive data structures because a tree is usually composed of smaller trees — often referred to as subtrees — inside of it.
- A simple way to think about the depth of a node is by answering the question: how far away is the node from the root of the tree?
- The height of a node can be simplified by asking the question: how far is this node from its furthest-away leaf?
- A tree is considered to be balanced if any two sibling subtrees do not differ in height by more than one level. However, if two sibling subtrees differ significantly in height (and have more than one level of depth of difference), the tree is unbalanced.

Binary Trees

Corner cases (Trees):

- Empty tree
- Single node
- Two nodes
- Very skewed tree (like a linked list)

Techniques:

- Use recursion
- Traversing by level
- Summation of nodes

PART 02

Problem Sets

Steps to approach the question:

Understand the problem

Take time to carefully read through the problem from start to finish is critical in finding the correct and complete solution to the problem in hand.

Code your solution

Map out your solution before you write any code. Avoid too much time trying to find the perfect solution. Validate your solution early and often.

Manage your time

Don’t forget, you have multiple questions to complete within a said time. Make sure you allocate enough time to carefully consider all problems.

PART 02

Problem 1: Invert Binary Tree

LeetCode Explore Problems Interview Contest Discuss Store

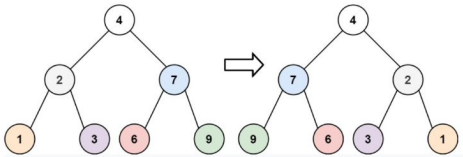
Description | Solution | Discuss (999+) | Submissions | Python3 | Autocomplete

226. Invert Binary Tree

Easy 10144 142 Add to List Share

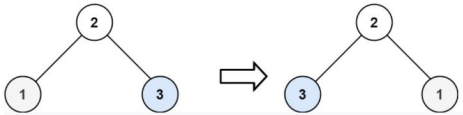
Given the `root` of a binary tree, invert the tree, and return *its root*.

Example 1:




```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
```

Example 2:




```
Input: root = [2,1,3]
Output: [2,3,1]
```

Example 3:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

```

Problems | Pick One | Prev | 9/69 | Next | Console | Contribute | Run Code | Submit

NYU TANDON

Approach: Iterative

```
def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
    if not root:
        return None

    queue = collections.deque([root])
    while queue:
        current = queue.popleft()
        current.left, current.right = current.right, current.left

        if current.left:
            queue.append(current.left)

        if current.right:
            queue.append(current.right)

    return root
```

The idea is that we need to swap the left and right child of all nodes in the tree. So we create a queue to store nodes whose left and right child have not been swapped yet. Initially, only the root is in the queue. As long as the queue is not empty, remove the next node from the queue, swap its children, and add the children to the queue. Null nodes are not added to the queue. Eventually, the queue will be empty and all the children swapped, and we return the original root.

Complexity Analysis

Time complexity : Since each node in the tree is visited / added to the queue only once, the time complexity is $O(n)$, where n is the number of nodes in the tree.

Space complexity : Space complexity is $O(n)$, since in the worst case, the queue will contain all nodes in one level of the binary tree.

Problem 2: Maximum difference between node and ancestor

LeetCode

Problem List

Description | Editorial | Solutions (2.2K) | Submissions

1026. Maximum Difference Between Node and Ancestor

Hint

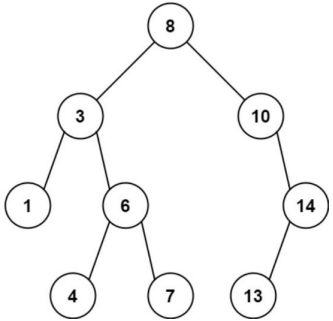
Medium 4K 95

Amazon Facebook Google

Given the `root` of a binary tree, find the maximum value `v` for which there exist **different** nodes `a` and `b` where `v = |a.val - b.val|` and `a` is an ancestor of `b`.

A node `a` is an ancestor of `b` if either: any child of `a` is equal to `b` or any child of `a` is an ancestor of `b`.

Example 1:




```
Input: root = [8,3,10,1,6,null,14,null,null,4,7,13]
Output: 7
Explanation: We have various ancestor-node differences, some of which are given
below :
|8 - 3| = 5
|3 - 7| = 4
```

Python3 | Auto

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def maxAncestorDiff(self, root: Optional[TreeNode]) -> int:

```

Testcase | Result

Accepted Runtime: 41 ms

Case 1 Case 2

Console | Run | Submit

NYU TANDON

Approach: Max minus Min

```
def maxAncestorDiff(self, root: TreeNode) -> int:
    if not root:
        return 0

    def helper(node, cur_max, cur_min):
        # if encounter leaves, return the max-min along the path
        if not node:
            return cur_max - cur_min
        # else, update max and min
        # and return the max of left and right subtrees
        cur_max = max(cur_max, node.val)
        cur_min = min(cur_min, node.val)
        left = helper(node.left, cur_max, cur_min)
        right = helper(node.right, cur_max, cur_min)
        return max(left, right)

    return helper(root, root.val, root.val)
```

Complexity Analysis

N is the *number of people*, and *E* is the *number of edges* (trust relationships).

Time complexity : $O(N)$.

Space complexity : $O(N)$.

Given any two nodes on the same root-to-leaf path, they must have the required ancestor relationship.

Therefore, we just need to record the maximum and minimum values of all root-to-leaf paths and return the maximum difference.

To achieve this, we can record the maximum and minimum values during the recursion and return the difference when encountering leaves.

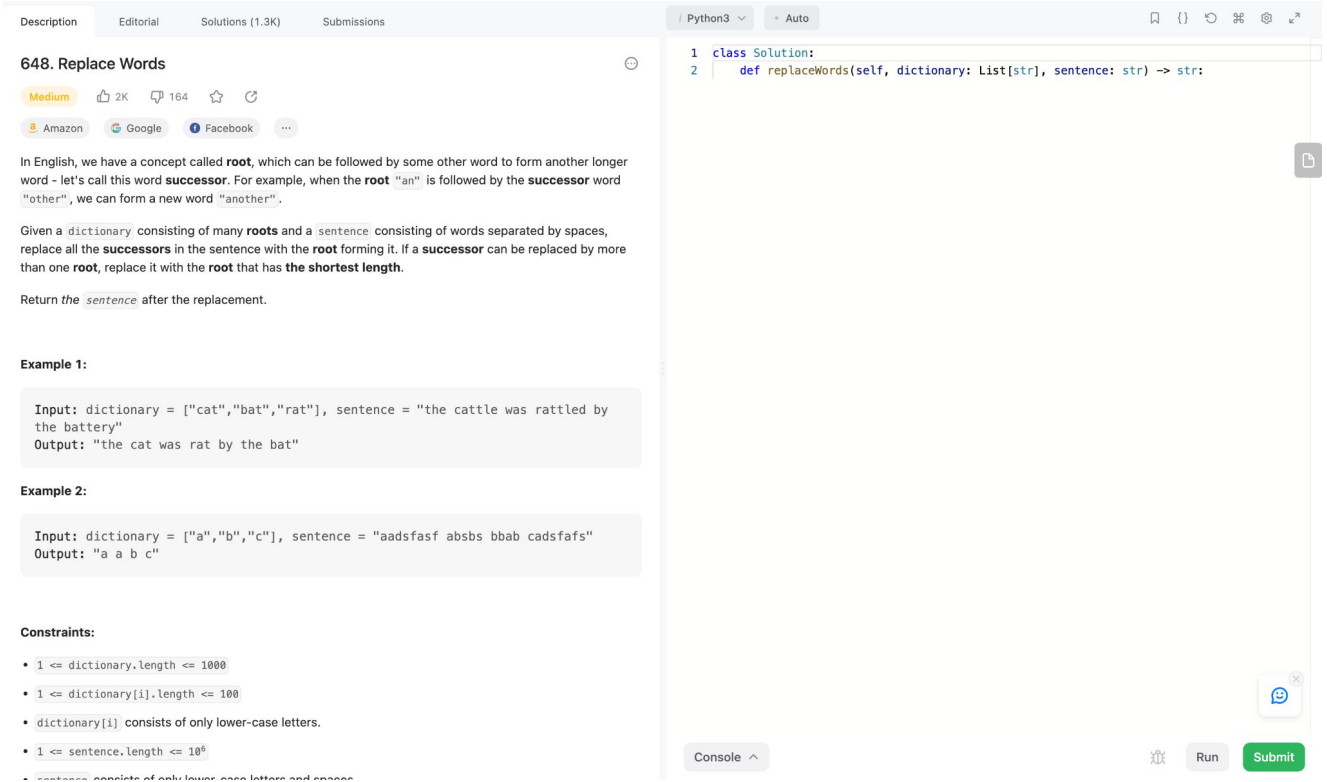
Algorithm

Step 1: Define a function helper, which takes three arguments as input and returns an integer.

- The first argument node is the current node, and the second argument cur_max and third argument cur_min are the maximum and minimum values along the root to the current node, respectively.
- Function helper returns cur_max - cur_min when encountering leaves. Otherwise, it calls helper on the left and right subtrees and returns their maximum.

Step 2: Run helper on the root and return the result.

Problem 3: Replace Words



Description | Editorial | Solutions (1.3K) | Submissions

648. Replace Words

Medium 2K 164

Amazon Google Facebook ...

In English, we have a concept called **root**, which can be followed by some other word to form another longer word - let's call this word **successor**. For example, when the **root** `"an"` is followed by the **successor** word `"other"`, we can form a new word `"another"`.

Given a `dictionary` consisting of many **roots** and a `sentence` consisting of words separated by spaces, replace all the **successors** in the sentence with the **root** forming it. If a **successor** can be replaced by more than one **root**, replace it with the **root** that has **the shortest length**.

Return *the* `sentence` after the replacement.

Example 1:

```
Input: dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by the battery"
Output: "the cat was rat by the bat"
```

Example 2:

```
Input: dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"
Output: "a a b c"
```

Constraints:

- `1 <= dictionary.length <= 1000`
- `1 <= dictionary[i].length <= 100`
- `dictionary[i]` consists of only lower-case letters.
- `1 <= sentence.length <= 10^6`
- `sentence` consists of only lower-case letters and spaces

Python3 Auto

```
1 class Solution:
2     def replaceWords(self, dictionary: List[str], sentence: str) -> str:
```

Console Run Submit

NYU TANDON

Approach: Trie

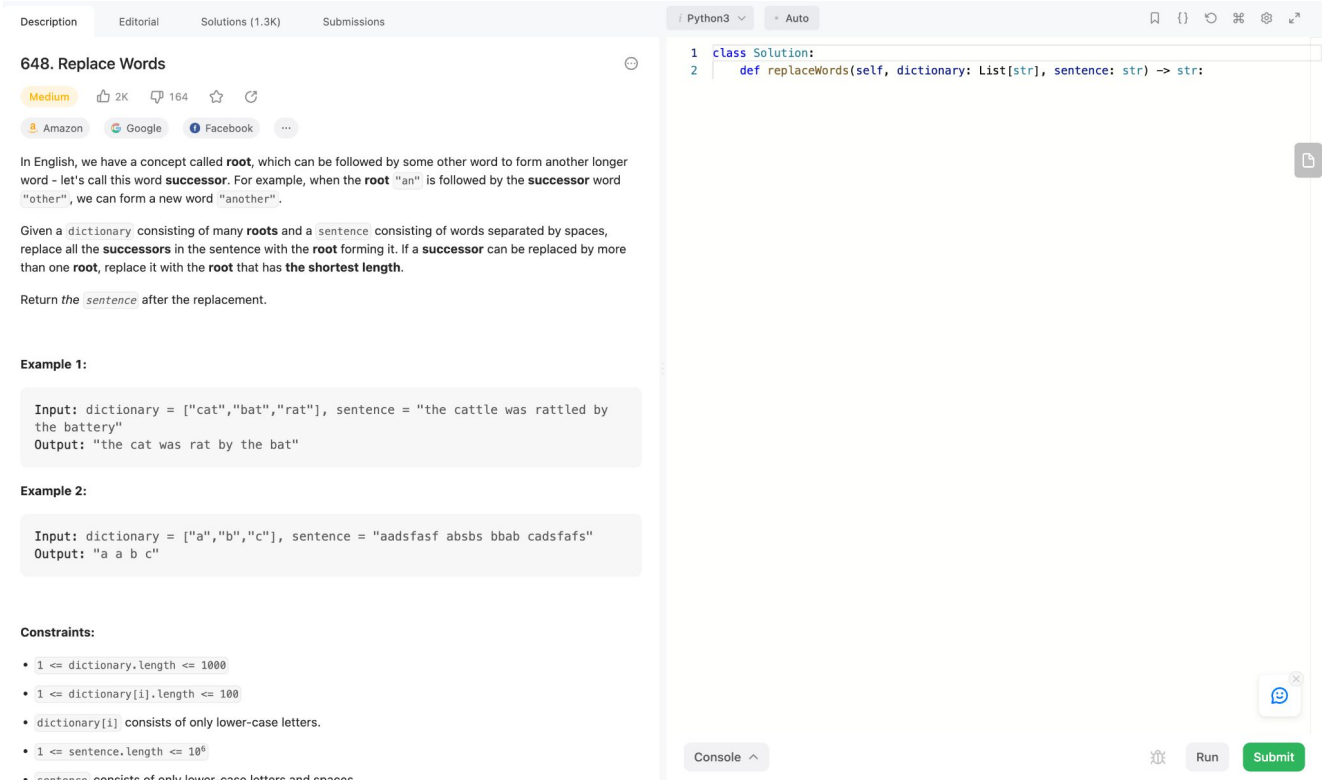
```
def replaceWords(self, roots, sentence):
    Trie = lambda: collections.defaultdict(Trie)
    trie = Trie()
    END = True

    for root in roots:
        reduce(dict.__getitem__, root, trie)[END] = root

    def replace(word):
        cur = trie
        for letter in word:
            if letter not in cur or END in cur: break
            cur = cur[letter]
        return cur.get(END, word)

    return " ".join(map(replace, sentence.split()))
```

Problem 3: Replace Words



Description | Editorial | Solutions (1.3K) | Submissions

648. Replace Words

Medium 2K 164

Amazon Google Facebook ···

In English, we have a concept called **root**, which can be followed by some other word to form another longer word - let's call this word **successor**. For example, when the **root** `"an"` is followed by the **successor** word `"other"`, we can form a new word `"another"`.

Given a `dictionary` consisting of many **roots** and a `sentence` consisting of words separated by spaces, replace all the **successors** in the sentence with the **root** forming it. If a **successor** can be replaced by more than one **root**, replace it with the **root** that has **the shortest length**.

Return *the* `sentence` after the replacement.

Example 1:

```
Input: dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by the battery"
Output: "the cat was rat by the bat"
```

Example 2:

```
Input: dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"
Output: "a a b c"
```

Constraints:

- `1 <= dictionary.length <= 1000`
- `1 <= dictionary[i].length <= 100`
- `dictionary[i]` consists of only lower-case letters.
- `1 <= sentence.length <= 10^6`

Python3 | Auto

```
1 class Solution:
2     def replaceWords(self, dictionary: List[str], sentence: str) -> str:
```

Console | Run | Submit

NYU TANDON

Approach: Max minus Min

```
def maxAncestorDiff(self, root: TreeNode) -> int:
    if not root:
        return 0

    def helper(node, cur_max, cur_min):
        # if encounter leaves, return the max-min along the path
        if not node:
            return cur_max - cur_min
        # else, update max and min
        # and return the max of left and right subtrees
        cur_max = max(cur_max, node.val)
        cur_min = min(cur_min, node.val)
        left = helper(node.left, cur_max, cur_min)
        right = helper(node.right, cur_max, cur_min)
        return max(left, right)

    return helper(root, root.val, root.val)
```

Given any two nodes on the same root-to-leaf path, they must have the required ancestor relationship.

Therefore, we just need to record the maximum and minimum values of all root-to-leaf paths and return the maximum difference.

To achieve this, we can record the maximum and minimum values during the recursion and return the difference when encountering leaves.

Algorithm

Step 1: Define a function helper, which takes three arguments as input and returns an integer.

- The first argument node is the current node, and the second argument cur_max and third argument cur_min are the maximum and minimum values along the root to the current node, respectively.
- Function helper returns cur_max - cur_min when encountering leaves. Otherwise, it calls helper on the left and right subtrees and returns their maximum.

Step 2: Run helper on the root and return the result.

PART 06

Q/A


NYU TANDON
Thank you!
Upcoming: BFS/DFS/BST