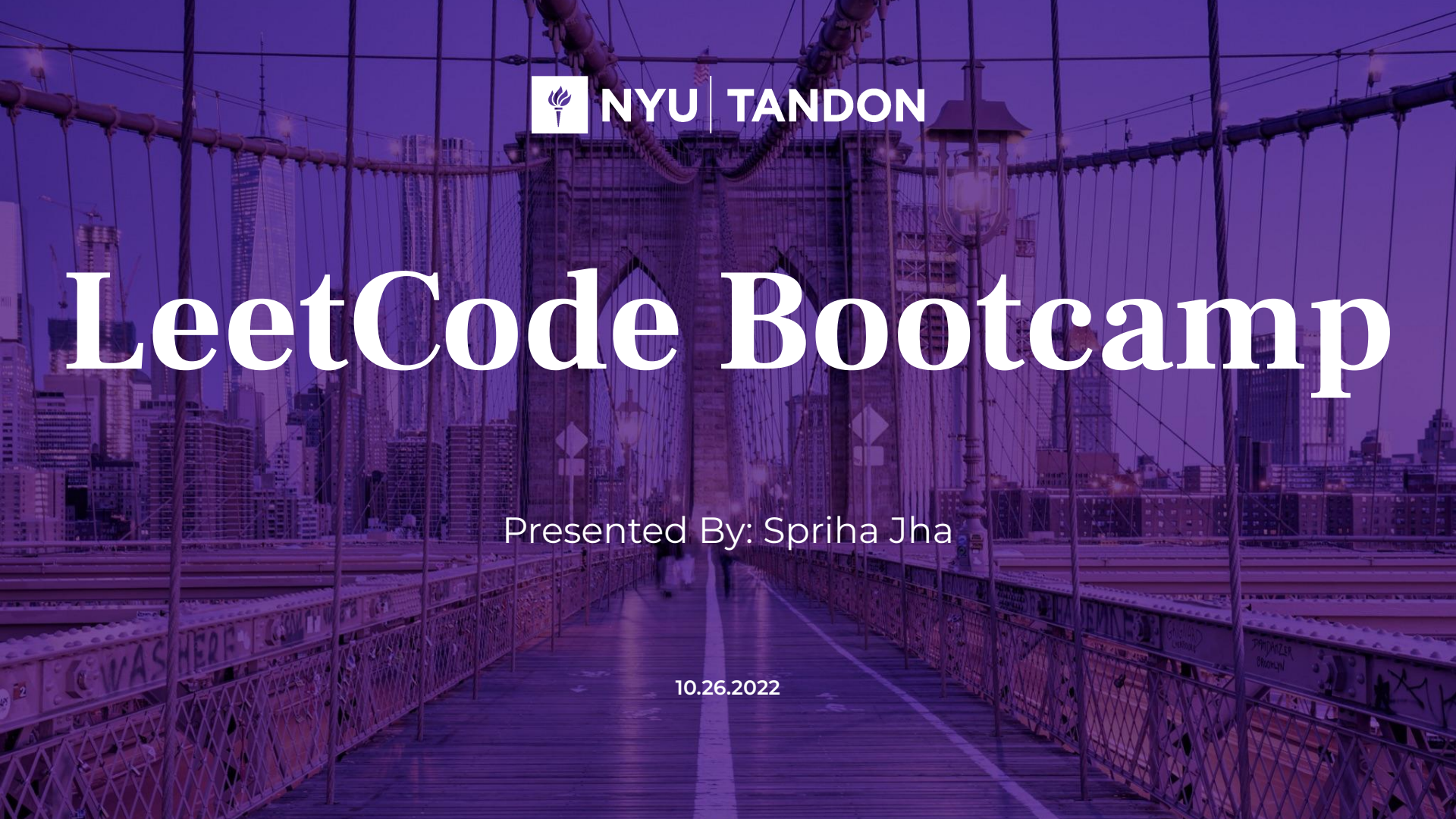


LeetCode Bootcamp

Presented By: Spriha Jha

10.26.2022

Session Outline

01. Introduction to Binary Trees & Graphs

02. Problem Sets

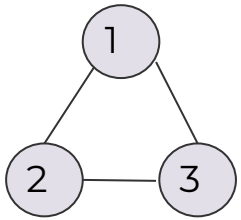
03. Debrief & Q/A

NYU TANDON

Graph

- A ***graph*** is a data structure with two distinct parts: a finite set of vertices, which are also called nodes, and a finite set of edges, which are references/links/pointers from one vertex to another.
- In directed graphs, the connections between nodes have a direction, and are called ***arcs***; in undirected graphs, the connections have no direction and are called ***edges***.
- Sometimes the nodes or arcs of a graph have weights or costs associated with them, and we are interested in finding the cheapest path.
- The characteristics of graph are strongly tied to what its vertices and edges look like. (Dense/Sparse)

Graph Representation



- As a list (or array) is called an ***edge list***, and is a representation of all the edges (*|E|*) in the graph. [[1, 2],[2, 3],[3, 1]]
- Or, an ***adjacency matrix*** is a matrix representation of exactly *which* nodes in a graph contain edges between them. [[0, 1, 1],[1, 0, 1],[1, 1, 0]]
- The matrix is kind of like a lookup table: once we've determined the two nodes that we want to find an edge between, we look at the value at the intersection of those two nodes.
- The values in the adjacency matrix are like boolean flag indicators; they are either present or not present. If the value is 1, that means that there is an edge between the two nodes; if the value is 0, that means an edge does not exist between them.
- We will have a value of 0 down the diagonal, since most graphs that we're dealing with won't be referential.

Graph Representation

Adjacency Matrix			
	0	**1**	**2**
0	0	1	1
1	1	0	1
2	1	1	0

Adjacency List		
0:	1	2
1:	0	2
2:	0	1

Edge List		
0:	0	1
1:	0	2
2:	1	2

- ***Adjacency list*** hybrid between an *edge list* and an *adjacency matrix*.
- Each vertex is given an index in its list, and has all of its neighboring vertices stored as an linked list (which could also be an array), adjacent to it.
- We can see that, because of the *structure* of an adjacency list, it's very easy to determine all the neighbors of one particular vertex.
- The ***degree*** of a vertex is the number of edges that it has, which is also known as the number of neighboring nodes that it has.

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 & Graphs

Corner cases (Graphs):

- Empty graph
- Graph with one or two nodes
- Disjoint graphs
- Graph with cycles

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

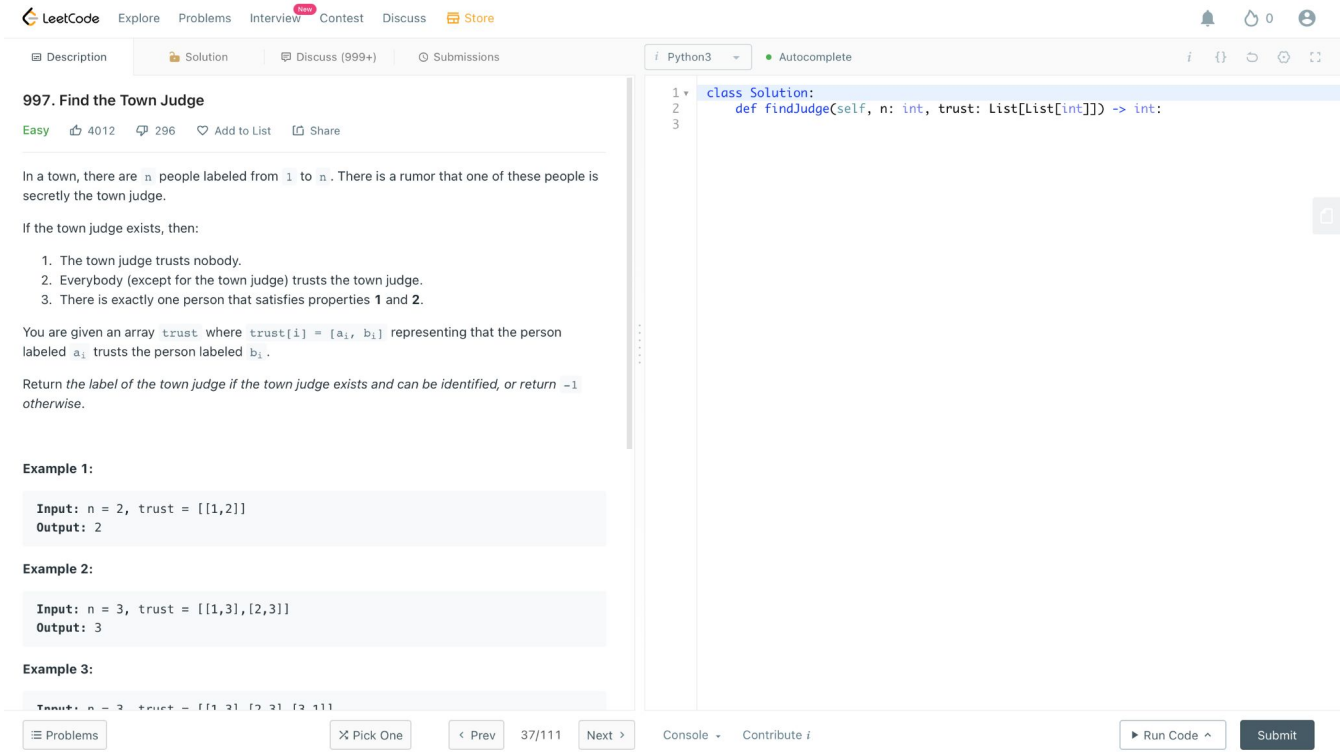
Map out your solution before you write any code. Avoid too much time trying to find the perfect solution. Validate your solution early and often.

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: Find the town judge



NYU TANDON

Approach: One Array

```
def findJudge(self, N: int, trust: List[List[int]]) -> int:

    if len(trust) < N - 1:
        return -1

    trust_scores = [0] * (N + 1)

    for a, b in trust:
        trust_scores[a] -= 1
        trust_scores[b] += 1

    for i, score in enumerate(trust_scores[1:], 1):
        if score == N - 1:
            return i
    return -1
```

Complexity Analysis

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

Time complexity : $O(E)$.

Space complexity : $O(N)$.

- Each person gains 1 "point" for each person they are trusted by, and loses 1 "point" for each person they trust. Then at the end, the town judge, if they exist, must be the person with N - 1 "points".
- Therefore, for a person to maximize their "score", they should be trusted by as many people as possible, and trust as few people as possible.
- In graph theory, we say the **outdegree** of a vertex (person) is the number of directed edges going out of it. For this graph, the outdegree of the vertex represents the number of other people that person trusts.
- Likewise, we say that the **indegree** of a vertex (person) is the number of directed edges going *into* it. So here, it represents the number of people *trusted by* that person.
- The maximum indegree is N - 1. This represents everybody trusting the person (except for themselves, they cannot trust themselves). The minimum indegree is 0. This represents not trusting anybody. Therefore, the maximum value for indegree - outdegree is **(N - 1) - 0 = N - 1**. These values also happen to be the definition of the town judge!

PART 02

Problem 2: Binary Tree Maximum Path Sum

LeetCode Explore Problems Interview Contest Discuss Store

Description Solution Discuss (999+) Submissions

Python3 Autocomplete

124. Binary Tree Maximum Path Sum

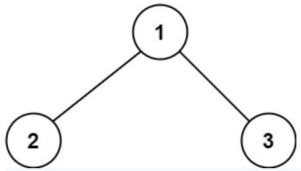
Hard 11948 592 Add to List Share

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the `root` of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

Example 1:




```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:


```
# 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 maxPathSum(self, root: Optional[TreeNode]) -> int:

```

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

NYU TANDON

Approach: Recursion

```
def maxPathSum(self, root):
    nonlocal max_sum
    if not node:
        return 0

    # max sum on the left and right sub-trees of node
    left_gain = max(max_gain(node.left), 0)
    right_gain = max(max_gain(node.right), 0)

    # the price to start a new path where `node` is a highest node
    price_newpath = node.val + left_gain + right_gain

    # update max_sum if it's better to start a new path
    max_sum = max(max_sum, price_newpath)

    # for recursion :
    # return the max gain if continue the same path
    return node.val + max(left_gain, right_gain)

max_sum = float('-inf')
max_gain(root)
return max_sum
```

1. Implement a function **max_gain(node)** which takes a node as an argument and computes a maximum contribution that this node and one/zero of its subtrees could add.
2. Check at each step what is better : to continue the current path or to start a new path with the current node as a highest node in this new path.
3. Initiate **max_sum** as the smallest possible integer and call max_gain(node = root).
4. Implement max_gain(node) with a check to continue the old path/to start a new path:
    a. Base case : if node is null, the max gain is 0.
    b. Call max_gain recursively for the node children to compute max gain from the left and right subtrees : left_gain = max(max_gain(node.left), 0) and right_gain = max(max_gain(node.right), 0).
    c. Now check to continue the old path or to start a new path. To start a new path would cost price_newpath = node.val + left_gain + right_gain. Update max_sum if it's better to start a new path.
    d. For the recursion return the max gain the node and one/zero of its subtrees could add to the current path : node.val + max(left_gain, right_gain).

Problem 3: Invert Binary Tree

LeetCode Explore Problems Interview Contest Discuss Store

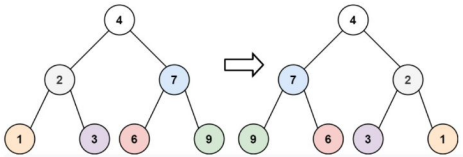
Description | Solution | Discuss (999+) | Submissions | Python3 | Autocomplete

226. Invert Binary Tree

Easy 10144 142 Add to List Share

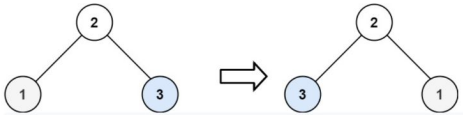
Given the `root` of a binary tree, invert the tree, and return *its root*.

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 | 69/69 | Next | Console | Contribute | Run Code | Submit

NYU TANDON

Approach: Recursion

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

    right = self.invertTree(root.right)
    left = self.invertTree(root.left)
    root.left = right
    root.right = left
    return root
```

The inverse of an empty tree is the empty tree. The inverse of a tree with root *r*, and subtrees right and left is a tree with root *r*, whose right subtree is the inverse of left, and whose left subtree is the inverse of right.

Complexity Analysis

Time complexity : Since each node in the tree is visited only once, the time complexity is $O(n)$, where n is the number of nodes in the tree. We cannot do better than that, since at the very least we have to visit each node to invert it.

Space complexity : Because of recursion, $O(h)$ function calls will be placed on the stack in the worst case, where h is the height of the tree. Because $h \in O(n)$, the space complexity is $O(n)$.

.

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.

PART 06

Q/A

Slack Hours

Join Slack Workspace!

Office Hours: Tuesday (10AM - 1PM)

Problem Assignments

Graphs

01. Flood fill (Easy)

02. Minimum height trees (Medium)

03. O1 Matrix (Medium)

04. Course Schedule (Medium)

05. Accounts Merge (Medium)

06. Word Ladder (Hard)

Problem Assignments

Binary Trees

01. Balance binary tree (Easy)

02. Maximum depth of binary tree (Easy)

03. Binary Tree level order traversal (Medium)

04. Lowest common ancestor of a Binary Tree (Medium)

05. Binary Tree right side view (Medium)

06. Serialize and Deserialize Binary Tree (Hard)


NYU TANDON
Thank you!
Upcoming: BFS/DFS (11/2)