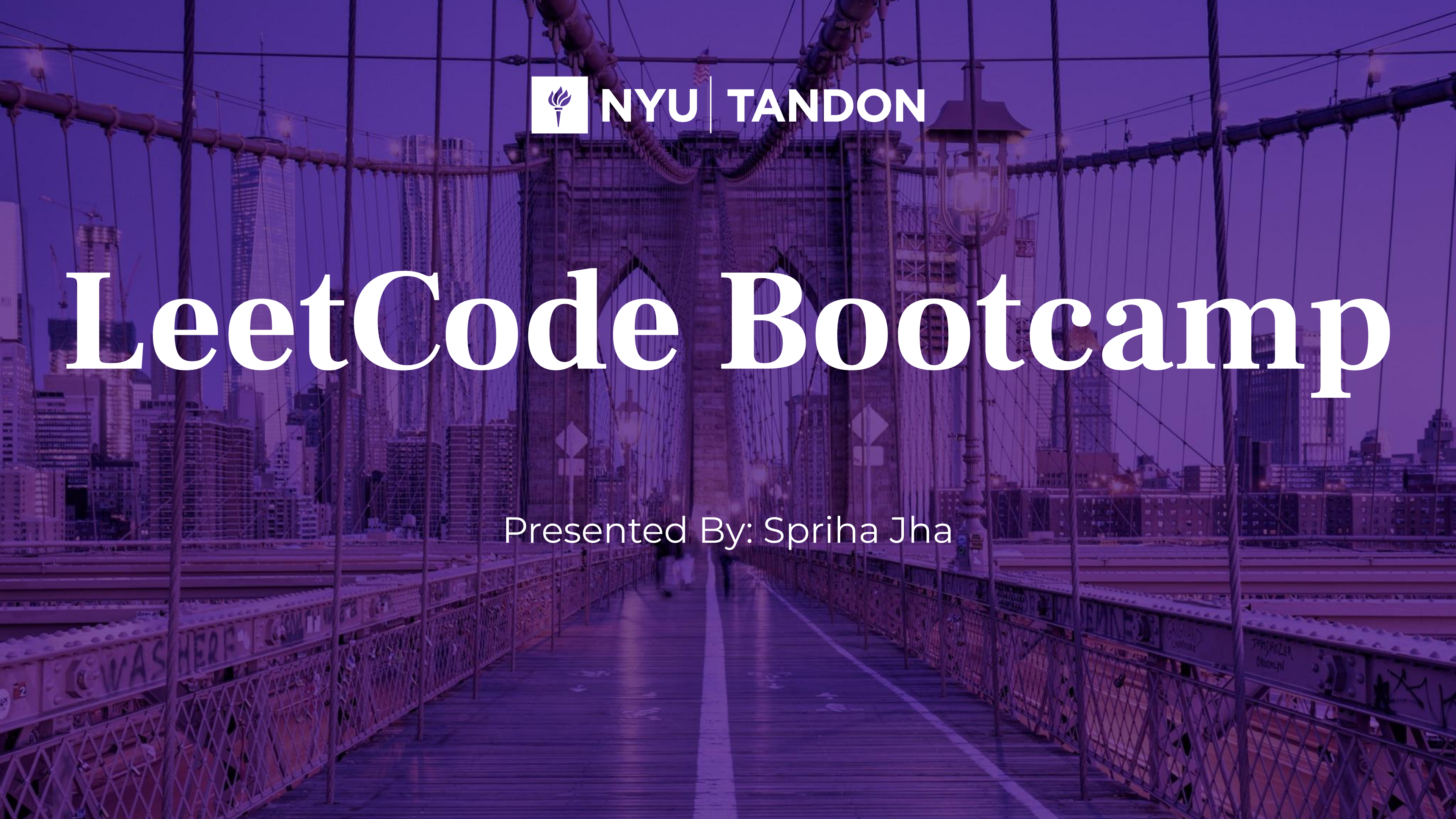


NYU | TANDON

LeetCode Bootcamp

Presented By: Spriha Jha

Session Outline

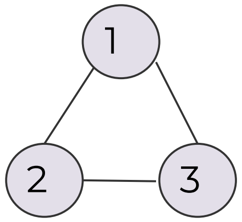
NYU | TANDON

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)

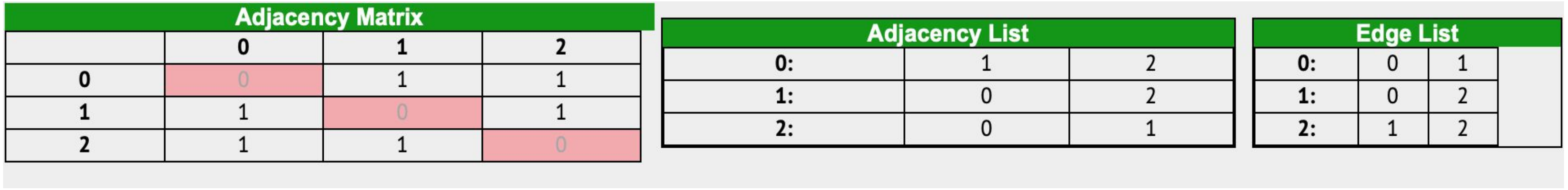
Graph Representation

-



- 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.

NYU | TANDON

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

NYU | TANDON

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 Search Tree

Properties of a BST:

1. Left subtree of a node N contains nodes whose values are lesser than or equal to node N's value.
2. Right subtree of a node N contains nodes whose values are greater than node N's value.
3. Both left and right subtrees are also BSTs.

Depth-first Search

Depth-first search is a graph traversal algorithm which explores as far as possible along each branch before backtracking. A stack is usually used to keep track of the nodes that are on the current search path. This can be done either by an implicit recursion stack, or an actual stack data structure. A simple template for doing depth-first searches on a matrix goes like this:

```
def dfs(matrix):
 # Check for an empty matrix/graph.
 if not matrix:
   return [ ]

 rows, cols = len(matrix), len(matrix[0])
 visited = set()
 directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

 def traverse(i, j):
   if (i, j) in visited:
     return

   visited.add((i, j))
   # Traverse neighbors.
   for direction in directions:
     next_i, next_j = i + direction[0], j + direction[1]
     if 0 <= next_i < rows and 0 <= next_j < cols:
       # Add in question-specific checks, where relevant.
       traverse(next_i, next_j)

 for i in range(rows):
   for j in range(cols):
     traverse(i, j)
```

NYU | TANDON

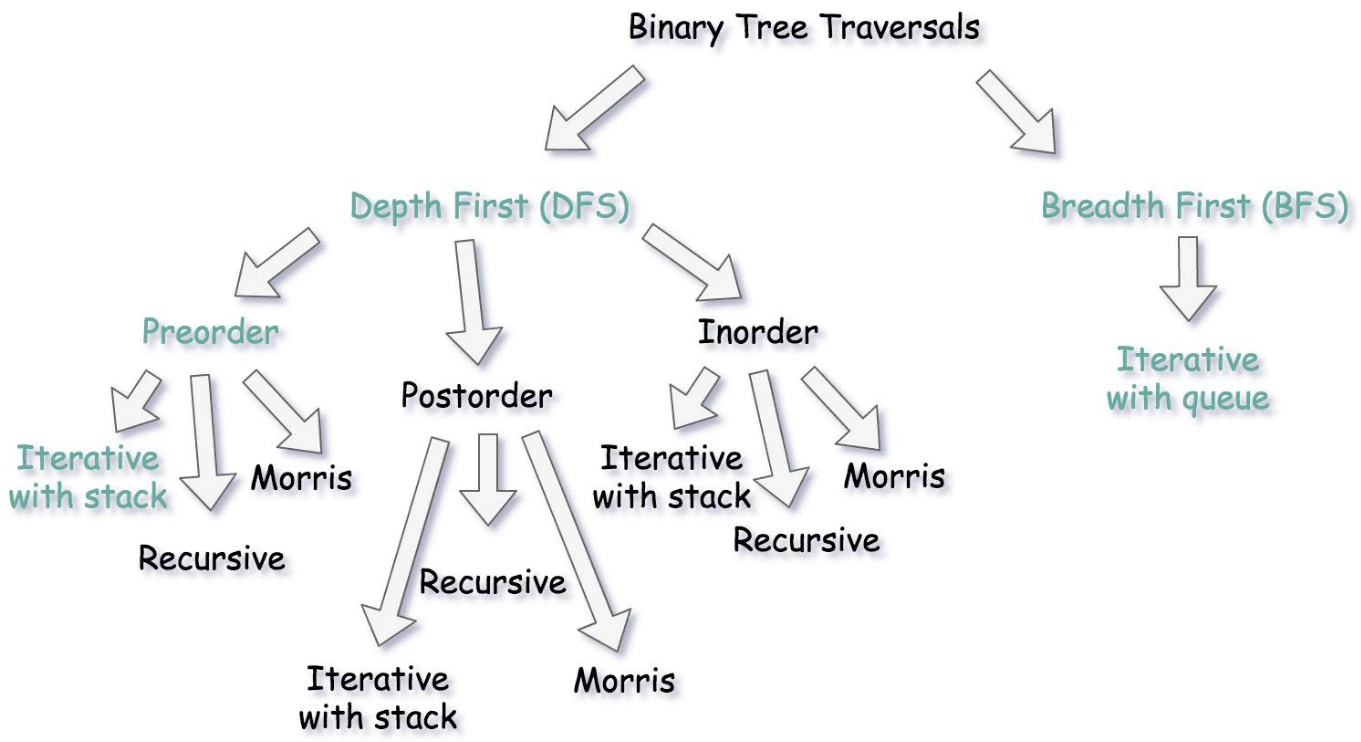
Breadth-first Search

Breadth-first search is a graph traversal algorithm which starts at a node and explores all nodes at the present depth, before moving on to the nodes at the next depth level. A queue is usually used to keep track of the nodes that were encountered but not yet explored.
A similar template for doing breadth-first searches on the matrix goes like this. It is important to use double-ended queues and not arrays/Python lists as enqueuing for double-ended queues is O(1) but it's O(n) for arrays.

```
from collections import deque

def bfs(matrix):
 # Check for an empty matrix/graph.
 if not matrix:
   return []

 rows, cols = len(matrix), len(matrix[0])
 visited = set()
 directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

 def traverse(i, j):
   queue = deque([(i, j)])
   while queue:
     curr_i, curr_j = queue.popleft()
     if (curr_i, curr_j) not in visited:
       visited.add((curr_i, curr_j))
       # Traverse neighbors.
       for direction in directions:
         next_i, next_j = curr_i + direction[0], curr_j + direction[1]
         if 0 <= next_i < rows and 0 <= next_j < cols:
           # Add in question-specific checks, where relevant.
           queue.append((next_i, next_j))

 for i in range(rows):
   for j in range(cols):
     traverse(i, j)
```

NYU | TANDON

Overview

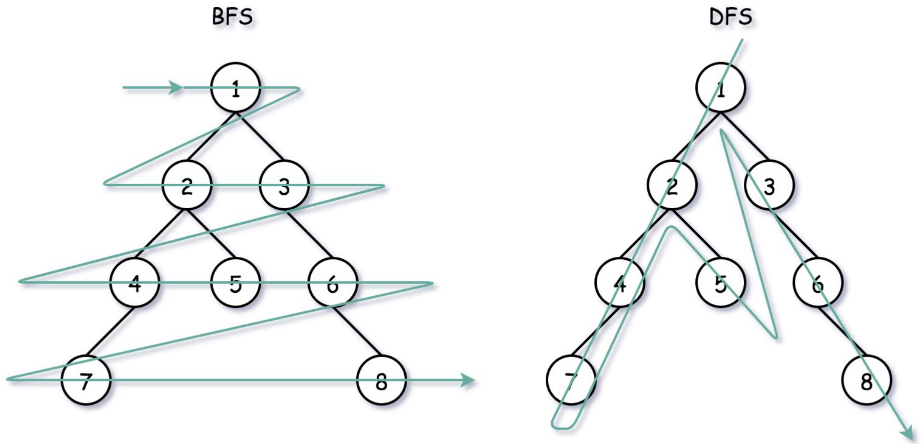


Binary Tree Traversals
Depth First (DFS)
Breadth First (BFS)
Preorder
Postorder
Inorder
Iterative with queue
Iterative with stack
Recursive
Morris
Recursive
Iterative with stack
Morris
Iterative with stack
Recursive
Morris

Overview

Both are starting from the root and going down, both are using additional structures, what's the difference?

Here is how it looks at the big scale: BFS traverses level by level, and DFS first goes to the leaves.



NYU | TANDON

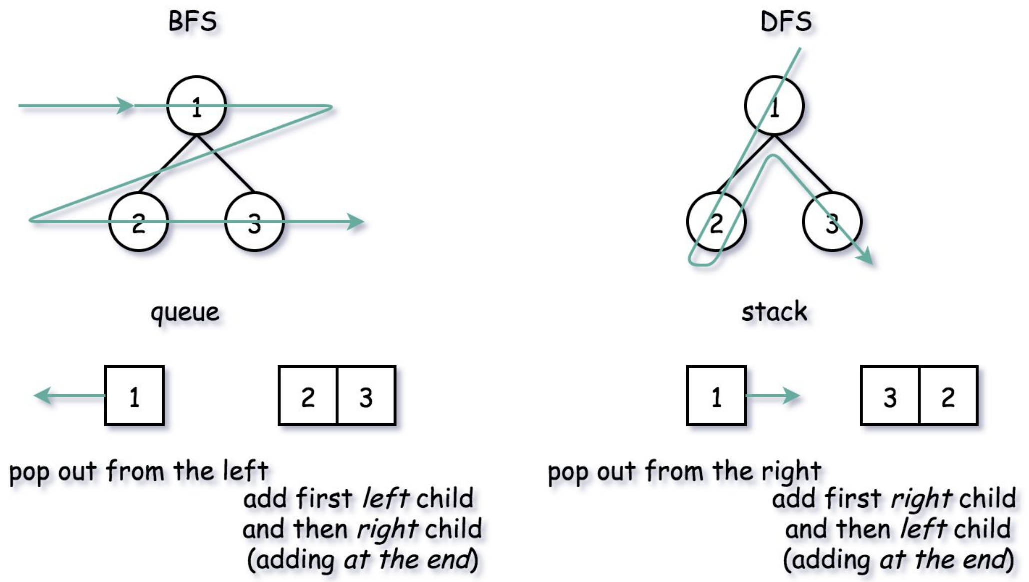
Overview

Now let's go down to the implementation. The idea is similar:

- Push root into queue (BFS) or stack (DFS).
- At each step pop out one node, and push its children into stack/queue.

For **BFS**: pop out from the *left*, first push the *left* child, and then the *right* one.

For **DFS**: pop out from the *right*, first push the *right* child, and then the *left* one.



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

NYU | TANDON

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

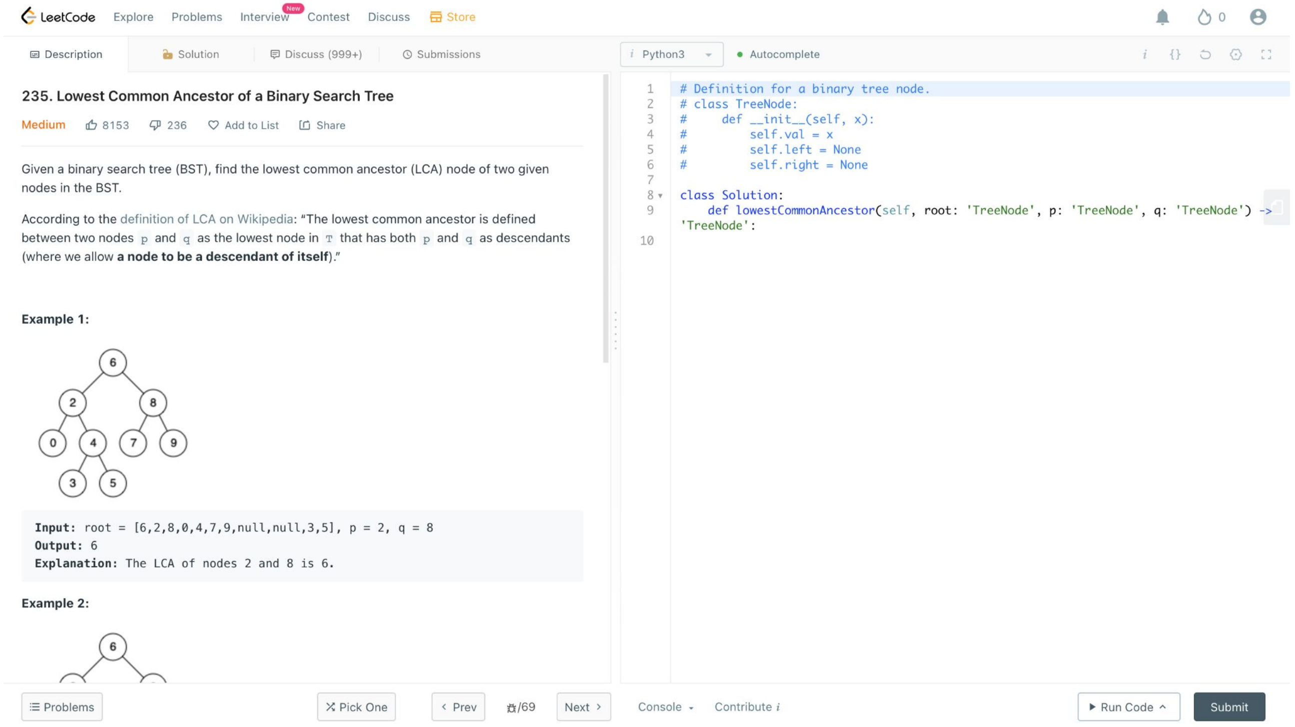
Map out your solution before you write any code. Avoid too much time trying to find the perfect solution. Validate your solution early and often.

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.

NYU | TANDON

Problem 1: Lowest Common Ancestor of a Binary Search Tree



235. Lowest Common Ancestor of a Binary Search Tree

Medium 8153 236 Add to List Share

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        
```

PROBLEM 1

Approach: Recursion

```
def lowestCommonAncestor(self, root, p, q):

    # Value of current node or parent node.
    parent_val = root.val

    # Value of p
    p_val = p.val

    # Value of q
    q_val = q.val

    # If both p and q are greater than parent
    if p_val > parent_val and q_val > parent_val:
        return self.lowestCommonAncestor(root.right, p, q)
    # If both p and q are lesser than parent
    elif p_val < parent_val and q_val < parent_val:
        return self.lowestCommonAncestor(root.left, p, q)
    # We have found the split point, i.e. the LCA node.
    else:
        return root
```

Algorithm

1. Start traversing the tree from the root node.
2. If both the nodes p and q are in the right subtree, then continue the search with right subtree starting step 1.
3. If both the nodes p and q are in the left subtree, then continue the search with left subtree starting step 1.
4. If both step 2 and step 3 are not true, this means we have found the node which is common to node p's and q's subtrees. and hence we return this common node as the LCA.

Complexity Analysis

Time complexity : $O(n)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.

Space complexity : $O(n)$, This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed BST could be N.

NYU | TANDON

Approach: Iterative

```
def lowestCommonAncestor(self, root, p, q):
    # Value of p
    p_val = p.val

    # Value of q
    q_val = q.val

    # Start from the root node of the tree
    node = root

    # Traverse the tree
    while node:

      # Value of current node or parent node.
      parent_val = node.val

      if p_val > parent_val and q_val > parent_val:
        # If both p and q are greater than parent
        node = node.right
      elif p_val < parent_val and q_val < parent_val:
        # If both p and q are lesser than parent
        node = node.left
      else:
        # We have found the split point, i.e. the LCA node.
        return node
```

Algorithm

The steps taken are also similar to approach 1. The only difference is instead of recursively calling the function, we traverse down the tree iteratively. This is possible without using a stack or recursion since we don't need to backtrace to find the LCA node. In essence of it the problem is iterative, it just wants us to find the split point. The point from where p and q won't be part of the same subtree or when one is the parent of the other.

Complexity Analysis

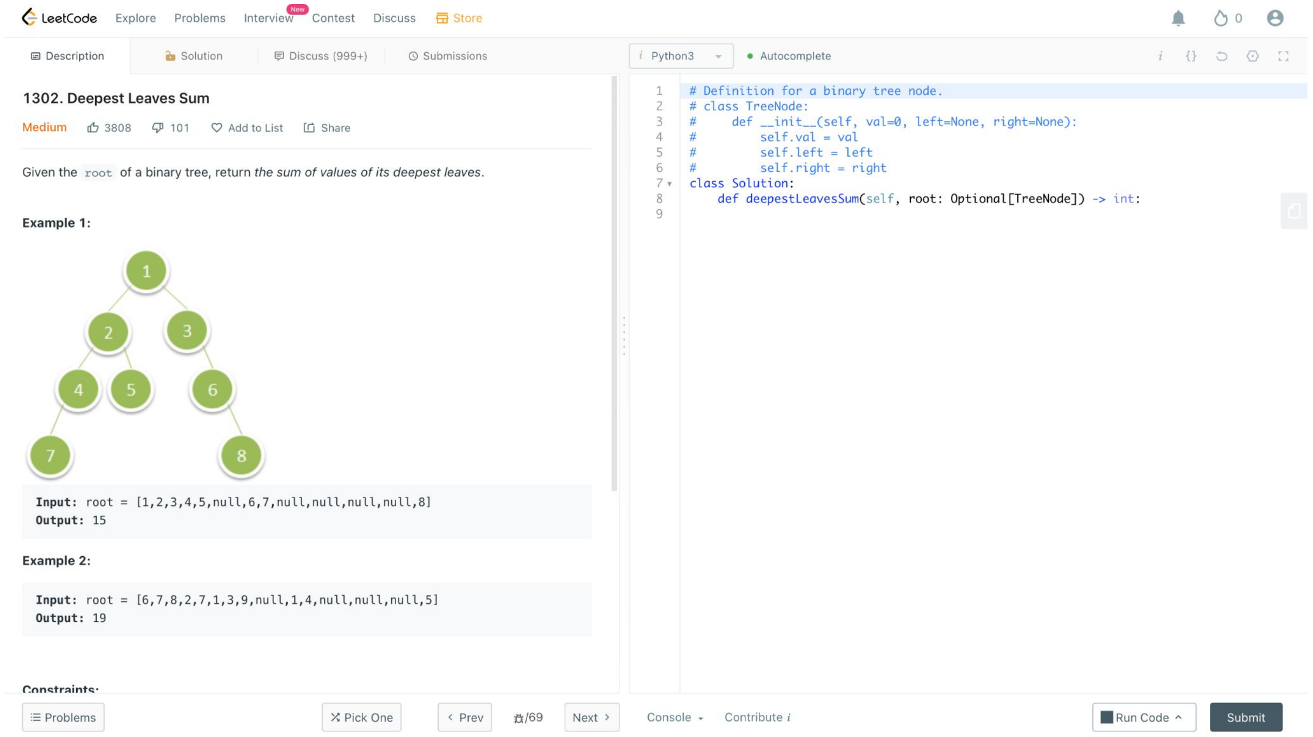
Time complexity : $O(n)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.

Space complexity : $O(1)$

NYU | TANDON

PART 02

Problem 2: Deepest Leaves Sum



1302. Deepest Leaves Sum

Given the `root` of a binary tree, return *the sum of values of its deepest leaves*.

Example 1:

```
Input: root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
Output: 15
```

Example 2:

```
Input: root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]
Output: 19
```

Constraints:

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

```

Approach: Iterative BFS Traversal

```
def deepestLeavesSum(self, root: TreeNode) -> int:
    next_level = deque([root,])

    while next_level:
        # prepare for the next level
        curr_level = next_level
        next_level = deque()

        for node in curr_level:
            # add child nodes of the current level
            # in the queue for the next level
            if node.left:
                next_level.append(node.left)
            if node.right:
                next_level.append(node.right)

    return sum([node.val for node in curr_level])
```

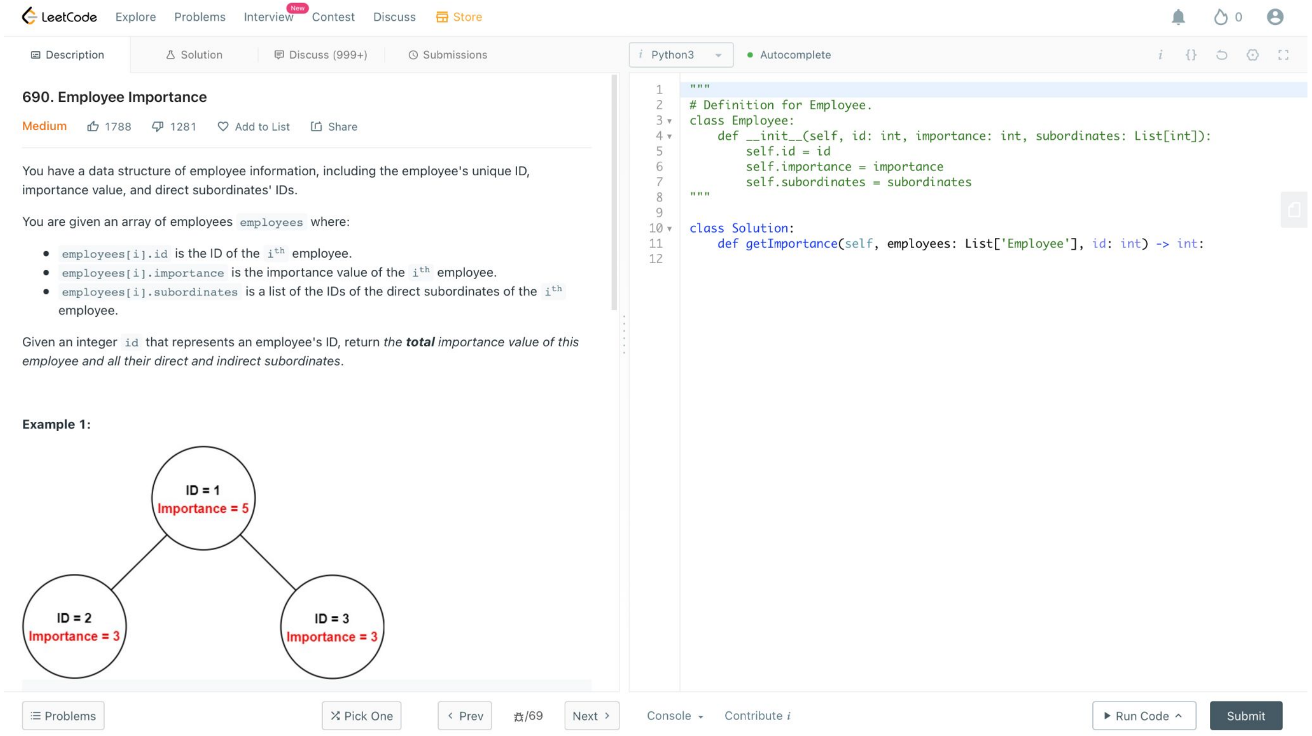
Traverse level by level, to check if this level is the last one. If it's the case, return the sum of all nodes values.

Complexity Analysis

Time complexity : $O(n)$, since one has to visit each node.

Space complexity : up to $O(N)$ to keep the queues. Let's use the last level to estimate the queue size. This level could contain up to $N/2$ tree nodes in the case of complete binary tree.

Problem 3: Employee Importance



690. Employee Importance

Medium 1788 1281 Add to List Share

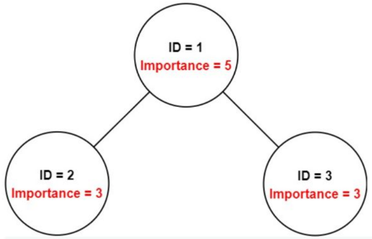
You have a data structure of employee information, including the employee's unique ID, importance value, and direct subordinates' IDs.

You are given an array of employees `employees` where:

- `employees[i].id` is the ID of the `i`th employee.
- `employees[i].importance` is the importance value of the `i`th employee.
- `employees[i].subordinates` is a list of the IDs of the direct subordinates of the `i`th employee.

Given an integer `id` that represents an employee's ID, return *the* ***total*** *importance value of this employee and all their direct and indirect subordinates*.

Example 1:



```
"""
# Definition for Employee.
class Employee:
    def __init__(self, id: int, importance: int, subordinates: List[int]):
        self.id = id
        self.importance = importance
        self.subordinates = subordinates
"""

class Solution:
    def getImportance(self, employees: List['Employee'], id: int) -> int:
```

Approach: DFS

```
def getImportance(self, employees, query_id):

    # Create a hashmap for employees.
    emap = {e.id: e for e in employees}

    # Define a recursive function
    def dfs(eid):
        employee = emap[eid]
        return (employee.importance +
                sum(dfs(eid) for eid in employee.subordinates))

    return dfs(query_id)
```

Let's use a hashmap emap = {employee.id -> employee} to query employees quickly.

Now to find the total importance of an employee, it will be the importance of that employee, plus the total importance of each of that employee's subordinates. This is a straightforward depth-first search.

Complexity Analysis

Time complexity : $O(n)$, where N is the number of employees. We might query each employee in dfs.

Space complexity : $O(N)$, the size of the implicit call stack when evaluating dfs.

PART 06

Q/A

NYU | TANDON

Thank you!

Upcoming: Greedy Algorithms, Dynamic Programming