Practice Interview

Objective

The partner assignment aims to provide participants with the opportunity to practice coding in an interview context. You will analyze your partner's Assignment 1. Moreover, code reviews are common practice in a software development team. This assignment should give you a taste of the code review process.

Group Size

Each group should have 2 people. You will be assigned a partner

Parts:

- Part 1: Complete 1 of 3 questions
- Part 2: Review your partner's Assignment 1 submission
- Part 3: Perform code review of your partner's assignment 1 by answering the questions below
- Part 3: Reflect on Assignment 1 and Assignment 2

Part 1:

You will be assigned one of three problems based of your first name. Enter your first name, in all lower case, execute the code below, and that will tell you your assigned problem. Include the output as part of your submission (do not clear the output). The problems are based-off problems from Leetcode.

```
import hashlib

def hash_to_range(input_string: str) -> int:
     hash_object = hashlib.sha256(input_string.encode())
     hash_int = int(hash_object.hexdigest(), 16)
     return (hash_int % 3) + 1
input_string = "abhijit"
result = hash_to_range(input_string)
print(result)
```

```
2
```

Starter Code for Question 1

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val = 0, left = None, right = None):
#         self.val = val
```

```
#         self.left = left
#         self.right = right
def is_duplicate(root: TreeNode) -> int:
  # TODO
```

Starter Code for Question 2

```
from typing import List

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def bt_path(root: TreeNode) -> List[List[int]]:
    """
    Find all root-to-leaf paths in a binary tree.

    Args:
        root: TreeNode representing the root of the binary tree

    Returns:
        List[List[int]]: List of all root-to-leaf paths, where each
path
                         is represented as a list of node values
    """
    if not root:
        return []

    result = []

    def dfs(node, path):
        # Add current node to path
        path.append(node.val)

        # If it's a leaf node, add the complete path to result
        if not node.left and not node.right:
            result.append(path[:])  # Make a copy of the path
        else:
            # Recursively explore left and right subtrees
            if node.left:
                dfs(node.left, path)
            if node.right:
                dfs(node.right, path)

        # Backtrack: remove current node from path
        path.pop()
```

```
    dfs(root, [])
    return result


def list_to_tree(arr: List[int]) -> TreeNode:
    """
    Convert a list of integers to a TreeNode using level-order
traversal.
    All values in the list are valid node values (no None
placeholders).

    Args:
        arr: List of integers representing nodes in level-order

    Returns:
        TreeNode: Root of the constructed binary tree

    Example:
        [1, 2, 3, 4, 5, 6, 7] creates:
            1
           / \
          2   3
         / \ / \
        4  5 6  7
    """
    if not arr:
        return None

    root = TreeNode(arr[0])
    queue = [root]
    i = 1

    while queue and i < len(arr):
        node = queue.pop(0)

        # Add left child
        if i < len(arr):
            node.left = TreeNode(arr[i])
            queue.append(node.left)
            i += 1

        # Add right child
        if i < len(arr):
            node.right = TreeNode(arr[i])
            queue.append(node.right)
            i += 1

    return root

def tree_to_list(root: TreeNode) -> List[int]:
```

```
    """
    Convert a TreeNode back to a list using level-order traversal.
    Useful for testing and visualization.

    Args:
        root: TreeNode representing the root of the binary tree

    Returns:
        List[int]: Level-order representation with None for missing
nodes
    """
    if not root:
        return []

    result = []
    queue = [root]

    while queue:
        node = queue.pop(0)

        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)

    # Remove trailing None values
    while result and result[-1] is None:
        result.pop()

    return result

# Example usage and test cases
if __name__ == "__main__":
    # Test the list_to_tree conversion
    print("=== Testing list_to_tree conversion ===")

    # Test case from your example
    arr_example = [1, 2, 2, 3, 5, 6, 7]
    root_example = list_to_tree(arr_example)
    print(f"Input list: {arr_example}")
    print(f"Converted back: {tree_to_list(root_example)}")
    print("All root-to-leaf paths:")
    paths = bt_path(root_example)
    print("Result:", paths)


=== Testing list_to_tree conversion ===
Input list: [1, 2, 2, 3, 5, 6, 7]
```

```
Converted back: [1, 2, 2, 3, 5, 6, 7]
All root-to-leaf paths:
Result: [[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]
```

Starter Code for Question 3

```
def missing_num(nums: List) -> int:
  # TODO
```

Part 2:

You and your partner must share each other's Assignment 1 submission.

Part 3:

Create a Jupyter Notebook, create 6 of the following headings, and complete the following for your partner's assignment 1:

- Paraphrase the problem in your own words.

While traversing a list left to right, output the first element which appears more than once or -1 if no elements repeat in the list

- Create 1 new example that demonstrates you understand the problem.
  Trace/walkthrough 1 example that your partner made and explain it.

```
nums = [1, 2, 3, 2, 1]
```

- Copy the solution your partner wrote.

```
from typing import List

def first_duplicate(nums: List[int]) -> int:
    # TODO
    seen  = set()
    for num in nums:
        if num in seen:
            return num
        seen.add(num)
    return -1
    pass
```

- Explain why their solution works in your own words.

```
print(first_duplicate(nums))
```

```
2
```

Tracing through the algorithm with the input [1, 2, 3, 2, 1]:

Start with empty set: seen = {} Process 1: 1 not in seen, add to seen → seen = {1} Process 2: 2 not in seen, add to seen → seen = {1, 2} Process 3: 3 not in seen, add to seen → seen = {1, 2, 3} Process 2: 2 IS in seen → Return 2

The result is 2. Even though 1 appears twice in the list, 2 is encountered as a duplicate first (at index 3), before we get to the second occurrence of 1 (at index 4). The algorithm returns the first value that appears for the second time as we scan from left to right.

- Explain the problem's time and space complexity in your own words.

Time complexity: O(n) where n is the length of the list, since set lookups and insertions are O(1) on average. Space complexity: O(k) where k is the number of unique elements encountered before finding the first duplicate.

- Critique your partner's solution, including explanation, and if there is anything that should be adjusted.

Partners solution looks optimum

Part 4:

Please write a 200 word reflection documenting your process from assignment 1, and your presentation and review experience with your partner at the bottom of the Jupyter Notebook under a new heading "Reflection." Again, export this Notebook as pdf.

Reflection

The algorithm works by:

1. Using a set to track seen values: As we iterate through the list, we keep track of numbers we've already encountered.
2. Early return on first duplicate: The moment we find a number that's already in our seen set, we return it immediately. This ensures we get the first duplicate in order of appearance.
3. Linear time complexity: The solution runs in O(n) time where n is the length of the list, since set lookups and insertions are O(1) on average.
4. Space complexity: O(k) where k is the number of unique elements encountered before finding the first duplicate.

The key insight is that we need to return the first value that appears for the second time, not necessarily the value that appears most frequently. The algorithm stops as soon as it encounters any duplicate, ensuring we get the earliest one in the list.

Evaluation Criteria

We are looking for the similar points as Assignment 1

- Problem is accurately stated
- New example is correct and easily understandable

- Correctness, time, and space complexity of the coding solution
- Clarity in explaining why the solution works, its time and space complexity
- Quality of critique of your partner's assignment, if necessary

Submission Information

 Please review our Assignment Submission Guide for detailed instructions on how to format, branch, and submit your work. Following these guidelines is crucial for your submissions to be evaluated correctly.

Submission Parameters:

- Submission Due Date: `HH:MM AM/PM - DD/MM/YYYY`
- The branch name for your repo should be: `assignment-2`
- What to submit for this assignment:
  - This Jupyter Notebook (assignment_2.ipynb) should be populated and should be the only change in your pull request.
- What the pull request link should look like for this assignment: `https://github.com/<your_github_username>/algorithms_and_data_structures/pull/<pr_id>`
  - Open a private window in your browser. Copy and paste the link to your pull request into the address bar. Make sure you can see your pull request properly. This helps the technical facilitator and learning support staff review your submission easily.

Checklist:

- ☐ Created a branch with the correct naming convention.
- ☐ Ensured that the repository is public.
- ☐ Reviewed the PR description guidelines and adhered to them.
- ☐ Verify that the link is accessible in a private browser window.

If you encounter any difficulties or have questions, please don't hesitate to reach out to our team via our Slack at `#cohort-6-help`. Our Technical Facilitators and Learning Support staff are here to help you navigate any challenges.