# assignment-2

November 15, 2024

## 1 Practice Interview

### 1.1 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.

### 1.2 Group Size

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

### 1.3 Part 1:

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

**Partnering and Working with Marcio Sugar.**

Marcio Sugar Assignment - https://github.com/msugar/algorithms_and_data_structures/pulls

Jyoti Narang Assignment - https://github.com/drop2jyoti/algorithms_and_data_structures/pull/1

### 1.4 Part 2:

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

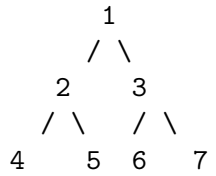
- Paraphrase the problem in your own words.

```
Find all the possible routes from a given root to its leafs
```

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

Example: input - [1, 2, 3, 4, 5, 6, 7, 8, 9, None, None, 10]

output - [[1, 2, 4, 8], [1, 2, 4, 9], [1, 2, 5], [1, 3, 6, 10], [1, 3, 7]]



```
  / \      /
 8   9    10
```

The function would:

- Start at 1, add it to path: [1]
- Go left to 2: [1,2]
- Go left to 4: [1,2,4]
- Go left to 8: [1,2,4,8]

**Found leaf! Save this path**

- Step back: [1,2,4]
- Go right to 4: [1,2,4,9]

**Found leaf! Save this path**

- step back: [1,2]
- Go right to 2: [1,2,5]

**Found leaf! Save this path**

- step back: [1]
- Go left to 3: [1,3]
- Go left to 6: [1,3,6]
- Go left to 10: [1,3,6,10]

**Found leaf! Save this path**

- step back: [1,3,6]
- No right path at 6, step back: [1,3]
- Go right to 3: [1,3,7]

**Found leaf! Save this path**

Done! Return [[1, 2, 4, 8], [1, 2, 4, 9], [1, 2, 5], [1, 3, 6, 10], [1, 3, 7]]

How is Marcio code working -

The provided code implements a binary tree structure and includes functionality to construct a binary tree from a breadth-first traversal array as well as to retrieve all root-to-leaf paths using depth-first search. The code has -

`TreeNode Class: Represents a node in a binary tree, with attributes for its value and left/rig`

`build_from_breadth_first_traversal: A class method to construct a binary tree from a list repr`

`bt_path: A function that retrieves all root-to-leaf paths using depth-first search, storing th`
`The function bt_path takes an optional TreeNode as input (the root of the binary tree) and retu`
`Empty Tree Check: If the tree is empty (i.e., the root is None), it returns an empty list.`

Initialization and Invocation: The all_paths list is initialized to store all complete paths. T

visit function - The visit function is defined within bt_path as a nested function. It takes th

- Copy the solution your partner wrote.

[164]:
```python
# Your answer here
# In a real interview, the code in this cell could be given or assumed.
# I'm conding it here to make it possible to test the code with unit tests␣
↪later.

from typing import List, Optional
from typing_extensions import Self

# Definition for a binary tree node.
class TreeNode:
    """
    A binary tree node with integer value and optional left and right children.

    Attributes:
        val (int): The node's value
        left (TreeNode | None): Left child node
        right (TreeNode | None): Right child node
    """
    def __init__(self, val: int = 0, left: Optional['TreeNode'] = None, right:␣
↪Optional['TreeNode'] = None) -> Self:
        self.val = val
        self.left = left
        self.right = right

    def is_leaf(self) -> bool:
        """Returns True if the node has no children."""
        return self.left is None and self.right is None

    def __repr__(self):
        """String representation of the node."""
        return str(self.val)

    @classmethod
    def build_from_breadth_first_traversal(cls, bfs_traversal:␣
↪List[Optional[int]]) -> Optional['TreeNode']:
        """
        Builds a binary tree from a level-order traversal array.
        None values in the array represent empty nodes.

        Args:
            bfs_traversal: List of integers or None representing the tree in␣
↪level-order
```

```
        Returns:
            The root node of the constructed tree, or None if input is empty

        Raises:
            ValueError: If the input array is invalid for tree construction
        """
        if not bfs_traversal:
            return None

        # Validate input
        if not all(isinstance(x, (int, type(None))) for x in bfs_traversal):
            raise ValueError("Array must contain only integers or None values")

        nodes = []
        for index, val in enumerate(bfs_traversal):
            if val is not None:
                node = cls(val)
                nodes.append(node)
                if index > 0:
                    parent_index = (index - 1) // 2
                    if index % 2 == 1:
                        nodes[parent_index].left = node
                    else:
                        nodes[parent_index].right = node
            else:
                nodes.append(None)

        return nodes[0] if len(nodes) > 0 else None
```

[165]:
```
def bt_path(root: Optional[TreeNode]) -> List[List[int]]:
    """
    Returns all root-to-leaf paths in a binary tree.

    Args:
        root: Root node of the binary tree

    Returns:
        List of all paths from root to leaves, where each path is a list of␣
 ↪node values
    """
    if not root:
        return []

    def visit(node: TreeNode, current_path: List[int], all_paths:␣
 ↪List[List[int]]) -> None:
        """
```

```
        DFS helper function to traverse the tree and collect paths.

        Args:
            node: Current node being visited
            current_path: Path from root to current node
            all_paths: Collection of all complete root-to-leaf paths
        """
        current_path.append(node.val)

        if node.is_leaf():
            all_paths.append(current_path.copy())
        else:
            if node.left:
                visit(node.left, current_path, all_paths)
            if node.right:
                visit(node.right, current_path, all_paths)

        current_path.pop()

    all_paths = []
    visit(root, [], all_paths)
    return all_paths
```

[166]:
```
# This is just to impress the interviewer. :-)

import unittest

class TestBinaryTreePathsToLeaves(unittest.TestCase):
    """Test cases for binary tree path finding algorithm."""

    def test_example_1(self):
        """Test Example 1."""
        input = [1, 2, 2, 3, 5, 6, 7]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]
        self.assertCountEqual(bt_path(root), expected)

    def test_example_2(self):
        """Test Example 2."""
        input = [10, 9, 7, 8]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[10, 7], [10, 9, 8]]
        self.assertCountEqual(bt_path(root), expected)

    def test_empty_tree(self):
        """Test handling of empty tree."""
        root = None
```

```
        self.assertEqual(bt_path(root), [])

    def test_single_node(self):
        """Test tree with only root node."""
        root = TreeNode(1)
        self.assertEqual(bt_path(root), [[1]])

    def test_invalid_input(self):
        """Test invalid input handling."""
        with self.assertRaises(ValueError):
            TreeNode.build_from_breadth_first_traversal(['invalid'])

    def test_none_values(self):
        """Test tree construction with None values."""
        input = [1, None, 3]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 3]]
        self.assertEqual(bt_path(root), expected)

    def test_balanced_tree(self):
        """Test balanced tree with multiple paths."""
        input = [1, 2, 3, 4, 5, 6, 7]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 4], [1, 2, 5], [1, 3, 6], [1, 3, 7]]
        self.assertCountEqual(bt_path(root), expected)

    def test_unbalanced_tree(self):
        """Test unbalanced tree."""
        input = [1, 2, None, 3, None, None, None, 4, 5]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 3, 4], [1, 2, 3, 5]]
        self.assertEqual(bt_path(root), expected)

    # def test_skewed_tree(self):
    #     input = [1, 2, 3, 4, 5, 6, 7, 8, 9, None, None, 10]
    #     root = TreeNode.build_from_breadth_first_traversal(input)
    #     expected = [[1, 2, 3, 4], [1, 2, 3, 5]]
    #     self.assertEqual(bt_path(root), expected)
```

[167]:
```
def run_tests():
    # Create a test suite
    suite = unittest.TestLoader().
 ↪loadTestsFromTestCase(TestBinaryTreePathsToLeaves)

    # Run the tests
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
```

```
run_tests()
```

```
test_balanced_tree (__main__.TestBinaryTreePathsToLeaves)
Test balanced tree with multiple paths. … ok
test_empty_tree (__main__.TestBinaryTreePathsToLeaves)
Test handling of empty tree. … ok
test_example_1 (__main__.TestBinaryTreePathsToLeaves)
Test Example 1. … ok
test_example_2 (__main__.TestBinaryTreePathsToLeaves)
Test Example 2. … ok
test_invalid_input (__main__.TestBinaryTreePathsToLeaves)
Test invalid input handling. … ok
test_none_values (__main__.TestBinaryTreePathsToLeaves)
Test tree construction with None values. … ok
test_single_node (__main__.TestBinaryTreePathsToLeaves)
Test tree with only root node. … ok
test_unbalanced_tree (__main__.TestBinaryTreePathsToLeaves)
Test unbalanced tree. … ok

----------------------------------------------------------------------
Ran 8 tests in 0.004s

OK
```

- Explain why their solution works in your own words.

### 1.4.1 Your answer here

The code works because it looks into each possible path from the root to each one of the roots, while at the same time keeping track of the paths so they don't get missed. The code effectively constructs a binary tree and retrieves all paths from the root to its leaves using a depth-first search approach. It is well-structured, with clear separation of concerns, making it easy to understand and maintain.

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

Time and Space Complexity Analysis For the bt_path function, we can analyze the time and space complexity as follows:

Time Complexity

Traversal:

The function traverses each node in the binary tree exactly once. In the worst case, if the tree is a complete binary tree with n nodes, we will visit each node once, leading to a time complexity of O(n).

Path Construction:

For each node, the function constructs paths from the root to the leaf. When a path is found, it copies the current path, which takes O(h) time, where h is the height of the tree. In the worst

case (for unbalanced trees), h can be equal to n for a skewed tree. However, since each node will be processed once, and the height of the tree can be at most n in the worst case, the overall time complexity remains O(n). Thus, the overall time complexity for the function is O(n).

Space Complexity

Call Stack:

The recursive function uses the call stack to keep track of function calls. In the worst case, the depth of the recursion can be equal to the height of the tree, which can be O(n) for a skewed tree.

Path Storage:

Each path stored in the all_paths list could take at most O(h) space, where h is the height of the tree. In the worst case, this can also be as large as O(n) for a skewed tree. Combining these factors, the overall space complexity is dominated by the storage of paths and the call stack, leading to a total space complexity of O(n).

Summary Time Complexity: O(n) Space Complexity: O(n) This analysis shows that the function is efficient in both time and space for constructing all root-to-leaf paths in a binary tree.

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

### 1.5 Code review comments -

**1. Use of Optional in Type Hints** You are using `Optional['TreeNode']` correctly, allowing for the possibility of `None`. However, be aware that while using string annotations (like `'TreeNode'`), it requires the `__future__` import in Python versions before 3.7. In your current setup, it's safe to use.

**2. The `is_leaf` method** This method is well-defined and should work correctly. No issues here.

**3. The `build_from_breadth_first_traversal` method** This method looks good, but just as a suggestion for clarity: - The `ValueError` message can be more descriptive. For example, you might want to specify what kind of values were found if they are not integers or `None`.

**4. The `bt_path` function** The function is defined correctly, and the logic within appears sound. However, to enhance readability and performance: - You can use list comprehension for the `visit` function to reduce the number of explicit checks for `node.left` and `node.right`.

**5. Error Handling** Currently, your code raises a `ValueError` for input validation, which is good. Ensure that your tests cover various edge cases, such as an empty list, a list with invalid types, etc.

**6. General Code Optimization**

- You may want to use `deque` from the `collections` module for your `current_path` if you expect a significant depth in the binary tree to avoid potential performance penalties with list append/pop operations due to resizing.

**Final Optimized Code using dequeus** Explanation: - We use a deque to handle our traversal. We start by adding the root node and its initial path to the deque. - We pop elements from the left (the front of the deque) for processing, checking if each node is a leaf. - If it's a leaf, we add the current path to the all_paths list. - If the node has children, we append them to the deque along with the updated path. - This approach maintains the efficiency of the traversal while utilizing the deque for better performance in terms of append and pop operations.

[168]:
```
from collections import deque
from typing import List, Optional

def bt_path_deque(root: Optional[TreeNode]) -> List[List[int]]:
    """
    Returns all root-to-leaf paths in a binary tree using a deque for iterative␣
 ↪traversal.

    Args:
        root: Root node of the binary tree

    Returns:
        List of all paths from root to leaves, where each path is a list of␣
 ↪node values
    """
    if not root:
        return []

    all_paths = []
    queue = deque([(root, [root.val])])  # Initialize deque with the root node␣
 ↪and its path

    while queue:
        node, path = queue.popleft()  # Dequeue the front element

        if node.is_leaf():
            all_paths.append(path)
        if node.right:
            queue.append((node.right, path + [node.right.val]))  # Enqueue␣
 ↪right child
        if node.left:
            queue.append((node.left, path + [node.left.val]))  # Enqueue left␣
 ↪child

    return all_paths
```

[169]:
```
# This is just to impress the interviewer. :-)

import unittest
```

```python
class TestBinaryTreePathsToLeavesDequeues(unittest.TestCase):
    """Test cases for binary tree path finding algorithm."""

    def test_example_1(self):
        """Test Example 1."""
        input = [1, 2, 2, 3, 5, 6, 7]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]
        self.assertCountEqual(bt_path(root), expected)

    def test_example_2(self):
        """Test Example 2."""
        input = [10, 9, 7, 8]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[10, 7], [10, 9, 8]]
        self.assertCountEqual(bt_path(root), expected)

    def test_empty_tree(self):
        """Test handling of empty tree."""
        root = None
        self.assertEqual(bt_path(root), [])

    def test_single_node(self):
        """Test tree with only root node."""
        root = TreeNode(1)
        self.assertEqual(bt_path(root), [[1]])

    def test_invalid_input(self):
        """Test invalid input handling."""
        with self.assertRaises(ValueError):
            TreeNode.build_from_breadth_first_traversal(['invalid'])

    def test_none_values(self):
        """Test tree construction with None values."""
        input = [1, None, 3]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 3]]
        self.assertEqual(bt_path(root), expected)

    def test_balanced_tree(self):
        """Test balanced tree with multiple paths."""
        input = [1, 2, 3, 4, 5, 6, 7]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 4], [1, 2, 5], [1, 3, 6], [1, 3, 7]]
        self.assertCountEqual(bt_path(root), expected)

    def test_unbalanced_tree(self):
```

```
        """Test unbalanced tree."""
        input = [1, 2, None, 3, None, None, None, 4, 5]
        root = TreeNode.build_from_breadth_first_traversal(input)
        expected = [[1, 2, 3, 4], [1, 2, 3, 5]]
        self.assertEqual(bt_path(root), expected)
```

[170]:
```
def run_tests():
    # Create a test suite
    suite = unittest.TestLoader().
 ↪loadTestsFromTestCase(TestBinaryTreePathsToLeavesDequeues)

    # Run the tests
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)

run_tests()
```

```
test_balanced_tree (__main__.TestBinaryTreePathsToLeavesDequeues)
Test balanced tree with multiple paths. … ok
test_empty_tree (__main__.TestBinaryTreePathsToLeavesDequeues)
Test handling of empty tree. … ok
test_example_1 (__main__.TestBinaryTreePathsToLeavesDequeues)
Test Example 1. … ok
test_example_2 (__main__.TestBinaryTreePathsToLeavesDequeues)
Test Example 2. … ok
test_invalid_input (__main__.TestBinaryTreePathsToLeavesDequeues)
Test invalid input handling. … ok
test_none_values (__main__.TestBinaryTreePathsToLeavesDequeues)
Test tree construction with None values. … ok
test_single_node (__main__.TestBinaryTreePathsToLeavesDequeues)
Test tree with only root node. … ok
test_unbalanced_tree (__main__.TestBinaryTreePathsToLeavesDequeues)
Test unbalanced tree. … ok

----------------------------------------------------------------------
Ran 8 tests in 0.004s

OK
```

### Time and Space Complexity Analysis

For the provided code (bt_path_deque function), we can analyze the time and space complexity as follows:

Time Complexity

Traversal:

The function traverses each node in the binary tree exactly once. In the worst case, if the tree is a complete binary tree with n nodes, we will visit each node once, leading to a time complexity of

O(n).

Path Construction:

For each node, we create a new path list that includes the current node's value. This involves copying the current path, which takes O(h) time, where h is the height of the tree. In the worst case (for unbalanced trees), h can be equal to n for a skewed tree. However, since each node will be processed once, and the height of the tree can be at most n in the worst case, the overall time complexity remains O(n). Thus, the overall time complexity for the function is O(n).

Space Complexity

Queue Storage:

The space used by the deque can grow up to the maximum number of nodes at any level of the tree. In the worst case (for a complete binary tree), the maximum number of nodes in the queue can be O(n/2), which simplifies to O(n).

Path Storage:

Each path stored in the queue could take at most O(h) space, where h is the height of the tree. In the worst case, this can also be as large as O(n) for a skewed tree. Combining these factors, the overall space complexity is dominated by the storage of nodes in the deque, leading to a total space complexity of O(n).

Summary Time Complexity: O(n) Space Complexity: O(n) This analysis shows that the function is efficient in both time and space for constructing all root-to-leaf paths in a binary tree.

## 1.6 Part 3:

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.

### 1.6.1 Reflection

During the assignment, I tackled the problem of finding missing numbers in a specified range using Python. I employed set operations to efficiently identify and collect missing integers between 0 and the maximum value in the given list. The solution utilized a set to store the numbers from the list and compared it against a full set of expected numbers to find missing elements. The benefit of using sets as opposed to a list has some advantage ie it has a highly optimized method for checking whether a specific element is contained in the set. This solution works for both sorted and unsorted list.

In reviewing my classmate's code for finding paths from a root to leaf nodes in a binary tree, I focused on clarity and efficiency. The solution effectively used depth-first search (DFS) to traverse the tree and collect paths, ensuring paths were correctly copied and stored without shared references. Recommendations included using Dequeues for better performance

Overall, the assignment and code review underscored the importance of clarity, efficiency, and thoughtful design in algorithmic solutions, reinforcing skills in problem-solving and code optimization in Python.

### 1.7 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

### 1.8 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.

#### 1.8.1 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-3-help`. Our Technical Facilitators and Learning Support staff are here to help you navigate any challenges.