# assignment_2

July 21, 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.

### 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.

[ ]:
```
# Your answer here
# my partner's github is https://github.com/cavengal/
  ↪algorithms_and_data_structures
# He did Question 2

# In my own words, the question is:
# We would like to return all the paths in a tree from the starting point (root)␣
  ↪to the ends of the branches (leaves) in any order.
```

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

[ ]:
```
# Your answer here
# new example:
#       1
#      / \
#     2   3
#    / \   \
#  4   5   6
# Input: root = [1, 2, 4, 5, 3, 6]
# Output: [[1, 2, 4], [1, 2, 5], [1, 3, 6]]

# example from my partner:
# Input: root = [8, 3, 10, 1, 6, 9, 12]
# Output: [[8, 3, 1], [8, 3, 6], [8, 10, 9], [8, 10, 12]]
# This is a full binary tree, every node has 2 or 0 leave
# So this is similar to the given example one in the question, root is 8
# 8 has two children: left is 3 and right is 10
# 3 has two children: left is 1, right is 6
# 10 has two children: left is 9 and right is 12
# We return all 4 paths
```

- Copy the solution your partner wrote.

[ ]:
```
# Your answer here
# credit from Carlos's github
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
# Defining the Main function (named pathfunction). It initializes the␣
 ↪Depth-First Search and returns the result.
def path_function(root: TreeNode) -> List[List[int]]:
    #Defining Helpe function for Recursive DFS: It tracks the current path and␣
 ↪adds it to the result when a leaf node is reached.
    def dfs(node, path, paths):
        #Base Case
        if not node:
            return
        # Add the current node to the path
        path.append(node.val)
        # If it's a leaf node, add the path to the result
        if not node.left and not node.right:
            paths.append(list(path))
        else:
```

```
            # Continue DFS on left and right child
            dfs(node.left, path, paths)
            dfs(node.right, path, paths)
        # Backtrack to explore other paths
        path.pop()

    paths = []
    dfs(root, [], paths)
    return paths
```

[ ]:
```
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)
path_function(root)
```

[ ]:
```
[[1, 2, 4], [1, 2, 5], [1, 3, 6]]
```

[ ]:
```
path_function(root=None)
```

[ ]:
```
[]
```

- Explain why their solution works in your own words.

[ ]:
```
# Your answer here
# He use recursive method to expand the path, until the node has no children, from the root to the end.
# and the helper function dfs recursively explores each path from the root to the leaves.
# Base Case: If the current node is None, it returns.
# Path Tracking: Adds the current node's value to the path.
# Leaf Node Check: If the node is a leaf (no left and right children), it adds the current path to the results.
# Recursive Calls: Continues the DFS on the left and right children.
# Backtracking: Removes the current node's value from the path to explore other paths.
```

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

[ ]:
```
# Your answer here
# The time complexity:
# The DFS function visits each node exactly once. In a binary tree, it's O(n)
# At most there are n nodes, and append the path takes at most O(n) to the return list, whch is O(n^2)
# Therefore, the the overall complexity is O(n^2)
```

```
# Space Complexity
# The maximum depth of the recursive call is the depth of the tree, so is O(n)
# In the worst case, we need to store O(n) paths, each of which could be O(n) in␣
 ↪length, total is O(n^2)
# Therefore, the space complecity is O(n^2)
```

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

[ ]:
```
# Your answer here
# The provided solution is generally efficient and correctly solves the problem.
# The use of a helper function with clear base case, path tracking, and␣
 ↪backtracking logic makes the code easy to understand and maintain.

#Adjustments:
# I would like to handle the case where the root is None at the beginning of the␣
 ↪path_function.

def path_function(root: TreeNode) -> List[List[int]]:
    def dfs(node, path, paths):
        if not node:
            return
        path.append(node.val)
        if not node.left and not node.right:
            paths.append(path[:])
        else:
            dfs(node.left, path, paths)
            dfs(node.right, path, paths)
        path.pop()

    if not root: # I only add the edge case here
        return []

    paths = []
    dfs(root, [], paths)
    return paths
```

[ ]:
```
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)
path_function(root)
```

[ ]:
```
[[1, 2, 4], [1, 2, 5], [1, 3, 6]]
```

[ ]:
```
path_function(root=None)
```

[ ]:
```
[]
```

### 1.5 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.

#### 1.5.1 Reflection

[ ]:
```
# Your answer here
# Plz see below
```

My process began with Question 3 on finding missing numbers in a range. Initially, I provided a straightforward Python solution only using List and explained the time complexity.
Sorting the list first was considered to improve the time complexity, but it did not contribute to saving time due to the inherent complexities involved in sorting and subsequent operations.
Upon further discussion and analysis, I realized that we could improve it by leveraging sets for faster membership checks. Using a set instead of a list reduces the time complexity.

Next, I review my partner's solution of Question 2: solving the tree paths question. I realize that it's important that we could rephase CS questions concisly with clear explanation.
I then reviewed his provided code solution, ensuring it worked correctly and discussed its time and space complexity in detail. I critiqued the solution, suggested handling edge cases at the begining, and made slight improvements to ensure robustness.

Throughout this process, I realized the importance of optimizing both time and space complexity, ensuring that explanations and solutions were accessible and efficient. Reflecting on and reviewing both problems highlighted the value of iterative improvement, clear communication, and thorough analysis in effective problem-solving.

### 1.6 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

## 1.7 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.

### 1.7.1 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.