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.

In [8]:
```python
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 = "Jayendran"
result = hash_to_range(input_string)
print(result)
```

```
1
```

▸ Question 1

Starter Code for Question 1

In [9]:
```python
# 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:
        if not root:
                return -1

        # Use a set to track seen values and a queue for breadth-first trave
        seen = set()
        # Start with the root node in the queue
        queue = [root]

        while queue:
                # Get the next node from the queue
                node = queue.pop(0)

                # Check if the current node's value has been seen before
                if node.val in seen:
                        return node.val

                # If not, add the value to the seen set
                seen.add(node.val)

                # Add the left and right children to the queue if they exist
                if node.left:
                        queue.append(node.left)

                if node.right:
                        queue.append(node.right)

        # If we finish processing all nodes without finding duplicates, retu
        return -1

# Constructing the binary tree from the list representation
def construct_tree(nodes):
        # If the list is empty, return None
        if not nodes:
                return None

        # Create the root node
        root = TreeNode(nodes[0])

        # Use a queue to keep track of nodes to which we will attach childre
        queue = [root]
        # Start from the second element in the list since the first is the r
        i = 1
        # Loop through the list and construct the tree
        while i < len(nodes) and queue:
                # Get the next node from the queue
                node = queue.pop(0)

                # Attach the left child if it exists
                if i < len(nodes) and nodes[i] is not None:
                        node.left = TreeNode(nodes[i])
                        queue.append(node.left)
                i += 1

                # Attach the right child if it exists
                if i < len(nodes) and nodes[i] is not None:
                        node.right = TreeNode(nodes[i])
                        queue.append(node.right)
                i += 1

        # After constructing the tree, return the root node
        return root
```

In [10]:
```
# Example 1
root = [1, 2, 2, 3, 5, 6, 7]

tree_root = construct_tree(root)

if tree_root is not None:
        print(is_duplicate(tree_root)) # The output should be 2 since it is
```

```
2
```

In [11]:
```
# Example 2
root = [1, 10, 2, 3, 10, 12, 12]

tree_root = construct_tree(root)

if tree_root is not None:
        print(is_duplicate(tree_root)) # The output should be 10 since it is
```

```
10
```

In [12]:
```
# Example 3
root = [10, 9, 8, 7]

tree_root = construct_tree(root)

if tree_root is not None:
        print(is_duplicate(tree_root)) # The output should be -1 since there
```

```
-1
```

▸ Question 2

Starter Code for Question 2

In [ ]:
```
# 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]]:
  # TODO
```

▸ Question 3

Starter Code for Question 3

In [ ]:
```
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.

We are given an input string containing only the characters '(', ')', '{', '}', '[', and ']'. We need to evaluate and conclude if the input string is a valid bracket sequence, consisting of a correctly matched bracket sequence. The string is valid if open brackets are closed by the same type of brackets, and open brackets are closed in the correct order.

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

New example input string: "{}" Expected output: True

Partner example of input string: "([]{})" Expected output: True

Algorithm logic:

Define pairs for braces/brackets as a dictionary (with keys as closing braces and values as matching opening braces) for verification.

Define a stack (as a list, that is initially empty) to push opening brace characters as we traverse the input string.

Loop through the characters in the input string.

For the character in each pass of the for loop

```
- if the character is a closing brace

        - if the stack is empty, or the last character in the
stack is a mismatch with the current closing
          brace, return False

        - else, a matching opening brace was found, pop the
stack

- else (the character being an open brace), append the
character to the stack
```

Return True, if the length of the stack is zero (which imples all braces have been matched and accounted for), else, return False.

Trace/walkthrough of the for loop for the example the partner made - "([]{})":

Pass 1: '(' is an open brace, so push to stack

Pass 2: '[' is an open brace, so push to stack

Pass 3: ']' is a closing brace, so pop the stack, since the last character in the stack '[' matches the closing brace

Pass 4: '{' is an open brace, so push to stack

Pass 5: '}' is a closing brace, so pop the stack, since the last character in the stack '{' matches the closing brace

Pass 6: ')' is a closing brace, so pop the stack, since the last character in the stack '(' matches the closing brace

Pass 7: Since we have traversed all the characters in the string, and the stack is empty, return True to signify, a valid brace matched string

- Copy the solution your partner wrote.

In [ ]:
```python
def is_valid_brackets(s: str) -> bool:
    pairs = {')': '(', '}': '{', ']': '['}
    stack = []

    for ch in s:
        if ch in pairs:  # closing bracket
            if not stack or stack[-1] != pairs[ch]:
                return False
            stack.pop()
        else:  # opening bracket
            # since input is only bracket chars, we can push directly
            stack.append(ch)
    return len(stack) == 0
```

- Explain why their solution works in your own words.

The solution uses a dictionary to map closing brackets to their corresponding opening brackets. It iterates through each character in the input string, using a stack to keep track of opening brackets. When it encounters a closing bracket, it checks if the top of the stack matches the expected opening bracket. If it doesn't match or if the stack is empty when a closing bracket is encountered, the function returns False. Finally, it checks if the stack is empty at the end, which would indicate that all brackets were properly closed and nested.

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

Time complexity: O(n)

- We iterate through the list at most once, leading to O(n)
- The dictionary lookup for a matching brace is O(1) on average
- Popping off the stack, or appending to the same is O(1)

Hence, O(n) being the largest time complexity.

Space complexity: O(n)

- Storing all the opening and closing brace characters in a dictionary is O(n)
- Storing all opening braces in an input string is O(n) in the worst case

Hence, O(n) being the largest space complexity.

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

This is a solution that is quite efficient and runs in O(n) time complexity, where n is the length of the input string. The space complexity is also O(n) in the worst case when all characters are opening brackets.

As such, the solution is already robust and most optimal, and I cannot fathom any other improvement.

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

Having gone through Assignment 1 and reviewing my partner's submission, I learned how to be efficient with algorithm design and choose the right data structure. The brace matching problem using a dictionary to define matching pairs is what I would have exactly come up with as well, only that the implementation of my partner Haimeng Wang was quite efficient and tight in code with a worst case performance of $O(n)$. The implementation was far better than coming up with nested loops that could cause the solution to have a worst case time complexity of $O(n^2)$.

Haimeng Wang's solution was clean, adequately commented, and demonstrated a very good understanding of data structures. Haimen explained the algorithm well and included multiple tests to handle both valid and invalid strings.

The reivew process emphasizes the importance of commenting and documenting code, so a reader can understand what the author has coded without difficulty.

I also learned that defining both time and space complexity helps comes up better designs and implementation in any computer programming.

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.

In [ ]:

```
```