Assignment 2 - Code Review

Partner Review for Assignment 1

1. Paraphrase the problem in your own words

> The goal of this problem is to determine whether a string that contains only brackets — specifically '(', ')', '{', '}', '[', and ']' — forms a valid bracket sequence. A sequence is valid when each opening bracket is properly closed by the corresponding type of closing bracket, and when the brackets are closed in the correct order. For instance, the sequence "({[]})" is valid because each bracket opens and closes correctly in a properly nested structure. On the other hand, "({[})]" is invalid because the brackets are not closed in the right order. The challenge lies in tracking the order and types of brackets to ensure the structure is balanced and properly nested.

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

New example:

Input: `"({[]}){}"`
Output: `True`

This is a valid bracket sequence.

- `(` opens → then `{` opens → then `[` opens.
- Then `]` closes the `[`, `}` closes the `{`, and `)` closes the `(`.
- After that, `{` opens and is immediately closed by `}`.
- All brackets are properly matched and closed in the correct order, so the sequence is valid.

Walkthrough of Sahil's example:

Input: `"({{}})"`
Output: `True`

Step-by-step:

1. `(` is an opening bracket → push to stack → stack = [`(`]
2. `{` is opening → push → stack = [`(` , `{`]

3. `{` is opening again → push → stack = [`(` , `{` , `{`]
4. `}` is closing → matches `{` on top → pop → stack = [`(` , `{`]
5. `}` is closing → matches `{` on top → pop → stack = [`(`]
6. `)` is closing → matches `(` on top → pop → stack = `[]`

Since the stack is empty at the end, all brackets are correctly matched and closed in order. Therefore, the output is `True` .

3. Copy the solution your partner wrote.

In [9]:
```python
def is_valid_brackets(s: str) -> bool:
    stack = []
    matching = {')': '(', ']': '[', '}': '{'}

    for char in s:
        if char in matching.values():
            stack.append(char)
        elif char in matching:
            if not stack or stack[-1] != matching[char]:
                return False
            stack.pop()
        else:
            return False

    return not stack
```

4. Explain why their solution works in your own words.

> The solution works because it uses a stack to keep track of opening brackets. As the function iterates through each character in the string, it pushes opening brackets onto the stack. When it encounters a closing bracket, it checks if the stack is not empty and whether the top element of the stack matches the corresponding opening bracket. If it does, it removes the top element from the stack; otherwise, it returns `False` immediately. At the end, if the stack is empty, it means that all brackets were properly matched and closed in the correct order, so the function returns `True` . This approach ensures that the string is both balanced and correctly nested.

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

> The time complexity of this solution is **O(n)**, where *n* is the length of the input string. This is because each character in the string is processed exactly once in a single pass through the loop.

> The space complexity is also **O(n)** in the worst case. This happens when the string contains only opening brackets (e.g., `"(([[{{"`), which are all pushed onto the

> stack without being popped. In such cases, the stack will grow to a size proportional to the length of the input. In summary:
>
> - **Time complexity:** O(n) — one pass through the string.
> - **Space complexity:** O(n) — for the stack used to store opening brackets.

6. Critique your partner's solution

Overall, the solution is clear and effective. It correctly uses a stack to validate the sequence of brackets, and the logic is well-structured to handle most cases, including mismatched or unbalanced inputs.

One small suggestion for improvement would be to make the code slightly more robust by removing the final `else` block. Currently, if a character is not a bracket, the function returns `False` immediately. However, based on the problem description, the input is expected to contain only valid brackets, so that case may be unnecessary. Removing the `else` clause could simplify the logic without affecting correctness.

Another minor point is to consider adding comments to make the purpose of each section more readable — especially for someone new reviewing the code.

In summary: the implementation is correct and efficient, but could benefit from small readability improvements.

Reflection

This was my first time participating in a peer code review activity, and I found it to be a very valuable experience. It helped me realize how communication can happen through code, even without speaking directly with someone. By reading my partner's solution, I was able to understand their logic and how they approached the problem — it felt like I was getting a glimpse into their way of thinking.

One key insight I gained is how much the readability of code matters. Although I was able to follow the logic, I now have a much clearer understanding of why adding comments is so important. Comments not only make the purpose of the code easier to understand for others, but also help the author maintain clarity when revisiting their own work later on.

This assignment also reinforced my understanding of how algorithms work and gave me an opportunity to practice explaining code in a structured way. Overall, it was a great opportunity to learn both from writing and reviewing code, and I appreciate how this kind of collaborative activity improves both technical and communication skills.