Paraphrase

Given a string that only contains bracket characters like (), [], {}, determine wether the string is a valid bracket sequence. A string is valid when every opening bracket is closed by the same type of bracket and brackets close in the correct nested order. Meaning if you open a curly brace first and then a square bracket, the square bracket has to close before the curly brace due to the fact that they need to be properly nested.

New example + walkthrough

New example: Input: s = "{[()]}" Output: True The curly brace opens, then the square bracket, then the parenthesis - each one closes in the reverse order they were opened which makes it valid.

Walkthrough of Jesse's Example 1: "((()))"

- Read (> push onto stack > stack: ['(']
- Read (> push > stack: ['(', '(']
- Read (> push > stack: ['(', '(', '(']
- Read) > pop (, it matches) > stack: ['(', '(']
- Read) > pop (, it matches) > stack: ['(']
- Read) > pop (, it matches) > stack: []
- Stack is empty at the end so return True

In [ ]:
```python
def is_valid_brackets(s: str) -> bool:
    match = {')': '(', ']': '[', '}': '{'}
    openers = set(match.values())
    stack = []
    for ch in s:
        if ch in openers:
            stack.append(ch)
        elif ch in match:
            if not stack or stack.pop() != match[ch]:
                return False
        else:
            # If input may contain other characters, treat as invalid
            return False
    return not stack
    pass
```

Explain why it works

The solution uses a stack which is a last-in first-out data structure - this is actually a perfect fit for bracket matching because the most recently opened bracket is the one that needs to close first. As we read the string left to right, every opening bracket gets pushed onto the stack. When we hit a closing bracket, we pop the top of the stack and check if it's the matching opener - if not, the string is invalid. If the stack is empty when we encounter a closer, that means theres no opener to match so it's also invalid. At the end, if the stack is empty it means every opener was matched and closed correctly. If anything remains on the stack, there are unclosed openers.

Time and space complexity

Time complexity: O(n), where n is the length of the string. We loop through each character exactly once, and each stack push/pop operation is O(1), so the total work scales linearly with the string length. Best case is O(1) if the first character is already a mismatch.

From a space complexity perspective, it's O(n) in the worst case. For example, if the string is all opening brackets like ((((((, every character gets pushed onto the stack meaning the memory required grows proportionally with the input size. In the best case space is O(1) because we return immediatly without adding anything to the stack.

Critique

Jesse's code is correct and handles all the test cases properly. The use of a dictionary for bracket matching and a set for the openers is clean and efficent. However there are a few things that should be adjusted:

The examples are weak - both examples only use parentheses and both return True. They don't demonstrate understanding of mixed bracket types like ([{}]) or invalid cases like "(]". At least one example should use multiple bracket types and one should show a False case.

The "explain why it works" section is missing an actual explanation. Jesse only wrote assert statements that prove the code passes test cases but theres no description of how the stack algorithm actually works. The asserts show that it works, not why.

Part 3 - Reflection

Working through Assignment 1, I was assigned the "Move All Zeros to End" problem. My initial instinct was to use pop() and append() but I quickly realized that modifying a list while iterating over it causes elements to shift and get skipped. This pushed me toward

the two pointer approach, where one pointer reads through the list and the other tracks where the next non-zero should go. Understanding why the naive approach fails was just as valuable as finding the elegent solution.

Reviewing Jesse's bracket validation solution gave me a different perspective. The stack data structure was a natural fit for that problem in the same way that the two pointer technique fit mine - choosing the right ADT simplifies everything. Walking through Jesse's code step by step helped me understand how a stack mirrors the nesting structure of brackets.

One thing I noticed during the review is how important clear explanations are. Jesse's code was correct and well written but the explanation section used assert statements instead of describing the algorithm. Reading someone else's work made me realize that working code alone isnt enough - being able to communicate why it works is equally important, especially in an interview or team setting.