I will review Janice's assignment 1: https://github.com/clv-techlead/algorithms_and_data_structures/pull/1

- Paraphrase the problem in your own words.

You're given a string made up only of the six bracket characters ()[]{}. Decide whether it's well-formed: every opening bracket must be matched by the same type of closing bracket, and brackets must close in the correct, nested order (no crossing or leftover opens). Return True if the whole string is well-formed; otherwise return False.

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

My examples:

Input: s = "(}[])"

Output: False

Input: s = "{[([]{})]}"

Output: True

Janice's example:

examples.append({"input": "({[]})", "expected_output": True})

String: "({[]})"

As per Stack data structure for this problem, we can check every step to confirm this string in the desired function is True or False

(is an opener → push → stack = [(]

{ is an opener → push → stack = [(, {]

[is an opener → push → stack = [(, {, []

] is a closer → pop top [→ matches pairs[']'] == '[' → stack = [(, {]

} is a closer → pop top { → matches pairs['}'] == '{' → stack = [(]

) is a closer → pop top (→ matches pairs[')'] == '(' → stack = []

So stack became empty, and the function result will be True

- Copy the solution your partner wrote.

```
# Your answer here
def is_valid_brackets(s: str) -> bool:
    stack = []
    bracket_map = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in bracket_map.values():  # It's an opening bracket
            stack.append(char)
        elif char in bracket_map.keys():  # It's a closing bracket
            if not stack:  # Stack is empty, no matching opening bracket
                return False
            top_element = stack.pop()
            if bracket_map[char] != top_element:  # Mismatch in opening and closing brackets
                return False
        # Ignore any other characters

    return not stack  # Stack should be empty if all brackets are matched
```

- Explain why their solution works in your own words.

Why it works

Core idea (stack): The algorithm keeps a stack of unclosed opening brackets.

When it sees an opening bracket ((, {, [), it pushes it onto the stack.

When it sees a closing bracket (), },]), it pops the most recent opener and checks that the types match using bracket_map.

Type matching: bracket_map = {')':'(', ']':'[', '}':'{'} guarantees a closer only matches the correct opener. If the popped opener doesn't equal bracket_map[closer], the sequence is invalid → False.

Order (nesting) correctness: Because the stack always compares a closer with the most recent unmatched opener, it enforces proper nesting (e.g., [(]) fails when] tries to close ().

Early failure checks:

If a closer appears when the stack is empty, there's nothing to match → False.

If a type mismatch happens, return False immediately.

Complete matching at the end: After scanning all chars, return not stack ensures no opener is left unmatched. Empty stack ⇒ every opener had a matching closer ⇒ valid.

Edge cases handled:

"" (empty string) → stack stays empty → True (commonly accepted).

")(" or ")" → detects empty stack on a closer → False.

"(((" → leftover openers make not stack false → False.

Non-bracket characters are ignored by design (though the prompt says only brackets will appear).

That's why the solution correctly recognizes valid bracket sequences.

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

--Time Complexity

Worst / Average case: O(n), where n is the length of the string.

Each character is examined once.

Stack operations (append, pop) are O(1).

Dictionary lookups (bracket_map[char], char in bracket_map) are O(1) on average.

Best case: O(1) early exit (e.g., first char is a closing bracket on an empty stack).

--Space Complexity

Worst case: O(n) — if the string is all opening brackets, the stack can grow to size n.

Best case: O(1) — immediate failure (e.g., starts with a closer) or an empty string.

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

--What's good

Uses the right idea: a stack to match brackets.

Runs in O(n) time and uses O(n) space worst-case.

Clean mapping bracket_map = {')':'(', ']':'[', '}':'{'} and a neat final check return not stack.

Stops early on mistakes (mismatch or empty stack) — efficient.

--What to tweak and adjust

Be clear about non-bracket characters: either reject them or say you're ignoring them.

Use elif char in bracket_map: (simpler than .keys()).

Define openers = set(bracket_map.values()) once, use it in the loop.

((Reflection))

Today I reviewed Janice's Assignment 1 solution and focused on explaining both correctness and opportunities for polish. I began by restating the specification in my own words and then made the core invariant explicit: the stack holds unmatched opening brackets, and every closing bracket must match the most recent opener. To demonstrate correctness, I walked through the example "({[]})" step by step, showing each push and pop and confirming the stack is empty at the end. I contrasted that with a failing case like "[(])", which exposes crossing pairs and triggers a mismatch when popping. I then discussed complexity: O(n) time because each character is processed once with O(1) stack and dictionary operations, and O(n) worst-case space for the stack if many openers appear

consecutively. My suggested tweaks were minimal and practical: use elif ch in bracket_map instead of .keys(), precompute openers = set(bracket_map.values()) for clarity, decide whether to reject or ignore any non-bracket characters, add a short docstring, and include edge tests (empty string, lone closer, leftover openers). Overall, pairing concrete examples with the invariant made the feedback clear and respectful, and reinforced my own ability to reason about stack-based validation problems.