Intuition

Use a Stack Data Structure: The stack is a fundamental data structure in solving problems related to bracket matching. It follows the Last In, First Out (LIFO) principle. In this problem, we use a stack to keep track of the opening brackets encountered so far.

Iterate Through the Input String: We iterate through the input string s character by character.

Checking Valid Pairs: For every character c in the input string, we check if the stack is not empty and if the pair (stack[-1], c) forms a valid bracket pair. If it does, that means we have found a matching opening and closing bracket. In this case, we remove the opening bracket from the stack, indicating that it has been successfully matched.

Handling Unmatched Brackets: If the pair (stack[-1], c) is not a valid bracket pair, we add the current character c to the stack. This step is crucial because it means we have encountered either an opening bracket without a corresponding closing bracket yet or an unmatched closing bracket.

Final Check: After processing all characters, we ensure that the stack is empty. An empty stack implies that all opening brackets have been successfully matched with their corresponding closing brackets. If the stack is not empty, there are unmatched opening brackets in the input string, so the expression is not valid.

Return the Result: If the stack is empty after processing the entire input string, we return True to indicate that the input string contains a valid expression with properly matched brackets. If there are unmatched brackets left in the stack, we return False.

Approach

Initialize an Empty Stack: Create an empty stack to store opening parentheses encountered in the input string.

Iterate Through the Input String: Traverse the given input string from left to right.

Process Each Character: For each character in the input string:

a. If It's an Opening Parenthesis: If the current character is an opening parenthesis ('(', '{', or '['), push it onto the stack. This indicates that an opening parenthesis has been encountered, and we need to find a matching closing parenthesis later.

b. If It's a Closing Parenthesis: If the current character is a closing parenthesis (')', '}', or ']'), do the following checks:

If the stack is empty, there is no corresponding opening parenthesis for the current closing parenthesis. In this case, the expression is invalid. Return False.

If the stack is not empty, pop the top element from the stack. If the popped opening parenthesis doesn't match the current closing parenthesis, the expression is invalid. Return False.

Final Check: After processing all characters in the input string, check if the stack is empty. If it's empty, all opening parentheses have been properly matched with closing parentheses, and the expression is valid. Return True. If the stack is not empty, there are unmatched opening parentheses, so the expression is invalid. Return False.

Complexity

- Time complexity: O(n)
- Space complexity: O(n)

Code

```
class Solution:
    def isValid(self, s: str) -> bool:
        valid_brack = [('{', '}'), ('(', ')'), ('[', ']')]
        stack = []
        for c in s:
            if len(stack)>0 and (stack[-1], c) in valid_brack:
                stack.pop()
            else:
                stack.append(c)
        return len(stack)==0
```

GitHub