Valid Parentheses

Problem Statement

The 'Valid Parentheses' problem is a popular coding question on platforms like LeetCode. In this problem, you are given a string containing just the characters '(', ')', '{', '}', '[' and ']', and you need to determine if the input string is valid.

To define valid parentheses:

- An open bracket must be closed by the same type of bracket.
- Open brackets must be closed in the correct order.
- An empty string is considered valid.

For example, "()" and "()[]{}" are considered valid, while "(]" and "([)]" are not.

One important aspect of this problem is understanding the concept of a stack data structure. When iterating through the input string, we can use a stack to keep track of the opening brackets we encounter. Whenever we encounter a closing bracket, we can check if it matches the top element of the stack. If it does, we pop the element from the stack; if not, the parentheses are not valid.

This problem is a good exercise in understanding stack operations and can be solved efficiently in linear time complexity. It is often used to assess a programmer's ability to implement fundamental data structures and algorithms.

In summary, the 'Valid Parentheses' problem requires checking whether a given string of parentheses is valid according to a specific set of rules regarding the proper opening and closing of brackets.

Examples

Valid Parentheses Examples:

Input: "()"
Expected Output: True
Explanation: The string contains only one pair of parentheses in the correct order, making it a valid expression.

Input: "()[]{}"
Expected Output: True
Explanation: This string contains multiple pairs of parentheses in correct order and nested properly, making it a valid expression.

Input: "([])"
Expected Output: True

Explanation: The parentheses are nested correctly, with square brackets inside round brackets, making it a valid expression.

Invalid Parentheses Examples:

Input: "(]"
Expected Output: False
Explanation: The closing bracket does not match the opening bracket, making this an invalid expression.

Input: "([)]"
Expected Output: False
Explanation: Although each type of bracket has a matching pair, the nesting is incorrect, resulting in an invalid expression.

Input: "{[}]"
Expected Output: False
Explanation: The types of brackets are mixed here, with curly brackets and square brackets not paired correctly, making it an invalid expression.

Constraints

The constraints provided by LeetCode for the 'Valid Parentheses' problem are crucial to consider while solving this coding question. The maximum length of the input string in this problem is typically constrained to be within the range of 0 to 10^4. It is essential to account for scenarios where the input string could be empty or contain the maximum number of characters allowed.

Additionally, when dealing with the input string, programmers should consider edge cases where the string consists of only opening or closing brackets without any pairs, as well as cases where the nesting of brackets is deliberately incorrect. These edge cases are important to test the robustness of the solution and ensure it can handle various scenarios effectively.

Understanding and adhering to these constraints not only helps in creating a solution that works efficiently within the specified limitations but also showcases the ability to write code that is adaptable and can handle unexpected inputs gracefully.

Approach to Solving the 'Valid Parentheses' Problem

When tackling the 'Valid Parentheses' problem, a common and effective strategy is to utilize a stack data structure. The use of a stack allows for efficient tracking of opening brackets as we traverse through the input string, enabling us to verify the validity of the parentheses.

Using a Stack Data Structure

Initialization: Start by creating an empty stack to store the opening brackets encountered during iteration.

Iterating Through the String: As you loop through each character in the input string, perform the following steps:

- If the current character is an opening bracket (i.e., '(', '[', '{'), push it onto the stack.
- If the current character is a closing bracket (')', ']', '}'), check if the stack is empty. If it is, the parentheses are not valid. Otherwise, compare the current closing bracket with the top element of the stack.
  - If they form a matching pair (e.g., '(' and ')', '[' and ']', '{' and '}'), pop the top element from the stack.
  - If they do not match, the parentheses are invalid.

Validation: After processing all characters in the string, check if the stack is empty. If it is, the parentheses are valid; otherwise, they are invalid.

Effectiveness of the Stack Approach

The stack data structure proves effective in solving the 'Valid Parentheses' problem due to its Last-In-First-Out (LIFO) property. This property aligns well with the requirement of maintaining the correct order of opening and closing brackets. By leveraging the stack to keep track of opening brackets, we can ensure that the nesting of parentheses is accurate and that each closing bracket matches the most recent opening bracket encountered.

Moreover, the stack allows for efficient checking of nested parentheses, enabling quick identification of invalid expressions. This approach offers a clear and structured method to validate parentheses, making it a preferred choice for solving this problem.

Referring back to the examples provided earlier, such as "()" and "()[]{}", you can see how the stack operation facilitates the validation process, confirming the correctness of the parentheses arrangement and order.

In conclusion, the stack-based approach offers a logical and efficient solution to the 'Valid Parentheses' problem, showcasing the significance of fundamental data structures in algorithmic problem-solving.

Simplified Explanation

Understanding Valid Parentheses

In the 'Valid Parentheses' problem, we need to check if a string of parentheses is arranged correctly. Imagine each opening bracket as a door that must be closed by the same type of door. The order in which you close these doors must match the order in which you opened them. An empty room (empty string) is always considered neat and tidy.

How It Works

To solve this problem, we can use a stack, like a stack of plates. As we go through the string of parentheses, we stack up the doors we open. When we reach a closing door, we check if it matches the last door we opened. If it does, we remove that door from the stack. If the doors don't match, the arrangement is incorrect.

Checking Validity

After going through all the doors, if our stack is empty, it means all doors were closed properly, making the arrangement valid. If there are still doors in the stack, the arrangement is invalid.

Why We Use a Stack

Using a stack helps us keep track of the order in which we opened the doors. It ensures that each opening door has a matching closing door and that they are closed in the right order. This method simplifies the process of checking the validity of the parentheses.

Key Takeaway

By visualizing the parentheses as doors and employing a stack to manage their opening and closing, we can efficiently determine if a string of parentheses is correctly arranged. This approach provides a straightforward and intuitive solution to the 'Valid Parentheses' problem.

Code

```
class Solution:
    def isValid(self, s: str) -> bool:
        # Dictionary to hold matching pairs of brackets
        bracket_map = {')': '(', '}': '{', ']': '['}
        # Stack to keep track of opening brackets
        stack = []

        # Iterate over each character in the string
        for char in s:
            # If the character is a closing bracket
            if char in bracket_map:
                # Pop the top element from the stack if it's not empty;
otherwise, use a dummy value
                top_element = stack.pop() if stack else '#'
                # Check if the popped element matches the expected opening
bracket
                if bracket_map[char] != top_element:
                    return False
            else:
                # If it's an opening bracket, push it onto the stack
                stack.append(char)

        # If the stack is empty, all brackets were properly closed; otherwise,
it's invalid
        return not stack
```

This Python code provides a solution to the 'Valid Parentheses' problem by implementing a function isValid that takes a string as input and returns a boolean indicating whether the parentheses in the input string are valid or not. The code utilizes a stack to keep track of the opening brackets encountered and verifies the validity of the parentheses by matching closing brackets with the corresponding opening brackets. The mapping dictionary helps in checking the matching pairs of brackets efficiently.

By running the function with the provided test cases, the code demonstrates its functionality in determining the validity of different sets of parentheses arrangements. The approach leverages the stack data structure to ensure the proper ordering and nesting of parentheses, offering a clear and concise solution to the problem.

Explanation of Each Line of Code

Let's break down the Python code provided above for solving the 'Valid Parentheses' problem step by step:

Function Definition:

- def isValid(s: str) -> bool:: This line defines a function named isValid that takes a string s as input and returns a boolean value indicating the validity of parentheses in the input string.

Initialization:

- stack = []: Initializes an empty list stack to serve as a stack data structure to store opening brackets encountered during iteration.
- mapping = {')': '(', '}': '{', ']': '['}: Creates a dictionary mapping that maps closing brackets to their corresponding opening brackets. This mapping helps in efficiently checking for matching pairs of brackets.

Iterating Through the Input String:

- for char in s:: Initiates a loop to iterate through each character in the input string.
  - if char in mapping:: Checks if the current character is a closing bracket.
    - top_element = stack.pop() if stack else '#': Pops the top element from the stack if the stack is not empty; otherwise, assigns '#' to top_element.
    - if mapping[char] != top_element:: Compares the current closing bracket with the top element of the stack to verify if they form a matching pair.
      - return False: Returns False if the brackets do not match, indicating an invalid parentheses arrangement.

  - else:: Executes if the current character is an opening bracket.
    - stack.append(char): Adds the opening bracket to the stack for tracking.

Validation:

- return not stack: Returns True if the stack is empty after processing all characters, indicating a valid parentheses arrangement; otherwise, returns False.

Function Testing:

- print(isValid("()")) # Output: True: Tests the function with a simple valid parentheses input.
- print(isValid("()[]{}")) # Output: True: Tests the function with multiple pairs of valid parentheses.
- print(isValid("([])")) # Output: True: Tests the function with nested valid parentheses.
- print(isValid("(]")) # Output: False: Tests the function with an invalid parentheses arrangement.
- print(isValid("([)]")) # Output: False: Tests the function with another invalid arrangement.
- print(isValid("{[}]")) # Output: False: Tests the function with mixed and invalid parentheses.

This code efficiently utilizes a stack data structure and a mapping dictionary to validate the correctness of parentheses arrangements in a given input string. By employing stack operations and bracket matching logic, the code provides a robust solution to the 'Valid Parentheses' problem, as demonstrated through the test cases.