20. Valid Parentheses

Easy Stack String

Problem Description

The problem presents a scenario in which we are given a string `s` consisting of six possible characters: the opening and closing brackets of three types—parentheses `()`, square brackets `[]`, and curly braces `{}`. The challenge is to determine whether this string represents a sequence of brackets that is considered valid based on certain rules. A string of brackets is deemed valid if it satisfies the following conditions:

1. Each opening bracket must be closed by a closing bracket of the same type.
2. Opening brackets must be closed in the correct order. That means no closing bracket should interrupt the corresponding pair of an opening bracket.
3. Each closing bracket must have an associated opening bracket of the same type before it.

For example, a string `"(())"` is valid because each opening parenthesis `(` has a corresponding closing parenthesis `)` that occurs later in the string, and they are properly nested and ordered.

Intuition

The intuition behind the solution utilizes a common data structure known as a stack, which operates on a last-in, first-out (LIFO) principle. This principle mimics the necessary behavior for tracking opening brackets and ensuring they are closed in the correct order. The steps followed in the solution are:

1. Initialize an empty stack to keep track of opening brackets.
2. Traverse the input string character by character.
3. When an opening bracket is encountered (`'('`, `'['`, or `'{'`), push it onto the stack. This represents waiting for a closing bracket to match.
4. When a closing bracket is encountered (`')'`, `']'`, or `'}'`), check if it forms a pair with the last opening bracket added to the stack (i.e., on top of the stack).
5. If the stack is empty (no opening bracket available to match) or the closing bracket does not form a valid pair with the opening bracket on top of the stack, we know the string is invalid, and we immediately return `False`.
6. If we successfully traverse the entire input string and the stack is empty, meaning all opening brackets have been matched correctly, we return `True`. If the stack is not empty, it indicates there are unmatched opening brackets, and therefore, the string is invalid.

The concise implementation of this algorithm ensures that both the correct type and order of brackets are validated for the string to be considered valid.

Solution Approach

The solution utilizes a stack data structure to track opening brackets and ensure they have the appropriate closing brackets. Let's walk through the implementation step by step:

1. **Initialize a Stack**: A list named `stk` is created to serve as a stack.
2. **Pairs Set**: We define a set `d` containing string representations of the valid bracket pairs: `'()'`, `'[]'`, and `'{}'`. This helps quickly check if an encountered closing bracket correctly pairs with the last opening bracket on the stack.
3. **Iterate Over the String**: The algorithm iterates over each character `c` in the input string `s`.
   - **Opening Bracket**: If `c` is an opening bracket (`'('`, `'['`, or `'{'`), it is pushed onto `stk`, waiting for the corresponding closing bracket.
   - **Closing Bracket**: If `c` is a closing bracket (`')'`, `']'`, or `'}'`):
     - The stack is checked to ensure it's not empty, which would mean there's no opening bracket to match the closing one.
     - If the stack is not empty, the top element is popped. We concatenate it with `c` to check if they form a valid pair by checking against set `d`.
     - If either condition fails – the stack was empty or the concatenation of the popped element with `c` does not form a valid pair – the function immediately returns `False` because we've detected an invalid bracket sequence.
4. **Final Stack Check**: After processing all characters in the string, the algorithm checks if the stack is empty. An empty stack implies all opening brackets were properly matched and closed.
   - If the stack is empty (`return not stk`), it indicates a valid bracket sequence and returns `True`.
   - If the stack is not empty, some opening brackets were not closed, thus the function returns `False`, representing an invalid bracket sequence.

The solution is efficient with a linear time complexity, O(n), where n is the length of the string `s`. It only requires a single pass through the string and constant-time operations for each character. The space complexity is also O(n), in the worst case where the string consists entirely of opening brackets, which would all be pushed onto the stack.

Example Walkthrough

Let's consider a small example to clearly understand the solution approach. Suppose we are given the string `s = "{[()()]}"`. We want to determine if this string represents a valid sequence of brackets. Following the solution steps:

1. **Initialize a Stack**: We start with an empty list `stk` that will be used as our stack.
2. **Pairs Set**: We have a predefined set `d` that contains `'()'`, `'[]'`, and `'{}'` to represent valid bracket pairs.
3. **Iterating Over the String**: We iterate through each character in the string `s`.
   - For the first character `{`, it's an opening bracket, so we push it onto `stk`.
   - For the second character `[`, it's also an opening bracket, so we push it onto `stk`.
   - For the third character `(`, another opening bracket gets pushed onto `stk`.
   - The fourth character `)` is a closing bracket, so we pop the last element `(`, which matches the closing bracket, forming a valid pair `()`. So far so good.
   - The fifth character `(` is once again an opening bracket and is pushed onto `stk`.
   - The sixth character `)` is a closing bracket, and popping from `stk` gives us the matching `(`, forming another valid pair. We continue this process for the remaining characters.
   - The seventh character `]` is a closing bracket, and popping from `stk` gives us `[`, which is the correct complement, forming the pair `[]`.
   - Finally, the last character `}` is a closing bracket, and after popping from `stk` we get `{`, its correct opening pair, forming `{}`.
4. **Final Stack Check**: At the end of iteration, `stk` is empty because every opening bracket has been matched with the correct closing bracket in the proper order.

Since the stack is empty, we return `True`, indicating that the given string `s = "{[()()]}"` is a valid sequence of brackets. Our solution has correctly determined the validity by using a stack to manage the ordering and pairing of the brackets.

Solution Implementation

Python

```python
class Solution:
    def isValid(self, s: str) -> bool:
        # Initialize an empty list to use as a stack
        stack = []
        # Create a set with valid parentheses pairs
        valid_pairs = {'()', '[]', '{}'}

        # Iterate over each character in the string
        for char in s:
            # If the character is an opening parenthesis, push it onto the stack
            if char in '({[':
                stack.append(char)
            # If the stack is empty or the formed pair is not valid, return False
            elif not stack or stack.pop() + char not in valid_pairs:
                return False

        # If the stack is empty, all parentheses were valid and correctly nested
        return not stack
```

Java

```java
class Solution {
    // Method to determine if an input string has valid parentheses
    public boolean isValid(String s) {
        // Use a deque as a stack to keep track of the opening brackets
        Deque<Character> stack = new ArrayDeque<>();
        // Iterate over each character in the input string
        for (char c : s.toCharArray()) {
            // If the current character is an opening bracket, push it onto the stack
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {
                // If the stack is empty or the current closing bracket doesn't match
                // the last opening bracket, the string is not valid
                if (stack.isEmpty() || !match(stack.pop(), c)) {
                    return false;
                }
            }
        }
        // If the stack is empty, all brackets were matched correctly
        return stack.isEmpty();
    }

    // Helper method to determine if two brackets are a matching pair
    private boolean match(char leftBracket, char rightBracket) {
        // Return true if pairs match, false otherwise
        return (leftBracket == '(' && rightBracket == ')') ||
               (leftBracket == '{' && rightBracket == '}') ||
               (leftBracket == '[' && rightBracket == ']');
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a given string has valid parentheses
    bool isValid(string s) {
        // Using a string as a stack to store opening brackets
        string stack;

        // Iterate through all characters in the input string
        for (char c : s) {
            // If the character is an opening bracket, push it onto the stack
            if (c == '(' || c == '{' || c == '[') {
                stack.push_back(c);
            }
            // If the stack is empty or characters don't match, return false
            else if (stack.empty() || !match(stack.back(), c)) {
                return false;
            }
            // If the characters match, pop the opening bracket from the stack
            else {
                stack.pop_back();
            }
        }

        // If the stack is empty, all brackets were properly closed, return true
        return stack.empty();
    }

    // Utility function to check if the opening and closing brackets match
    bool match(char left, char right) {
        return (left == '(' && right == ')') ||
               (left == '[' && right == ']') ||
               (left == '{' && right == '}');
    }
};
```

TypeScript

```typescript
// Map that associates opening braces with their corresponding closing braces.
const bracketPairs = new Map<string, string>([
    ['(', ')'],
    ['[', ']'],
    ['{', '}'],
]);

/**
 * Function to determine if the given string has valid bracket pairing.
 * @param {string} str - The input string containing brackets to be checked.
 * @return {boolean} - Returns true if the string is valid, otherwise false.
 */
function isValid(str: string): boolean {
    // A stack to keep track of the expected closing brackets.
    const expectedBracketsStack: string[] = [];

    // Iterate over each character in the string.
    for (const char of str) {
        // Check if the character is an opening bracket and get its pair.
        if (bracketPairs.has(char)) {
            // Push the expected closing bracket onto the stack.
            expectedBracketsStack.push(bracketPairs.get(char)!);
        } else {
            // If the character is not a matching closing bracket,
            // or the stack is empty (mismatched brackets), return false.
            if (expectedBracketsStack.pop() !== char) {
                return false;
            }
        }
    }

    // If the stack is empty, all brackets were properly closed; otherwise, return false.
    return expectedBracketsStack.length === 0;
}
```

```python
class Solution:
    def isValid(self, s: str) -> bool:
        # Initialize an empty list to use as a stack
        stack = []
        # Create a set with valid parentheses pairs
        valid_pairs = {'()', '[]', '{}'}

        # Iterate over each character in the string
        for char in s:
            # If the character is an opening parenthesis, push it onto the stack
            if char in '({[':
                stack.append(char)
            # If the stack is empty or the formed pair is not valid, return False
            elif not stack or stack.pop() + char not in valid_pairs:
                return False

        # If the stack is empty, all parentheses were valid and correctly nested
        return not stack
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input string. This is because the algorithm iterates over each character in the input string exactly once.

The space complexity of the code is also `O(n)`, as in the worst case (when all characters in the input string are opening brackets), the stack `stk` will contain all characters in the input string.

Time Complexity:

- **Best Case:** When the string is empty or consists of a single pair of brackets, the time complexity is `O(1)` because it takes a constant amount of time.
- **Average Case:** For a typical string with a mix of opening and closing brackets, the time complexity remains `O(n)` because each character is processed once.
- **Worst Case:** In the worst case scenario, where there are `n` characters and the string is properly nested to the deepest level, each character is still processed exactly once, giving us a time complexity of `O(n)`.

Space Complexity:

- The space complexity is `O(n)` which occurs when all characters are opening brackets and hence, all are pushed onto the stack. This represents the maximum space utilized by the algorithm.
- As the stack grows with each opening bracket and shrinks with each closing bracket, the actual space used depends on the number of unmatched opening brackets at any point in the algorithm. Therefore, space usage can vary from `O(1)` (for an empty string or a string with just one pair of brackets) to `O(n)` (when all characters are opening brackets).