

valid parentheses leetcode solution

Valid Parentheses LeetCode Solution

The problem of validating parentheses is a common interview question that can be found on platforms like LeetCode. It challenges candidates to develop a solution that verifies whether a string containing parentheses is valid. A string is considered valid if every opening parenthesis has a corresponding closing parenthesis and the pairs are correctly nested and ordered. In this article, we will explore the problem in detail, discuss various approaches to solving it, and provide a comprehensive solution using different programming languages.

Understanding the Problem

Before diving into the solution, it is essential to grasp the problem's requirements. The string can contain three types of parentheses:

1. Round brackets: `()`
2. Square brackets: `[]`
3. Curly braces: `{}`

To determine if the string is valid, the following conditions must be met:

- Every opening parenthesis must have a matching closing parenthesis.
- Parentheses must close in the correct order. For instance, `{([])}` is valid, while `{([)]}` is not.

Examples

To clarify the requirements, let's look at some examples:

- Input: `()`
- Output: `true`
- Input: `()[]{}`
- Output: `true`
- Input: `[ ]`
- Output: `false`
- Input: `{([)]}`
- Output: `false`
- Input: `{[]}`
- Output: `true`

Approaches to Solve the Problem

There are several approaches to solving the valid parentheses problem, including:

1. Stack Data Structure
2. Using a Counter
3. Recursion

Let's delve into the most common and efficient approach, which is using a stack.

Stack Data Structure

The stack data structure is particularly suitable for this problem because it operates on a Last In First Out (LIFO) principle. This allows us to easily manage the order of opening and closing parentheses.

Algorithm:

1. Initialize an empty stack.
2. Iterate through each character in the input string:
 - If it is an opening parenthesis ('(', '{', or '['), push it onto the stack.
 - If it is a closing parenthesis (')', '}', or ']'):
 - Check if the stack is empty. If it is, return `false` because there is no corresponding opening parenthesis.
 - Pop the top element from the stack and check if it matches the current closing parenthesis. If it does not match, return `false`.
3. After iterating through the string, check if the stack is empty. If it is, the parentheses are valid; otherwise, they are not.

Time Complexity:

- The time complexity of this approach is $O(n)$, where n is the length of the input string, since we are iterating through the string only once.

Space Complexity:

- The space complexity is $O(n)$ in the worst case when all characters are opening parentheses.

Implementing the Solution

Now that we've established the algorithm, let's implement it in several popular programming languages: Python, Java, and JavaScript.

Python Implementation

```
```python
def isValid(s: str) -> bool:
```

```

stack = []
mapping = {')': '(', '}': '{', ']': '['}

for char in s:
 if char in mapping:
 top_element = stack.pop() if stack else ""
 if mapping[char] != top_element:
 return False
 else:
 stack.append(char)

return not stack
```

```

Explanation:

- We use a dictionary to map closing parentheses to their corresponding opening ones.
- The loop checks if the character is a closing parenthesis; if so, it checks the top of the stack. If it's not a match, it returns `false`.
- Finally, we return `True` if the stack is empty, indicating all parentheses were matched.

Java Implementation

```

```java
import java.util.Stack;

public class ValidParentheses {
 public boolean isValid(String s) {
 Stack stack = new Stack<>();
 for (char c : s.toCharArray()) {
 if (c == '(' || c == '{' || c == '[') {
 stack.push(c);
 } else {
 if (stack.isEmpty()) {
 return false;
 }
 char top = stack.pop();
 if ((c == ')' && top != '(') ||
 (c == '}' && top != '{') ||
 (c == ']' && top != '[')) {
 return false;
 }
 }
 }
 return stack.isEmpty();
 }
}
```

```

Explanation:

- Similar to the Python implementation, we utilize a stack to keep track of opening parentheses.
- We check for matches and return `false` if any mismatches occur.

JavaScript Implementation

```
```javascript
function isValid(s) {
 const stack = [];
 const mapping = { ')': '(', '}': '{', ']': '[' };

 for (let char of s) {
 if (char in mapping) {
 const topElement = stack.length === 0 ? "" : stack.pop();
 if (mapping[char] !== topElement) {
 return false;
 }
 } else {
 stack.push(char);
 }
 }

 return stack.length === 0;
}
```
```

Explanation:

- The JavaScript implementation follows the same logic as the Python and Java versions, using an array as a stack.

Handling Edge Cases

When solving the valid parentheses problem, it's also crucial to consider edge cases:

1. Empty String: An empty string should return `true`, as there are no unmatched parentheses.
2. Single Parenthesis: Strings such as `(` or `[` should return `false`, as they are unmatched.
3. Long Strings: Performance may be tested with very long strings to ensure that the algorithm runs efficiently.

Conclusion

The valid parentheses problem is a classic example of using a stack to solve problems that involve nested structures. The approach discussed here is both efficient and straightforward, making it an excellent choice for interviews and real-world applications. By mastering this problem, you can gain a deeper understanding of stack data structures and their applications in various programming challenges.

Frequently Asked Questions

What is the problem statement for 'Valid Parentheses' on LeetCode?

The problem asks to determine if a string containing just the characters '(', ')', '{', '}', '[' and ']' is valid. An input string is valid if the brackets are closed in the correct order.

What is the optimal time complexity for solving the 'Valid Parentheses' problem?

The optimal time complexity for solving the 'Valid Parentheses' problem is $O(n)$, where n is the length of the input string.

What data structure is commonly used to solve the 'Valid Parentheses' problem?

A stack data structure is commonly used to solve the 'Valid Parentheses' problem, as it allows for the Last-In-First-Out (LIFO) processing required for matching brackets.

How do you handle edge cases in the 'Valid Parentheses' problem?

Edge cases include checking for an empty string (which is valid) and ensuring that the parentheses are correctly matched and nested. If the count of opening and closing brackets ever mismatches during traversal, the string is invalid.

Can you provide a sample Python code solution for 'Valid Parentheses'?

Sure! Here is a sample solution in Python:

```
```python
def isValid(s):
 stack = []
 parentheses_map = {'(': ')', '{': '}', '[': ']'}
 for char in s:
 if char in parentheses_map:
 top_element = stack.pop() if stack else ""
 if parentheses_map[char] != top_element:
 return False
 else:
 stack.append(char)
 return not stack
```
```

What are the common pitfalls to avoid when implementing the solution?

Common pitfalls include not properly handling cases where the stack is empty when trying to pop an element, and not checking if the stack is empty at the end of the function, which indicates unmatched opening brackets.

What is the significance of using a hash map in the solution?

Using a hash map allows for $O(1)$ time complexity when checking for corresponding opening brackets, improving the efficiency of the solution compared to using a linear search.

[Valid Parentheses Leetcode Solution](#)

Valid Parentheses Leetcode Solution

Related Articles

- [viviscal hair therapy beauty rest](#)
- [vista 128fbp user manual](#)
- [volume of rectangular prism worksheet](#)

[Back to Home](#)