Valid Parentheses Problem DSA

Give DSA notes for the below problem. Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Every close bracket has a corresponding open bracket of the same type.

Here are **detailed notes** to help you understand and solve the problem effectively:

Problem Analysis

The task is to check whether a given string containing only parentheses and brackets is **valid** based on the following rules:

1. **Matching Type**: Each opening bracket must have a matching closing bracket of the same type (`()` , `{}` , `[]`).
2. **Order of Closure**: Brackets must be closed in the correct order (e.g., `({})` is valid, but `{(})` is not).
3. **Balanced Pairs**: Each closing bracket must have a corresponding opening bracket before it.

Approach

The problem can be solved efficiently using a **stack**. A stack is an ideal data structure for problems where you need to process items in a "last-in, first-out" manner.

Steps to Solve

1. **Initialize a Stack**: Use a stack to store opening brackets as you traverse the string.
2. **Iterate Through the String**:
   - If you encounter an **opening bracket** (`(` , `{` , `[`), push it onto the stack.
   - If you encounter a **closing bracket** (`)` , `}` , `]`):
     - Check if the stack is empty (if yes, return `false` because there's no matching opening bracket).
     - Otherwise, pop the top of the stack and verify whether it matches the closing bracket.
3. **Check the Stack After Traversal**:
   - If the stack is empty, all brackets were properly matched and closed, so the string is valid.
   - If the stack is not empty, there are unmatched opening brackets, and the string is invalid.

Algorithm

Pseudocode

```text
1. Create a stack (empty initially).
2. Define a map (hash table) to store pairs of brackets:
   matching_pairs = { ')': '(', '}': '{', ']': '[' }.
3. Loop through each character in the string:
   - If the character is an opening bracket ('(', '{', '['), push it onto the stack.
   - If the character is a closing bracket (')', '}', ']'):
       - If the stack is empty, return false (no matching opening bracket).
       - Otherwise, pop the top element from the stack and check if it matches the
closing bracket.
       - If not, return false.
4. After the loop, check if the stack is empty:
   - If yes, return true (all brackets matched).
   - If no, return false (unmatched opening brackets).
```

Complexity Analysis

1. **Time Complexity**:
   - Each character is processed exactly once (push or pop operation).
   - Time complexity: **O(n)**, where `n` is the length of the string.
2. **Space Complexity**:
   - The stack may contain up to `n` elements in the worst case (e.g., all opening brackets).
   - Space complexity: **O(n)**.

Edge Cases

1. **Empty String**: An empty string is considered valid (`"" → true`).
2. **Unbalanced Brackets**:
   - `"(]"` → False (mismatched closing bracket).
   - `"((("` → False (unmatched opening brackets).
3. **Correctly Nested Brackets**:
   - `"({[]})"` → True.
4. **Single Closing Bracket**:
   - `")"` → False.

Key Points for Implementation

1. Use a **stack** to track opening brackets.
2. **Match Closing Brackets**:
   - Use a hash map to quickly match pairs of brackets.

- Example: `{')': '(', ']': '[', '}': '{'}`.

3. **Early Exit**:
   - If a closing bracket appears without a corresponding opening bracket, you can return `false` immediately.

Example Walkthrough

Input: `"({[]})"`

- Step 1: Initialize stack: `stack = []`.
- Step 2: Traverse the string:
  - `'('` → Push to stack: `stack = ['(']`.
  - `'{'` → Push to stack: `stack = ['(', '{']`.
  - `'['` → Push to stack: `stack = ['(', '{', '[']`.
  - `']'` → Matches top of stack. Pop: `stack = ['(', '{']`.
  - `'}'` → Matches top of stack. Pop: `stack = ['(']`.
  - `')'` → Matches top of stack. Pop: `stack = []`.
- Step 3: Stack is empty after traversal → Valid string → Output: `true`.

Input: `"([)]"`

- Step 1: Initialize stack: `stack = []`.
- Step 2: Traverse the string:
  - `'('` → Push to stack: `stack = ['(']`.
  - `'['` → Push to stack: `stack = ['(', '[']`.
  - `')'` → Does not match top of stack. Output: `false`.

Let me know if you'd like to implement this step-by-step in Java!

Add java code

Here's the Java code implementation for the problem:

java

```java
import java.util.Stack;

public class ValidParentheses {
    public static boolean isValid(String s) {
        // Use a stack to track opening brackets
        Stack<Character> stack = new Stack<>();

        // Map closing brackets to their corresponding opening brackets
        for (char ch : s.toCharArray()) {
            // If the character is an opening bracket, push it to the stack
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            }
            // If the character is a closing bracket
            else {
                // If the stack is empty, there is no matching opening bracket
                if (stack.isEmpty()) {
                    return false;
                }

                // Pop the top of the stack and check if it matches
                char top = stack.pop();
                if ((ch == ')' && top != '(') ||
                    (ch == '}' && top != '{') ||
                    (ch == ']' && top != '[')) {
                    return false;
                }
            }
        }

        // If the stack is empty, all brackets were matched
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println(isValid("()"));         // true
```

```
        System.out.println(isValid("()[]{}"));     // true
        System.out.println(isValid("(]"));         // false
        System.out.println(isValid("([)]"));       // false
        System.out.println(isValid("{[]}"));       // true
    }
}
```

Explanation of Code

1. **Stack Usage**:
   - Push opening brackets onto the stack.
   - Pop and check for matching brackets when a closing bracket is encountered.
2. **Bracket Matching**:
   - Use simple `if` conditions to match opening and closing brackets.
3. **Edge Case Handling**:
   - If the stack is empty when a closing bracket is encountered, return `false`.
   - After traversing the string, check if the stack is empty to ensure all opening brackets were matched.

How to Run

1. Copy the code into a Java file, for example, `ValidParentheses.java`.
2. Compile the file using `javac ValidParentheses.java`.
3. Run the program using `java ValidParentheses`.
4. Observe the outputs for the given test cases.

Let me know if you'd like further explanation or additional test cases!

Also you can add one condition. If length of string is odd we can return false;

Great suggestion! Adding a check for odd-length strings can improve performance since a valid string of brackets must always have an even length. Here's the updated Java code with that condition:

java

```java
import java.util.Stack;

public class ValidParentheses {
    public static boolean isValid(String s) {
        // Early exit: If the length is odd, it cannot be valid
        if (s.length() % 2 != 0) {
            return false;
        }

        // Use a stack to track opening brackets
        Stack<Character> stack = new Stack<>();

        // Loop through each character in the string
        for (char ch : s.toCharArray()) {
            // If the character is an opening bracket, push it to the stack
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            }
            // If the character is a closing bracket
            else {
                // If the stack is empty, there is no matching opening bracket
                if (stack.isEmpty()) {
                    return false;
                }

                // Pop the top of the stack and check if it matches
                char top = stack.pop();
                if ((ch == ')' && top != '(') ||
                    (ch == '}' && top != '{') ||
                    (ch == ']' && top != '[')) {
                    return false;
                }
            }
        }

        // If the stack is empty, all brackets were matched
```

```
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println(isValid("()"));         // true
        System.out.println(isValid("()[]{}"));     // true
        System.out.println(isValid("(]"));         // false
        System.out.println(isValid("([)]"));       // false
        System.out.println(isValid("{[]}"));       // true
        System.out.println(isValid("{[}"));        // false
        System.out.println(isValid("(((("));       // false
    }
}
```

Changes Made

1. **Added Odd-Length Check:**

   ```java
   if (s.length() % 2 != 0) {
       return false;
   }
   ```

   - If the string length is odd, return `false` immediately.

2. **Early Exit for Efficiency:**
   - This avoids unnecessary processing for invalid strings upfront.

Benefits

- **Performance Optimization:** Reduces runtime for invalid strings by skipping the stack operations.

- **Code Readability**: Highlights that valid parentheses must have an even number of characters.

Let me know if you need further tweaks or additional enhancements!