1190. Reverse Substrings Between Each Pair of Parentheses

Medium Stack String Leetcode Link

Problem Description

In this problem, you are given a string `s` which is a sequence of lower case English letters and parentheses (brackets). Your task is to process the string by reversing the substrings within each pair of matching parentheses. Importantly, you need to begin this process with the innermost pair of brackets and work your way outwards. The expected output is a string that has no parentheses and reflects the reversals that happened within each (formerly) enclosed section.

Intuition

To solve this problem, we need to simulate the process of reversing characters within the parentheses. We could approach this problem iteratively or recursively, but regardless of the approach, the challenge is to manage the parentheses and the elements between them efficiently.

Intuitively, we can think of each pair of parentheses as a signal to reverse the string within them. When we encounter an opening parenthesis, it indicates the start of a section that will eventually be reversed. A closing parenthesis signals the end of such a section. Since we want to start with the innermost pair, we can't just reverse as we go because they could be nested. We must find a way to pair opening and closing parentheses and then reverse the substrings when we know we've reached the innermost pair.

To achieve this, we use a stack to keep track of the indices of the opening parentheses. When we encounter a closing parenthesis, we pop the last index from the stack which corresponds to the matching opening parenthesis. By marking the positions with indices, we can then navigate back and forth in the string.

The code uses a stack and an additional array, `d`, to store the indices of where to "jump" next when a parenthesis is encountered. When processing the string, if we hit a closing parenthesis, we use the associated opening index to jump back to the opening parenthesis. The `x` variable changes the direction of traversal each time we encounter a parenthesis. This allows us to traverse the string outwards from the innermost nested parentheses.

The solution is very clever because it linearizes the steps needed to reverse the substrings. If you picture the parenthesis matching, it's like unfolding the string by making connections between each pair; you hop from one parenthesis to its partner, and back, but each time you ignore the pair you just left, effectively collapsing the string's nested structure.

At the end of this process, the string is traversed, taking into account all reversals, and we join together all non-parenthesis characters to form the result.

Solution Approach

The solution approach is centered around identifying and processing the nested structures within the input string, `s`. The key algorithmic technique used in this implementation is a stack, which is a last-in-first-out (LIFO) data structure. The stack is utilized to keep track of opening parentheses indices.

Here is a step-by-step explanation of how the algorithm executes:

1. Initialize a stack, `stk`, to store indices of opening parentheses.
2. Initialize an array, `d`, of the same length as the input string, `s`. This array will be used to track the jump positions after we encounter a parenthesis.
3. Iterate through each character in the string `s` by its index `i`:
   - If the current character is an opening parenthesis `'('`, push its index onto the stack.
   - If it's a closing parenthesis `')'`, pop an index from the stack (this represents the matching opening parenthesis), and update the `d` array at both the opening and closing parentheses' positions, setting them to point at each other (i.e., `d[i]` and `d[j]` are set to `j` and `i` respectively to create a 'jump' from one parenthesis to the matching one).
4. After setting up the 'jumps' with the `d` array, we initialize two variables, `i` with the value of 0 (to start at the beginning of the string) and `x` with the value of 1 (to move forward initially).
5. Create an empty list `ans` to store the characters of the final result.
6. While `i` is within the bounds of the string length:
   - If `s[i]` is a parenthesis, use the `d` array at index `i` to jump to the matching parenthesis, and reverse the traversal direction by negating `x` (`x = -x`).
   - Otherwise, append the current character `s[i]` to the `ans` list.
   - Move to the next character by updating `i` (`i += x`), with `x` managing the direction of traversal.
7. After completing the traversal, return the result by joining all elements in `ans` to form the final string without any brackets.

This method allows the program to "skip" over the parts of the string that have been processed and dynamically reverse the direction of traversal when it moves between matching parentheses. The stack ensures that we always find the innermost pair of parentheses first, and the `d` array allows us to traverse the string efficiently without recursion and without modifying the input string.

By using this stack and jump array technique, we intelligently collapse nested structures, mimicking the effect of recursive reversal, and ultimately, we achieve a linear-time solution with respect to the input size.

Example Walkthrough

Let's use a small example to illustrate the solution approach with the input string `s = "(ed(et(oc))el)"`. We want to reverse the substrings within each pair of parentheses, starting from the innermost one.

Here is how the algorithm would execute on this example:

1. Initialize an empty stack `stk` and an array `d` of the same length as `s`. In our case, `s` has a length of 13, so `d` would be an array of 13 uninitialized elements.
2. Iterate over each character in `s`.
   - At index 0, we encounter `'('`, so we push 0 onto the stack.
   - At index 1, the character is `'e'`, and we do nothing.
   - At index 2, the character is `'d'`, and we do nothing.
   - At index 3, we encounter `'('`, so we push 3 onto the stack.
   - At index 4, the character is `'e'`, and we do nothing.
   - At index 5, we encounter `'t'`, and we do nothing.
   - At index 6, the character is `'('`, so we push 6 onto the stack.
   - At index 7, we encounter `'o'`, and again, we do nothing.
   - At index 8, we encounter `'c'`, and we do nothing.
   - At index 9, we encounter the closing parenthesis `')'`. We pop 6 from the stack, which is the index of the matching opening parenthesis. In `d`, we set `d[6]` to 9 and `d[9]` to 6 to create a 'jump' between these indices.
   - At index 10, we encounter `')'`, so we pop 3 from the stack and in `d` set `d[3]` to 10 and `d[10]` to 3.
   - At index 11, we encounter another `')'`, so we pop 0 from the stack and in `d` set `d[0]` to 11 and `d[11]` to 0.
   - At index 12, we have `'l'`, and we do nothing.
3. Now we start traversing the string with `i = 0` and `x = 1`.
   - We skip the parenthesis at `s[0]` by jumping to `s[d[0]]`, which is `s[11]` and change the direction of `x` to `-1`.
   - We continue to traverse backward, appending 'l', 'e', skipping `s[10]` by jumping to `s[d[10]]` which is `s[3]`, append 'e', 't', skipping `s[6]` by jumping to `s[d[6]]` which is `s[9]`, append 'c', 'o', then move backward from `s[6]` since our direction is still `-1` and append 't', 'e'.
   - Eventually, we hit `s[0]` again, and we reverse the direction to forward (`x = 1`) and continue appending to `ans` from `s[1]` onwards.
4. After `i` has traversed all characters, the list `ans` consists of the characters: `['l', 'e', 'e', 't', 'c', 'o', 'd', 'e']`.
5. Finally, we join all elements in `ans` to form the final string `'leetcode'`, which is returned as the result.

The given example successfully demonstrates the clever use of a stack to track the opening parentheses and a jump array to manage traversal effectively, allowing us to unfold and reverse nested string structures efficiently.

Python Solution

```python
class Solution:
    def reverseParentheses(self, s: str) -> str:
        length_of_string = len(s)
        # Create a mapping array of the same length as the input string
        mapping = [0] * length_of_string
        # Stack to hold the indexes of the opening parentheses
        stack = []

        # For each character in the string with its index
        for index, character in enumerate(s):
            if character == '(':
                # If it's an opening bracket, append its index to stack
                stack.append(index)
            elif character == ')':
                # If it's a closing bracket, pop the last opening bracket's index
                opening_index = stack.pop()
                # Update the mapping array where the closing bracket points to the opening bracket and vice versa
                mapping[index], mapping[opening_index] = opening_index, index

        # Initialize pointers and direction
        index = 0
        direction = 1
        # List to accumulate the characters of the final answer
        result = []
        # Iterate over the input string
        while index < length_of_string:
            if s[index] in '()':
                # If the current character is a parenthesis,
                # flip the direction and jump to the corresponding parenthesis
                index = mapping[index]
                direction = -direction
            else:
                # Else, append the current character to the result
                result.append(s[index])
            # Move to the next character considering the current direction
            index += direction

        # Combine the characters in the result to form the final string
        return ''.join(result)

```

Java Solution

```java
class Solution {
    public String reverseParentheses(String s) {
        int length = s.length(); // Length of the input string.
        int[] pairIndex = new int[length]; // Array to keep track of the indices of matching parentheses.
        Deque<Integer> stack = new ArrayDeque<>(); // Stack to hold the indices of the '(' characters.

        // First pass: Find pairs of parentheses and record their indices.
        for (int i = 0; i < length; ++i) {
            if (s.charAt(i) == '(') {
                // When we encounter an opening parenthesis, we push its index onto the stack.
                stack.push(i);
            } else if (s.charAt(i) == ')') {
                // When we encounter a closing parenthesis, we pop the index of the corresponding
                // opening parenthesis from the stack and record the pairing in the pairIndex array.
                int j = stack.pop();
                pairIndex[i] = j;
                pairIndex[j] = i;
            }
        }

        StringBuilder result = new StringBuilder(); // StringBuilder to construct the resulting string.
        int index = 0; // Current index in the input string.
        int direction = 1; // Direction of iteration: 1 for forward, -1 for backward.

        // Second pass: Construct the result using the paired indices to reverse substrings as needed.
        while (index < length) {
            if (s.charAt(index) == '(' || s.charAt(index) == ')') {
                // If the current character is a parenthesis, we switch direction and jump to its pair.
                index = pairIndex[index];
                direction = -direction;
            } else {
                // Otherwise, we append the current character to the result.
                result.append(s.charAt(index));
            }
            index += direction; // Move to the next character in the current direction.
        }

        return result.toString(); // Convert the StringBuilder to a String and return it.
    }
}

```

C++ Solution

```cpp
#include <stack>
#include <vector>
#include <string>

class Solution {
public:
    // Function for reversing substrings within parentheses.
    std::string reverseParentheses(std::string s) {
        int length = s.size(); // Length of the input string.
        std::vector<int> pairedIndex(length); // This vector holds paired indices of parentheses.
        std::stack<int> openParensStack; // Stack to keep track of the positions of '('.

        // First pass: identify and record positions of paired parentheses
        for (int i = 0; i < length; ++i) {
            if (s[i] == '(') {
                // When we encounter '(', push the index on the stack.
                openParensStack.push(i);
            } else if (s[i] == ')') {
                // For ')', pop the top of the stack to find the matching '('.
                int matchedIndex = openParensStack.top();
                openParensStack.pop();
                pairedIndex[i] = matchedIndex; // Set the pair for ')'.
                pairedIndex[matchedIndex] = i; // Set the pair for '('.
            }
        }

        std::string result; // This will hold the final output string.
        int index = 0; // Current index in the string.
        int direction = 1; // Direction of traversal. 1 for forward, -1 for backward.

        // Second pass: build the result string by navigating through the pairs.
        while (index < length) {
            if (s[index] == '(' || s[index] == ')') {
                // On hitting a parenthesis, jump to its pair and invert direction
                index = pairedIndex[index];
                direction = -direction;
            } else {
                // Otherwise, just add the current character to the result
                result.push_back(s[index]);
            }
            index += direction; // Move in the current direction.
        }

        return result; // Return the final reversed string.
    }
};

```

Typescript Solution

```typescript
/**
 * Takes a string s, reverses the substrings that are within each pair of parentheses,
 * and returns a single string as a result.
 * @param {string} s - The input string containing parentheses and other characters
 * @return {string} - The string with the parentheses' contents reversed
 */
function reverseParentheses(s: string): string {
    // Length of the input string
    const length: number = s.length;
    // Array to keep track of mirror indices of parentheses
    const mirror: number[] = new Array(length).fill(0);
    // Stack to keep indices of opening parentheses
    const stack: number[] = [];

    // Loop through the string to find and record mirror indices
    for (let i = 0; i < length; ++i) {
        if (s[i] === '(') {
            // Push the index of the opening parenthesis onto the stack
            stack.push(i);
        } else if (s[i] === ')') {
            // Pop the stack to get the matching opening parenthesis index
            const j = stack.pop() as number; // as number is safe here, because stack is guaranteed to be non-empty
            // Record the mirror index for both parentheses
            mirror[i] = j;
            mirror[j] = i;
        }
    }

    // Initialize the index and step variables for traversing the string
    let index: number = 0;
    let step: number = 1;
    // Initialize the result array to construct the reversed string
    const result: string[] = [];

    // Traverse the string to construct the result while handling parentheses
    while (index < length) {
        const char: string = s.charAt(index);
        if (char === '(' || char === ')') {
            // Jump to mirror index and reverse traversal direction upon encountering a parenthesis
            index = mirror[index];
            step = -step;
        } else {
            // Otherwise, add the current character to the result array
            result.push(char);
        }
        // Move to the next character in the specified direction
        index += step;
    }

    // Return the reconstructed string without parentheses
    return result.join('');
}

```

Time and Space Complexity

The time complexity of the given code can be analyzed as follows:

1. Iterating over the string `s` to populate the `d` array and handling the stack operations takes `O(n)` time, where `n` is the length of the string. This is because each character in the string is visited once.
2. The second while loop also takes `O(n)` time as it iterates over each character of the string and performs constant time operations (unless `i` is set to a previously visited index, but due to the nature of the problem this cannot lead to an overall time complexity worse than `O(n)`).

Thus, the total time complexity of the algorithm is `O(n)`.

The space complexity of the given code can be analyzed as follows:

1. The `d` array takes `O(n)` space to store the indices that should be jumped to when a parenthesis is encountered.
2. The stack `stk` takes at most `O(n/2)` space in the case where half of the string consists of `'('` characters before any `')'` character is encountered. However, `O(n/2)` simplifies to `O(n)`.
3. The list `ans` also grows to hold `O(n)` characters, as it holds the result string with parentheses characters removed.

Therefore, the total space complexity of the algorithm is `O(n)` (since `O(n) + O(n)` still simplifies to `O(n)`).