1249. Minimum Remove to Make Valid Parentheses

Problem Description

Stack

String

Medium

The given problem presents us with a string s that consists of parentheses ('(' and ')') and lowercase English letters. Our goal is to make the string a valid parentheses string by removing the fewest number of parentheses possible. The criteria for a valid parentheses string is:

Leetcode Link

It is an empty string, or

It contains only lowercase characters, or

- It can be written in the form of AB, where A and B are valid strings, or It can be written in the form of (A), where A is a valid string.
- closing parentheses without a matching opening one.

Intuition The solution can be devised based on the properties of a valid parentheses sequence, where the number of opening '(' and closing

Ultimately, we must ensure that every opening parenthesis '(' has a corresponding closing parenthesis ')', and there are no extra

')' parentheses are the same, and at any point in the sequence, there can't be more closing parenthesis before an opening one.

To approach this problem, we need to monitor the balance of the parentheses and remove any excess closing or opening parentheses. A stack can be beneficial in such situations, but this solution optimizes space by using two passes through the string and counting variables to keep track of the balance.

parentheses to the stack if they maintain the balance, but we increment or decrement the counting variable x based on whether we encounter an opening '(' or a closing ')'. This ensures we don't add any extra closing parentheses. However, after this first pass, we might still have excess opening parentheses, which become evident when looked at from the

In the first pass, we iterate over the string from left to right, ignoring any characters that are not parentheses. We only add

opposite direction (since they won't have a corresponding closing parenthesis). Hence, we conduct a second pass from right to left, this time using a similar approach to selectively keep valid parentheses and maintaining the count in the opposite manner. By reversing this result, we achieve a string where the parentheses are balanced.

Finally, we combine the characters into a string and return it as the valid parentheses string with the minimal removal of invalid parentheses characters.

• Ensuring that every closing parenthesis has a corresponding opening parenthesis to its left. No stack data structure is used to save space; instead, a counter is employed to ensure the balance of parentheses as we iterate through the string.

1. Left-to-right pass: We iterate through the string, and for each character:

Solution Approach

The solution relies on two main ideas:

∘ If it's a closing parenthesis ')' and the current balance x is zero, we skip adding it to the stack because there's no corresponding opening parenthesis.

parentheses, which the second pass addresses.

corresponding closing parenthesis.

The algorithm employs two passes:

o If it's an opening parenthesis '(', we increment the counter x since we have an additional opening parenthesis that needs a match.

• Ensuring that every opening parenthesis has a corresponding closing parenthesis to its right.

∘ If it's a closing parenthesis ')', we decrement x because we have found a potential match for an earlier opening parenthesis.

• We add all non-parenthesis characters and valid parenthesis characters to stk.

∘ If it's a closing parenthesis ')', we increment the counter x since we need to match it with an opening parenthesis. ∘ If it's an opening parenthesis '(', we decrement x since we found a match for a closing parenthesis. We collect the characters to form the answer, skipping the unmatched opening parentheses.

After this pass, we have a string that contains no unmatched parentheses, and by reversing the result, we get the valid parentheses

By only keeping track of the balance of the parentheses and using two linear passes, we efficiently construct a balanced string by

ignoring characters that would violate the balance. This approach avoids the overhead of using extra data structures like a stack.

∘ If it's an opening parenthesis '(' and the current balance x is zero, we skip adding it to the answer because there's no

This pass ensures that we don't have unmatched closing parentheses at this stage. However, we may still have unmatched opening

Example Walkthrough

We start with an empty stk and a balance counter x set to 0.

4. '(' - It is an opening parenthesis, so we increment x (x=1) and add it to stk.

6. ')' - It is a closing parenthesis, and x is 1, so we decrement x (x=0) and add it to stk.

8. ')' - Again, x is 0, so this closing parenthesis does not have a match. We do not add it to stk.

1. 'a' - It is a lowercase letter, so we add it to stk.

5. 'c' - It is a lowercase letter, so we add it to stk.

7. 'd' - It is a lowercase letter, so we add it to stk.

Now, we iterate through each character in reverse order:

We iterate over each character of the string:

2. Right-to-left pass: We reverse the stk and then iterate through it:

- Let's consider an example string s: "a)b(c)d)". We want to remove the fewest number of parentheses to make this a valid parentheses string. Let's apply the solution approach to this string step by step.
- 2. ')' The balance x is 0, meaning there's no open parenthesis to match with, so we do not add this to stk. 3. 'b' - It is a lowercase letter, so we add it to stk.

Right-to-left pass:

number of parentheses.

class Solution:

10

11

17

18

19

20

22

23

24

25

26

27

28

29

36

37

38

39

40

41

9

13

14

15

16

19

20

21

24

25

26

27

28

29

30

31

32

33

39

43

44

45

46

47

48

49

50

51

52

53 }

according to the provided criteria.

stack = []

open count = 0

for char in s:

open_count = 0

answer = []

elif char == ')':

stack.append(char)

for char in reversed(stack):

elif char == '(':

answer.append(char)

int openCount = 0;

open_count -= 1

open_count -= 1

Left-to-right pass:

string.

4. 'c' - It is a lowercase letter, so it remains part of the final string. 5. ')' - It is a closing parenthesis, so we increment x (x=1) and keep it since we expect to find a matching opening parenthesis.

At the end of the first pass, stk is ["a","b","(","c",")","d"], and x is 0.

We reverse stk to get ["d",")","c","(","b","a"] and set x to 0 again.

1. 'a' - It is a lowercase letter, so it remains part of the final string.

2. 'b' - It is a lowercase letter, so it remains part of the final string.

6. 'd' - It is a lowercase letter, so it remains part of the final string.

3. '(' - x is 0, so this has no matching closing parenthesis and should be removed.

Python Solution

Stack to keep track of characters that lead to a valid string

def minRemoveToMakeValid(self, s: str) -> str:

Counter to track the balance of parentheses

Reset the open counter for the second pass

List to store the characters for the final answer

// Use a deque as a stack to manage parentheses.

// First pass: remove invalid closing parentheses.

if (current == ')' && openCount == 0) {

// Push the current character onto the stack.

// StringBuilder to construct the output string.

// Second pass: remove invalid opening parentheses.

// Append the current character to the result.

// Reverse the result string to restore the original order

// since the second pass processed characters in reverse.

StringBuilder result = new StringBuilder();

// Reset openCount for the second pass.

// Counter to keep track of unmatched opening parentheses.

// Skip this character if it's an unmatched closing parenthesis.

// If a valid opening parenthesis is found, increment openCount.

// If a valid closing parenthesis is found, decrement openCount.

// If an opening parenthesis is found, decrement openCount.

Deque<Character> stack = new ArrayDeque<>();

for (int i = 0; i < s.length(); ++i) {</pre>

char current = s.charAt(i);

} else if (current == ')') {

continue;

if (current == '(') {

++openCount;

--openCount;

stack.push(current);

while (!stack.isEmpty()) {

if (current == ')') {

++openCount;

--openCount;

result.append(current);

return result.reverse().toString();

} else if (current == '(') {

openCount = 0;

First pass to remove invalid closing parentheses

If a closing parenthesis is encountered with no matching open, skip it if char == ')' and open_count == 0: 12 continue 13 14 # If an opening parenthesis is found, increment the open count if char == '(': 16 open_count += 1

If a closing parenthesis is found and there is a matching open, decrement the open count

Add the character to the stack as it's part of valid substring so far

Second pass to remove invalid opening parentheses; process characters in reverse

Add the character to the answer as it is part of a valid substring

After reversing the result of this pass, we get "ab(c)d" which indeed is the valid parentheses string after removing the fewest

This example clearly shows how the two passes effectively remove unnecessary parentheses while keeping the string valid

30 # If an opening parenthesis is encountered with no matching close, skip it if char == '(' and open_count == 0: 31 continue 32 33 # If a closing parenthesis is found, increment the open count if char == ')': 34 35 open_count += 1

If an opening parenthesis is found and there is a matching close, decrement the open count

- 42 # The characters in answer are in reverse order, reverse them back to the correct order return ''.join(reversed(answer))
- Java Solution class Solution { public String minRemoveToMakeValid(String s) {

char current = stack.pop(); // Skip this character if it's an unmatched opening parenthesis. 35 if (current == '(' && openCount == 0) { 36 continue; 37 38 // If a closing parenthesis is found, increment openCount.

54 C++ Solution #include <algorithm> // Required for std::reverse class Solution { public: // Function to remove the minimum number of parentheses to make a valid parentheses string. string minRemoveToMakeValid(string s) { string tempStack; // Simulates a stack to hold valid characters. int openCount = 0; // Counter for unmatched '(' characters. 8 9 // First pass: Remove any ')' that doesn't have a matching '('. 10 for (char& c : s) { 11 if (c == ')' && openCount == 0) continue; // Skip invalid ')'. 12 13 openCount++; // Increment counter for '('. } else if (c == ')') { 15 openCount--; // Decrement counter for ')'. 16 17 tempStack.push_back(c); // Add the character to the stack. 18 19 20 21 string result; // String that will store the valid parentheses sequence. 22 openCount = 0; // Reset counter for the next pass. 23 24 // Second pass: Remove any '(' that doesn't have a matching ')' from the end. 25 while (!tempStack.empty()) { 26 char c = tempStack.back(); // Take the last character from the stack. tempStack.pop_back(); // Remove that character from the stack. 27 28 29 if (c == '(' && openCount == 0) continue; // Skip invalid '('. if (c == ')') { 30 31 openCount++; // Increment counter for ')'. 32 } else if (c == '(') { 33 openCount--; // Decrement counter for '('. 34 35 result.push_back(c); // Add the character to the result string. 36 37 38 // The result is built in reverse order, so we need to reverse it. 39 std::reverse(result.begin(), result.end()); 40

26 27 28 if (matchedLeftCount < rightToMatch) {</pre> 29 30 matchedLeftCount++;

} else {

return result;

result += char;

rightToMatch--;

result += char;

matchedLeftCount--;

} else if (char === ')') {

result += char;

Time and Space Complexity

track of parentheses and a counter to validate them.

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

48 }

Time Complexity The time complexity of this algorithm is O(n), where n is the length of the string s. It performs two separate passes through the string: The first for-loop iterates through each character of the input string once to count the parentheses and remove invalid closing

After the first loop, the stack stk contains a modified version of the string without excess closing parentheses.

parentheses in the reverse direction and remove the invalid opening parentheses.

• The second for-loop iterates through the modified string (stack stk) in reverse, again once for each element, to count the

The given Python code processes a string s to remove the minimum number of parentheses to make the input string valid (all

opened parentheses must be closed). The code uses two passes through the string, forward and backward, utilizing stacks to keep

// If there is a '(' we can match with, we include the ')' in the result.

// For any character that is not a parenthesis, we always include it in the result.

if (matchedLeftCount !== 0 && rightToMatch !== 0) {

// Return the final result string with all valid matched parentheses.

Each character is processed once in each direction, resulting in a total of 2 * n operations, which simplifies to 0(n) as constants are dropped in Big O notation.

parentheses.

- **Space Complexity** The space complexity of the code is also O(n), due to the use of additional data structures that store elements proportional to the size of the input string.
- The stk list is used to store the characters of the string after the first pass. In the worst case, it could store the entire string if no closing parentheses need to be removed, leading to O(n) space. The ans list is used to store the characters after the second pass has checked for the validity of the remaining opening
- parentheses. Similarly, in the worst case, it could store the entire string if no opening parentheses need to be removed, also leading to O(n) space. Even though we use two stacks (stk and ans), they do not exist at the same time, as ans is only created after stk's processing is
- completed. Therefore, we don't consider this as 2n but simply n, and the space complexity remains 0(n). Note: The space needed for counters and individual characters is constant and therefore is not included in the space complexity

analysis. The space complexity is determined by the significant data structures that grow with the input size.

return result; // Return the valid parentheses string. 41 42 43 **}**; 44 Typescript Solution function minRemoveToMakeValid(inputString: string): string { // Count how many '(' must be matched. let leftToMatch = 0; // Count how many ')' could be possibly matched. let rightToMatch = 0; // First pass to calculate the minimum number of ')' to match for (const char of inputString) { **if** (char === '(') { // Increment if we find a '(' that potentially could be matched later. leftToMatch++; } else if (char === ')') { // Only increment the rightToMatch if there are unmatched '(' available. if (rightToMatch < leftToMatch) {</pre> rightToMatch++; // Reset the count of '(' that has been matched so far. let matchedLeftCount = 0; // Initialize the result string to be built. let result = ''; // Second pass to build the result string with matched parentheses for (const char of inputString) { **if** (char === '(') { // If we still have '(' that can be matched, we take it and add to the result string.