## 1249. Minimum Remove to Make Valid Parentheses Medium Stack

**Problem Description** 

make the string a valid parentheses string by removing the fewest number of parentheses possible. The criteria for a valid parentheses string is: It is an empty string, or

The given problem presents us with a string s that consists of parentheses ('(' and ')') and lowercase English letters. Our goal is to

- 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.

It contains only lowercase characters, or

String

- Ultimately, we must ensure that every opening parenthesis '(' has a corresponding closing parenthesis ')', and there are no extra
- closing parentheses without a matching opening one.

and counting variables to keep track of the balance.

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

')' 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

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

Solution Approach The solution relies on two main ideas: Ensuring that every opening parenthesis has a corresponding closing parenthesis to its right.

No stack data structure is used to save space; instead, a counter is employed to ensure the balance of parentheses as we iterate

# The algorithm employs two passes:

through the string.

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

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

parentheses string. Let's apply the solution approach to this string step by step.

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

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

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

corresponding closing parenthesis. ∘ 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.

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.

Left-to-right pass:

3. 'b' - It is a lowercase letter, so we add it to stk. 4. '(' - It is an opening parenthesis, so we increment x (x=1) and add it to stk. 5. 'c' - It is a lowercase letter, so we add it to stk.

## We reverse stk to get ["d",")","c","(","b","a"] and set x to 0 again. Now, we iterate through each character in reverse order:

number of parentheses.

10

11

12

13

14

16

17

24

32

33

34

35

37

38

39

40

42

14

15

16

19

20

21

24

25

26

27

according to the provided criteria.

for char in s:

open\_count = 0

continue

**if** char == '(':

open\_count += 1

return ''.join(reversed(answer))

Right-to-left pass:

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

  - class Solution: def minRemoveToMakeValid(self, s: str) -> str: # Stack to keep track of characters that lead to a valid string stack = []
- elif char == ')': 18 open\_count -= 1 19 # Add the character to the stack as it's part of valid substring so far 20 21 stack.append(char) 22

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

25 # List to store the characters for the final answer 26 answer = [] 27 28 # Second pass to remove invalid opening parentheses; process characters in reverse 29 for char in reversed(stack): # If an opening parenthesis is encountered with no matching close, skip it 30 if char == '(' and open\_count == 0: 31

# The characters in answer are in reverse order, reverse them back to the correct order

# If a closing parenthesis is encountered with no matching open, skip it

# If an opening parenthesis is found, increment the open count

- class Solution { public String minRemoveToMakeValid(String s) { // Use a deque as a stack to manage parentheses. Deque<Character> stack = new ArrayDeque<>(); // Counter to keep track of unmatched opening parentheses. int openCount = 0; // First pass: remove invalid closing parentheses. for (int i = 0; i < s.length(); ++i) {</pre> 9 char current = s.charAt(i); 10 // Skip this character if it's an unmatched closing parenthesis. if (current == ')' && openCount == 0) { 13
- 37 38 // If a closing parenthesis is found, increment openCount. 39 **if** (current == ')') { ++openCount; } else if (current == '(') { // If an opening parenthesis is found, decrement openCount. 43 --openCount; 44 // Append the current character to the result. 45 result.append(current); 46 47 48
- 35 36 37 38 // The result is built in reverse order, so we need to reverse it. 39 std::reverse(result.begin(), result.end());

class Solution {

public:

8

9

10

11

12

13

15

16

17

18

19

20

21

22

23

24

25

40

41

42

44

43 };

### 11 13 14 if (rightToMatch < leftToMatch) {</pre> 15 rightToMatch++;

16

17

18

19

20

21

22

23

24

35

36

37

38

39

40

41

42

43

44

45

46

48 }

opened parentheses must be closed). The code uses two passes through the string, forward and backward, utilizing stacks to keep track of parentheses and a counter to validate them. **Time Complexity** 

The time complexity of this algorithm is O(n), where n is the length of the string s.

- parentheses in the reverse direction and remove the invalid opening 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. **Space Complexity**

The first for-loop iterates through each character of the input string once to count the parentheses and remove invalid closing

- 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.
- 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).

parentheses characters.

1. Left-to-right pass: We iterate through the string, and for each character: If it's a closing parenthesis ')' and the current balance x is zero, we skip adding it to the stack because there's no

match. parenthesis.

Example Walkthrough

parentheses, which the second pass addresses.

- 2. Right-to-left pass: We reverse the stk and then iterate through it: ∘ If it's an opening parenthesis '(' and the current balance x is zero, we skip adding it to the answer because there's no
- After this pass, we have a string that contains no unmatched parentheses, and by reversing the result, we get the valid parentheses string.

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

2. ')' - The balance x is 0, meaning there's no open parenthesis to match with, so we do not add this to stk.

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

- We start with an empty stk and a balance counter x set to 0. We iterate over each character of the string: 1. 'a' - It is a lowercase letter, so we 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. At the end of the first pass, stk is ["a","b","(","c",")","d"], and x is 0.

5. ')' - It is a closing parenthesis, so we increment x (x=1) and keep it since we expect to find a matching opening parenthesis.

6. 'd' - It is a lowercase letter, so it remains part of the final string. After reversing the result of this pass, we get "ab(c)d" which indeed is the valid parentheses string after removing the fewest

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.

4. 'c' - 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.

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

Python Solution

if char == ')' and open\_count == 0:

- # Counter to track the balance of parentheses open\_count = 0 # First pass to remove invalid closing parentheses
- 23 # Reset the open counter for the second pass
- continue # If a closing parenthesis is found, increment the open count **if** char == ')': open\_count += 1 36 # If an opening parenthesis is found and there is a matching close, decrement the open count elif char == '(': open\_count -= 1 # Add the character to the answer as it is part of a valid substring answer.append(char) 41
- Java Solution
  - continue; // If a valid opening parenthesis is found, increment openCount.
  - **if** (current == '(') { ++openCount; } else if (current == ')') { // If a valid closing parenthesis is found, decrement openCount. --openCount; // Push the current character onto the stack. stack.push(current); // StringBuilder to construct the output string. StringBuilder result = new StringBuilder();
- // Reset openCount for the second pass. 28 openCount = 0; 30 31 // Second pass: remove invalid opening parentheses. 32 while (!stack.isEmpty()) { char current = stack.pop(); // Skip this character if it's an unmatched opening parenthesis. 35 if (current == '(' && openCount == 0) { 36 continue;
- // Reverse the result string to restore the original order 49 // since the second pass processed characters in reverse. 50 51 return result.reverse().toString(); 52 53 } 54 C++ Solution #include <algorithm> // Required for std::reverse

string tempStack; // Simulates a stack to hold valid characters.

// First pass: Remove any ')' that doesn't have a matching '('.

if (c == ')' && openCount == 0) continue; // Skip invalid ')'.

tempStack.push\_back(c); // Add the character to the stack.

string result; // String that will store the valid parentheses sequence.

// Second pass: Remove any '(' that doesn't have a matching ')' from the end.

int openCount = 0; // Counter for unmatched '(' characters.

openCount++; // Increment counter for '('.

openCount--; // Decrement counter for ')'.

openCount = 0; // Reset counter for the next pass.

return result; // Return the valid parentheses string.

// Function to remove the minimum number of parentheses to make a valid parentheses string.

char c = tempStack.back(); // Take the last character from the stack. 26 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 result.push\_back(c); // Add the character to the result string.

1 function minRemoveToMakeValid(inputString: string): string {

// Count how many ')' could be possibly matched.

// First pass to calculate the minimum number of ')' to match

// Reset the count of '(' that has been matched so far.

// Initialize the result string to be built.

rightToMatch--;

result += char;

result += char;

matchedLeftCount--;

// Count how many '(' must be matched.

while (!tempStack.empty()) {

string minRemoveToMakeValid(string s) {

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

for (char& c : s) {

for (const char of inputString) { **if** (char === '(') { 9 // Increment if we find a '(' that potentially could be matched later. 10 leftToMatch++; } else if (char === ')') { // Only increment the rightToMatch if there are unmatched '(' available.

let matchedLeftCount = 0;

let result = '';

} else {

return result;

parentheses.

let leftToMatch = 0;

let rightToMatch = 0;

Typescript Solution

25 // Second pass to build the result string with matched parentheses 26 for (const char of inputString) { 27 **if** (char === '(') { // If we still have '(' that can be matched, we take it and add to the result string. 28 29 if (matchedLeftCount < rightToMatch) {</pre> matchedLeftCount++; 30 result += char; 31 32 33 } else if (char === ')') { 34 // If there is a '(' we can match with, we include the ')' in the result.

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

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

Time and Space Complexity

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

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

- After the first loop, the stack stk contains a modified version of the string without excess closing parentheses. • The second for-loop iterates through the modified string (stack stk) in reverse, again once for each element, to count the

It performs two separate passes through the string:

- 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
- 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.