32. Longest Valid Parentheses

Summary

We need to determine the length of the largest valid substring of parentheses from a given string.

Solution

Approach 1: Brute Force

Algorithm

In this approach, we consider every possible non-empty even length substring from the given string and check whether it's a valid string of parentheses or not. In order to check the validity, we use the Stack's Method.

Every time we encounter a '(', we push it onto the stack. For every ')' encountered, we pop a '(' from the stack. If '(' isn't available on the stack for popping at anytime or if stack contains some elements after processing complete substring, the substring of parentheses is invalid. In this way, we repeat the process for every possible substring and we keep on storing the length of the longest valid string found so far.

```
Example:
"((())"

(( --> invalid
(( --> invalid
() --> valid, length=2
)) --> invalid
((()--> invalid
(())--> valid, length=4
maxlength=4
```

Java Copy

Complexity Analysis

- Time complexity : $O(n^3)$. Generating every possible substring from a string of length n requires $O(n^2)$. Checking validity of a string of length n requires $O(n)$.
- Space complexity : $O(n)$. A stack of depth n will be required for the longest substring.

Approach 2: Using Dynamic Programming

Algorithm

This problem can be solved by using Dynamic Programming. We make use of a dp array where ith element of dp represents the length of the longest valid substring ending at ith index. We initialize the complete dp array with 0's. Now, it's obvious that the valid substrings must end with ‘)’. This further leads to the conclusion that the substrings ending with ‘(’ will always contain '0' at their corresponding dp indices. Thus, we update the dp array only when ‘)’ is encountered.

To fill dp array we will check every two consecutive characters of the string and if

1. $\text{s}[i] = \text{‘)’}$ and $\text{s}[i-1] = \text{‘(’}$, i.e. string looks like $\text{``.......()"} \Rightarrow$

$$\text{dp}[i] = \text{dp}[i-2] + 2$$

   We do so because the ending "()" portion is a valid substring anyhow and leads to an increment of 2 in the length of the just previous valid substring's length.

2. $\text{s}[i] = \text{‘)’}$ and $\text{s}[i-1] = \text{‘)’}$, i.e. string looks like $\text{``.......))"} \Rightarrow$

   if $\text{s}[i - \text{dp}[i-1] - 1] = \text{‘(’}$ then

$$\text{dp}[i] = \text{dp}[i-1] + \text{dp}[i - \text{dp}[i-1] - 2] + 2$$

The reason behind this is that if the 2nd last ‘)’ was a part of a valid substring (say sub_s), for the last ‘)’ to be a part of a larger substring, there must be a corresponding starting ‘(’ which lies before the valid substring of which the 2nd last ‘)’ is a part (i.e. before sub_s). Thus, if the character before sub_s happens to be ‘(’, we update the $\text{dp}[i]$ as an addition of 2 in the length of sub_s which is $\text{dp}[i-1]$. To this, we also add the length of the valid substring just before the term "(,sub_s,)" , i.e. $\text{dp}[i - \text{dp}[i-1] - 2]$.

For better understanding of this method, see this example:

1 / 8

Java

Copy

Complexity Analysis

- Time complexity : $O(n)$. Single traversal of string to fill dp array is done.
- Space complexity : $O(n)$. dp array of size n is used.

Approach 3: Using Stack

Algorithm

Instead of finding every possible string and checking its validity, we can make use of stack while scanning the given string to check if the string scanned so far is valid, and also the length of the longest valid string. In order to do so, we start by pushing -1 onto the stack.

For every '$($' encountered, we push its index onto the stack.

For every '$)$' encountered, we pop the topmost element and subtract the current element's index from the top element of the stack, which gives the length of the currently encountered valid string of parentheses. If while popping the element, the stack becomes empty, we push the current element's index onto the stack. In this way, we keep on calculating the lengths of the valid substrings, and return the length of the longest valid string at the end.

See this example for better understanding.

1 / 11

Java

Copy

Complexity Analysis

- Time complexity : $O(n)$. n is the length of the given string..
- Space complexity : $O(n)$. The size of stack can go up to n.

Approach 4: Without extra space

Algorithm

In this approach, we make use of two counters $left$ and $right$. First, we start traversing the string from the left towards the right and for every $‘(’$ encountered, we increment the $left$ counter and for every $‘)’$ encountered, we increment the $right$ counter. Whenever $left$ becomes equal to $right$, we calculate the length of the current valid string and keep track of maximum length substring found so far. If $right$ becomes greater than $left$ we reset $left$ and $right$ to 0.

Next, we start traversing the string from right to left and similar procedure is applied.

Example of this approach:

1 / 21

Java

Copy

Complexity Analysis

- Time complexity : $O(n)$. Two traversals of the string.

- Space complexity : $O(1)$. Only two extra variables $left$ and $right$ are needed.

Comments: 35

Sort By

Type comment here... (Markdown is supported)

Preview Post

appenthused0418 (/appenthused0418) ★ 1 2 days ago

(/appenthused0418)

I have to admit this is too hard for me..

0 Share Reply

muslimov (/muslimov) ★ 17 September 30, 2018 6:03 PM

(/muslimov)

In dynamic approach, actually, first case is corner case of second (`dp[i-1] =0`). Here my solution https://leetcode.com/problems/longest-valid-parentheses/discuss/176229/Python-readable-dp-solution (https://leetcode.com/problems/longest-valid-parentheses/discuss/176229/Python-readable-dp-solution)

0 Share Reply

sivanesanms (/sivanesanms) ★ 2 September 28, 2018 12:36 AM

(/sivanesanms)

For solution one, "Generating every possible substring from a string of length nn requires O(n^2)O(n2)."
How ??? Could someone explain?

0 Share Reply

bmunot9 (/bmunot9) ★ 20 September 16, 2018 8:05 PM

(/bmunot9)

I am finding it little difficult to digest the second part of 4th approach. I am convinced that it is required and I get it up to certain extent, but still not getting the precise intuition behind that. Could someone please explain?

6 Share Reply

SHOW 2 REPLIES

zem_nezer (/zem_nezer) ★ 1 September 4, 2018 5:47 PM

(/zem_nezer)

Anyone can explain why approach #3 can work? What is its mechanism?

1 Share Reply

SHOW 2 REPLIES

Timmy_Black (/timmy_black) ★ 2 September 4, 2018 6:59 AM

(/timmy_black)

I have my python solution beats 86.59% though just once...

Read More

0 Share Reply

meganlee (/meganlee) ★ 210 July 31, 2018 12:24 AM

(/meganlee)

solution 1 is `Time Limit Exceeded`

0 Share Reply