**32. Longest Valid Parentheses**

**Question**

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**Quick Navigation**

* [Summary](https://leetcode.com/articles/longest-valid-parentheses/#summary)
* [Solution](https://leetcode.com/articles/longest-valid-parentheses/#solution)
  + [Approach #1 Brute Force [Time Limit Exceeded]](https://leetcode.com/articles/longest-valid-parentheses/#approach-1-brute-force-time-limit-exceeded)
  + [Approach #2 Using Dynamic Programming [Accepted]](https://leetcode.com/articles/longest-valid-parentheses/#approach-2-using-dynamic-programming-accepted)
  + [Approach #3 Using Stack [Accepted]](https://leetcode.com/articles/longest-valid-parentheses/#approach-3-using-stack-accepted)
  + [Approach #4 Without extra space [Accepted]](https://leetcode.com/articles/longest-valid-parentheses/#approach-4-without-extra-space-accepted)

**Summary**

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

**Solution**

**Approach #1 Brute Force [Time Limit Exceeded]**

**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**

**public** **class** **Solution** **{**

**public** **boolean** **isValid(**String s**)** **{**

Stack**<**Character**>** stack **=** **new** Stack**<**Character**>();**

**for** **(int** i **=** 0**;** i **<** s**.**length**();** i**++)** **{**

**if** **(**s**.**charAt**(**i**)** **==** '('**)** **{**

stack**.**push**(**'('**);**

**}** **else** **if** **(!**stack**.**empty**()** **&&** stack**.**peek**()** **==** '('**)** **{**

stack**.**pop**();**

**}** **else** **{**

**return** **false;**

**}**

**}**

**return** stack**.**empty**();**

**}**

**public** **int** **longestValidParentheses(**String s**)** **{**

**int** maxlen **=** 0**;**

**for** **(int** i **=** 0**;** i **<** s**.**length**();** i**++)** **{**

**for** **(int** j **=** i **+** 2**;** j **<=** s**.**length**();** j**+=**2**)** **{**

**if** **(**isValid**(**s**.**substring**(**i**,** j**)))** **{**

maxlen **=** Math**.**max**(**maxlen**,** j **-** i**);**

**}**

**}**

**}**

**return** maxlen**;**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(n3)O(n^3)O(n​3​​). Generating every possible substring from a string of length nnn requires O(n2)O(n^2)O(n​2​​). Checking validity of a string of length nnn requires O(n)O(n)O(n).
* Space complexity : O(n)O(n)O(n). A stack of depth nnn will be required for the longest substring.

**Approach #2 Using Dynamic Programming [Accepted]**

**Algorithm**

This problem can be solved by using Dynamic Programming. We make use of a dp array where ith element of dp\text{dp}dp represents the length of the longest valid substring ending at iiith 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. s[i]=‘)’ and s[i−1]=‘(’, i.e. string looks like ‘‘.......()"⇒``.......()" ⇒

dp[i]=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.

1. s[i]=‘)’ and s[i−1]=‘)’, i.e. string looks like ‘‘.......))"⇒``.......))" ⇒

if s[i−dp[i−1]−1]=‘(’ then

dp[i]=dp[i−1]+dp[i−dp[i−1]−2]+2

The reason behind this is that if the 2nd last ‘)’ was a part of a valid substring (say subs), 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 subs ​​). Thus, if the character before subs happens to be ‘(’, we update the dp[i] as an addition of 2 in the length of subs ​​ which is dp[i−1]. To this, we also add the length of the valid substring just before the term "(,sub\_s,)" , i.e. dp[i−dp[i−1]−2].

For better understanding of this method, see this example:

**Java**

**public** **class** **Solution** **{**

**public** **int** **longestValidParentheses(**String s**)** **{**

**int** maxans **=** 0**;**

**int** dp**[]** **=** **new** **int[**s**.**length**()];**

**for** **(int** i **=** 1**;** i **<** s**.**length**();** i**++)** **{**

**if** **(**s**.**charAt**(**i**)** **==** ')'**)** **{**

**if** **(**s**.**charAt**(**i **-** 1**)** **==** '('**)** **{**

dp**[**i**]** **=** **(**i **>=** 2 **?** dp**[**i **-** 2**]** **:** 0**)** **+** 2**;**

**}** **else** **if** **(**i **-** dp**[**i **-** 1**]** **>** 0 **&&** s**.**charAt**(**i **-** dp**[**i **-** 1**]** **-** 1**)** **==** '('**)** **{**

dp**[**i**]** **=** dp**[**i **-** 1**]** **+** **((**i **-** dp**[**i **-** 1**])** **>=** 2 **?** dp**[**i **-** dp**[**i **-** 1**]** **-** 2**]** **:** 0**)** **+** 2**;**

**}**

maxans **=** Math**.**max**(**maxans**,** dp**[**i**]);**

**}**

**}**

**return** maxans**;**

**}**

**}**

**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 [Accepted]**

**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.

**Java**

**public** **class** **Solution** **{**

**public** **int** **longestValidParentheses(**String s**)** **{**

**int** maxans **=** 0**;**

Stack**<**Integer**>** stack **=** **new** Stack**<>();**

stack**.**push**(-**1**);**

**for** **(int** i **=** 0**;** i **<** s**.**length**();** i**++)** **{**

**if** **(**s**.**charAt**(**i**)** **==** '('**)** **{**

stack**.**push**(**i**);**

**}** **else** **{**

stack**.**pop**();**

**if** **(**stack**.**empty**())** **{**

stack**.**push**(**i**);**

**}** **else** **{**

maxans **=** Math**.**max**(**maxans**,** i **-** stack**.**peek**());**

**}**

**}**

**}**

**return** maxans**;**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(n). nnn 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 [Accepted]**

**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 000.

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

Example of this approach:

**Java**

**public** **class** **Solution** **{**

**public** **int** **longestValidParentheses(**String s**)** **{**

**int** left **=** 0**,** right **=** 0**,** maxlength **=** 0**;**

**for** **(int** i **=** 0**;** i **<** s**.**length**();** i**++)** **{**

**if** **(**s**.**charAt**(**i**)** **==** '('**)** **{**

left**++;**

**}** **else** **{**

right**++;**

**}**

**if** **(**left **==** right**)** **{**

maxlength **=** Math**.**max**(**maxlength**,** 2 **\*** right**);**

**}** **else** **if** **(**right **>=** left**)** **{**

left **=** right **=** 0**;**

**}**

**}**

left **=** right **=** 0**;**

**for** **(int** i **=** s**.**length**()** **-** 1**;** i **>=** 0**;** i**--)** **{**

**if** **(**s**.**charAt**(**i**)** **==** '('**)** **{**

left**++;**

**}** **else** **{**

right**++;**

**}**

**if** **(**left **==** right**)** **{**

maxlength **=** Math**.**max**(**maxlength**,** 2 **\*** left**);**

**}** **else** **if** **(**left **>=** right**)** **{**

left **=** right **=** 0**;**

**}**

**}**

**return** maxlength**;**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(n). Two traversals of the string.
* Space complexity : O(1). Only two extra variables left and right are needed.