Length of the longest valid substring

Given a string consisting of opening and closing parenthesis, find length of the longest valid parenthesis substring.

Examples:

```
Input : (((()
Output : 2
Explanation : ()

Input: )()())
Output : 4
Explanation: ()()

Input:  ()((((((
Output: 6
Explanation:  ()((()))
```

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Approach** is to find all the substrings of given string. For every string, check if it is a valid string or not. If valid and length is more than maximum length so far, then update maximum length. We can check whether a substring is valid or not in linear time using a stack (See this for details). Time complexity of this solution is $O(n^2$.

An **Efficient Solution** can solve this problem in O(n) time. The idea is to store indexes of previous starting brackets in a stack. The first element of stack is a special element that provides index before beginning of valid substring (base for next valid string).

```
1) Create an empty stack and push -1 to it. The first element
   of stack is used to provide base for next valid string.

2) Initialize result as 0.

3) If the character is '(' i.e. str[i] == '(', push index
   'i' to the stack.

2) Else (if the character is ')')
   a) Pop an item from stack (Most of the time an opening bracket)
   b) If stack is not empty, then find length of current valid
      substring by taking difference between current index and
      top of the stack. If current length is more than result,
      then update the result.
   c) If stack is empty, push current index as base for next
      valid substring.

3) Return result.
```

Below are C++ and Python implementations of above algorithm.

C++

```
// C++ program to find length of the longest valid
// substring
#include<bits/stdc++.h>
using namespace std;

int findMaxLen(string str)
{
    int n = str.length();

    // Create a stack and push -1 as initial index to it.
    stack<int> stk;
    stk.push(-1);

    // Initialize result
    int result = 0;

    // Traverse all characters of given string
    for (int i=0; i<n; i++)
    {
        // If opening bracket, push index of it
        if (str[i] == '(')
          stk.push(i);

        else // If closing bracket, i.e.,str[i] = ')'
        {
            // Pop the previous opening bracket's index
            stk.pop();

            // Check if this length formed with base of
            // current valid substring is more than max
            // so far
            if (!stk.empty())
                result = max(result, i - stk.top());

            // If stack is empty. push current index as
            // base for next valid substring (if any)
            else stk.push(i);
        }
    }

    return result;
}

// Driver program
int main()
{
    string str = "((()()";
    cout << findMaxLen(str) << endl;

    str = "()((((((";
    cout << findMaxLen(str) << endl ;

    return 0;
}
```

Python

```
# Python program to find length of the longest valid
# substring

def findMaxLen(string):
    n = len(string)

    # Create a stack and push -1 as initial index to it.
    stk = []
    stk.append(-1)

    # Initialize result
    result = 0

    # Traverse all characters of given string
    for i in xrange(n):

        # If opening bracket, push index of it
        if string[i] == '(':
            stk.append(i)

        else:    # If closing bracket, i.e., str[i] = ')'

            # Pop the previous opening bracket's index
            stk.pop()

            # Check if this length formed with base of
            # current valid substring is more than max
            # so far
            if len(stk) != 0:
                result = max(result, i - stk[len(stk)-1])

            # If stack is empty. push current index as
            # base for next valid substring (if any)
            else:
                stk.append(i)

    return result

# Driver program
string = "((()()"
print findMaxLen(string)

string = "()((()))))"
print findMaxLen(string)

# This code is contributed by Bhavya Jain
```

Output:

```
4
6
```

Explanation with example:

```
Input: str = "(()()"

Initialize result as 0 and stack with one item -1.

For i = 0, str[0] = '(', we push 0 in stack

For i = 1, str[1] = '(', we push 1 in stack

For i = 2, str[2] = ')', currently stack has [-1, 0, 1], we pop
from the stack and the stack now is [-1, 0] and length of current
valid substring becomes 2 (we get this 2 by subtracting stack top
from current index).
Since current length is more than current result, we update result.

For i = 3, str[3] = '(', we push again, stack is [-1, 0, 3].

For i = 4, str[4] = ')', we pop from the stack, stack becomes
[-1, 0] and length of current valid substring becomes 4 (we get
this 4 by subtracting stack top from current index).
Since current length is more than current result, we update result.
```