# 题目

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

输入：s = "(()"

输出：2

解释：最长有效括号子串是 "()"

示例 2：

输入：s = ")()())"

输出：4

解释：最长有效括号子串是 "()()"

示例 3：

输入：s = ""

输出：0

提示：

0 <= s.length <= 3 \* 10^4

s[i] 为 '(' 或 ')'

# 分析

## 方法一：动态规划

要找出最长有效括号子串的长度，可以使用动态规划的方法。具体思路如下：

1、创建一个长度为n的数组dp，其中dp[i]表示以字符s[i]结尾的最长有效括号子串的长度。

2、初始化数组dp全部为0。

3、从左到右遍历字符串s，对于每个字符 s[i]：

- 如果s[i]是'('，则dp[i]必定为0，因为任何以'('结尾的子串都不可能是有效的。

- 如果s[i]是')'，分两种情况讨论：

- 如果s[i-1]是'('，则dp[i] = dp[i-2] + 2。

- 如果s[i-1]是')'且i-dp[i-1]-1 >= 0且s[i-dp[i-1]-1]是'('，则dp[i] = dp[i-1] + 2 + dp[i-dp[i-1]-2]。

4、遍历完整个字符串后，dp数组中最大的值即为最长有效括号子串的长度。

具体实现如下：

class Solution {

public:

int longestValidParentheses(string s) {

int n = s.length();

if (n == 0) return 0;

vector<int> dp(n, 0);

int maxLen = 0;

for (int i = 1; i < n; ++i) {

if (s[i] == ')') {

if (s[i - 1] == '(') {

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

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

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

}

maxLen = max(maxLen, dp[i]);

}

}

return maxLen;

}

};

这个算法的时间复杂度为 O(n)，空间复杂度为 O(n)，其中 n 是字符串 `s` 的长度。

## 方法二：栈

要使用单调栈实现最长有效括号子串的长度的计算，可以遵循以下思路：

1、创建一个栈，用于存储字符在字符串中的下标。

2、遍历字符串，对于每个字符s[i]：

- 如果s[i]是'('，将其下标入栈。

- 如果s[i]是')'，需要判断当前情况是否能构成有效括号子串：

- 如果栈为空，说明当前')'没有对应的'('，将当前字符的下标入栈。

- 如果栈不为空，弹出栈顶元素表示匹配了一个'('，此时需要更新最长有效括号子串的长度：

- 如果弹出后栈为空，说明之前的括号全部匹配，更新最长长度为当前字符下标加一。

- 如果弹出后栈不为空，更新最长长度为当前字符下标减去栈顶元素的值（即截止到上一个未匹配的'('的长度）。

具体实现如下：

class Solution {

public:

int longestValidParentheses(string s) {

stack<int> stk;

stk.push(-1); // 初始放入一个-1，用于计算整个字符串长度

int maxLen = 0;

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

if (s[i] == '(') {

stk.push(i); // 将'('的下标入栈

} else { // 遇到')'

stk.pop(); // 匹配了一个'('

if (stk.empty()) {

stk.push(i); // 栈空，将当前')'下标入栈（当前是为匹配的')'）

} else {

maxLen = max(maxLen, i - stk.top()); // 更新最长长度

}

}

}

return maxLen;

}

};

这个算法的时间复杂度为 O(n)，空间复杂度为 O(n)，其中 n 是字符串 `s` 的长度。

上述代码优化：

可以使用一个变量start来记录当前有效括号子串的起始位置，遇到左括号时将start入栈，遇到右括号时判断栈是否为空，若为空则更新start为当前位置；若不为空，则出栈一个元素表示匹配了一个左括号，此时更新最长长度。具体实现如下：

class Solution {

public:

int longestValidParentheses(string s) {

stack<int> stk;

int maxLen = 0, start = 0;

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

if (s[i] == '(') {

stk.push(i);

} else {

if (stk.empty()) {

start = i + 1; // 无法匹配，更新起始位置

} else {

stk.pop(); // 匹配成功，出栈

if (stk.empty()) {

maxLen = max(maxLen, i - start + 1); // 更新最长长度

} else {

maxLen = max(maxLen, i - stk.top()); // 更新最长长度

}

}

}

}

return maxLen;

}

};

**思路：**

撇开方法一提及的动态规划方法，相信大多数人对于这题的第一直觉是找到每个可能的子串后判断它的有效性，但这样的时间复杂度会达到 O(n^3)，无法通过所有测试用例。但是通过栈，我们可以在遍历给定字符串的过程中去判断到目前为止扫描的子串的有效性，同时能得到最长有效括号的长度。

具体做法是我们始终保持栈底元素为当前已经遍历过的元素中「最后一个**没有被匹配**的右括号的下标」，这样的做法主要是考虑了边界条件的处理，栈里其他元素维护左括号的下标：

1、对于遇到的每个‘(’，我们将它的下标放入栈中

2、对于遇到的每个‘)’，我们先弹出栈顶元素表示匹配了当前右括号：

1）如果栈为空，说明当前的右括号为没有被匹配的右括号，我们将其下标放入栈中来更新我们之前提到的「最后一个没有被匹配的右括号的下标」

2）如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」

我们从前往后遍历字符串并更新答案即可。

需要注意的是，如果一开始栈为空，第一个字符为左括号的时候我们会将其放入栈中，这样就不满足提及的「最后一个没有被匹配的右括号的下标」，为了保持统一，我们在一开始的时候往栈中放入一个值为-1的元素。

**代码：**

class Solution {

public:

int longestValidParentheses(string s) {

int maxans = 0;

stack<int> stk;

stk.push(-1);

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

if (s[i] == '(') {

stk.push(i); // 起始左括号的位置

} else {

stk.pop(); // 如果是')'，则表示匹配了，直接弹出

if (stk.empty()) {

stk.push(i);

} else {

maxans = max(maxans, i - stk.top());

}

}

}

return maxans;

}

};

**复杂度分析：**

时间复杂度：O(n)，n是给定字符串的长度。我们只需要遍历字符串一次即可。

空间复杂度：O(n)。栈的大小在最坏情况下会达到n，因此空间复杂度为O(n)。