<https://leetcode.com/problems/longest-valid-parentheses/description/>

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

**Example 1:**

**Input:** s = "(()"

**Output:** 2

**Explanation:** The longest valid parentheses substring is "()".

**Example 2:**

**Input:** s = ")()())"

**Output:** 4

**Explanation:** The longest valid parentheses substring is "()()".

**Example 3:**

**Input:** s = ""

**Output:** 0

**Constraints:**

0 <= s.length <= 3 \* 104

s[i] is '(', or ')'.

**Attempt 1: 2025-07-13**

**Solution 1: Stack (60 min)**

**Style 1: With push -1 as dummy begin first**

class Solution {

public int longestValidParentheses(String s) {

// Stores the maximum length of valid parentheses substring found

int maxLen = 0;

// Stack to keep track of indices of parentheses

// The indices help us calculate the length of valid substrings

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

// Push -1 as initial base index onto the stack

// This serves three important purposes:

// 1. Provides a reference point for calculating lengths starting from index 0

// 2. Acts as a boundary marker when the stack becomes empty

// 3. Helps handle edge cases like valid substrings starting at index 0

stack.push(-1);

// Iterate through each character in the string

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

char c = s.charAt(i);

if(c == '(') {

// For opening '(', push its index onto the stack

// These represent potential starting points for valid substrings

stack.push(i);

} else {

// For closing ')', we first pop the top element

// This could be either:

// 1. The index of a matching '(' (which we'll use to calculate length)

// 2. A boundary marker (either -1 or index of a previous unmatched ')')

stack.pop();

if(stack.isEmpty()) {

// If stack becomes empty after popping, push current index

// This serves as a new boundary marker because:

// 1. We've encountered an unmatched ')'

// 2. Future valid substrings cannot extend past this point

stack.push(i);

} else {

// If stack is not empty, the current top represents:

// - For matched pairs: the index before the valid substring starts

// - For boundary cases: the last unmatched position

// Calculate current valid length as current index - stack's top index

// Update maxLen if this is the longest valid substring found so far

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

}

}

}

// Return the maximum length of valid parentheses substring found

return maxLen;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: Without push -1 as dummy begin first**

class Solution {

    public int longestValidParentheses(String s) {

        // Stores the maximum length of valid parentheses found

        int maxLen = 0;

        // Stack to keep track of indices

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

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

            char c = s.charAt(i);

            if(c == '(') {

                // Push the index of '(' onto the stack

                stack.push(i);

            // When encountering ')'

            } else {

                if(!stack.isEmpty() && s.charAt(stack.peek()) == '(') {

                    // Found a matching pair - pop the matching '(' index

                    stack.pop();

                    // Calculate the start of the current valid substring:

                    // If stack is empty, the valid substring starts from -1 (before index 0)

                    // Otherwise, it starts from the index after the last unmatched ')'

                    maxLen = Math.max(maxLen, stack.isEmpty() ? i + 1 : i - stack.peek());

                } else {

                    // Push unmatched ')'

                    stack.push(i);

                }

            }

        }

        return maxLen;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

**Problem Statement**

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

**Approaches**

**1. Stack Approach (Optimal)**

Use a stack to keep track of indices of parentheses

Initialize stack with -1 to handle base case

For every '(', push its index

For every ')':

Pop from stack

If stack becomes empty, push current index

Else, calculate length from current top of stack to current index

**2. Dynamic Programming (Alternative)**

Maintain a DP array where dp[i] represents longest valid parentheses ending at i

Handle different cases based on previous characters

**Solution Code (Stack Approach)**

class Solution {

public int longestValidParentheses(String s) {

int maxLen = 0;

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

stack.push(-1); // Base for length calculation

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

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

stack.push(i);

} else {

stack.pop();

if (stack.isEmpty()) {

stack.push(i); // New base

} else {

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

}

}

}

return maxLen;

}

}

**Solution Code (DP Approach)**

class Solution {

public int longestValidParentheses(String s) {

int maxLen = 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 {

int prevLen = dp[i-1];

int prevIndex = i - prevLen - 1;

if (prevIndex >= 0 && s.charAt(prevIndex) == '(') {

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

}

}

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

}

}

return maxLen;

}

}

**Explanation**

**Stack Approach**

**Initialization**: Start with -1 in stack to handle base case for length calculation

**Processing**:

For '(', push index onto stack

For ')':

Pop from stack (matches with last '(')

If stack empty, push current index (new base)

Else, calculate length from current top to current index

**Time Complexity**: O(n) - single pass through string

**Space Complexity**: O(n) - stack space

**DP Approach**

**DP Array**: dp[i] stores longest valid parentheses ending at i

**Cases**:

Immediate pair "()": dp[i] = dp[i-2] + 2

Nested case "(())": check matching '(' before the valid substring

**Time Complexity**: O(n) - single pass through string

**Space Complexity**: O(n) - for DP array

**Example**

**Input:** ")()())"

**Initial State:**

Stack: [-1] (base index)

maxLen: 0

**Processing each character:**

**i=0, ')'**:

Pop: removes -1

Stack is empty → push current index 0

Stack: [0]

maxLen remains 0

**i=1, '('**:

Push current index 1

Stack: [0, 1]

maxLen remains 0

**i=2, ')'**:

Pop: removes 1 (matches with '(' at index 1)

Stack not empty → length = 2 - 0 = 2

maxLen = max(0, 2) = 2

Stack: [0]

**i=3, '('**:

Push current index 3

Stack: [0, 3]

maxLen remains 2

**i=4, ')'**:

Pop: removes 3 (matches with '(' at index 3)

Stack not empty → length = 4 - 0 = 4

maxLen = max(2, 4) = 4

Stack: [0]

**i=5, ')'**:

Pop: removes 0

Stack is empty → push current index 5

Stack: [5]

maxLen remains 4

**Key Points:**

**Base Index Handling**: The initial -1 helps calculate lengths starting from index 0

**Stack Operations**:

Every '(' pushes its index

Every ')' pops and either:

Sets new base if stack becomes empty

Calculates new valid length otherwise

**Efficiency**:

Time: O(n) - single pass through the string

Space: O(n) - worst case stack size

This corrected implementation properly handles all cases and matches the expected behavior for the given example. The longest valid parentheses substring "()()" (indices 1-4) is correctly identified with length 4.

**Why we have to write a tricky statement as "stack.push(-1);" first ? and if i don't write this line, what will be the changed solution ?**

Great question! The stack.push(-1) initialization is indeed crucial for the algorithm's correctness. Let me explain why it's needed and what would happen without it.

**Why We Need stack.push(-1):**

**Base Case Handling**:

The -1 serves as a dummy index representing the position before the string starts (imaginary index -1)

It provides a reference point for calculating the length of valid parentheses starting from index 0

**Length Calculation**:

When we find a valid pair () at indices (i,j), the length is j - stack.peek()

Without -1, we wouldn't have a starting reference for valid pairs beginning at index 0

**Empty Stack Handling**:

When we pop and the stack becomes empty, we know we've encountered an unmatched )

We then push the current index as a new boundary marker

**What Happens Without stack.push(-1):**

**For Input Starting with '('**:

Input: "(()"

Without -1:

- Stack becomes empty after first ')', pushes index 1

- Misses valid length calculation for index 0-1

**For Input Starting with ')'**:

Input: ")()"

Without -1:

- First ')' causes EmptyStackException (nothing to pop)

**Modified Solution Without stack.push(-1):**

If you really want to avoid the -1 initialization, you'd need to:

Handle empty stack cases differently

Add special checks for the start of the string

Make the code more complex

Here's how it would look (but I don't recommend this):

class Solution {

public int longestValidParentheses(String s) {

int maxLen = 0;

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

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

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

stack.push(i);

} else {

if (!stack.isEmpty() && s.charAt(stack.peek()) == '(') {

stack.pop();

int start = stack.isEmpty() ? -1 : stack.peek();

maxLen = Math.max(maxLen, i - start);

} else {

stack.push(i); // Push unmatched ')'

}

}

}

return maxLen;

}

}

**Why the Original is Better:**

**Cleaner Logic**:

The -1 initialization provides a uniform way to handle all cases

No need for special checks for the string start

**Easier to Understand**:

The i - stack.peek() formula works consistently for all valid pairs

Without -1, you need conditional logic for empty stack cases

**More Robust**:

Handles edge cases like (), ((), )() uniformly

No risk of EmptyStackException

**Example Comparison:**

**Input:** "()"

With -1:

Stack: [-1]

'(' at 0: push → [-1,0]

')' at 1: pop → peek is -1 → length = 1 - (-1) = 2

Without -1:

Stack: []

'(' at 0: push → [0]

')' at 1: pop → stack empty → need special handling

The -1 initialization makes the algorithm more elegant and uniform in handling all cases. While it's possible to write a solution without it, the resulting code would be more complex and harder to maintain.

**For Style 1 below explain and the statement, please how an example to for: A boundary marker (either -1 or index of a previous unmatched ')')**

// For closing ')', we first pop the top element

// This could be either:

// 1. The index of a matching '(' (which we'll use to calculate length)

// 2. A boundary marker (either -1 or index of a previous unmatched ')')

stack.pop();

**Explanation with Examples: Popping from the Stack for ')'**

When we encounter a closing parenthesis ')', we always pop() from the stack. The popped element can be one of two things:

**Case 1: Index of a Matching '(' (Valid Pair)**

**Example String:** "()"

Stack starts: [-1]

i=0 '(' → push 0 → stack: [-1, 0]

i=1 ')' → **pop() removes 0**

This 0 was the index of a matching '('

Stack becomes: [-1] (not empty)

Calculate length: 1 - (-1) = 2

**Result:** Valid substring "()" (length=2)

**Case 2: Boundary Marker (Unmatched ')')**

**Example String:** "())"

Stack starts: [-1]

i=0 '(' → push 0 → stack: [-1, 0]

i=1 ')' → pop() removes 0 (matching '(')

Stack becomes: [-1]

Length: 1 - (-1) = 2

i=2 ')' → **pop() removes -1**

This -1 was a boundary marker (not a '(')

Stack becomes empty → push current index 2

New stack: [2] (this 2 is a boundary marker for future calculations)

**Why Boundary Markers Matter**

**Extended Example:** "())()"

After processing "())" (above):

Stack: [2] (boundary marker at index 2)

Current max length: 2

Continue processing:

i=3 '(' → push 3 → stack: [2, 3]

i=4 ')' → pop() removes 3 (matching '(')

Stack becomes: [2] (not empty)

**Calculate length:** 4 - 2 = 2

New max length: max(2, 2) = 2

**Final valid substring:** "()" (indices 3-4)

Separated from earlier valid "()" (indices 0-1) by boundary marker 2 (the unmatched ')')

**Key Insights**

**Boundary markers** are indices that represent:

The start of the string (initial -1)

Positions of **unmatched ')'** (like index 2 in the example)

**Length calculation** i - stack.peek():

Measures distance from current position back to:

The last boundary marker (if no matches since then)

Or the position after the last valid match

**Purpose of pushing unmatched ')':**

Creates a new boundary preventing future valid substrings from merging across unmatched parentheses.

**Visualization**

For "())()":

Indices: 0 1 2 3 4

Chars: ( ) ) ( )

└─┘ └─┘

len=2 len=2 (separated by boundary at index 2)

The unmatched ')' at index 2 acts as a permanent divider between the two valid substrings.

**Refer to**

[L20.P11.5.Valid Parentheses (Ref.L32)](note://8CA124C3F0314C4A9CC7078E4E81EB04)