<https://leetcode.com/problems/score-of-parentheses/>

Given a balanced parentheses string s, return *the* ***score*** *of the string*.

The **score** of a balanced parentheses string is based on the following rule:

* "()" has score 1.
* AB has score A + B, where A and B are balanced parentheses strings.
* (A) has score 2 \* A, where A is a balanced parentheses string.

**Example 1:**

Input: s = "()"

Output: 1

**Example 2:**

Input: s = "(())"

Output: 2

**Example 3:**

Input: s = "()()"

Output: 2

**Constraints:**

* 2 <= s.length <= 50
* s consists of only '(' and ')'.
* s is a balanced parentheses string.

**Attempt 1: 2023-03-24**

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

**Style 1: No initial '0' on stack,  Use local variable 'cur\_score' to identify "))" and "()" two different cases (read '(' before read ')')**

class Solution {

public int scoreOfParentheses(String s) {

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

int result = 0;

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

char c = s.charAt(i);

if(c == '(') {

stack.push(0);

} else {

int cur = stack.pop();

// == 0 indicates no inner parentheses

// > 0 indicates have inner parentheses

if(cur == 0) {

cur = 1;

} else {

cur \*= 2;

}

if(!stack.isEmpty()) {

stack.push(stack.pop() + cur);

} else {

result += cur;

}

}

}

return result;

}

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/score-of-parentheses/solutions/197544/Java-Basic-Stack-Based-Solution-O(n)-time-O(n)-space/>

class Solution {

public int scoreOfParentheses(String S) {

int result = 0;

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

for(char c : S.toCharArray()) {

if(c == '(') {

// When encounter open bracket, push 0 on stack

// to initialize calculate for this section (the

// actual calculate will happen when encounter

// its corresponding close bracket)

stack.push(0);

} else {

// When encounter a close bracket, pop current

// score from stack and start actual calculate

int cur\_score = stack.pop();

// Based on given example, we have two scenarios:

// 1. If current close bracket's corresponding open bracket just

// one index ahead, and construct "()", e.g "(()(()))" when encounter

// ')' at index = 2, cur\_score is 0 and it pushed onto stack because

// '(' at index = 1, these 2 chars able to construct "()", so when

// calculate we update the current score from 0 to 1

// 2. If current close bracket's corresponding open bracket not

// one index ahead, e.g "(()(()))" when encounter ')' at index = 6,

// its corresponding '(' at index = 3, cur\_score is 1 and we should

// update from 1 to 1 \* 2 = 2

if(cur\_score == 0) {

cur\_score = 1;

} else {

cur\_score \*= 2;

}

// Most important part: after pop current score (which actually means

// pop out current ')'), if the stack not empty, it means at same

// level (from current ')' close bracket's perspective), we have

// previous calculated section stored onto stack, we have to sum up

// previous section and current section, so pop peek value which

// means previous section score plus current score and push the sum

// onto stack to finish sum up

if(!stack.isEmpty()) {

stack.push(stack.pop() + cur\_score);

} else {

// Or if stack is empty, which means we reach the out most level

// of brackets, just add onto result

result += cur\_score;

}

}

}

return result;

}

}

Our line of thinking is like this: when you see a new opening bracket, you'll later need to double the "score" of everything inside of it. When you see a closing bracket, pop the stack to retrieve the "score" of what's inside the pair, double the score (because of (A) = 2 \* A, given in problem), then use it as a sum term; if it's inside another bracket, then add that to what's now at the top of the stack, because it needs to be part of that "doubling" operation. If the stack is empty, then just add it straight to your final answer.

The ternary operator in the popping operation area is because if you end up with '()', there is nothing inside of it, but the score should be 1, not 0. So, we treat that as a special case.

**Style 2: Prepare an initial '0' on stack**

Not gonna work for me

**Refer to**

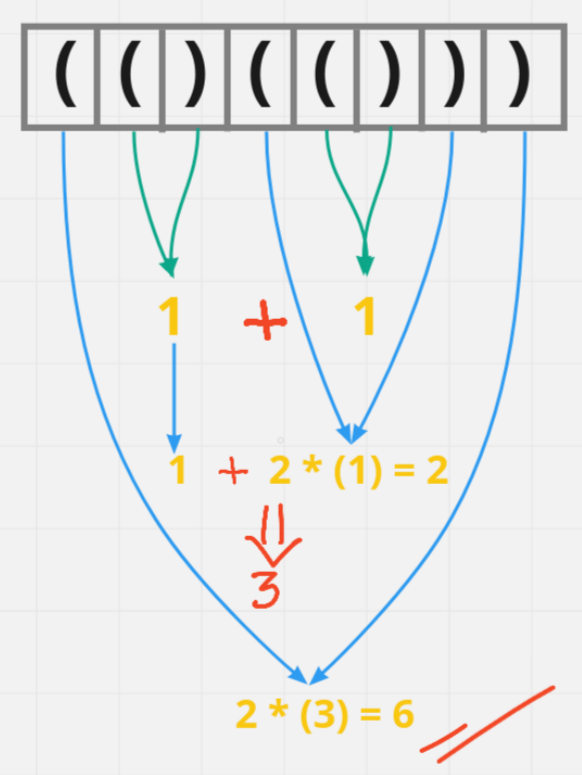
<https://leetcode.com/problems/score-of-parentheses/solutions/1856519/java-c-visually-explained/>

Let's understand our rule's first of all :-

* If our input string is **"()"** then our score will be **1**, so this is our first rule
* If we have input string **"()()"** then our score will be **1 + 1** i.e. **2**, where 1 is a score of a balanced parentheses. So, this is our second rule
* Now last rule say's, if we have given something like **"(())"** in our input string, our score will be **2 X 1** i.e. **2**, it means that we have something like nested parenthesis. Then multiply it by 2.

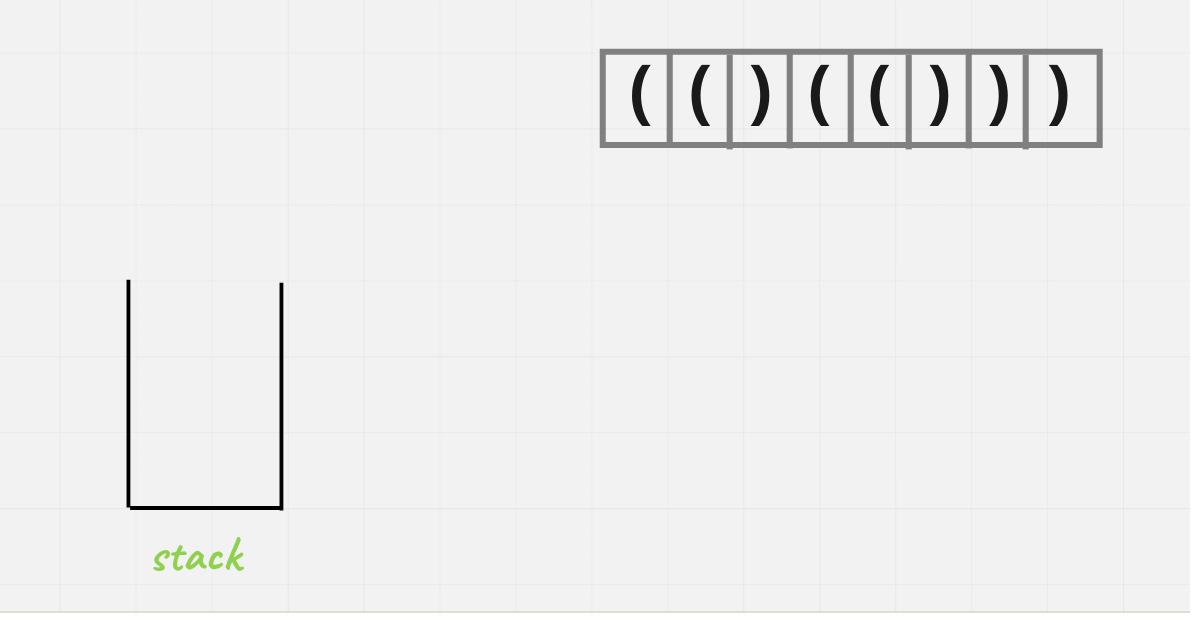
Let's take one example, inorder to solve this problem :

**Input :** "( ( ) ( ( ) ) )"**Output :** 6



Okay, so first thing came in our mind is can we solve this problem using stack, and I say yes we'll solve this problem using stack.

* First create one stack of **Integer** not
* So, as we are using Integer, what we gonna put in stack is initially **0** when we encounter (
* And we'll calculate the score when we encounter )



class Solution {

public int scoreOfParentheses(String s) {

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

int score = 0;

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

char ch = s.charAt(i);

if(ch == '('){

st.push(score);

score = 0;

}

else {

score = st.pop() + Math.max(2 \* score, 1);

}

}

return score;

}

}

* **Time Complexity :-** BigO(N)
* **Space Complexity :-** BigO(N)

**Style 3: Prepare an initial '0' on stack, an intuitive solution by handling two cases on second last char as '(' or ')' when last char is ')', and each time when encounter ')' will pop out previous continuously two '(' if exists two, OR almost same but judge two cases on temporary sum > 0 or = 0 when last char is ')', and each time when encounter ')' will pop out previous continuously two '(' if exists two**

Judge by last second char

class Solution {

public int scoreOfParentheses(String s) {

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

stack.push(0);

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

char c = s.charAt(i);

if(c == '(') {

stack.push(0);

} else {

char p = s.charAt(i - 1);

int cur = stack.pop();

// second last position p as '(' indicates no inner parentheses

// second last position p as ')' indicates have inner parentheses

if(p == '(') {

stack.push(stack.pop() + 1);

} else {

stack.push(stack.pop() + cur \* 2);

}

}

}

return stack.peek();

}

}

=======================================================

Judge by last depth value

class Solution {

public int scoreOfParentheses(String s) {

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

stack.push(0);

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

char c = s.charAt(i);

if(c == '(') {

stack.push(0);

} else {

int cur = stack.pop();

// == 0 indicates no inner parentheses

// > 0 indicates have inner parentheses

if(cur == 0) {

stack.push(stack.pop() + 1);

} else {

stack.push(stack.pop() + cur \* 2);

}

}

}

return stack.peek();

}

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/score-of-parentheses/editorial/>

<https://leetcode.com/problems/score-of-parentheses/solutions/141975/c-solution-using-stack-5ms-detail-explained/comments/1311390>

Every position in the string has a *depth* - some number of matching parentheses surrounding it. For example, the dot in (()(.())) has depth 2, because of these parentheses: (\_\_(.\_\_))

Our goal is to maintain the score at the current depth we are on. When we see an opening bracket, we increase our depth, and our score at the new depth is 0. **When we see a closing bracket, we add twice the score of the previous deeper part - except when counting (), which has a score of 1.**

For example, when counting (()(())), our stack will look like this:

[0, 0] after parsing (

[0, 0, 0] after (

[0, 1] after )

[0, 1, 0] after (

[0, 1, 0, 0] after (

[0, 1, 1] after )

[0, 3] after )

1. after )

public int scoreOfParentheses(String s) {

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

// Prepare a score of current frame

stack.push(0);

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

char c = s.charAt(i);

if(c == '(') {

stack.push(0);

} else {

// c= ')'

char p = s.charAt(i - 1);

if(p == '(') {

stack.pop();

stack.push(stack.pop() + 1);

} else {

int current = stack.pop();

stack.push(stack.pop() + current \* 2);

}

}

}

return stack.pop();

}

**Refer to**

<https://leetcode.com/problems/score-of-parentheses/solutions/141975/c-solution-using-stack-5ms-detail-explained/>

When travel through S, only meets ')', we need to calculate the score of this pair of parentheses. If we know the score of inner parentheses, such as 3, we can double it and pass to the outer parentheses. But the question is how do we know the score of inner parentheses? Using stack.

Explain with "( ( ) ( ( ) ) )"

1.stack: 0->string\_traveled :""When start only 0 in stack, this int will store the total score

2.stack: 0->0->0->string\_traveled:"( ("Meet two '(', push two zeros to the stack

3.stack: 0->1->string\_traveled: "( ( )"First time meets ')', it balance the last '(', so pop the stack. But 0 indicates no inner 4.parentheses exists, so just pass 1 to parent parenthese.

5.stack: 0->1->0->0->string\_traveled: "( ( ) ( ("Keep pushing zeros

6.stack 0->1->1->string\_traveled: "( ( ) ( ( )"Balance one '(', and still no inner parenthese, so pass 1 to parent

7.stack 0->3->string\_traveled: "( ( ) ( ( ) )"Balance another '(', but the inner is not zero, so double it and add to parent's score

8.stack 6->string\_traveled: "( ( ) ( ( ) ) )"Same as last step, double the inner score and add to parent's

int scoreOfParentheses(string S)

{

stack<int> m\_stack;

m\_stack.push(0); // to keep the total score

for(char c:S){

if(c=='(')

m\_stack.push(0); //When meets '(', just push a zero to stack

else{

int tmp=m\_stack.top(); // balance the last '(', it stored the score of inner parentheses

m\_stack.pop();

int val=0;

if(tmp>0) // not zero means inner parentheses exists and double it

val=tmp\*2;

else // zero means no inner parentheses, just using 1

val=1;

m\_stack.top()+=val; // pass the score of this level to parent parenthese

}

}

return m\_stack.top();

}

**Solution 2: Recursion (60 min, hard to get)**

class Solution {

public int scoreOfParentheses(String s) {

return helper(s, 0, s.length());

}

private int helper(String s, int start, int end) {

int result = 0;

int balance = 0;

for(int i = start; i < end; i++) {

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

balance++;

} else {

balance--;

}

if(balance == 0) {

if(i - start == 1) {

result++;

} else {

result += 2 \* helper(s, start + 1, i);

}

start = i + 1;

}

}

return result;

}

}

Time complexity: O(n^2)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/score-of-parentheses/editorial/>

#### **Approach 1: Divide and Conquer**

**Intuition**

Split the string into S = A + B where A and B are balanced parentheses strings, and A is the smallest possible non-empty prefix of S.

**Algorithm**

Call a balanced string *primitive* if it cannot be partitioned into two non-empty balanced strings.

By keeping track of balance (the number of ( parentheses minus the number of ) parentheses), we can partition S into primitive substrings S = P\_1 + P\_2 + ... + P\_n. Then, score(S) = score(P\_1) + score(P\_2) + ... + score(P\_n), by definition.

For each primitive substring (S[i], S[i+1], ..., S[k]), if the string is length 2, then the score of this string is 1. Otherwise, it's twice the score of the substring (S[i+1], S[i+2], ..., S[k-1]).

class Solution {

public int scoreOfParentheses(String S) {

return F(S, 0, S.length());

}

public int F(String S, int i, int j) {

//Score of balanced string S[i:j]

int ans = 0, bal = 0;

// Split string into primitives

for (int k = i; k < j; ++k) {

bal += S.charAt(k) == '(' ? 1 : -1;

if (bal == 0) {

if (k - i == 1) ans++;

else ans += 2 \* F(S, i+1, k);

i = k+1;

}

}

return ans;

}

}

**Complexity Analysis**

* Time Complexity: O(N^2), where N is the length of S. An example worst case is (((((((....))))))).
* Space Complexity: O(N), the size of the implied call stack.