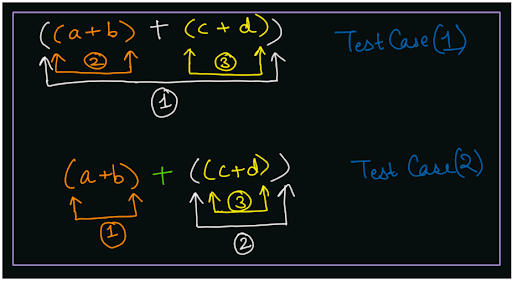
**1. Problem Discussion :**

You are given a string exp representing an expression. Assume that the expression is balanced i.e. the opening and closing brackets match with each other. There is always a closing bracket for each opening bracket. But, some of the pairs of brackets may be extra/needless. You are required to print true if you detect extra brackets and false otherwise. Example ((a + b) + (c + d)) -> false: There is no redundant or duplicate braces found (a + b) + ((c + d)) -> true: There is a pair of redundant or duplicate braces around c+d. I recommend you to watch the question video, where the problem statement is explained in detail.

**2. Approach :**

Deducing Algorithm:

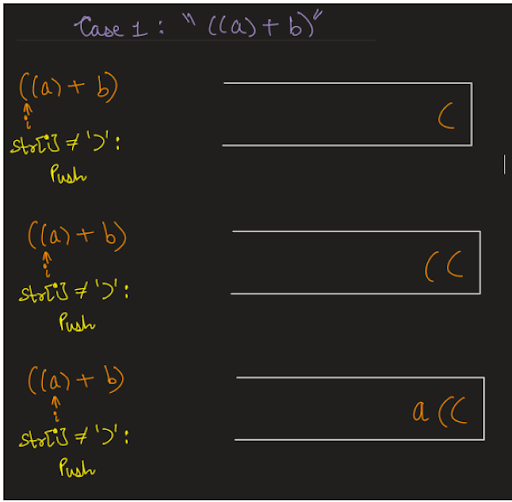
Let us analyze the condition which should be fulfilled so that a pair of brackets can be said as duplicate/redundant or not, using the two test cases:

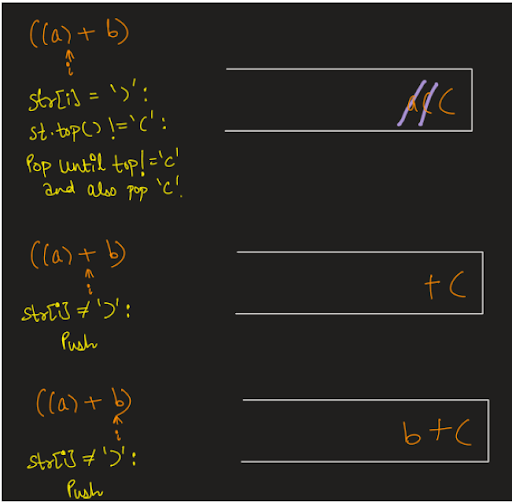
****

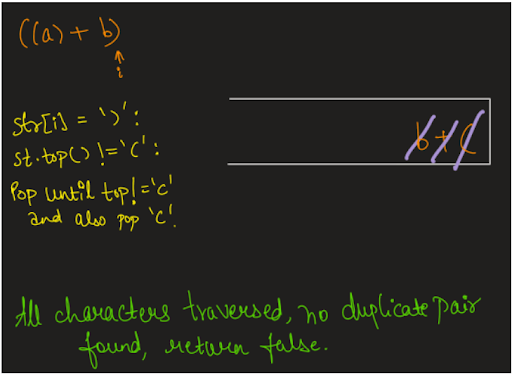
In the first test case, if you will see that, for each pair of brackets, there is at least one character inside the brackets corresponding to the pair. There is one '+' character for pair (1), three characters 'a', '+', 'b' for pair (2) and three characters 'c', '+', 'd' for pair (3). Hence, no pair is redundant or duplicate. But in the second test case, you can see that although there are 3-3 characters each corresponding to the pairs (1) and (3), there is no character corresponding to pair (2). Hence this pair (2) is redundant/duplicate. Although characters {c, +, d} are inside the pair (2), they do not belong exclusively to the pair (2), but instead belong to pair (3). Hence, each character can belong to only 1 pair of brackets. Note: Characters corresponding to a pair are marked with the color same as the brackets color in the diagram. Hence, what we have found out is: there must be at least one character exclusively within a bracket pair, so that this pair is not a duplicate/redundant pair.

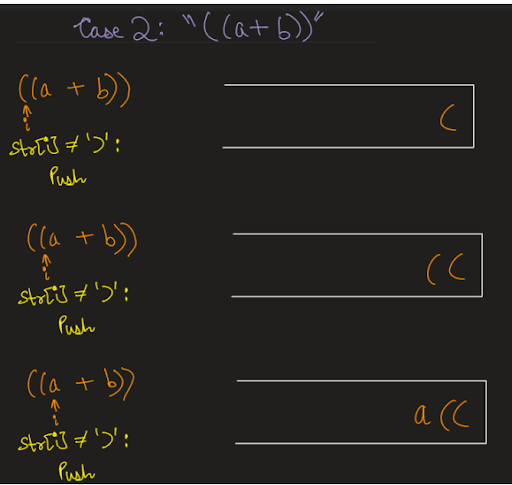
Pseudo Code

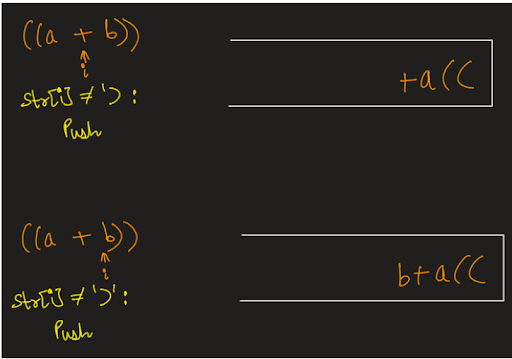
We can use the stack data structure to check whether there exists at least one character exclusively between each pair of brackets or not. Initialize an empty stack of characters st. Iterate over the expression string and pick the current character at index i. If the current character is not ')', {can be any alphabet, digit, '(' opening bracket or arithmetic operator}, then: Push the character into the stack Else, {current character is ')'} Check if the character at the top of the stack is '(' or not. If it is '(', then return true (as pair of redundant bracket is found) Else, (there is at least 1 character b/w the current pair) Pop characters from the stack until you get '(' opening bracket at the top. Pop the opening bracket '(' as well. If all characters are traversed, and no duplicate bracket is found, then return false. Q) You may think that we are not even checking whether the stack is empty or not before checking whether the top element is '(' or not. But why so? R) This is because the input expression is balanced. It means before every ')' closing bracket, there will be at least one '(' opening bracket for sure. Hence, we cannot get a string like ")(", because this is not balanced. Let us look at the algorithm using an example:

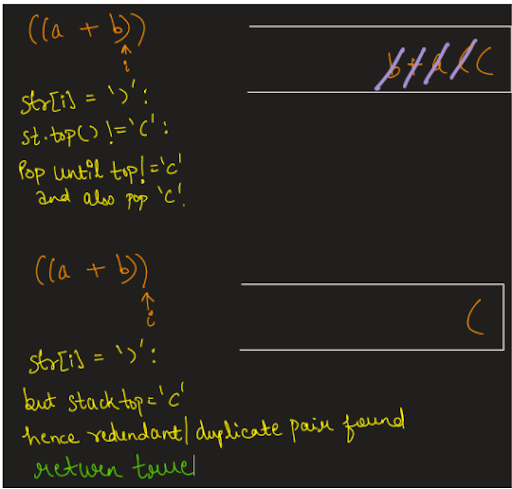
****

****

****

****

****

****

Suggestion:

Please don't worry if the algorithm is not intuitive for you at the moment. It is a very classic application of stack data structure, and you will get used to it, after solving some problems using this technique. Please refer to the solution video if you find difficulty in understanding the algorithm completely. Seems simple, right? Do give it a try before reading the code.

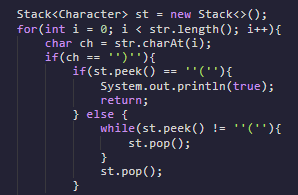
Code

import java.io.\*; import java.util.\*; public class Main { public static void main(String[] args) throws Exception { BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); String str = br.readLine(); Stack< Character> st = new Stack< >(); for (int i = 0; i < str.length(); i++) { char ch = str.charAt(i); if (ch == ')') { if (st.peek() == '(') { System.out.println(true); return; } else { while (st.peek() != '(') { st.pop(); } st.pop(); } } else { st.push(ch); } } System.out.println(false); } }

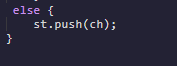
This code is written and explained by our team in the solution video. Please refer to it if you are stuck somewhere.

Code Explanation:

In the below code we are checking every character in the string if it is a closing parenthesis ‘)’ then if the top of the stack contains an opening parenthesis then it means it is an extra bracket and we return true, Else we pop all the elements until we find an opening parenthesis corresponding to this closing parenthesis.When we find an opening parenthesis we pop it out also.

****

In the else part i.e. if the character encountered at index ‘i’ is not a closing parenthesis we simply push the character.

****

**3. Analysis**

This is a very crucial point where a lot of readers get confused. Please read it carefully and try to focus on the analysis.

Time Complexity :

We are traversing the expression string once. Now, you can see that inside the for loop, whenever we encounter ')' character, we are running a while loop which will run until '(' is encountered at top of stack. So, should not time complexity be O(n \* n) as there is a while loop nested inside a for loop? No, the time complexity is O(n) only. Let us figure out why it is so. For each character in the expression string (other than the closing bracket), we are pushing it in the stack only once. Hence, although there is a nested while loop inside a for loop, the number of characters that will get popped out of the stack are not equal to the size of string in each iteration. We will pop n characters from the stack at maximum during the entire for loop and not single iteration of for loop. Thus, we are pushing and popping each character only once, thus the time complexity will be O(2\*n) = O(n) only.

SPACE COMPLEXITY :

We are using the stack data structure, which can take space equal to the number of characters in the expression string, i.e. equal to the size of the string, in the worst case, hence O(n) auxiliary space is required. I hope you enjoyed solving the problem with me. I will come with a similar problem ' Balanced Brackets' for you to solve, until then Happy Coding!