DSA Post #1:

leetcode problem #32 Longest Valid Parenthesis (Hard)

Parenthesis such as (), ()(), ((())), ()(()) ... are considered valid.

Given a string, find the longest length of the substring which is formed by valid parenthesis

Eg:

)() - 2

()(()) - 6

))((- 0

) - 0

Approach 1:

Technique: *Stack*

Rationale:

Screen all the characters of the input string and push or pop "the index of character" based on following conditions:

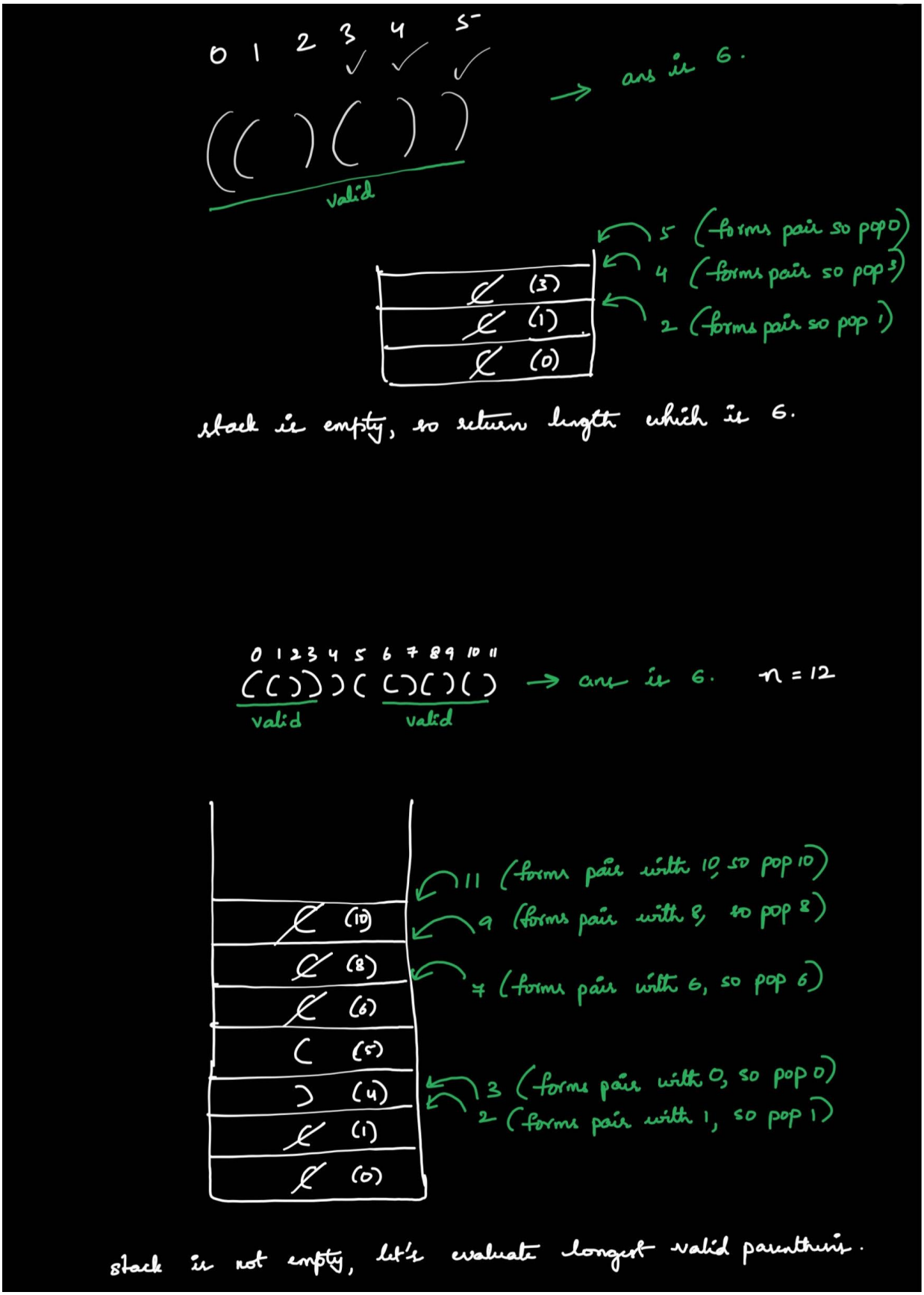
If character is ')' check if the top of stack is '(' and if true, pop the character from stack

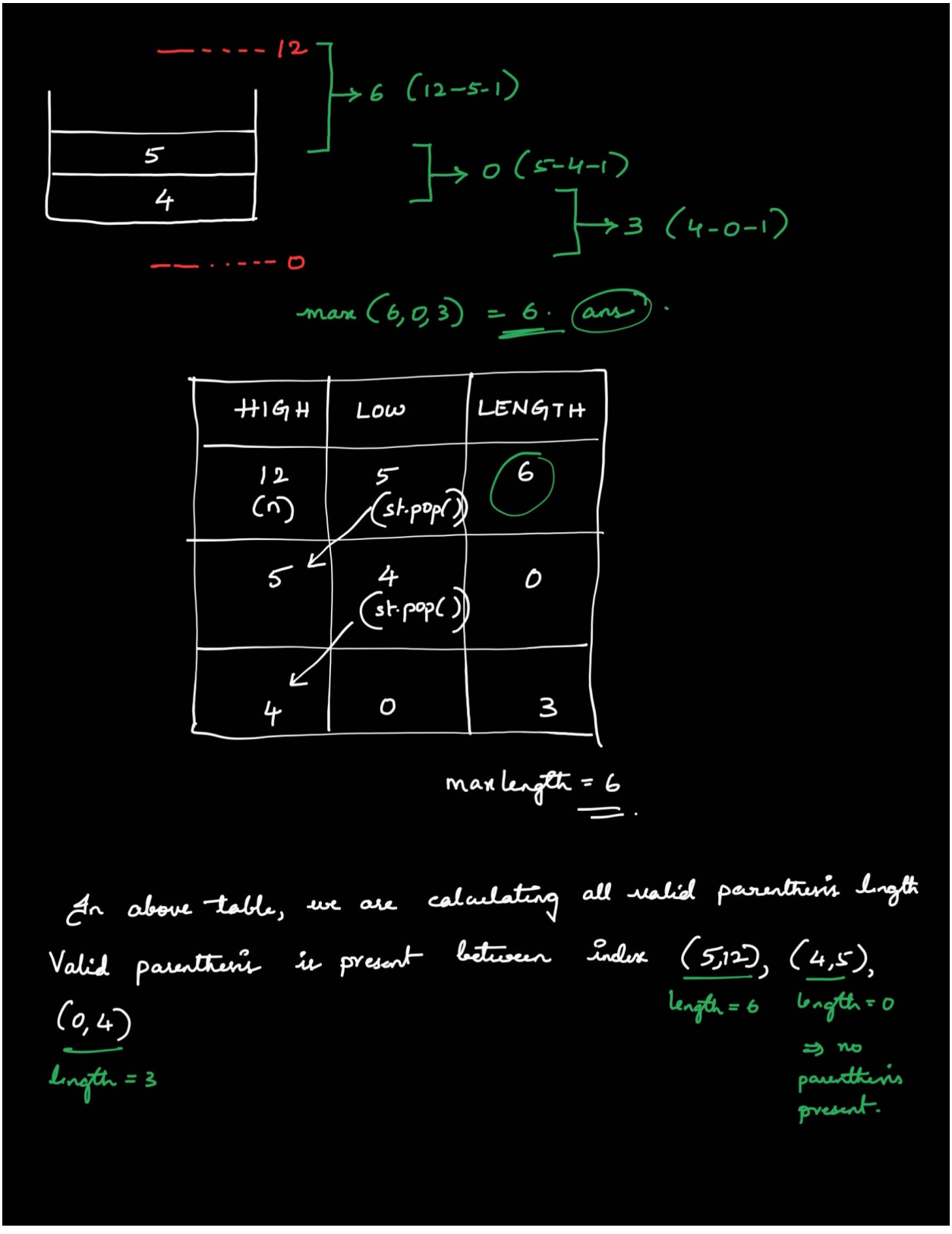
Else push the character to stack

Once above screening is finished, validate the stack.

If stack is empty, complete string is valid and return the size of input

Otherwise, Find the lengths between the indices in the stack (these lengths are valid lengths) and get the "longest", return the longest


0 1 2 3 4 5
((()()))
valid
→ ans is 6.
5 (forms pair so pop 0)
4 (forms pair so pop 3)
2 (forms pair so pop 1)
(3)
(1)
(0)
stack is empty, so return length which is 6.
0 1 2 3 4 5 6 7 8 9 10 11
(()))(()()()
valid
valid
→ ans is 6. n = 12
11 (forms pair with 10, so pop 10)
9 (forms pair with 8, so pop 8)
7 (forms pair with 6, so pop 6)
3 (forms pair with 0, so pop 0)
2 (forms pair with 1, so pop 1)
(10)
(8)
(6)
(5)
(4)
(1)
(0)
stack is not empty, let's evaluate longest valid parenthesis.


12
5
4
0
6 (12-5-1)
0 (5-4-1)
3 (4-0-1)
max (6,0,3) = 6. ans.
HIGH
LOW
LENGTH
12 (n)
5 (st.pop())
6
5
4 (st.pop())
0
4
0
3
max length = 6
An above table, we are calculating all valid parenthesis length
Valid parenthesis is present between index (5,12), (4,5),
length = 6
length = 0
⇒ no parenthesis present.
(0,4)
length = 3

Code:

```
class Solution {
    public int longestValidParentheses(String s) {
        char[] chArray = s.toCharArray();
        int n = s.length();
        Stack<Integer> chStack = new Stack<>();
        int longest = 0;
        for (int i = 0; i < n; i++) {
            if (chArray[i] == ')' && !chStack.isEmpty() && chArray[chStack.peek()] == '(') {
                chStack.pop();
            } else {
                chStack.push(i);
            }
        }
        if (chStack.isEmpty()) {
            //when stack is empty, complete string is a valid one, return length of string
            longest = n;
        } else {
            int high = n, low = 0;
            //when stack is not empty, evaluate all valid parenthesis between indexes in the stack
            while (!chStack.isEmpty()) {
                low = chStack.pop();
                longest = Math.max(longest, high - low - 1);
                high = low;
            }
            longest = Math.max(longest, high);
        }
```

```
        return longest;
    }
}
```

Time complexity: O(n)

Space complexity: O(n) - for creating stack

Approach 2:

Technique: *Dynamic programming*

Rationale:

In approach 1, we are evaluating the string lengths and updating the variable “longest” when we encounter longer length.

In this approach, we avoid storing indexes in stack and calculate lengths.

Instead, we leverage dp array to store the longest string lengths. Which means, at dp[i], we store longest valid parenthesis until index i.

Note: if current bracket is ')' and previous bracket is also ')' then we check valid length at previous bracket ')' dp[i-1]. We jump to the index before to last valid string (i - dp[i-1] - 1) and check if it is '(' which makes it a valid pair () like at index 2 and 5 in example below:

eg: ()(())

For this sample, dp array is {0, 2, 0, 0, 2, 0} until i = 4

when i = 5,

charAt(5) is ')' and charAt(4) is ')' and charAt(5 - 2 - 1) is '('

dp[5] = 2 + 2 + 2 = 6

dp array finally is {0, 2, 0, 0, 2, 6}

ans: 6

Code:

```
public int longestValidParentheses(String s) {
        int n = s.length();
        if (n <= 1) return 0;
        int[] dp = new int[n];
        int longest = 0;
        for (int i = 1; i < n; i++) {
            int prevOfLastValid = i - dp[i - 1] - 1;
            if (s.charAt(i) == ')' && prevOfLastValid >= 0 && s.charAt(prevOfLastValid) == '(') {
                dp[i] = dp[i - 1] + 2 +
                    ((prevOfLastValid - 1 >= 0) ? dp[prevOfLastValid - 1] : 0);
                longest = Math.max(longest, dp[i]);
            }
        }
        return longest;
    }
```

Time complexity: O(n)
Space Complexity: O(n)