**1)Remove Outermost Parentheses**

**Example 1:**

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

Output: "(())"

Explanation: The input string is a single primitive: "((()))".

Removing the outermost layer yields: "(())".

**Example 2:**

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

Output: "(()())()"

Explanation: Primitive decomposition: "()" + "(()())" + "(())"

After removing outermost parentheses: "" + "()()" + "()"

Final result: "(()())()".

class Solution {

public String removeOuterParentheses(String s) {

StringBuilder result = new StringBuilder();

int openCount = 0;

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

if (c == '(') {

if (openCount > 0) {

result.append(c);

}

openCount++;

} else if (c == ')') {

openCount--;

if (openCount > 0) {

result.append(c);

}

}

}

return result.toString();

}

}

**Time Complexity: O(n)**, since we are performing a single traversal of the string.  
**Space Complexity: O(1)**, since we are using a few variables to track the current state.

2)**Reverse Words in a String**

**Approach:**

* We start traversing the string from the end until we hit a space. It indicates that we have gone past a word and now we need to store it.
* We check if our answer variable is empty or not
* If it’s empty, it indicates that this is the last word we need to print, and hence, there shouldn’t be any space after this word.
* If it’s empty we add it to our result with a space after it. Here’s a quick demonstration of the same

class Solution {

    public String reverseWords(String s) {

        String[] words = s.split("\\s+");

        StringBuilder res = new StringBuilder();

        for (int i = words.length - 1; i >= 0; i--) {

            res.append(words[i]);

            if (i != 0) {

                res.append(" ");

            }

        }

        return res.toString().trim();

    }

}

**Time Complexity: O(N),** N~length of string

**Space Complexity: O(1),** Constant Space