**Valid Parenthesis String**

**Medium**

Given a string s containing only three types of characters: '(', ')' and '\*', return true *if* s *is****valid***.

The following rules define a **valid** string:

* Any left parenthesis '(' must have a corresponding right parenthesis ')'.
* Any right parenthesis ')' must have a corresponding left parenthesis '('.
* Left parenthesis '(' must go before the corresponding right parenthesis ')'.
* '\*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string "".

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** true

**Example 3:**

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

**Output:** true

**Constraints:**

* 1 <= s.length <= 100
* s[i] is '(', ')' or '\*'.

class Solution {

public:

    bool checkValidString(string s) {

        stack<pair<char, int>> s1, s2;

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

            if (s[i]=='(') s1.push({s[i], i});

            else if (s[i]=='\*') s2.push({s[i], i});

            else {

                if (!s1.empty()) s1.pop();

                else if (!s2.empty()) s2.pop();

                else return false;

            }

        }

        while (!s1.empty()) {

            if (s2.empty()) return false;

            else {

                pair<char, int> p=s2.top();

                s2.pop();

                pair<char, int> p1=s1.top();

                s1.pop();

                if (p.second<p1.second) return false;

            }

        }

        return true;

    }

};

**Check if a Parentheses String Can Be Valid**

**Medium**

A parentheses string is a **non-empty** string consisting only of '(' and ')'. It is valid if **any** of the following conditions is **true**:

* It is ().
* It can be written as AB (A concatenated with B), where A and B are valid parentheses strings.
* It can be written as (A), where A is a valid parentheses string.

You are given a parentheses string s and a string locked, both of length n. locked is a binary string consisting only of '0's and '1's. For **each** index i of locked,

* If locked[i] is '1', you **cannot** change s[i].
* But if locked[i] is '0', you **can** change s[i] to either '(' or ')'.

Return true *if you can make s a valid parentheses string*. Otherwise, return false.

**Example 1:**

Shape

Description automatically generated

**Input:** s = "))()))", locked = "010100"

**Output:** true

**Explanation:** locked[1] == '1' and locked[3] == '1', so we cannot change s[1] or s[3].

We change s[0] and s[4] to '(' while leaving s[2] and s[5] unchanged to make s valid.

**Example 2:**

**Input:** s = "()()", locked = "0000"

**Output:** true

**Explanation:** We do not need to make any changes because s is already valid.

**Example 3:**

**Input:** s = ")", locked = "0"

**Output:** false

**Explanation:** locked permits us to change s[0].

Changing s[0] to either '(' or ')' will not make s valid.

**Constraints:**

* n == s.length == locked.length
* 1 <= n <= 105
* s[i] is either '(' or ')'.
* locked[i] is either '0' or '1'.
* class Solution {
* public:
* bool canBeValid(string s, string locked) {
* int len=s.length();
* if (len%2) return false;
* stack<int> s1, s2;
* for (int i=0; i<len; i++) {
* if (locked[i]=='0') s1.push(i);
* else {
* if (s[i]=='(') s2.push(i);
* else {
* if (!s2.empty()) s2.pop();
* else if (!s1.empty()) s1.pop();
* else return false;
* }
* }
* }
* while (!s2.empty()) {
* if (s1.empty()) return false;
* int x=s2.top();
* s2.pop();
* int y=s1.top();
* s1.pop();
* if (y<x) return false;
* }
* int len1=s1.size();
* if (s1.empty() or len1%2==0) return true;
* else return false;
* }
* };