2116. Check if a Parentheses String Can Be Valid

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

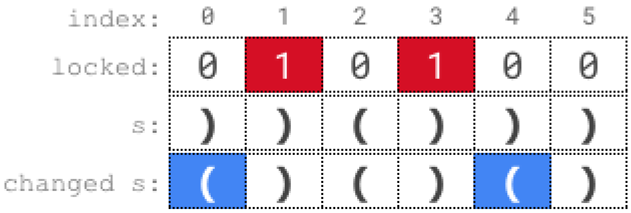
- It is `()`.
- It can be written as `AB` (`A` concatenated with `B`), where `A` and `B` are valid parentheses strings.
- It can be written as `(A)`, where `A` is a valid parentheses string.

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:



index:	0	1	2	3	4	5
locked:	0	1	0	1	0	0
s:	)	)	(	)	)	)
changed s:	(	)	(	)	(	)

```
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 <=` 10^5
- `s[i]` is either `'('` or `')'`.
- `locked[i]` is either `'0'` or `'1'`.

Overview

We are given two strings, `s` and `locked`. The string `s` is a sequence of parentheses, consisting of opening brackets `(` and closing brackets `)`. The string locked is a binary string of the same length as `s`, where:

- If `locked[i]` is 1, the character at index `i` in `s` cannot be changed.
- If `locked[i]` is 0, the character can be modified: an opening bracket `(` can become a closing bracket `)` and vice versa.

Our task is to determine if it's possible to make the sequence in `s` balanced by modifying the characters marked as changeable (`locked[i] = 0`).

What does a balanced parentheses sequence mean?

A sequence of parentheses is considered balanced if:

1. Every opening bracket `(` has a corresponding closing bracket `)`.
2. The brackets are properly nested. For example, `(())` is balanced, but `())(` is not.

To gain familiarity with similar parentheses-based problems, you may first solve an easier version: 20. Valid Parentheses.

Approach 1: Stack

Intuition

To get a good intuition to this problem, we need to ensure that at any point while iterating through `s`, the number of closing brackets `)` should not exceed the number of opening brackets `(` and by the end of the string, the total number of opening and closing brackets must be equal.

Observe that the locked characters (`locked[i] = 1`) cannot be modified, so they must remain fixed. However, we have the flexibility to assign the unlocked characters (`locked[i] = 0`) as either opening or closing brackets, depending on what is needed to maintain balance.

The main challenge is that if at any point the number of closing brackets exceeds the number of opening brackets and there are no unlocked characters available to "fix" the imbalance, it’s impossible to balance the string, and we return false.

And to address this, we need a way to keep track of all previously encountered unlocked characters so we can use them later if needed. Thus a stack is a suitable data structure for this, because it follows the Last In, First Out (LIFO) principle, which works well for keeping track of unmatched brackets.

To implement this, we iterate through the string, whenever we encounter an unlocked character (locked[i] = 0), we push its index onto the stack.

If we encounter a closing bracket `)` and find that the number of closing brackets exceeds the number of opening brackets at that point, we can "fix" the imbalance by popping an index from the stack and treating that unlocked character as an opening bracket `(`.

If at any point we need an unlocked character to balance the string but the stack is empty (i.e., there are no more unlocked characters left), it means balancing the string is impossible, and we return false.

After processing all the characters in the string:

- If the stack still contains indices of unused unlocked characters, we can pair them up to form balanced brackets, such as `()()()`.
- As long as the number of opening and closing brackets is equal by the end, the string is balanced, and we return true.

Algorithm

1. If the length of the string `s` is odd, return `false` because an odd-length string cannot have balanced parentheses.
2. Use a stack `openBrackets` to keep track of the indices of open parentheses `'('` in the locked positions and a stack `unlocked` to keep track of the indices of positions where parentheses can be changed (`locked[i] == '0'`).
3. For each character in the string `s`, check:
   - If the position is unlocked (i.e., `locked[i] == '0'`), add its index to the `unlocked` stack.
   - If the character is an open parenthesis `'('`, add its index to the `openBrackets` stack.
   - If the character is a close parenthesis `')'`:
     - If there is a matching open parenthesis (i.e., the `openBrackets` stack is not empty), pop the stack.
     - If no open parenthesis is available, try to use an unlocked position and pop the `unlocked` stack to match with it.
     - If neither an open parenthesis nor an unlocked position is available to match, return `false`.
4. After processing all characters, check if there are any unmatched open parentheses remaining in the `openBrackets` stack.
   - If there are unmatched open parentheses, try to match them with the available unlocked positions and pop the stacks.
   - If any open parentheses remain unmatched, return `false`. Otherwise, return `true`.

Implementation

```
/*
    Stack
    Time compelxity: O(2n)
    Space compelxity: O(n)
*/
class Solution {
public:
    bool canBeValid(std::string s, std::string locked){
        int n=s.size();
        if (n%2!=0) return false;
        std::stack<int> open,unlocked;
        for(int i=0;i<n;++i){
            if(locked[i]=='0') unlocked.push(i);
            else if(s[i]=='(') open.push(i);
            else if(s[i]==')'){
                if(!open.empty()) open.pop();
                else if (!unlocked.empty()) unlocked.pop();
                else return false;
            }
        }
        // Match remaining open brackets with unlocked characters
        while(!open.empty() && !unlocked.empty() && open.top()<unlocked.top()) {
            open.pop();
            unlocked.pop();
        }
        return open.empty();
    }
};
```

Approach 2: Constant Space

Intuition

In the previous approach, we used a stack to store the unlocked characters and open brackets in the order they appear in the string. However, do we actually need a stack, or is a simple count of the unlocked characters and open brackets sufficient?

The stack indices are required when matching the remaining opening brackets with the unlocked characters, as shown in the code snippet below:

```
// Match remaining open brackets with unlocked characters
 while(!open.empty() && !unlocked.empty() && open.top()<unlocked.top()) {
         open.pop();
         unlocked.pop();
 }
```

To address this, we could explore a trick to match the brackets using only the counts of the unpaired opening brackets and unlocked characters.

Since we want to balance the remaining opening brackets, note that the unlocked characters towards the end of the string can be converted into closing brackets to pair them up. This allows us to iterate from the end of the string `s` while maintaining a `balance` variable to check whether the parentheses are balanced.

We use the integer counters `openBrackets` and `unlocked` from the previous steps:

- If we encounter an unlocked character, we can treat it as a closing bracket.
- If the `balance` variable indicates that the string is unbalanced at any point, we return `false`.

Finally, if all the `openBrackets` are balanced by the end of the iteration, we can return `true`. Otherwise, we return `false`.

Algorithm

1. Initialize `length` as the size of the string `s`.
2. Check if the `length` is odd:
   - If `length % 2 == 1`, return `false`.
3. Initialize variables:
   - `openBrackets` to count the unmatched opening brackets.
   - `unlocked` to count the wildcard positions.
4. Perform a forward pass to process the string:
   - Iterate through `s` from left to right.
   - For each character:
     - If `locked[i] == '0'`, increment `unlocked`.
     - If `s[i] == '('`, increment `openBrackets`.
     - If `s[i] == ')'`:
       - If `openBrackets > 0`, decrement `openBrackets`.
       - Else if `unlocked > 0`, decrement `unlocked`.
       - Else, return `false`.
5. Perform a reverse pass to match remaining open brackets:
   - Initialize `balance` to track excess unmatched opening brackets.
   - Iterate through `s` from right to left.
   - For each character:
     - If `locked[i] == '0'`, decrement `balance` and `unlocked`.
     - If `s[i] == '('`, increment `balance` and decrement `openBrackets`.
     - If `s[i] == ')'`, decrement `balance`.
     - If `balance > 0`, return `false`.
     - If `unlocked == 0` and `openBrackets == 0`, break out of the loop.
6. After the reverse pass:
   - If `openBrackets > 0`, return `false`.
7. Return `true` if no unmatched brackets remain.

Implementation

```
/*
    Space optimization
    Time compelxity: O(2n)
    Space compelxity: O(1)
*/
class Solution {
public:
    bool canBeValid(std::string s, std::string locked){
        int n=s.size();
        if (n%2!=0) return false;
        int open=0,unlocked=0;
        for(int i=0;i<n;++i){
            if(locked[i]=='0') unlocked++;
            else if(s[i]=='(') open++;
            else if(s[i]==')'){
                if(open>0) open--;
                else if (unlocked>0) unlocked--;
                else return false;
            }
        }
        // Match remaining open brackets with unlocked characters
        int level=0;
        for(int i=n-1;i>=0;--i){
            if(locked[i]=='0'){
                level--;
                unlocked--;
            }
            else if(s[i]=='('){
                level++;
                open--;
            }
            else if(s[i]==')') level--;
            if(level>0) return false;
            if(unlocked==0&&open==0) break;
        }
        return open<=0;
    }
};
```