Valid Parenthesis String

Given a string containing only three types of characters: '(', ')' and '*', write a function to check whether this string is valid. We define the validity of a string by these rules:

1. Any left parenthesis `'('` must have a corresponding right parenthesis `')'`.
2. Any right parenthesis `')'` must have a corresponding left parenthesis `'('`.
3. Left parenthesis `'('` must go before the corresponding right parenthesis `')'`.
4. `'*'` could be treated as a single right parenthesis `')'` or a single left parenthesis `'('` or an empty string.
5. An empty string is also valid.

Example 1:

```
Input: "()"
Output: True
```

Example 2:

```
Input: "(*)"
Output: True
```

Example 3:

```
Input: "(*))"
Output: True
```

Note:

1. The string size will be in the range [1, 100].

Solution 1

The idea is to similar to validate a string only contains '(' and ')'. But extend it to tracking the lower and upper bound of valid '(' counts. My thinking process is as following.

scan from left to right, and record counts of unpaired ‘(’ for all possible cases. For ‘(’ and ‘)’, it is straightforward, just increment and decrement all counts, respectively.
When the character is '*', there are three cases, ‘(’, empty, or ‘)’, we can think those three cases as three branches in the ongoing route.
Take “(**())” as an example. There are 6 chars:
----At step 0: only one count = 1.
----At step 1: the route will be diverted into three branches.
so there are three counts: 1 - 1, 1, 1 + 1 which is 0, 1, 2, for ‘)’, empty and ‘(’ respectively.
----At step 2 each route is diverged into three routes again. so there will be 9 possible routes now.
-- For count = 0, it will be diverted into 0 – 1, 0, 0 + 1, which is -1, 0, 1, but when the count is -1, that means there are more ‘)’s than ‘(’s, and we need to stop early at that route, since it is invalid. we end with 0, 1.
-- For count = 1, it will be diverted into 1 – 1, 1, 1 + 1, which is 0, 1, 2
-- For count = 2, it will be diverted into 2 – 1, 2, 2 + 1, which is 1, 2, 3
To summarize step 2, we end up with counts of 0,1,2,3
----At step 3, increment all counts --> 1,2,3,4
----At step 4, decrement all counts --> 0,1,2,3
----At step 5, decrement all counts --> -1, 0,1,2, the route with count -1 is invalid, so stop early at that route. Now we have 0,1,2.
In the very end, we find that there is a route that can reach count == 0. Which means all ‘(’ and ‘)’ are cancelled. So, the original String is valid.
Another finding is counts of unpaired ‘(’ for all valid routes are consecutive integers. So we only need to keep a lower and upper bound of that consecutive integers to save space.
One case doesn’t show up in the example is: if the upper bound is negative, that means all routes have more ‘)’ than ‘(’ --> all routes are invalid --> stop and return false.

Hope this explanation helps.

```java
    public boolean checkValidString(String s) {
        int low = 0;
        int high = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                low++;
                high++;
            } else if (s.charAt(i) == ')') {
                if (low > 0) {
                    low--;
                }
                high--;
            } else {
                if (low > 0) {
                    low--;
                }
                high++;
            }
            if (high < 0) {
                return false;
            }
        }
        return low == 0;
    }
```

written by howei original link here

Solution 2

1. How to check valid parenthesis w/ only `(` and `)`? Easy. Count each char from left to right. When we see `(`, count++; when we see `)` count--; if count < 0, it is invalid (`)` is more than `(`); At last, count should == 0.
2. This problem added `*`. The easiest way is to try `3` possible ways when we see it. Return true if one of them is valid.

```
class Solution {
    public boolean checkValidString(String s) {
        return check(s, 0, 0);
    }

    private boolean check(String s, int start, int count) {
        if (count < 0) return false;

        for (int i = start; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c == '(') {
                count++;
            }
            else if (c == ')') {
                if (count <= 0) return false;
                count--;
            }
            else if (c == '*') {
                return check(s, i + 1, count + 1) || check(s, i + 1, count - 1) ||
check(s, i + 1, count);
            }
        }

        return count == 0;
    }
}
```

written by shawngao original link here

Solution 3

The basic idea is to track the index of the left bracket and star position.
Push all the indices of the star and left bracket to their stack respectively.
STEP 1
Once a right bracket comes, **pop left bracket stack first if it is not empty.** If the left bracket stack is empty, pop the star stack if it is not empty. **A false return can be made provided that both stacks are empty.**

STEP 2
Now attention is paid to the remaining stuff in these two stacks. Note that the left bracket **CANNOT** appear after the star as there is NO way to balance the bracket. In other words, whenever there is a left bracket index appears after the Last star, **a false statement can be made.** Otherwise, pop out each from the left bracket and star stack.

STEP 3
A correct sequence should have an empty left bracket stack. You don't need to take care of the star stack.

Final Remarks:
Greedy algorithm is used here. We always want to use left brackets to balance the right one first as the * symbol is a wild card. There is probably an O(1) space complexity but I think this is worth mentioning.

```
    public boolean checkValidString(String s) {
        Stack<Integer> leftID = new Stack<>();
        Stack<Integer> starID = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch == '(')
                leftID.push(i);
            else if (ch == '*')
                starID.push(i);
            else {
                if (leftID.isEmpty() && starID.isEmpty())   return false;
                if (!leftID.isEmpty())
                    leftID.pop();
                else
                    starID.pop();
            }
        }
        while (!leftID.isEmpty() && !starID.isEmpty()) {
            if (leftID.pop() > starID.pop())
                return false;
        }
        return leftID.isEmpty();
    }
```

written by MichaelPhelps original link here