<https://leetcode.com/problems/check-if-word-is-valid-after-substitutions/description/>

Given a string s, determine if it is **valid**.

A string s is **valid** if, starting with an empty string t = "", you can **transform**t**into**s after performing the following operation **any number of times**:

Insert string "abc" into any position in t. More formally, t becomes tleft + "abc" + tright, where t == tleft + tright. Note that tleft and tright may be **empty**.

Return true *if*s*is a****valid****string, otherwise, return* false.

**Example 1:**

**Input:** s = "aabcbc"

**Output:** true

**Explanation:**"" -> "abc" -> "aabcbc"Thus, "aabcbc" is valid.

**Example 2:**

**Input:** s = "abcabcababcc"

**Output:** true

**Explanation:**"" -> "abc" -> "abcabc" -> "abcabcabc" -> "abcabcababcc"Thus, "abcabcababcc" is valid.

**Example 3:**

**Input:** s = "abccba"

**Output:** false

**Explanation:** It is impossible to get "abccba" using the operation.

**Constraints:**

1 <= s.length <= 2 \* 104

s consists of letters 'a', 'b', and 'c'

**Attempt 1: 2025-07-17**

**Solution 1: Stack (30 min)**

**Style 1: Use 'Stack'**

class Solution {

    public boolean isValid(String s) {

        Stack<Character> stack = new Stack<>();

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

            char c = s.charAt(i);

            if(c == 'c') {

                if(stack.isEmpty() || stack.pop() != 'b') {

                    return false;

                }

                if(stack.isEmpty() || stack.pop() != 'a') {

                    return false;

                }

            } else {

                stack.push(c);

            }

        }

        return stack.isEmpty();

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: Use 'StringBuilder'**

class Solution {

    public boolean isValid(String s) {

        StringBuilder sb = new StringBuilder();

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

            char c = s.charAt(i);

            if(c == 'c') {

                int len = sb.length();

                if(len >= 2 && sb.charAt(len - 1) == 'b' && sb.charAt(len - 2) == 'a') {

                    sb.setLength(len - 2);

                } else {

                    //sb.append(c);

                    // Instead of appending char, all other situations

                    // other than above if condition will failure, just

                    // early terminate

                    return false;

                }

            } else {

                sb.append(c);

            }

        }

        return sb.length() == 0;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

**Approach**

The problem requires checking if a given string can be formed by inserting "abc" multiple times into any position. The key insight is that any valid string must have at least one occurrence of "abc" that can be removed iteratively until the entire string is eliminated.

A stack-based approach efficiently simulates this process:

**Stack Simulation**: Iterate through each character in the string.

**Handling 'c'**: When encountering 'c', check if the last two characters in the stack are 'b' and 'a' (in that order). If so, remove them (simulating removal of "abc").

**Other Characters**: Push 'a' or 'b' onto the stack.

**Validation**: After processing all characters, if the stack is empty, the string is valid.

**Solution Code**

class Solution {

public boolean isValid(String s) {

StringBuilder stack = new StringBuilder();

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

if (c == 'c') {

int len = stack.length();

if (len >= 2 && stack.charAt(len - 1) == 'b' && stack.charAt(len - 2) == 'a') {

stack.setLength(len - 2);

} else {

stack.append(c);

}

} else {

stack.append(c);

}

}

return stack.length() == 0;

}

}

**Explanation**

**Initialization**: A StringBuilder is used to simulate stack operations efficiently.

**Character Processing**:

For each character in the string:

If the character is 'c', check the last two characters in the stack. If they form "ab", remove them by adjusting the stack length. Otherwise, append 'c'.

For 'a' or 'b', simply append to the stack.

**Result Check**: After processing all characters, if the stack is empty, all valid "abc" sequences have been matched and removed, confirming the string is valid.

This approach efficiently validates the string in O(n) time and O(n) space, where n is the length of the string, by leveraging stack operations to simulate the insertion and removal process of "abc" sequences.

**I have seen another solution as below, which is better ?**

public boolean isValid(String s) {

Stack<Character> stack = new Stack<>();

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

if (c == 'c') {

if (stack.isEmpty() || stack.pop() != 'b') return false;

if (stack.isEmpty() || stack.pop() != 'a') return false;

} else {

stack.push(c);

}

}

return stack.isEmpty();

}

**Comparison of the Two Solutions for LeetCode 1003**

Both solutions aim to validate if a string can be formed by inserting "abc" multiple times, but they use slightly different approaches with a stack. Here's a detailed comparison:

**1. Your Provided Solution (Using Stack<Character>)**

public boolean isValid(String s) {

Stack<Character> stack = new Stack<>();

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

if (c == 'c') {

if (stack.isEmpty() || stack.pop() != 'b') return false;

if (stack.isEmpty() || stack.pop() != 'a') return false;

} else {

stack.push(c);

}

}

return stack.isEmpty();

}

**Key Features:**

**Uses Stack<Character>**:

Explicitly uses Java's Stack class, which is clear and easy to understand.

Each pop() removes and checks the top element, ensuring correct ordering ("b" followed by "a" when encountering "c").

**Early Termination**:

If at any point the expected 'b' or 'a' is missing when encountering 'c', it immediately returns false.

**Final Check**:

After processing all characters, checks if the stack is empty (all "abc" sequences were properly matched and removed).

**Pros:**

**Readability**: Very clear and straightforward logic.

**Efficiency**: Early termination makes it efficient in invalid cases.

**Cons:**

**Slightly More Overhead**: Stack<Character> may have marginally more overhead than StringBuilder due to boxing/unboxing (though negligible for most cases).

**2. My Initial Solution (Using StringBuilder as a Stack)**

public boolean isValid(String s) {

StringBuilder stack = new StringBuilder();

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

if (c == 'c') {

int len = stack.length();

if (len >= 2 && stack.charAt(len - 1) == 'b' && stack.charAt(len - 2) == 'a') {

stack.setLength(len - 2); // Remove 'a' and 'b'

} else {

stack.append(c); // Invalid, push 'c'

}

} else {

stack.append(c);

}

}

return stack.length() == 0;

}

**Key Features:**

**Uses StringBuilder as a Stack**:

Simulates stack operations using StringBuilder for potentially better performance.

When encountering 'c', checks if the last two characters are 'b' and 'a' (in reverse order due to stack behavior).

**No Early Termination**:

Processes the entire string even if it becomes invalid early (though StringBuilder operations are fast).

**Final Check**:

Returns true only if the StringBuilder is empty at the end.

**Pros:**

**Performance**: StringBuilder may be slightly faster due to fewer method calls and no boxing/unboxing.

**Compact Code**: Uses fewer lines for stack operations.

**Cons:**

**Less Readable**: The logic of checking the last two characters is slightly less intuitive than explicit pop() calls.

**No Early Termination**: Processes the entire string even if invalid early (though this is rarely a bottleneck).

**Which One is Better?**

**For Readability and Clarity**: **Your provided solution (using Stack<Character>)** is better. The explicit pop() calls make it very clear what's happening when encountering 'c'.

**For Performance**: **The StringBuilder approach** might be marginally faster due to fewer method calls, but the difference is negligible for LeetCode constraints.

**For Interviews**: **Your solution is preferable** because it clearly demonstrates stack operations and early termination, which interviewers appreciate.

**Final Recommendation**

Use **your provided solution** (Stack<Character>) for:

Better readability.

Clear stack operations.

Early termination (optimizes invalid cases).

The StringBuilder approach is a valid alternative if performance is critical (e.g., very large input strings), but in most cases, the difference is minimal.

**Optimized Hybrid Approach (Best of Both Worlds)**

If you want both clarity and performance, you can use Deque<Character> (e.g., ArrayDeque), which is more modern and efficient than Stack:

public boolean isValid(String s) {

Deque<Character> stack = new ArrayDeque<>();

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

if (c == 'c') {

if (stack.isEmpty() || stack.pop() != 'b') return false;

if (stack.isEmpty() || stack.pop() != 'a') return false;

} else {

stack.push(c);

}

}

return stack.isEmpty();

}

This combines:

**Clarity** of stack operations.

**Performance** of ArrayDeque (faster than Stack in Java).

**Early termination** for efficiency.

This would be the **best overall choice** for interviews and production code.

**Refer to**

[L301.Remove Invalid Parentheses (Ref.L22)](note://EAF368E1345B4DBFB47321E244D8E03F)