<https://leetcode.com/problems/backspace-string-compare/description/>

Given two strings s and t, return true **if they are equal when both are typed into empty text editors**. '#' means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

**Example 1:**

**Input:** s = "ab#c", t = "ad#c"

**Output:** true

**Explanation:** Both s and t become "ac".

**Example 2:**

**Input:** s = "ab##", t = "c#d#"

**Output:** true

**Explanation:** Both s and t become "".

**Example 3:**

**Input:** s = "a#c", t = "b"

**Output:** false

**Explanation:** s becomes "c" while t becomes "b".

**Constraints:**

1 <= s.length, t.length <= 200

s and t only contain lowercase letters and '#' characters.

**Follow up:** Can you solve it in O(n) time and O(1) space?

**Attempt 1: 2023-03-04**

**Solution 1: Stack (10 min)**

**Wrong Solution:**

**Test out by:**

Input: s = "y#fo##f", t = "y#f#o##f"

Output: false

Expected: true

class Solution {

    public boolean backspaceCompare(String s, String t) {

        Stack<Character> s1 = new Stack<>();

        Stack<Character> s2 = new Stack<>();

        char[] s\_chars = s.toCharArray();

        char[] t\_chars = t.toCharArray();

        for(char c : s\_chars) {

            if(c == '#' && !s1.isEmpty()) {

                s1.pop();

            } else  {

                s1.push(c);

            }

        }

        for(char c : t\_chars) {

            if(c == '#' && !s2.isEmpty()) {

                s2.pop();

            } else {

                s2.push(c);

            }

        }

        while(!s1.isEmpty() && !s2.isEmpty()) {

            if(s1.pop() != s2.pop()) {

                return false;

            }

        }

        if(!s1.isEmpty() || !s2.isEmpty()) {

            return false;

        }

        return true;

    }

}

**Correct Solution:**

class Solution {

    public boolean backspaceCompare(String s, String t) {

        Stack<Character> s1 = new Stack<>();

        Stack<Character> s2 = new Stack<>();

        char[] s\_chars = s.toCharArray();

        char[] t\_chars = t.toCharArray();

        for(char c : s\_chars) {

            if(c == '#' && !s1.isEmpty()) {

                s1.pop();

            } else if(c != '#') {

                s1.push(c);

            }

        }

        for(char c : t\_chars) {

            if(c == '#' && !s2.isEmpty()) {

                s2.pop();

            } else if(c != '#') {

                s2.push(c);

            }

        }

        while(!s1.isEmpty() && !s2.isEmpty()) {

            if(s1.pop() != s2.pop()) {

                return false;

            }

        }

        if(!s1.isEmpty() || !s2.isEmpty()) {

            return false;

        }

        return true;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 2: Two Pointers (30 min)**

class Solution {

    public boolean backspaceCompare(String s, String t) {

        int i = s.length() - 1;

        int j = t.length() - 1;

        int sSkip = 0;

        int tSkip = 0;

        while(i >= 0 || j >= 0) {

            // Find position of next possible char in String s

            while(i >= 0) {

                if(s.charAt(i) == '#') {

                    sSkip++;

                    i--;

                } else if(sSkip > 0) {

                    sSkip--;

                    i--;

                } else {

                    break;

                }

            }

            // Find position of next possible char in String t

            while(j >= 0) {

                if(t.charAt(j) == '#') {

                    tSkip++;

                    j--;

                } else if(tSkip > 0) {

                    tSkip--;

                    j--;

                } else {

                    break;

                }

            }

            // If two actual characters are different

            if(i >= 0 && j >= 0 && s.charAt(i) != t.charAt(j)) {

                return false;

            }

            // If expecting to compare char vs nothing

            // Test out by: s = "ab##", t = "c#d#"

                return false;

            }

            i--;

            j--;

        }

        return true;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/844>

**Problem Description**

This problem involves comparing two strings to determine if they are the same after processing all backspace operations. In the strings, each # character represents a backspace operation. The backspace operation removes the character immediately before it, or does nothing if there is no character to remove (i.e., at the beginning of the string). The goal is to return true if, after applying all backspace operations, the two strings are equal, otherwise return false.

Let's consider an example. If we have the string "ab#c", processing the backspace operations would result in the string "ac", since the # removes the preceding b. On the other hand, "a#d#" after processing would become "", as both characters are removed by backspaces.

The challenge is to do this comparison efficiently without actually reconstructing the strings after applying the backspace operations.

**Intuition**

The solution is based on traversing both strings from the end to the start, simulating the backspace operations as we go. This way, we can compare characters that would appear on screen without building the resultant strings.

Here's how we can think about the problem:

We start by pointing at the last characters of both s and t.

We move backwards through each string. Whenever we encounter a #, it signifies that we need to skip the next non-# character since it is "backspaced." We keep a count of how many characters to skip.

Whenever we are not supposed to skip characters (the skip count is zero), we compare the characters at the current position in both s and t. If they are different, we return false.

If we reach the beginning of one string but not the other (meaning one string has more characters that would appear on screen than the other), the strings are not equal, and we return false.

If both pointers reach the beginning without finding any mismatch, the strings are the same after processing backspaces, and we return true.

In summary, the intuition is to iterate from the end to the beginning of the strings while keeping track of backspaces, hence ensuring that only characters that would appear on the screen are compared.

**Solution Approach**

The implementation uses a two-pointer approach. This means we have a pointer for each string (s and t), starting from the end of the strings and moving towards the beginning. The variables i and j serve as pointers for strings s and t, respectively.

Here's a step-by-step explanation of the solution:

Initialize pointers i and j to the last indices of s and t respectively.

Use two additional variables skip1 and skip2 to keep track of the number of backspaces (#) encountered in each string. These variables indicate how many characters we should skip over as we move backwards.

Use a while loop to walk through both strings concurrently until both pointers reach the beginning of their respective strings.

For each string s and t, if the current character is #, increment the respective skip variable (skip1 for s and skip2 for t) and move the pointer one step back.

If the current character is not # and the skip variable is greater than zero, decrement the skip variable and move the pointer one step back without comparing any characters. This simulates the backspace operation.

If the current character is not # and the skip variable is zero, this is a character that would actually appear on screen, and we can pause this step to compare it against the character in the other string.

Compare the characters from each string that are at the current positions:

If both pointers are within the bounds of their strings and the characters are different, return false.

If one pointer is within the bounds of its string and the other is not, return false, because one string has more visible characters than the other.

Decrement both pointers i and j and return to step 3, continuing this process.

Once both strings have been fully processed, if no mismatches were found, the function returns true.

The beauty of this algorithm is that it simulates the text editing process without needing extra space to store the resultant strings after backspace operations, making it an efficient solution in terms of space complexity, which is O(1). The time complexity of the algorithm is O(N + M), where N and M are the lengths of strings s and t respectively, as each character in both strings is visited at most twice.

**Example Walkthrough**

Let's use the solution approach to compare two example strings, s = "ab##" and t = "c#d#" to determine if they are equal after processing backspace operations.

We'll walk through each step of the solution:

Initialize pointers i to index 3 (last index of s) and j to index 3 (last index of t).

Initialize skip variables skip1 and skip2 to 0.

Step-by-step processing:

**Iteration 1:**

s[i] is # so we increment skip1 to 1 and decrement i to 2.

t[j] is # so we increment skip2 to 1 and decrement j to 2.

**Iteration 2:**

s[i] is # again, so now skip1 becomes 2 and i is decremented to 1.

t[j] is d, but skip2 is 1, so we decrement skip2 to 0 and j to 1 without comparing the characters.

**Iteration 3:**

s[i] is b, but skip1 is 2, so we decrement skip1 to 1 and i to 0.

t[j] is c, and skip2 is 0, so we should compare t[j] with s[i]. However, we notice skip1 is still greater than 0, so we decrement skip1 to 0 and i is now -1 (out of bounds).

**Iteration 4:**

i is out of bounds, so we can't process s anymore.

t[j] is c and skip2 is 0, so c would be a character that should appear on the screen. Since i is out of bounds, we compare an out-of-bounds s[i] with t[j] which has a visible character 'c'.

**Conclusion:**

Since i is out of bounds and j points to a visible character, the strings are not the same after processing the backspace operations. We don't need to check the remaining characters in s since we know at this point that the visible characters are different.

Hence, the function would return false. This example demonstrates that string s becomes empty after applying all backspace operations whereas string t results in the character 'c', making the strings unequal.

**Java Solution**

class Solution {

    public boolean backspaceCompare(String s, String t) {

        // Initialize two pointers for iterating through the strings in reverse.

        int pointerS = s.length() - 1, pointerT = t.length() - 1;

        // Variables to keep track of the number of backspaces found.

        int skipS = 0, skipT = 0;

        // Continue comparing characters until both pointers go beyond the start of the string.

        while (pointerS >= 0 || pointerT >= 0) {

            // Process backspaces in string s.

            while (pointerS >= 0) {

                if (s.charAt(pointerS) == '#') {

                    skipS++; // We found a backspace character.

                    pointerS--; // Move one character back.

                } else if (skipS > 0) {

                    skipS--; // Reduce the backspace count.

                    pointerS--; // Skip over this character.

                } else {

                    break; // Found a character to compare.

                }

            }

            // Process backspaces in string t.

            while (pointerT >= 0) {

                if (t.charAt(pointerT) == '#') {

                    skipT++; // We found a backspace character.

                    pointerT--; // Move one character back.

                } else if (skipT > 0) {

                    skipT--; // Reduce the backspace count.

                    pointerT--; // Skip over this character.

                } else {

                    break; // Found a character to compare.

                }

            }

            // Compare the characters of both strings.

            if (pointerS >= 0 && pointerT >= 0) {

                // If characters do not match, return false.

                if (s.charAt(pointerS) != t.charAt(pointerT)) {

                    return false;

                }

            } else if (pointerS >= 0 || pointerT >= 0) {

                // If one pointer has reached the start but the other has not, they do not match.

                return false;

            }

            // Move to the next characters to compare.

            pointerS--;

            pointerT--;

        }

        // All characters match considering the backspace characters.

        return true;

    }

}

**Time and Space Complexity**

The time complexity of the given code is O(N + M), where N is the length of string s and M is the length of string t. This is because in the worst case, the algorithm may have to iterate through all the characters in both strings once. The backspace character (#) processing only increases the number of iterations by a constant factor, not the overall complexity.

The space complexity of the code is O(1). This is because the space required for the variables i, j, skip1, and skip2 does not depend on the size of the input strings, making it constant space.

**Refer to**

<https://leetcode.com/problems/backspace-string-compare/editorial/>

**Approach #2: Two Pointer [Accepted]**

**Intuition**

When writing a character, it may or may not be part of the final string depending on how many backspace keystrokes occur in the future.

If instead we iterate through the string in reverse, then we will know how many backspace characters we have seen, and therefore whether the result includes our character.

**Algorithm**

Iterate through the string in reverse. If we see a backspace character, the next non-backspace character is skipped. If a character isn't skipped, it is part of the final answer.

See the comments in the code for more details.

class Solution {

public boolean backspaceCompare(String S, String T) {

int i = S.length() - 1, j = T.length() - 1;

int skipS = 0, skipT = 0;

while (i >= 0 || j >= 0) { // While there may be chars in build(S) or build (T)

while (i >= 0) { // Find position of next possible char in build(S)

if (S.charAt(i) == '#') {skipS++; i--;}

else if (skipS > 0) {skipS--; i--;}

else break;

}

while (j >= 0) { // Find position of next possible char in build(T)

if (T.charAt(j) == '#') {skipT++; j--;}

else if (skipT > 0) {skipT--; j--;}

else break;

}

// If two actual characters are different

if (i >= 0 && j >= 0 && S.charAt(i) != T.charAt(j))

return false;

// If expecting to compare char vs nothing

if ((i >= 0) != (j >= 0))

return false;

i--; j--;

}

return true;

}

}

**Complexity Analysis**

Time Complexity: O(M+N), where M,N are the lengths of S and T respectively.

Space Complexity: O(1).