**Backspace String Compare**

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?

/\*\*

\* @param {string} s

\* @param {string} t

\* @return {boolean}

\*/

var backspaceCompare = function(s, t) {

};

Approach #1: Build String [Accepted]

**Intuition**

Let's individually build the result of each string (build(S) and build(T)), then compare if they are equal.

**Algorithm**

To build the result of a string build(S), we'll use a stack based approach, simulating the result of each keystroke.

class Solution {

public boolean backspaceCompare(String S, String T) {

return build(S).equals(build(T));

}

public String build(String S) {

Stack<Character> ans = new Stack();

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

if (c != '#')

ans.push(c);

else if (!ans.empty())

ans.pop();

}

return String.valueOf(ans);

}

}

**Complexity Analysis**

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

#### 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)*O*(*M*+*N*), where M, N*M*,*N* are the lengths of S and T respectively.
* Space Complexity: O(1)*O*(1).