# 844. Backspace String Compare



## 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 `"abc#d"`, processing the backspace operations would result in the string `"acd"`, since the `#` removes the preceding `b`. On the other hand, `"a#c#"` 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:

1. We start by pointing at the last characters of both `s` and `t`.
2. 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.
3. 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`.
4. 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`.
5. 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 (`i` and `j`), 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:

1. Initialize pointers `i` and `j` to the last indices of `s` and `t` respectively.
2. 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.
3. 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.
4. 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.
5. Decrement both pointers `i` and `j` and return to step 3, continuing this process.
6. 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#d"` and `t = "c#d#"` to determine if they are equal after processing backspace operations.

We'll walk through each step of the solution:

1. Initialize pointers `i` to index 3 (last index of `s`) and `j` to index 3 (last index of `t`).
2. Initialize skip variables `skip1` and `skip2` to 0.

Step-by-step processing:

- **Iteration 1:**
  - `s[i]` is `d` 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.

## Python Solution

```python
class Solution:
    def backspace_compare(self, string1: str, string2: str) -> bool:
        # Initialize pointers for both strings starting from the end
        index1, index2 = len(string1) - 1, len(string2) - 1
        # Initialize counters for skip characters ('#')
        skip_count1, skip_count2 = 0, 0

        # Compare characters in the strings from the end
        while index1 >= 0 or index2 >= 0:
            # Find the position of next valid character in string1
            while index1 >= 0:
                if string1[index1] == '#':
                    skip_count1 += 1
                    index1 -= 1
                elif skip_count1 > 0:
                    skip_count1 -= 1
                    index1 -= 1
                else:
                    break  # Found a valid character

            # Find the position of next valid character in string2
            while index2 >= 0:
                if string2[index2] == '#':
                    skip_count2 += 1
                    index2 -= 1
                elif skip_count2 > 0:
                    skip_count2 -= 1
                    index2 -= 1
                else:
                    break  # Found a valid character

            # If both strings have valid characters to compare
            if index1 >= 0 and index2 >= 0:
                if string1[index1] != string2[index2]:
                    # Characters do not match
                    return False
            # If one index is negative, it means one string has more characters left after processing backspaces
            elif index1 >= 0 or index2 >= 0:
                return False

            # Move to the next character
            index1, index2 = index1 - 1, index2 - 1

        # If all characters matched, return True
        return True
```

## Java Solution

```java
class Solution {
    public boolean backspaceCompare(String s, String t) {
        // Initialize pointers for traversing 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++; // Hm 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 (s.charAt(pointerT) == '#') {
                    skipT++; // Hm 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;
    }
}
```

## C++ Solution

```cpp
class Solution {
public:
    /// Function to compare two strings considering '#' as a backspace character.
    bool backspaceCompare(string s, string t) {
        // Initialize two pointers for the end of the strings.
        int tIndex = s.size() - 1, tIndex = t.size() - 1;
        // Initialize counters for the number of backspaces in s and t.
        int sSkip = 0, tSkip = 0;

        // While there are characters to compare in either string.
        while (sIndex >= 0 || tIndex >= 0) {
            // Find position of next possible character in s.
            while (sIndex >= 0) {
                if (s[sIndex] == '#') { // If a backspace char found, increment the skip counter.
                    ++sSkip;
                    --sIndex;
                } else if (sSkip > 0) { // If we have backspaces to apply, decrement the counter and index.
                    --sSkip;
                    --sIndex;
                } else {
                    break; // Found a valid character to compare.
                }
            }

            // Find position of next possible character in t.
            while (tIndex >= 0) {
                if (t[tIndex] == '#') { // If a backspace char found, increment the skip counter.
                    ++tSkip;
                    --tIndex;
                } else if (tSkip > 0) { // If we have backspaces to apply, decrement the counter and index.
                    --tSkip;
                    --tIndex;
                } else {
                    break; // Found a valid character to compare.
                }
            }

            // If both current indices are valid, compare the characters from s and t.
            if (sIndex >= 0 && tIndex >= 0) {
                // If characters don't match, return false.
                if (s[sIndex] != t[tIndex]) return false;
            } else if (sIndex >= 0 || tIndex >= 0) {
                // If one string has ended, but the other has not, they are not the same.
                return false;
            }

            // Move to the next character in each string.
            --sIndex;
            --tIndex;
        }

        // All compared characters are equal, return true.
        return true;
    }
};
```

## Typescript Solution

```typescript
function backspaceCompare(S: string, T: string): boolean {
    // Initialize two pointers, starting from the end of each string.
    let pointerS = S.length - 1;
    let pointerT = T.length - 1;

    // Continue comparing as long as there's a character in either string.
    while (pointerS >= 0 || pointerT >= 0) {
        let skipS = 0; // To count the backspaces in S.
        // Processing backspaces for S.
        while (pointerS >= 0) {
            if (S[pointerS] === '#') {
                skipS++; // Found a backspace, increase count.
                pointerS--; // Move backwards in string S.
            } else if (skipS > 0) {
                skipS--; // Skip the character due to a previous backspace.
                pointerS--; // Move backwards in string S.
            } else {
                break; // Found a valid character to compare.
            }
        }

        let skipT = 0; // To count the backspaces in T.
        // Processing backspaces for T.
        while (pointerT >= 0) {
            if (T[pointerT] === '#') {
                skipT++; // Found a backspace, increase count.
                pointerT--; // Move backwards in string T.
            } else if (skipT > 0) {
                skipT--; // Skip the character due to a previous backspace.
                pointerT--; // Move backwards in string T.
            } else {
                break; // Found a valid character to compare.
            }
        }

        // Compare the characters of S and T at the pointers.
        if (pointerS >= 0 && pointerT >= 0 && S[pointerS] !== T[pointerT]) {
            return false; // Characters do not match.
        }

        // If one string is at the end, make sure the other is too.
        if ((pointerS >= 0) !== (pointerT >= 0)) {
            return false; // One string ended prematurely.
        }

        // Move to the next valid character in the string.
        pointerS--;
        pointerT--;
    }

    // All compared characters matched.
    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.