392. Is Subsequence

Easy | Two Pointers | String | Dynamic Programming

Problem Description

The problem presents us with two strings, `s` and `t`. The task is to determine whether `s` is a subsequence of `t`. A subsequence of a string is formed by removing zero or more characters from the original string without changing the order of the remaining characters. The function should return `true` if `s` can be obtained from `t` by such a deletion process, otherwise `false`.

For example, if `s` is "ace" and `t` is "abcde", then `s` is a subsequence of `t` because you can delete characters 'b' and 'd' from "abcde" without changing the order of the remaining characters to get "ace".

Intuition

To determine if `s` is a subsequence of `t`, we iterate through both strings concurrently from the beginning, using pointers. If a character in `s` matches a character in `t`, we move forward in both strings. If the characters don't match, we only move forward in `t`, since we're looking to delete or skip characters from `t`. When the end of `s` is reached, it signifies that all characters of `s` have been matched in `t` in order, proving that `s` is a subsequence of `t`.

We arrive at this solution approach because it efficiently traverses the strings without the need for backtracking. This approach leverages the fact that the relative order of characters in the subsequence must remain consistent with the larger string.

Solution Approach

The solution provided uses a simple yet effective algorithm: the two-pointer technique. It involves iterating over both strings `s` and `t` simultaneously but independently, with pointers `i` for string `s` and `j` for string `t`.

Here is how the algorithm is implemented step by step:

1. Initialize two variables, `i` and `j`, to zero. These will act as pointers to the current character being compared in strings `s` and `t`, respectively.
2. Use a `while` loop to continue the iteration as long as `i` is less than the length of string `s` and `j` is less than the length of string `t`. This helps in making sure we do not go out of bounds of either string.
3. Inside the `while` loop, compare the characters at the current pointers. If `s[i]` equals `t[j]`, this means the current character in `s` is matched in `t`, and we move the pointer `i` to the next position (`i += 1`).
4. Regardless of whether we found a match or not, move the pointer `j` to the next position (`j += 1`). This is because even if the current characters do not match, we want to continue looking for the next character of `s` in the remainder of `t`.
5. After the loop, if `i` is equal to the length of string `s`, it indicates that all characters of `s` have successfully been found in `t` in the correct order, and the function returns `true`.
6. If the loop ends because `j` reaches the end of `t` but `i` has not yet reached the end of `s`, the function returns `false` because not all characters from `s` could be matched in `t`.

By using this approach, the solution is efficient and eliminates the need for extra data structures, making the space complexity O(1) (since the pointers use a constant amount of extra space) and time complexity O(n), where `n` is the length of string `t`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have two strings, `s` is "axc" and `t` is "ahbgdc".

1. Initialize pointers `i` and `j` to 0, representing the starting positions in `s` and `t`.
2. Enter the `while` loop since `i < len(s)` and `j < len(t)`.
3. Check if `s[i]` is equal to `t[j]` (compare 'a' with 'a'):
   - Since they match, increment `i` to 1 (now `s[i]` is 'x') and increment `j` to 1 (now `t[j]` is 'h').
4. `s[i]` is 'x' and `t[j]` is 'h', they do not match:
   - Increment only `j` to 2 (`t[j]` is now 'b').
5. Continue checking:
   - `s[i]` is 'x' and `t[j]` is 'b' (no match), increment `j` to 3
   - `s[i]` is 'x' and `t[j]` is 'g' (no match), increment `j` to 4
   - `s[i]` is 'x' and `t[j]` is 'd' (no match), increment `j` to 5
   - `s[i]` is 'x' and `t[j]` is 'c' (no match), since `'x'` is not found and `j` has reached the end of `t`, we exit the loop.
6. After exiting the loop, we check if `i` is equal to `len(s)` which in this case it is not (`i` is 1 and `len(s)` is 3).
7. Since `i != len(s)`, we can conclude that not all characters of `s` were matched in `t`, and therefore return `false`.

Throughout this example, we were able to test whether `s` is a subsequence of `t` without needing any extra data structures and by only incrementing the pointers as needed.

Python Solution

```python
class Solution:
    def is_subsequence(self, subsequence: str, sequence: str) -> bool:
        # Initialize two pointers for both the subsequence and the sequence
        subsequence_index = 0
        sequence_index = 0

        # Iterate over the sequence while there are characters left in both
        # the subsequence and the sequence
        while subsequence_index < len(subsequence) and sequence_index < len(sequence):
            # If the current characters match, move to the next character in the subsequence
            if subsequence[subsequence_index] == sequence[sequence_index]:
                subsequence_index += 1
            # Move to the next character in the sequence
            sequence_index += 1

        # Return True if all characters in the subsequence have been matched
        # This is indicated by subsequence_index pointing to the end of subsequence
        return subsequence_index == len(subsequence)

```

Java Solution

```java
class Solution {
    public boolean isSubsequence(String s, String t) {
        // Lengths of the strings s and t
        int lengthS = s.length(), lengthT = t.length();

        // Initialize pointers for both the strings
        int indexS = 0, indexT = 0;

        // Iterate over both strings
        while (indexS < lengthS && indexT < lengthT) {
            // Check if the current character of s matches the current character of t
            if (s.charAt(indexS) == t.charAt(indexT)) {
                // If they match, move the pointer of s forward
                ++indexS;
            }
            // Move the pointer of t forward
            ++indexT;
        }

        // If indexS is equal to the length of s, all characters of s are found in t in sequence
        // Therefore, s is a subsequence of t
        return indexS == lengthS;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // The function checks if 's' is a subsequence of 't'.
    bool isSubsequence(string s, string t) {
        int sLength = s.size(), tLength = t.size(); // Store lengths of both strings.
        int sIndex = 0, tIndex = 0; // Initialize indices for both strings.

        // Loop through both strings.
        for (; sIndex < sLength && tIndex < tLength; ++tIndex) {
            // If characters match, move to the next character in 's'.
            if (s[sIndex] == t[tIndex]) {
                ++sIndex;
            }
        }

        // If we have gone through the entire 's' string, it is a subsequence of 't',
        return sIndex == sLength;
    }
};

```

Typescript Solution

```typescript
function isSubsequence(subString: string, mainString: string): boolean {
    const subStringLength = subString.length;  // Store the length of the subsequence string
    const mainStringLength = mainString.length;  // Store the length of the main sequence string
    let subStringIndex = 0;  // Initialize an index to track the current position in subString

    // Loop over the mainString while there are characters left in both strings
    for (let mainStringIndex = 0; subStringIndex < subStringLength && mainStringIndex < mainStringLength; ++mainStringIndex) {
        // If characters match, increment the index of the subString
        if (subString[subStringIndex] === mainString[mainStringIndex]) {
            ++subStringIndex;
        }
    }

    // If subStringIndex equals to subStringLength, all characters of subString were found in order in mainString
    return subStringIndex === subStringLength;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of string `t`. This is because the function iterates through each character of `t` at most once. The pointer `i` moves only when there is a match in `s` and `t`, which does not change the overall linear time complexity with respect to `t`. It does not iterate more than `n` times even if every character of `s` is found in `t`.

The space complexity of the code is `O(1)` since no additional space is used that grows with the input size. The only extra memory used is for the two pointers `i` and `j`, which use a constant amount of space.