

# 2486. Append Characters to String to Make Subsequence

MediumGreedyTwo PointersString

## Problem Description

In this problem, we're given two strings `s` and `t`, which consist only of lowercase English letters. Our task is to determine the minimum number of characters we must append to the end of `s` to make `t` a subsequence of `s`. To clarify, a subsequence of a string is a new string that's formed from the original string by deleting some (possibly no) characters without changing the order of the remaining characters. For example, if `s` is "abcde" and `t` is "ace", then `t` is already a subsequence of `s`, and we don't need to append any characters. However, if `s` is "abc" and `t` is "dabc", we'd need to append a "d" at the start. Since the problem only allows appending at the end, we count the number of characters in `t` that are not yet in `s` as a subsequence, and that's the number we must append.

## Intuition

The solution for this problem is based on the two-pointer technique, which is often used for problems involving sequences or arrays. The key idea is to iterate through both strings `s` and `t` simultaneously using [two pointers](#) (or indices) and find the parts of `t` that are already a subsequence in `s`. Whenever we find that a character in `t` does not match any further characters in `s` (because we've reached the end of `s` without finding a match), we realize that the remaining characters of `t` must be appended to `s`.

We use [two pointers](#)—`i` for tracking the position in `s` and `j` for tracking the position in `t`. We increment `i` to find a match for `t[j]` in `s`. If we reach the end of `s` (`i == m`) before finding a match, it means the rest of `t` starting from `t[j]` must be appended to `s`. If we successfully find a match for every character in `t`, it implies that `t` is already a subsequence of `s`, and no characters need to be appended.

The solution method effectively compares each character of `t` with the characters in `s`, ensuring the subsequence order of `t` in `s`. The number of iterations (advance of the pointer) in `s` gives us the minimal insertions at the end of `s` to make `t` a subsequence.

## Solution Approach

The solution uses a straightforward approach without any additional data structures, relying only on two indices to traverse the strings.

Here's a step-by-step explanation of how the code works:

- Initialize [two pointers](#): `i` starting at 0 for string `s`, and `j` also starting at 0 for string `t`. These pointers are used to traverse each string.
- Begin a loop over string `t` using `j` as the loop index. For each character `t[j]` in `t`:
  - Increment `i` to find a matching character in `s`. We move `i` forward as long as `i < m` (not at the end of `s`) and `s[i]` is not equal to `t[j]`. This is done with `while i < m and s[i] != t[j]: i += 1`.
  - Once a character in `s` matches `t[j]`, or we have reached the end of `s`, we proceed to check if we have exhausted `s`. If `i == m`, it means that there are no more characters left in `s` to match with `t[j]`, hence the remaining characters of `t` need to be appended.
    - In this case, we return `n - j`, where `n` is the length of `t`, and `j` is the current index. `n - j` gives us the count of additional characters needed from `t` to complete the subsequence.
  - If a match is found, and `i` has not reached the end of `s`, we increment the pointer `i` to continue searching for the next characters of `t`.
- After the loop, if all characters in `t` have a corresponding character in `s`, then we do not need to append any additional characters. Hence we return `0`.

The algorithm's time complexity is  $O(n + m)$  where `n` is the length of `t` and `m` is the length of `s`, because in the worst case we might have to traverse both strings entirely. No extra space is needed except for the pointers, so the space complexity is  $O(1)$ .

This approach is efficient because it minimizes the number of operations and only uses memory for the pointers to the current character in both strings `s` and `t`. It uses the inherent order of the input strings to find the solution without backtracking, making it an elegant solution for the given problem.

## Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose `s` is "xyza" and `t` is "ayz".

- Initialize two pointers: `i = 0` for "xyza" (`s`) and `j = 0` for "ayz" (`t`).
- We are now looking to match "a" from `t` in `s`. Since `s[0]` is "x", and "x"  $\neq$  "a", we increment `i`.
- `i = 1` and `s[1]` is "y", and "y"  $\neq$  "a", we increment `i` again.
- `i = 2` and `s[2]` is "z", and "z"  $\neq$  "a", we increment `i` once more.
- `i = 3` and `s[3]` is "a", and "a" = "a", we found our first match. Now, we increment `j` to look for the next character in `t`.
- `j = 1` and `t[1]` is "y". `i` is also moved to look for "y" in `s`.
- `i` stays at 3 because "a" = `s[3]` and "a"  $\neq$  "y", but we cannot increment `i` anymore since we have reached the end of `s`.
- Since `i == m` (length of `s`), we realize that we need to append the remaining characters of `t` starting from `t[j]` to `s`.
- Therefore, we return `n - j`, which is the length of `t` (`n = 3`) minus the current index of `t` (`j = 1`), giving us `3 - 1 = 2`. So, we must append two characters, "yz", to `s`.

This example confirms that the minimum number of characters we must append to `s` to make `t` a subsequence is 2, with the result being "xyzayz". The result does not have to be the original `t`; it only needs to contain `t` as a subsequence.

## Solution Implementation

### Python

```
class Solution:
    def appendCharacters(self, s: str, t: str) -> int:
        # Initialize the lengths of strings s and t
        s_length, t_length = len(s), len(t)
        # Initialize the index for string s
        index_s = 0

        # Iterate over each character in string t
        for index_t in range(t_length):
            # Move index s in string s until the character matches the current character in string t
            while index_s < s_length and s[index_s] != t[index_t]:
                index_s += 1

            # If index s has reached the end of s, return the remaining number of characters in t
            # that need to be appended to s to make t a substring of the modified s
            if index_s == s_length:
                return t_length - index_t

            # Move to the next character in s after a match is found
            index_s += 1

        # If all characters in t are matched before reaching the end of s, no characters need to be appended
        return 0
```

### Java

```
class Solution {
    public int appendCharacters(String s, String t) {
        // m is the length of string s
        int lengthS = s.length();
        // n is the length of string t
        int lengthT = t.length();

        // Initialize pointers for both strings
        // i for traversing s, j for traversing t
        int i = 0, j = 0;

        // Iterate through string t to find if characters are in string s
        while (i < lengthT) {
            // Move i forward in string s until character at s[i] matches t[j],
            // or until the end of s is reached
            while (i < lengthS && s.charAt(i) != t.charAt(j)) {
                i++;
            }

            // If the end of s is reached before finding a match, return
            // the number of remaining characters to append from t to s
            if (i++ == lengthS) {
                return lengthT - j;
            }

            // Move j forward to the next character in t
            j++;
        }

        // If we reach here, all characters of t are matched in order within s,
        // so there is no need to append any characters, hence return 0
        return 0;
    }
}
```

### C++

```
class Solution {
public:
    int appendCharacters(string s, string t) {
        int m = s.size(); // Length of string s
        int n = t.size(); // Length of string t

        // Initialize pointers for strings s (i) and t (j)
        for (int i = 0, j = 0; i < n; ++i) {
            // Increment i until we find a matching character in s for t[j]
            while (i < m && s[i] != t[j]) {
                ++i;
            }
            // If we reach the end of s, return how many characters to append from t
            if (i == m) {
                return n - j;
            }
            // Move to the next character in s
            i++;
        }
        // If all characters of t are found in s in order, no need to append any character
        return 0;
    }
};
```

### TypeScript

```
function appendCharacters(s: string, t: string): number {
    let m = s.length; // Length of string s
    let n = t.length; // Length of string t

    // Initialize pointers for strings s (i) and t (j)
    let i = 0, j = 0;
    while (i < n) {
        // Increment i until we find a matching character in s for t[j]
        while (i < m && s[i] !== t[j]) {
            i++;
        }
        // If we reach the end of s, return how many characters to append from t
        if (i === m) {
            return n - j;
        }
        // Move to the next character in s and t
        i++;
        j++;
    }
    // If all characters of t are found in s in the given order, no need to append any character
    return 0;
}
```

```
class Solution:
    def appendCharacters(self, s: str, t: str) -> int:
        # Initialize the lengths of strings s and t
        s_length, t_length = len(s), len(t)
        # Initialize the index for string s
        index_s = 0

        # Iterate over each character in string t
        for index_t in range(t_length):
            # Move index s in string s until the character matches the current character in string t
            while index_s < s_length and s[index_s] != t[index_t]:
                index_s += 1

            # If index s has reached the end of s, return the remaining number of characters in t
            # that need to be appended to s to make t a substring of the modified s
            if index_s == s_length:
                return t_length - index_t

            # Move to the next character in s after a match is found
            index_s += 1

        # If all characters in t are matched before reaching the end of s, no characters need to be appended
        return 0
```

## Time and Space Complexity

### Time Complexity

The time complexity of the given code can be analyzed by looking at the nested loops (represented implicitly through the sequential searches within the string `s`):

- For each character in string `t` (`n` characters total), the code potentially iterates through the entirety of string `s` (`m` characters) to find a matching character.
- In the worst case, each character of `t` will be compared to each character in `s` until a match is found or until the end of `s` is reached.
- Therefore, the worst-case scenario would result in a time complexity of  $O(m \times n)$ , where `m` is the length of the string `s` and `n` is the length of the string `t`.

### Space Complexity

The space complexity of the code is quite straightforward:

- The code uses only a fixed number of variables (`m`, `n`, `i`, and `j`) to keep track of the indices and lengths of the strings `s` and `t`.
- No additional data structures are created that grow in size with the input.
- Thus, the space complexity is  $O(1)$ , which means it is constant.