161. One Edit Distance

Medium Java Problems String

Problem Description

The problem provides two strings, `s` and `t`, and asks us to determine if they are one edit distance apart. Being "one edit distance apart" means that there is exactly one simple edit that can transform one string into the other. This edit can be one of the following types:

- Inserting exactly one character into `s` to get `t`.
- Deleting exactly one character from `s` to get `t`.
- Replacing exactly one character in `s` with a different character to get `t`.

To solve this problem, we need to carefully compare the two strings and verify if they differ by exactly one of the aforementioned operations or, in contrast, they are either identical or differ by more than one operation.

Intuition

The solution exploits the fact that if two strings are one edit distance apart, their lengths must differ by at most one. If the lengths differ by more than one, more than one edit is required, which violates the problem constraints. Therefore, the solution begins by comparing the lengths of `s` and `t`:

1. If `s` is shorter than `t`, swap them to ensure that we only have one case to handle for insertions or deletions (the case where `s` is longer than or equal to `t`).
2. If the difference in lengths is greater than one, return `false` immediately.
3. Iterate over the shorter string and compare the characters at each index with the corresponding characters in the longer string.
   - The moment a mismatch is encountered, there are two scenarios to consider based on the length of the strings:
     - If `s` and `t` are of the same length, the only possible edit is to replace the mismatching character in `s` with the character from `t`. Check if the substrings of `s` and `t` after the mismatch indices are identical.
     - If `s` is longer than `t` by one, the only possible edit is a deletion. Check if the substring of `s` after the mismatch index plus one is identical to the substring of `t` after the index of the mismatch.
   - If no mismatch is found and the loop completes, one last check is needed: if `s` is longer than `t` by one character, then the last character of `s` could be the extra character, satisfying the condition for being one edit distance apart.

In conclusion, this solution systematically eliminates scenarios that would require more than one edit and carefully checks for the one allowed edit between the given strings.

Solution Approach

The implementation of the solution provided uses a straightforward approach that does not rely on any additional data structures or complex patterns. Let's take a closer look at the algorithm:

1. **Check Lengths and Swap:** The function begins by making sure that `s` is the longer string. If it's not, it calls itself with the strings reversed.

```
if len(s) < len(t):
    return self.isOneEditDistance(t, s)
```

   This optimizes the code by allowing us to only think about two scenarios (instead of three): replacing a character or deleting a character from `s`.

2. **Check Length Difference:** If the length of `s` is more than one character longer than `t`, they cannot be one edit apart, so the function returns `False`.

```
if m - n > 1:
    return False
```

3. **Iterate and Compare Characters:** The next step involves iterating over the shorter string `t` and checking character by character to find any differences.

```
for i, c in enumerate(t):
    if c != s[i]:
```

   As soon as a difference is found, the function moves to the next step.

4. **Check Possible One Edit Scenarios:** Upon encountering a mismatch, there are now two cases to consider:
   - For strings of the same length (`n == m`), a replacement might be the one edit. The function checks if the substrings after the different characters are the same.

```
return s[i + 1 :] == t[i + 1 :]
```

   - If `s` is longer by one character (`n == n + 1`), it indicates a possible deletion. The code checks if the substring of `s` beyond the next index is the same as the substring of `t` from the current index.

```
return s[i + 1 :] == t[i:]
```

5. **Final Check for Deletion at the End:** If the `for` loop finishes without finding any mismatches, the function finally checks if `s` is exactly one character longer than `t`, which would imply the possibility of the one edit being a deletion at the end.

```
return m == n + 1
```

Throughout its process, this approach relies purely on the properties of strings and their indices. It uses string slicing to compare substrings and eliminate the need for additional operations. This makes for an efficient approach both in terms of space, not requiring any additional storage, and in terms of time, as it can short-circuit and exit as soon as it finds the strings are not one edit distance apart.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider two strings `s = "cat"` and `t = "cut"`. We want to determine whether they are one edit distance apart using the above solution approach.

1. **Check Lengths and Swap:** Since the length of `s` is equal to `t`, no swapping occurs.

```
if len(s) < len(t):
    return self.isOneEditDistance(t, s)
```

2. **Check Length Difference:** The lengths of `s` and `t` are the same, so this condition is not triggered.

```
if m - n > 1:
    return False
```

3. **Iterate and Compare Characters:** We iterate over each character in `t` and compare it with the corresponding character in `s`.

```
for i, c in enumerate(t):
    if c != s[i]:
```

   The loop encounters a mismatch at index 1: `'a'` in `s` is different from `'u'` in `t`.

4. **Check Possible One Edit Scenarios:** Since `s` and `t` are of the same length, we check if replacing the second character in `s` with the corresponding character in `t` would make the strings equal.

```
return s[i + 1 :] == t[i + 1 :]
```

   This returns `True` as `s[2:]` ('t') is identical to `t[2:]` ('t'), indicating that replacing 'a' with 'u' in `s` results in `t`.

So, `s` and `t` are indeed one edit distance apart as we only need to replace one character ('a' with 'u') to transform `s` into `t`.

Solution Implementation

Python

```
class Solution:
    def isOneEditDistance(self, s: str, t: str) -> bool:
        # If s is shorter than t, swap them to make sure s is longer or equal to t
        # This helps in reducing further checks
        if len(s) < len(t):
            return self.isOneEditDistance(t, s)

        # Extract lengths of both strings.
        len_s, len_t = len(s), len(t)

        # There cannot be a one edit distance if the length difference is more than 1
        if len_s - len_t > 1:
            return False

        # Iterate through both strings
        for idx in range(len_t):
            # If the characters at current position are different,
            # It must either be a replace operation when lengths are same,
            # Or it must be an insert operation when lengths are different.
            if s[idx] != t[idx]:
                if len_s == len_t:
                    # The remainder of the strings after this character should
                    # be the same if it is a replace operation.
                    return s[idx + 1:] == t[idx + 1:]
                else:
                    # In case of insert operation, the remainder of the longer string
                    # starting from the next character should be the same as the
                    # rest of shorter string starting from current character.
                    return s[idx + 1:] == t[idx:]

        # If all previous chars are same, the only possibility for one edit distance
        # is when the longer string has one extra character at the end.
        return len_s == len_t + 1
```

Java

```
class Solution {

    public boolean isOneEditDistance(String s, String t) {
        int lengthS = s.length(), lengthT = t.length();

        // Ensure s is the longer string.
        if (lengthS < lengthT) {
            return isOneEditDistance(t, s);
        }

        // If the length difference is more than 1, it's not one edit distance.
        if (lengthS - lengthT > 1) {
            return false;
        }

        // Check each character to see if there's a discrepancy.
        for (int i = 0; i < lengthT; ++i) {
            if (s.charAt(i) != t.charAt(i)) {
                // If the lengths are the same, the rest of the strings must be equal after this character.
                if (lengthS == lengthT) {
                    return s.substring(i + 1).equals(t.substring(i + 1));
                }
                // If the lengths are not the same, the rest of the longer string must be equal to the rest of the shorter string aft
                return s.substring(i + 1).equals(t.substring(i));
            }
        }

        // Covers the case where there is one extra character at the end of the longer string.
        return lengthS == lengthT + 1;
    }
}
```

C++

```
class Solution {
public:
    // Check if string 's' can be converted to string 't' with exactly one edit
    bool isOneEditDistance(string s, string t) {
        int lenS = s.length(), lenT = t.length(); // Use more descriptive variable names

        // Guarantee that 's' is not shorter than 't'
        if (lenS < lenT) return isOneEditDistance(t, s);

        // If the strings differ in length by more than 1, it can't be one edit
        if (lenS - lenT > 1) return false;

        // Iterate through characters in both strings
        for (int i = 0; i < lenT; ++i) {
            // If characters don't match, check the types of possible one edit
            if (s[i] != t[i]) {
                // If lengths are the same, check for replace operation
                if (lenS == lenT) return s.substr(i + 1) == t.substr(i + 1);
                // If lengths differ, check for insert operation
                return s.substr(i + 1) == t.substr(i);
            }
        }

        // If all previous characters matched, it might be an append operation
        return lenS == lenT + 1;
    }
};
```

TypeScript

```
// Check if string 's' can be converted to string 't' with exactly one edit
function isOneEditDistance(s: string, t: string): boolean {
    let lengthS = s.length; // Length of string 's'
    let lengthT = t.length; // Length of string 't'

    // Ensure 's' is not shorter than 't'
    if (lengthS < lengthT) return isOneEditDistance(t, s);

    // If the lenaths differ by more than 1, it can't be a single edit
    if (lengthS - lengthT > 1) return false;

    // Iterate through characters in both strings
    for (let i = 0; i < lengthT; i++) {
        // If characters don't match, check the types of possible one edit
        if (s[i] !== t[i]) {
            // If lengths are the same, check for replace operation
            if (lengthS === lengthT) return s.substring(i + 1) === t.substring(i + 1);
            // If lengths differ, check for insert operation
            return s.substring(i + 1) === t.substring(i);
        }
    }

    // If all previous characters matched, it might be an append operation
    return lengthS === lengthT + 1;
}
```

```
class Solution:
    def isOneEditDistance(self, s: str, t: str) -> bool:
        # If s is shorter than t, swap them to make sure s is longer or equal to t
        # This helps in reducing further checks
        if len(s) < len(t):
            return self.isOneEditDistance(t, s)

        # Extract lengths of both strings.
        len_s, len_t = len(s), len(t)

        # There cannot be a one edit distance if the length difference is more than 1
        if len_s - len_t > 1:
            return False

        # Iterate through both strings
        for idx in range(len_t):
            # If the characters at current position are different,
            # It must either be a replace operation when lengths are same,
            # Or it must be an insert operation when lengths are different.
            if s[idx] != t[idx]:
                if len_s == len_t:
                    # The remainder of the strings after this character should
                    # be the same if it is a replace operation.
                    return s[idx + 1:] == t[idx + 1:]
                else:
                    # In case of insert operation, the remainder of the longer string
                    # starting from the next character should be the same as the
                    # rest of shorter string starting from current character.
                    return s[idx + 1:] == t[idx:]

        # If all previous chars are same, the only possibility for one edit distance
        # is when the longer string has one extra character at the end.
        return len_s == len_t + 1
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(N)` where `N` is the length of the shorter string between `s` and `t`. This complexity arises from the fact that we iterate through each character of the shorter string at most once, comparing it with the corresponding character of the longer string. In the worst case, if all characters up to the last are the same, we perform `N` comparisons.

Space Complexity

The space complexity of the code is `O(1)`. No additional space is required that scales with the input size. The only extra space used is for a few variables to store the lengths of strings and for indices, which does not depend on the size of the input strings.