2399. Check Distances Between Same Letters

Easy Array Hash Table String Leetcode Link

Problem Description

You are tasked with determining if a given string `s` is "well-spaced". A string is well-spaced if it follows a specific distance rule. The string `s` only contains lowercase English letters, and each letter appears exactly twice in the string. In addition to the string, you're given an integer array `distance` with 26 elements corresponding to the letters of the English alphabet, which are indexed from `0` for 'a' to `25` for 'z'.

The distance rule is as follows: The number of letters between the two occurrences of any given letter in the string must equal the value specified in the `distance` array. For instance, if `s` has two 'a's and the `distance[0]` (since 'a' corresponds to index 0) is 3, there should be exactly three letters between the two 'a's in `s`. The rule applies to all letters present in `s`, and you do not need to consider `distance` values for letters that do not appear in `s`.

Your goal is to return `true` if the string meets this well-spaced criterion for all the letters it contains. Otherwise, you will return `false`.

Intuition

To verify if the string `s` is well-spaced according to the given `distance` array, you need to track the position of each letter's first appearance and check the spacing when you find its second appearance.

By iterating through the string, each time you encounter a character, check if it's the first or second appearance. If it's the first appearance, you record its position (index). If it's the second appearance, you calculate the number of characters between this and the first appearance. You then compare this calculated number with the corresponding value in the `distance` array.

To efficiently keep track of the positions, you can use a dictionary `d`, mapping each character to its first occurrence index. As you go through the string, you update the dictionary with the current position for the first occurrence. Upon finding the second occurrence of a letter, you immediately check if the distance between the first and second occurrence matches the `distance` value for that letter. If there's a mismatch for any letter, you return `false` right away since the string is not well-spaced. If no mismatch is found throughout the string, you return `true` after the iteration.

The `ord(c) - ord('a')` part of the solution is used to convert a character to its corresponding `distance` index (i.e., 'a' to 0, 'b' to 1, and so on).

Using a dictionary for tracking allows for an efficient solution with just one pass through the string, making the algorithm run in linear time relative to the length of the string.

Solution Approach

The solution implements a straightforward and efficient approach to check if the given string `s` is well-spaced according to the `distance` array. Here's a step-by-step explanation of how the code works:

- The solution first initializes a dictionary `d` using `defaultdict(int)` from the `collections` module. This dictionary will keep track of the first occurrence index of each letter in the string.
- Then, it iterates over each character `c` in the string `s` along with its index `i` (1-indexed, because we want to compute the number of letters between the occurrences, not the distance in terms of indices). The `enumerate` function is essential here as it provides both the character and its index in one go.
- Inside the loop, for each character `c` encountered, the code checks if it's already present in the dictionary `d` (which would mean this is its second occurrence).
  - If it is not in the dictionary, it means this is the first time the character has appeared, so the current index `i` is recorded in `d[c]`.
  - If it is present, meaning this is the second occurrence, the code immediately calculates the distance between the two occurrences. This is done by subtracting `d[c]` (where the first occurrence of `c` was recorded) from the current index `i` and subtracting one more to account for the character itself (since we want the number of characters between the first and second occurrences).
- After calculating the distance between occurrences for the character `c`, the code compares this number to the expected distance provided in the `distance` array. If the calculated distance does not match the expected `distance` value, which is `distance[ord(c) - ord('a')]` (turns the letter `c` into its respective alphabet index), the function returns `False` indicating that `s` isn't a well-spaced string.
- If the iteration completes without returning `False`, it means all characters in `s` satisfy the well-spaced criteria, and the function finally returns `True`.

This solution uses a dictionary, which is a hash map, to store and access the first occurrence indices of the characters efficiently. The decision to check the distance immediately upon the second occurrence of each character ensures that we don't need a second pass through the string or additional storage, leading to a time complexity of O(n), where n is the length of the string, and a constant space complexity since the dictionary will store at most one entry per unique character in `s`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have a string `s = "aibjca"` and a `distance` array such that `distance[0] = 2`, `distance[1] = 2`, `distance[2] = 1`, and `distance[3] = 1` (all other values in `distance` array are irrelevant since the corresponding characters are not in `s`). We need to verify if `s` is well-spaced according to the distance rule defined.

Now we walk through the solution steps using this example:

1. Initialize a dictionary `d` to track the first occurrence index of each letter.
2. Start iterating over the string `s`:
   - Index 1: `c = 'a'`. `'a'` is not in `d`, so we add `'a'` with its index `d['a'] = 1`.
   - Index 2: `c = 'i'`. `'i'` is not in `d`, so we add `'i'` with its index `d['i'] = 2`.
   - Index 3: `c = 'b'`. `'b'` is not in `d`, so we add `'b'` with its index `d['b'] = 3`.
   - Index 4: `c = 'j'`. `'j'` is not in `d`, so we add `'j'` with its index `d['j'] = 4`.
   - Index 5: `c = 'c'`. `'c'` is not in `d`, so we add `'c'` with its index `d['c'] = 5`.
   - Index 6: `c = 'a'`. `'a'` is in `d`, so we check the spacing. The first occurrence was at index 1, and the current index is 6. The distance between them is `6 - 1 - 1 = 4`, but `distance[0]` (`distance` for 'a') is 2. The calculated distance doesn't match the expected distance, so we return `False`.

If, however, the `distance` for `'a'` was given as `4`, then the example would continue as:

- Index 7: `c = 'i'`. `'i'` is in `d`, so we check the spacing. The first occurrence was at index 2, and the current index is 7. The distance between them is `7 - 2 - 1 = 4`, which does not match `distance[8] = 2` (`distance` for 'i'), we return `False`.
- Index 8: `c = 'b'`. `'b'` is in `d`, so we check the spacing. The first occurrence was at index 3, and the current index is 8. The distance between them is `8 - 3 - 1 = 4`, which does not match `distance[1] = 2` (`distance` for 'b'), we return `False`.
- Index 9: `c = 'j'`. `'j'` is in `d`, so we check the spacing. The first occurrence was at index 4, and the current index is 9. The distance between them is `9 - 4 - 1 = 4`, which does not match `distance[9] = 1` (`distance` for 'j'), we return `False`.

No exact matches occurred, and we return `False` in each case, so `s` is not well-spaced. If all distances were to match, we would continue through the entire string and return `True` at the end.

This example demonstrates how the positions are recorded and how the solution approach identifies if a string is well-spaced or not by comparing the actual distance between recurring characters with the specified distances in the given `distance` array.

Python Solution

```python
from collections import defaultdict

class Solution:
    def checkDistances(self, s: str, distances: List[int]) -> bool:
        # Dictionary to keep track of the first occurrence index of each character
        first_occurrence = defaultdict(int)

        # Iterate through the string while enumerating, which provides both
        # the index and the character
        for index, char in enumerate(s, 1): # Starting index at 1 for calculation convenience
            # If the character has been seen before and the distance constraint is not met
            if first_occurrence[char] and index - first_occurrence[char] - 1 != distances[ord(char) - ord('a')]:
                return False  # The distance constraint is violated
            # Record the first occurrence index of the character
            first_occurrence[char] = index

        # If all characters meet their distance constraints, return True
        return True

```

Java Solution

```java
class Solution {
    public boolean checkDistances(String s, int[] distance) {
        // Create an array to store the positions where characters 'a' to 'z' were seen last.
        int[] lastSeenPositions = new int[26];

        // Loop over characters of the string.
        for (int index = 1, stringLength = s.length(); index <= stringLength; ++index) {
            // Calculate the array index for the character (0 for 'a', 1 for 'b', etc.)
            int charIndex = s.charAt(index - 1) - 'a';

            // If the character has been seen before (i.e., its position is recorded in lastSeenPositions)
            // check if the current distance is equal to the distance specified in the array.
            if (lastSeenPositions[charIndex] > 0 && index - lastSeenPositions[charIndex] - 1 != distance[charIndex]) {
                return false; // If the distance doesn't match, return false.
            }

            // Record the current position (index) for the character.
            lastSeenPositions[charIndex] = index;
        }

        // If all distances match, return true.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function checks if the distances between each identical pair of characters in the string 's'
    // are as described in the 'distance' vector, where 'distance[i]' represents the distance for the character 'a' + i.
    // Returns true if distances are as expected, false otherwise.

    bool checkDistances(string s, vector<int>& distance) {
        // Create an array to store the most recent index (1-based) of each letter encountered in the string 's'.
        // Initially all set to zero indicating that letters haven't been encountered yet.
        int lastIndex[26] = {};

        // Iterate through the string characters
        for (int i = 1; i <= s.size(); ++i) {
            // Calculate the 0-based index of the letter in 'lastIndex' array, 'a' corresponds to 0, 'b' to 1, and so on.
            int charIndex = s[i - 1] - 'a';

            // If the letter has been encountered before (index non-zero),
            // and if the current distance doesn't match the expected distance from the 'distance' vector,
            // return false because the distances don't match.
            if (lastIndex[charIndex] && i - lastIndex[charIndex] - 1 != distance[charIndex]) {
                return false;
            }

            // Update the index for the current letter to the current position (1-based).
            lastIndex[charIndex] = i;
        }

        // If all the distances match, return true.
        return true;
    }
};

```

Typescript Solution

```typescript
function checkDistances(s: string, distances: number[]): boolean {
    // Initialize the length of the string for future use.
    const stringLength = s.length;
    // Create an array to keep track of the last seen position of each character.
    const lastSeenPositions: number[] = new Array(26).fill(0);

    // Loop through each character of the string.
    for (let index = 0; index < stringLength; ++index) {
        // Calculate the alphabet index of the current character (0 for 'a', 1 for 'b', etc.).
        const charIndex = s.charCodeAt(index) - 'a'.charCodeAt(0);

        // Check if this character has been seen before.
        if (lastSeenPositions[charIndex] !== 0) {
            // Calculate the actual distance between the current and the last occurrence.
            const actualDistance = index - lastSeenPositions[charIndex];

            // Compare the actual distance with the given distance in the distances array.
            if (actualDistance !== distances[charIndex]) {
                // If the distances do not match, return false as the condition hasn't been met.
                return false;
            }
        }

        // Update the last seen position of the current character.
        lastSeenPositions[charIndex] = index + 1; // Adding 1 as we're working with 1-based positions in this approach.
    }

    // If all conditions are satisfied for each character, return true.
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input string `s`. The function iterates over each character of the string exactly once.

Space Complexity

The space complexity of the function is `O(1)`. This is because the additional data structure used, `d`, is a dictionary that holds each unique character in the string and its latest index. Since there can be at most 26 unique characters (assuming `s` consists of lowercase English letters), the dictionary size is bounded by a constant, which does not scale with the input size.