2414. Length of the Longest Alphabetical Continuous Substring

Medium String

Problem Description

The given problem involves finding the length of the longest continuous alphabetical substring within a given string `s`. A continuous alphabetical string is defined as one where each character is the immediate successor of the previous one in the English alphabet. For instance, 'abc' progresses directly from 'a' to 'b' to 'c', so it counts as a continuous alphabetical string. Conversely, 'acb' doesn't meet this criterion because 'c' doesn't directly follow 'a'. The string 's' consists only of lowercase letters. Our task is to compute the maximum length of such a substring found in `s`.

To sum it up, we ought to traverse the string `s`, find every alphabetical continuous substring and keep track of the longest one we encounter along this traversal.

Intuition

The crux of the solution lies in sequential character comparison within the string. We utilize two pointers, `i` and `j`, where `i` denotes the starting index of a continuous alphabetical substring, and `j` acts as the explorer or the runner that moves ahead to find the end of this substring. The idea is to iterate through `s` using `j`. As the iteration occurs, we compare adjacent characters.

The insight is to notice that a continuous alphabetical substring will have characters whose ASCII values are consecutive. This implies that the difference between the ASCII values of such characters is exactly 1. Thus, as long as the difference `ord(s[j]) - ord(s[j - 1])` is 1, `j` can keep moving, indicating the substring starting at `i` and ending before `j` is still continuous and alphabetical.

Upon finding a character that doesn't follow this rule, we calculate the length of the continuous substring by computing `j - i`, which is then compared with the current answer. Subsequently, `i` is updated to the current location of `j`, as this marks the beginning of a potential new continuous alphabetical substring.

After the loop, there is a final comparison to ensure we account for the situation where the longest continuous alphabetical substring is at the tail of `s`.

Solution Approach

The approach uses a simple linear scan algorithm to walk through the string `s`. The main data structures used here are two pointers, named `i` and `j`. There is no complex pattern or algorithm beyond this two-pointer approach, and no additional space is required, making it an `O(1)` space complexity solution.

We start by initializing `ans` to `0`, which will hold the maximum length found, and two pointers `i` and `j` to `0` and `1` respectively. Pointer `i` represents the start of the current continuous alphabetical substring, and `j` is the runner that iterates through the string to determine the end of this substring.

Below is a step-by-step walkthrough of the implementation:

1. Begin a `while` loop that will run as long as `j` is less than the length of `s`.
2. On each iteration, first check if the current answer `ans` needs to be updated by comparing it with the length of the current continuous alphabetical substring `j - i`. The `max` function is used for this purpose.
3. Then, check if the current character at index `j` and the one preceding it (`j - 1`) form a part of a continuous alphabetical substring. This is done by comparing their ASCII values and verifying if `ord(s[j]) - ord(s[j - 1])` equals `1`.
4. If the condition is not met, this means the current character doesn't follow the previous character alphabetically, and thus, `i` is updated to `j`. This effectively starts a new continuous substring from position `j`.
5. After the condition check, increment `j` with `j += 1` to continue scanning the string.
6. After the loop ends, there may be a continuous substring still under consideration, which reaches the end of the string `s`. Hence, a final update to `ans` is required to include this last substring's length by again using `max(ans, j - i)`.
7. Finally, return the answer `ans`, which holds the length of the longest continuous alphabetical substring found in `s`.

This method performs just one scan through the string, which makes its time complexity `O(n)`, where `n` is the length of the string `s`, satisfying the requirement for an efficient solution.

Example Walkthrough

Let's go through a sample string `s = "abcdfghij"` to illustrate the solution approach outlined above.

With the given string `s = "abcdfghij"`, we aim to find the length of the longest substring where each character is followed by its immediate successor in the alphabet.

1. We initialize `ans = 0`, `i = 0`, and `j = 1` as our starting points.
2. Begin the `while` loop since `j < len(s)` (1 < 9).
3. Since `ord('b') - ord('a')` equals `1`, we continue without updating `i`. `ans` remains `0` for now because `j - i` (which equals `1`) is not greater than `ans`.
4. Increment `j` to 2. The characters at `s[1]` and `s[2]` ('b' and 'c') also satisfy the continuous condition, so we proceed without updating `i`.
5. Continuing the process, `j` increments to 3 ('d'), 4 ('f'), and now `ord('f') - ord('d')` does not equal `1`. This means we have a break in our continuous alphabetical sequence.
6. At this point, we update `ans` to `max(ans, j - i)` which is `max(0, 4 - 0) = 4`. We establish a new potential substring, and thus `i = j`, setting `i` to 4.
7. `j` becomes 5 and since `ord('g') - ord('f')` equals `1`, we continue scanning. This goes on with `j` taking the values 6 ('h'), 7 ('i'), and 8 ('j') as each of these characters continues the alphabet sequence.
8. We reach the end of the while loop when `j` is 9, and `ans` is updated for a final time: `ans = max(ans, j - i) = max(4, 9 - 4) = 5`.
9. With no more characters left to inspect, we exit the loop and return `ans`, which now holds the value `5`, the length of the longest continuous alphabetical substring in `s` ('fghij').

This walkthrough demonstrates the implementation of a two-pointer approach for finding the longest continuous alphabetical substring in a given string with an example.

Solution Implementation

Python

```python
class Solution:
    def longestContinuousSubstring(self, s: str) -> int:
        # Initialize the maximum length of the continuous substring
        max_length = 0

        # Initialize pointers i and j for start and end of the current substring
        start_index, end_index = 0, 1

        # Iterate through the string while end_index is less than the length of the string
        while end_index < len(s):
            # Update max length with the length of the current continuous substring
            max_length = max(max_length, end_index - start_index)

            # Check if the current character and the previous character are not consecutive
            if ord(s[end_index]) - ord(s[end_index - 1]) != 1:
                # If they are not consecutive, reset the start_index to the current position
                start_index = end_index

            # Move to the next character
            end_index += 1

        # Outside the loop, update the max length one last time for the ending substring
        max_length = max(max_length, end_index - start_index)

        # Return the maximum length of the continuous substring found
        return max_length
```

Java

```java
class Solution {

    public int longestContinuousSubstring(String s) {
        // Initialize the maximum substring length.
        int maxLen = 0;

        // 'start' is the starting index of the current continuous substring.
        // 'end' will be used to explore ahead in the string.
        int start = 0, end = 1;

        // Iterate through the characters of the string, starting from the second character.
        for (; end < s.length(); ++end) {
            // Update the maximum substring length found so far.
            maxLen = Math.max(maxLen, end - start);

            // Check whether the current character is not consecutive to the previous one.
            if (s.charAt(end) - s.charAt(end - 1) != 1) {
                // If not consecutive, move 'start' to the current character's index.
                start = end;
            }
        }

        // Update the maximum substring length for the last continuous substring.
        // This covers the case where the longest substring ends at the last character.
        maxLen = Math.max(maxLen, end - start);

        // Return the maximum length of continuous substring found.
        return maxLen;
    }
}
```

C++

```cpp
class Solution {
public:
    int longestContinuousSubstring(string s) {
        // Initialize the answer to zero length
        int maxLength = 0;

        // Pointers to keep track of the start and end of current continuous substring
        // Set 'start' to 0 and 'end' to 1 since we'll compare elements in pairs
        int start = 0, end = 1;

        // Loop through the string starting from the second character
        for (; end < s.size(); ++end) {
            // Update the maximum length found so far
            maxLength = max(maxLength, end - start);

            // If the current and previous characters are not consecutive
            if (s[end] - s[end - 1] != 1) {
                // Move the 'start' to the current character's index as a new substring begins
                start = end;
            }
        }

        // Update the maxLength for the last continuous substring which is terminated at the string's end
        maxLength = max(maxLength, end - start);

        // Return the length of the longest continuous substring
        return maxLength;
    }
};
```

TypeScript

```typescript
/**
 * Finds the length of the longest continuous substring where each
 * character appears to be in consecutive alphabetical order.
 * @param {string} s The input string.
 * @return {number} The length of the longest continuous substring.
 */
function longestContinuousSubstring(s: string): number {
    // n holds the length of the input string
    const n: number = s.length;
    // res (result) will store the length of the longest substring found
    let res: number = 1;
    // i marks the beginning of the current substring being examined
    let i: number = 0;

    // Loop through the string starting from the second character
    for (let j = 1; j < n; j++) {
        // If the current character is not the consecutive character
        // following the previous one in ASCII value
        if (s[j].charCodeAt(0) - s[j - 1].charCodeAt(0) !== 1) {
            // Update the result if a longer substring has been found
            res = Math.max(res, j - i);
            // Reset i to the current position for the next substring
            i = j;
        }
    }

    // Return the maximum of the result or the length of the substring
    // from the last reset point to the end of the string
    return Math.max(res, n - i);
}
```

```python
class Solution:
    def longestContinuousSubstring(self, s: str) -> int:
        # Initialize the maximum length of the continuous substring
        max_length = 0

        # Initialize pointers i and j for start and end of the current substring
        start_index, end_index = 0, 1

        # Iterate through the string while end_index is less than the length of the string
        while end_index < len(s):
            # Update max length with the length of the current continuous substring
            max_length = max(max_length, end_index - start_index)

            # Check if the current character and the previous character are not consecutive
            if ord(s[end_index]) - ord(s[end_index - 1]) != 1:
                # If they are not consecutive, reset the start_index to the current position
                start_index = end_index

            # Move to the next character
            end_index += 1

        # Outside the loop, update the max length one last time for the ending substring
        max_length = max(max_length, end_index - start_index)

        # Return the maximum length of the continuous substring found
        return max_length
```

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`. This is because the code uses a single while loop that iterates through each character of the string exactly once. The `while` loop starts with `j` at 1 and increments it until it reaches the end of the string. The evaluation of whether `ord(s[j]) - ord(s[j - 1]) == 1` is a constant-time operation, as is the `max` function that is called with constant arguments. Since there are no nested loops or recursive calls that depend on the size of the input, the time complexity remains linear in terms of the length of the string.

Space Complexity

The space complexity of the code is `O(1)`. Only a fixed number of integer variables (`ans`, `i`, `j`) are used, and their amount of space does not change with the size of the input. No additional data structures or dynamic memory allocation are used that would scale with the input size, so the space used by the algorithm is constant.