1624. Largest Substring Between Two Equal Characters

Easy Hash Table String

Leetcode Link

Problem Description

The problem presents a string `s` and requires finding the length of the longest substring that is present between two identical characters in the string, *excluding* the characters themselves. If no such substring exists (i.e., there aren't two identical characters in the string with anything between them), the function should return `-1`. The term "substring" indicates a consecutive sequence of characters within the string `s`.

For example, in the string "abca", the longest substring between two equal characters 'a' is "bc", which has the length of 2. In a case where the string is "abc", there are no two equal characters with something between them, so the result would be `-1`.

Intuition

To solve this problem, we must find characters in the string that appear more than once and calculate the distance between their first and last occurrences, since that will determine the length of the substring between them. To efficiently do this, we should remember the index of the first occurrence of each character when we see it for the first time.

With these considerations in mind, we use a dictionary `d` to store characters as keys and their first encountered index positions as values. We iterate over the string `s` and for each character `c`:

- If `c` is already in the dictionary `d`, this means we have previously seen it, and hence we calculate the distance between this occurrence and its first occurrence, which is `i - d[c] - 1`, and update the answer `ans` if this distance is larger.
- If `c` is not in the dictionary, we record its index `i` in the dictionary `d`, marking its first occurrence.

The variable `ans` is used to keep track of the maximum length found so far. If no such length is found (meaning `ans` never gets updated), it remains `-1`, which is also the default value we return when there are no two identical characters with substrings between them.

Thus, by using a single pass over the string and a dictionary to track the indices of characters, we arrive at an efficient solution to the problem.

Solution Approach

The implementation uses a dictionary to store the first occurrence of each character in the given string. A dictionary is an excellent choice for this problem due to its fast lookups and insertions, which operate on average in O(1) time complexity.

Let's walk through the implementation step by step:

- A dictionary `d` is initialized to keep a record of the first index where each character appears.
- A variable `ans` is initiated with a value of `-1`. This will eventually hold the maximum length of any substring found between two equal characters, excluding the characters themselves.
- We iterate over the characters of the string `s` using a `for` loop, where `i` is the index and `c` is the character at that index.
- Inside the loop, we check if the character `c` is already in the dictionary `d`:
  - If `c` is in `d`, it means we have encountered `c` before. The substring length between the two `c` characters is `i - d[c] - 1`.
    - We compare this length with the current maximum stored in `ans`.
    - If it's larger, we update `ans` with this new value.
  - If `c` is not in `d`, we add `c` to the dictionary with the current index `i` as its value. This marks the location of the first occurrence of `c`.
- Once the loop is complete, `ans` will have the maximum length found or remain `-1` if no such substring exists.

This approach is efficient because we maintain a sliding window between two occurrences of the same character by remembering only the first occurrence and using the current index to calculate the length of the in-between substring. As a result, the time complexity is O(n), where n is the length of the string, because we are making a single pass through the string.

With this understanding, the provided Python code realizes the approach effectively and delivers the correct result for the problem at hand.

Example Walkthrough

Let's consider a string `s` with the value `"characher"`. We are trying to find the length of the longest substring present between two identical characters (excluding those characters).

Following the solution approach, let's break it down step by step:

- Initialize a dictionary `d` for keeping the first index occurrence of each character.
- Initialize `ans` as `-1`.
- Start iterating through the string:

  1. For index `0`, character `c`: it's not in `d`, so we add `c: 0` to the dictionary.
  2. For index `1`, character `h`: it's also not in `d`, add `h: 1`.
  3. For index `2`, character `a`: it's not in `d`, add `a: 2`.
  4. For index `3`, character `r`: not in `d`, add `r: 3`.
  5. For index `4`, character `a`: a is already in `d` at index `2`. Calculate the distance: `4 - 2 - 1 = 1`. Update `ans` to `1`.
  6. For index `5`, character `c`: c is in `d` with index `0`. The distance is `5 - 0 - 1 = 4`. Since `4` is greater than the current `ans` value of `1`, update `ans` to `4`.
  7. For index `6`, character `t`: not in `d`, add `t: 6`.
  8. For index `7`, character `e`: not in `d`, add `e: 7`.
  9. For index `8`, character `r`: r is in `d` with index `3`. The distance is `8 - 3 - 1 = 4`. `ans` is already `4`, so no update is needed.

- End of iteration. The final `ans` is `4`, which is the length of the longest substring ("aract") between two identical characters without including them ("c...c" and "r...r").

The code returns `4` as the length of the longest substring between two repeating characters, excluding the characters themselves.

Python Solution

```python
class Solution:
    def maxLengthBetweenEqualCharacters(self, s: str) -> int:
        # Dictionary to store the first occurrence of each character
        first_occurrence = {}

        # Initialize the answer with -1 as per problem constraints
        max_length = -1

        # Iterate over the string to find the max length between equal characters
        for index, char in enumerate(s):
            # If character is already seen, calculate the length between the current and first occurrence
            if char in first_occurrence:
                length_between = index - first_occurrence[char] - 1
                max_length = max(max_length, length_between)
            else:
                # Store the first occurrence of the character
                first_occurrence[char] = index

        # Return the maximum length found
        return max_length

```

Java Solution

```java
class Solution {
    public int maxLengthBetweenEqualCharacters(String s) {
        // Array to store the first occurrence index of each character.
        // Initialized to -1 indicating that the character hasn't been seen yet.
        int[] firstOccurrence = new int[26];
        Arrays.fill(firstOccurrence, -1);

        // Variable to store the maximum length found.
        int maxLength = -1;

        // Loop through the string to find the maximum length.
        for (int i = 0; i < s.length(); ++i) {
            // Calculate the index for the character 'a' as 0, 'b' as 1, etc.
            int charIndex = s.charAt(i) - 'a';

            // If the character has not been seen before,
            // store its index in firstOccurrence.
            if (firstOccurrence[charIndex] == -1) {
                firstOccurrence[charIndex] = i;
            } else {
                // If we have seen the character before,
                // calculate the length between the current and first index.
                // Then, update the maxLength if the calculated length is larger.
                maxLength = Math.max(maxLength, i - firstOccurrence[charIndex] - 1);
            }
        }

        // Return the maximum length found.
        return maxLength;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxLengthBetweenEqualCharacters(string s) {
        // Create a vector 'firstIndex' to store the first occurrence index of each lowercase letter.
        // Initialized with -1, indicating we have not seen the character yet.
        vector<int> firstIndex(26, -1);

        // Variable to store the maximum length found between two equal characters.
        int maxLen = -1;

        // Iterate through the string to check each character.
        for (int i = 0; i < s.size(); ++i) {
            // Converting the current character to its corresponding index (0-25 for 'a'-'z').
            int index = s[i] - 'a';

            // If we have not seen this character before, store its index.
            if (firstIndex[index] == -1) {
                firstIndex[index] = i;
            } else {
                // If we've seen this character before, calculate the length between the two equal characters.
                // Update 'maxLen' if we find a longer length.
                maxLen = max(maxLen, i - firstIndex[index] - 1);
            }
        }

        // Return the maximum length found.
        return maxLen;
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the maximum length of a substring between two identical characters
 * in the input string `s`.
 *
 * @param s The string to be evaluated.
 * @return The maximum length of a substring.
 */
function maxLengthBetweenEqualCharacters(s: string): number {
    // 'n' holds the length of the string 's'.
    const n: number = s.length;
    // 'positionIndex' is an array to track the first occurrence position of each alphabet character.
    const positionIndex: number[] = new Array(26).fill(-1);
    // 'maxLength' will keep track of the maximum length found.
    let maxLength: number = -1;

    // Iterate over each character in the string.
    for (let i = 0; i < n; i++) {
        // Compute the zero-based alphabet index of the current character.
        const charIndex: number = s[i].charCodeAt(0) - 'a'.charCodeAt(0);
        // If it is the first occurrence of the character, store its position.
        if (positionIndex[charIndex] === -1) {
            positionIndex[charIndex] = i;
        } else {
            // If a pair is found, update the 'maxLength', if necessary.
            maxLength = Math.max(maxLength, i - positionIndex[charIndex] - 1);
        }
    }

    // Return the maximum length found, or -1 if no such length exists.
    return maxLength;
}

```

Time and Space Complexity

Time Complexity

The given Python code iterates over the string `s` only once with a for loop. For each character in the string, it performs a constant-time operation to check if the character is in the dictionary `d`, update the maximum length using `max()`, and set the value in the dictionary. None of these operations depend on the size of the input string inside the loop, so each iteration of the loop runs in constant time, `O(1)`.

The time complexity of the entire function is thus proportional to the number of characters `n` in the string `s`, as each character is processed just once. This results in an overall time complexity of:

```
O(n)
```

where `n` is the length of the string `s`.

Space Complexity

The space complexity of the code depends on the number of unique characters in the string `s`, as a dictionary `d` stores the first index at which each character appears. In the worst case, all characters of the string `s` are unique, which would require storing each character in the dictionary `d`.

Assuming the input string `s` contains `n` characters and taking into account the possibility of `n` unique characters, the space complexity of maintaining the dictionary is:

```
O(n)
```

where `n` is the length of the string `s`.

However, if we consider the constraint of the problem that the input string `s` consists of only lowercase English letters, then there is a constant maximum of 26 unique characters that can be stored in the dictionary `d`. This would imply a constant space complexity:

```
O(1)
```

Depending on the input constraints specified in the problem statement, you should use the appropriate analysis for the space complexity.