2716. Minimize String Length

Easy Hash Table String

Leetcode Link

Problem Description

In this problem, we are given a 0-indexed string `s`. The operation we can perform on this string is to delete the closest occurrences of a character `c` to the left and right of index `i`, where `c` is the character at index `i`. We can perform this operation any number of times with the goal to minimize the length of string `s` as much as possible. The task is to return the length of the string after performing the operation optimally to minimize the string length.

Intuition

To solve this problem, we need to find a way to reduce the string length as much as possible by deleting characters. One might initially think that keeping track of the occurrences and their positions is required. However, on careful observation, we can note that in the best case scenario, each unique character can completely remove its closest occurrence to the left and right for every other duplicate character.

If we consider an example: Given a string `s = "abcddcba"`, we can see this:

- For 'a', the closest 'a' to the left of the second 'a' is the first 'a', and similarly for the closest to the right.
- The same argument applies to 'b', 'c', and 'd'.

The optimal removal would remove all characters except for one instance of each unique character. Thus, our problem simplifies to counting the number of unique characters in the string since in the minimized string, every character that has duplicates would have only one instance left.

Therefore, the solution approach to finding the length of the minimized string is to simply return the count of unique characters in the string, which can be found using the `set()` function in Python that returns all unique elements, and then using the `len()` function to count these unique elements.

Solution Approach

The implementation of the solution is straightforward because of the simplicity of the problem once we understand the pattern. We utilize the fact that for any character that has duplicates in the string, all its occurrences except one will eventually be removed. By following this logic, we can infer that the solution does not require complex algorithms or data structures.

The approach consists of two steps:

1. Convert the string into a set of characters, which eliminates any duplicate occurrences of characters. We use the `set()` constructor in Python, which takes an iterable, such as a string, and returns a new set object with distinct elements.
2. Calculate the length of this set, which gives us the number of unique characters in the original string. To get the count, we use the `len()` function which returns the number of items in a container.

By employing these built-in functions, `set` and `len`, we efficiently solve the problem without the need for any additional data structure or algorithm. The pattern recognized here is essentially a unique element count problem which often leads to the use of sets for their property of containing non-duplicate elements.

The Python code is as follows:

```python
class Solution:
    def minimizedStringLength(self, s: str) -> int:
        return len(set(s))
```

In the code, `s` is the input string. We pass `s` to the `set()` function which removes all duplicate characters. Then, we pass the resulting set to the `len()` function to get the total count of unique characters. This count represents the length of the minimized string, according to the problem statement.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the string `s = "abacabad"`. According to the problem, we want to delete occurrences of characters that are closest to the left and right of any instance of that character. Following the solution approach, here's a step-by-step walkthrough:

1. Convert the string `s` into a set of characters to get the unique characters. For our example, converting `s` to a set would look like this:

```python
unique_chars = set("abacabad")
print(unique_chars)  # Output will be {'a', 'b', 'c', 'd'}
```

After converting to a set, we have `{'a', 'b', 'c', 'd'}`, which represents the unique characters in the string.

2. Calculate the length of this set to find out how many unique characters we have:

```python
length = len(unique_chars)
print(length)  # Output will be 4
```

The length of the unique character set is `4`, which means there are four unique characters in the string `s`.

Following the intuition of the problem, these four unique characters would be the characters that remain after optimally performing the given operation (i.e., deleting the closest occurrences of a character to the left and right of index `i`). Therefore, the length of the string after performing the operations is the same as the number of unique characters in the original string, which is `4` in this case.

Thus, using the `set()` function in Python, we eliminated all the duplicates, and by using the `len()` function, we efficiently found the minimized string length. This example perfectly illustrates how the solution approach leads to an optimal and simplistic answer.

Python Solution

```python
class Solution:
    def minimizedStringLength(self, s: str) -> int:
        # Create a set from the string 's' to eliminate any duplicate characters.
        unique_characters = set(s)

        # The minimized string length is equal to the number of unique characters.
        minimized_length = len(unique_characters)

        # Return the length of the minimized string.
        return minimized_length
```

Java Solution

```java
class Solution {
    // Function to compute the minimized string length based on unique characters
    public int minimizedStringLength(String inputString) {
        // Create a HashSet to store unique characters
        Set<Character> uniqueCharacters = new HashSet<>();

        // Iterate through all characters in the input string
        for (int i = 0; i < inputString.length(); ++i) {
            // Add each character to the Set, duplicates are automatically ignored
            uniqueCharacters.add(inputString.charAt(i));
        }

        // The size of the Set represents the number of unique characters
        // Return this size as the minimized string length
        return uniqueCharacters.size();
    }
}
```

C++ Solution

```cpp
#include <unordered_set> // Include necessary header for unordered_set
#include <string> // Include necessary header for string

class Solution {
public:
    // Function to calculate the minimized string length
    int minimizedStringLength(std::string s) {
        // Create an unordered_set to store unique characters
        std::unordered_set<char> unique_characters(s.begin(), s.end());

        // The size of the set gives us the count of unique characters
        return unique_characters.size();
    }
};
```

Typescript Solution

```typescript
// This function calculates the minimized length of a string
// by removing all duplicate characters.
// @param {string} inputString - The string to be processed.
// @returns {number} The count of unique characters in the input string.

function minimizedStringLength(inputString: string): number {
    // Split the input string into its individual characters,
    // convert it into a Set to remove duplicates,
    // and then return the size of the Set.
    return new Set(inputString.split('')).size;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the `minimizedStringLength` function is `O(n)`, where `n` is the length of the string `s`. This is because the function iterates through each character in the string exactly once to create a set of unique characters.

Space Complexity

The space complexity of the function is `O(k)`, where `k` is the number of unique characters in the string `s`. In the worst case, if all characters are unique, `k` would be equal to `n`, resulting in a space complexity of `O(n)`. However, considering the input is a string with a fixed character set (e.g., ASCII characters), `k` can also be considered a constant, and hence the space complexity can be `O(1)` in the context of a fixed character set.