1930. Unique Length-3 Palindromic Subsequences

Medium | Hash Table | String | Prefix Sum

Problem Description

The problem requires us to determine the number of unique palindromic subsequences of length three that can be extracted from a given string `s`. Recall the definitions for a palindrome and a subsequence:

- A **palindrome** is a sequence of characters that reads the same forward and backward, like "abcba" or "mom".
- A **subsequence** is derived from the original string by deleting some characters without changing the order of the remaining characters. For instance, "ace" is a subsequence of "abcde".

With this in mind, we are to focus on palindromes that have a length of exactly three characters. The key term here is *unique*—if a palindromic subsequence can be formed in multiple ways, it still counts as one unique instance.

Intuition

To devise a solution, let's consider the characteristics of a three-character palindrome: it begins and ends with the same character, with a different third character sandwiched in between. A three-character palindrome will always have the form "XYX". To find such palindromes, we need to:

1. Find the first and last occurrence of a character "X" in the string.
2. Look between those occurrences for any characters "Y" that can form a palindrome "XYX".

Our algorithm iterates through each letter of the alphabet. For each letter, it finds the leftmost (`l`) and the rightmost (`r`) occurrence in the string `s`. If the indices `l` and `r` are the same, it means there is only one instance of that letter, and thus a three-character palindrome cannot be formed with that letter as the start and end. If the indices are different, and particularly if `r - l > 1`, it ensures that there is at least one character between them to potentially create a palindrome.

Inside the slice `s[l + 1 : r]` (which excludes the outer characters), we convert the substring into a set to get unique characters. The size of this set gives the number of unique characters that can form a palindrome with the current character as the first and last letter. We sum up these counts for all characters of the alphabet to get our answer.

The solution optimizes searching using the `find()` and `rfind()` functions. `find()` returns the lowest index of the substring if found, else returns -1. `rfind()` functions similarly but returns the highest index.

Solution Approach

The implementation of the solution provided above leverages a couple of Python's string methods and a simple loop to find a solution efficiently. Here's the breakdown of the approach:

1. **Initialization**: We start by setting up a counter `ans`. This variable will hold the cumulative count of unique palindromes.
2. **Iterate over the alphabet**: Using a `for` loop, we iterate through each character `c` in the `ascii_lowercase`. The `ascii_lowercase` from the `string` module contains all lowercase letters of the alphabet.
3. **Finding character positions**: For each character `c`, we use `s.find(c)` to get the index `l` of the first occurrence and `s.rfind(c)` to get the index `r` of the last occurrence of that character in the string `s`.
4. **Check for valid palindrome positions**: We then check if `r - l > 1`. This condition is crucial, as it ensures there are at least two characters between the first and last occurrence to potentially form a 3-character palindrome.
   - If `l` and `r` are the same, it would mean that only one instance of the character `c` exists in `s` and thus a palindrome cannot be formed.
   - If `l` and `r` are different but `r - l <= 1`, it would mean the character `c` occurs in consecutive positions, leaving no room for a middle character to form a palindrome.
5. **Counting unique middle characters**: If the condition `r - l > 1` is satisfied, we take a slice of the string from `l + 1` to `r` to isolate the characters between the first and last occurrence of `c`. We then convert this substring into a set to remove duplicate characters.
6. **Update the count**: The length of the set gives the number of unique characters that can form a palindrome with the character `c` sandwiched between them. We add this number to our counter `ans`.
7. **Return the result**: Once the loop has gone through each letter in the alphabet, `ans` will contain the total count of unique palindromic subsequences of length three in our input string `s`. This value is then returned.

Here is the critical algorithm part in the code enclosed in backticks for markdown display:

```
for c in ascii_lowercase:
    l, r = s.find(c), s.rfind(c)
    if r - l > 1:
        ans += len(set(s[l + 1 : r]))
return ans
```

This code snippet clearly highlights the search for the leftmost and rightmost occurrences of each character and the calculation of the number of distinct middle characters that can form a palindrome with the current character. It does so effectively by utilizing the built-in functions provided by Python for string manipulation.

Example Walkthrough

Let's take the string `s = "abcbabb"` as an example to illustrate the solution approach. The strategy is to identify unique palindromic subsequences of the format "XYX" within `s`.

1. **Initialization**: We initiate `ans = 0` to accumulate the count of unique palindromes.
2. **Iterate over the alphabet**: We start checking for each character in `ascii_lowercase`. We'll illustrate this with a few selected characters from `s`: 'a', 'b', and 'c'.
3. **Finding character positions**:
   - For character 'a':
     - `l = s.find('a')` results in `0`. (first occurrence of 'a')
     - `r = s.rfind('a')` results in `0`. (last occurrence of 'a')
     - Since `r - l` is not greater than 1, we do not have enough space to form a 3-character palindrome with 'a', so we move on.
   - For character 'b':
     - `l = s.find('b')` results in `1`. (first occurrence of 'b')
     - `r = s.rfind('b')` results in `6`. (last occurrence of 'b')
     - Since `r - l` is greater than 1, we have a valid situation.
4. **Check for valid palindrome positions**: For character 'b', the condition `r - l > 1` is true (6 - 1 > 1), indicating potential for 3-character palindromes.
5. **Counting unique middle characters**: We extract the substring `s[l + 1 : r]` which is `"cbab"`. Converting this to a set gives `{'c', 'b', 'a'}`.
6. **Update the count**: The set length for character 'b' as the outer character is `3`, meaning we have 'bcb', 'bab', and 'bab' as unique palindromic subsequences. Since 'bab' can be formed by different indices, it still counts as one. We add `3` to `ans`, making `ans = 3`.
   - For character 'c':
     - Similarly, we would find `l = 2`, `r = 2`, so no palindrome can be formed, and we move on.
7. **Return the result**: After iterating through all alphabet characters, assume we found no additional characters that can form a unique palindromic subsequence. Therefore, our final answer for string `s` is `ans = 3`.

This walkthrough provides a clear example of the steps outlined in the solution approach, demonstrating the counting of unique palindromic subsequences within the given string `s`.

Solution Implementation

Python

```python
from string import ascii_lowercase

class Solution:
    def countPalindromicSubsequence(self, s: str) -> int:
        # Initialize the count of unique palindromic subsequences
        count = 0

        # Iterate through each character in the lowercase English alphabet
        for char in ascii_lowercase:
            # Find the first (leftmost) and last (rightmost) indices of the character in the string
            left_index = s.find(char)
            right_index = s.rfind(char)

            # Check if there is more than one character between the leftmost and rightmost occurrence
            if right_index - left_index > 1:
                # If so, add the number of unique characters between them to the count
                # This creates a palindromic subsequence of the form "cXc"
                # where 'c' is the current character and 'X' represents any unique set of characters
                count += len(set(s[left_index + 1 : right_index]))

        # Return the final count of unique palindromic subsequences
        return count
```

Java

```java
class Solution {
    // Method to count the unique palindromic subsequences in the given string
    public int countPalindromicSubsequence(String s) {
        // Initialize the count of unique palindromic subsequences to 0
        int count = 0;

        // Iterate through all lowercase alphabets
        for (char currentChar = 'a'; currentChar <= 'z'; ++currentChar) {
            // Find the first and last occurrence of 'currentChar' in the string
            int leftIndex = s.indexOf(currentChar);
            int rightIndex = s.lastIndexOf(currentChar);

            // Create a HashSet to store unique characters between the first and
            // last occurrence of 'currentChar'
            Set<Character> uniqueChars = new HashSet<>();

            // Iterate over the substring that lies between the first and
            // last occurrence of 'currentChar'
            for (int i = leftIndex + 1; i < rightIndex; ++i) {
                // Add each character in the substring to the HashSet
                uniqueChars.add(s.charAt(i));
            }

            // The number of unique characters added to the HashSet is the number
            // of palindromic subsequences starting and ending with 'currentChar'
            count += uniqueChars.size();
        }

        // Return the total count of unique palindromic subsequences
        return count;
    }
}
```

C++

```cpp
#include <string>
#include <unordered_set>
using namespace std;

class Solution {
public:
    int countPalindromicSubsequence(string s) {
        // Initialize the count of palindromic subsequences to 0
        int countPaliSubseq = 0;

        // Iterate over all lowercase alphabets
        for (char c = 'a'; c <= 'z'; ++c) {
            // Find the first and last occurrence of the current character
            int firstIndex = s.find_first_of(c);
            int lastIndex = s.find_last_of(c);

            // Use an unordered set to store unique characters between the first and last occurrence
            unordered_set<char> uniqueChars;

            // Iterate over the characters between the first and last occurrence
            for (int i = firstIndex + 1; i < lastIndex; ++i) {
                // Insert unique characters into the set
                uniqueChars.insert(s[i]);
            }

            // Increment the count by the number of unique characters found
            countPaliSubseq += uniqueChars.size();
        }

        // Return the total count of palindromic subsequences
        return countPaliSubseq;
    }
};
```

TypeScript

```typescript
// Import necessary features from the standard utility library
import { Set } from "typescript-collections";

// Function that counts the number of unique palindromic subsequences in a string
function countPalindromicSubsequence(s: string): number {
    // Initialize the count of palindromic subsequences to 0
    let countPaliSubseq = 0;

    // Iterate over all lowercase alphabets
    for (let c = 'a'.charCodeAt(0); c <= 'z'.charCodeAt(0); c++) {
        let currentChar = String.fromCharCode(c);

        // Find the first and last occurrence of the current character
        let firstIndex = s.indexOf(currentChar);
        let lastIndex = s.lastIndexOf(currentChar);

        // Use a Set to store unique characters between the first and last occurrence
        let uniqueChars = new Set<string>();

        // Iterate over the characters between the first and last occurrence
        for (let i = firstIndex + 1; i < lastIndex; i++) {
            // Insert unique characters into the Set
            uniqueChars.add(s[i]);
        }

        // Increment the count by the number of unique characters found
        countPaliSubseq += uniqueChars.size();
    }

    // Return the total count of palindromic subsequences
    return countPaliSubseq;
}
```


```python
from string import ascii_lowercase

class Solution:
    def countPalindromicSubsequence(self, s: str) -> int:
        # Initialize the count of unique palindromic subsequences
        count = 0

        # Iterate through each character in the lowercase English alphabet
        for char in ascii_lowercase:
            # Find the first (leftmost) and last (rightmost) indices of the character in the string
            left_index = s.find(char)
            right_index = s.rfind(char)

            # Check if there is more than one character between the leftmost and rightmost occurrence
            if right_index - left_index > 1:
                # If so, add the number of unique characters between them to the count
                # This creates a palindromic subsequence of the form "cXc"
                # where 'c' is the current character and 'X' represents any unique set of characters
                count += len(set(s[left_index + 1 : right_index]))

        # Return the final count of unique palindromic subsequences
        return count
```


Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed as follows:

- The code iterates over all lowercase ASCII characters, which are constant in number (26 characters). This loop runs in O(1) with respect to the input size.
- Inside the loop for each character `c`, the code performs `s.find(c)` and `s.rfind(c)`, each of which operates in O(n), where n is the length of the string `s`.
- If a character `c` is found in the string, the code computes the set of unique characters in the substring `s[l + 1 : r]`, where `l` is the index of the first occurrence of `c`, and `r` is the index of the last occurrence of `c`. Since the substring extraction `s[l + 1 : r]` is O(n) and computing the set of unique characters could also be O(n) (since in the worst case, the substring could have all unique characters), this operation is O(n).
- Therefore, for each character iteration, the total time complexity is O(n) + O(n) + O(n), which simplifies to O(n).
- Given that the outer loop runs 26 times, the time complexity in total will be O(26*n), which simplifies to O(n).

Space Complexity

The space complexity of the algorithm can be analyzed as follows:

- A set is created for each character in the loop to hold the unique characters in the substring. The largest this set can be is the total alphabet set size, so at most 26 characters.
- However, because the set is re-used for each character (i.e., it doesn't grow for each character found in `s`) and does not depend on the size of the input `s`, the space complexity is O(1).

In conclusion, the time complexity is `O(n)` and the space complexity is `O(1)`.