467. Unique Substrings in Wraparound String

Medium String Dynamic Programming Leetcode Link

Problem Description

The given problem is to count the number of unique non-empty substrings of a string `s` that can also be found in the infinite `base` string, which is essentially the English lowercase alphabet letters cycled indefinitely. The string `s` is a finite string, possibly containing repeated characters.

The task is to figure out how many unique substrings from `s` fit consecutively into `base`. The challenge here is figuring out an efficient way to calculate this without the need to generate all possible substrings of `s` and check them against `base`, which would be computationally expensive.

Intuition

The intuition behind the solution leverages the properties of the `base` string. Since `base` is comprised of the English alphabet in order and repeated indefinitely, any substring following the pattern of contiguous letters ('abc', 'cde', etc.) will be found in `base`. Furthermore, if we have found a substring that fits into `base`, any shorter substrings starting from the same character will also fit.

The solution uses dynamic programming to keep track of the maximum length for substrings starting with each letter of the alphabet found in `s`. The intuition is that if you can form a substring up to a certain length starting with a particular character (like 'a'), then you can also form all smaller substrings that start with that character.

Here's our approach in steps:

1. Create a list `dp` of 26 elements representing each letter of the alphabet to store the maximum substring length starting with that letter found in `s`.
2. Go through each character of string `s` and determine if it is part of a substring that follows the contiguous pattern. We do this by checking if the current and previous characters are adjacent in `base`.
3. If adjacent, increment our running length `k`. If not, reset `k` to one, because the current character does not continue from the previous character.
4. Determine the list index corresponding to the current character and update `dp[idx]` with the maximum of its current value or `k`.
5. After processing the entire string, the sum of the values in `dp` is the total number of unique substrings found in `base`.

This approach prevents checking each possible substring and efficiently aggregates the counts by keeping track of the longest contiguous substring ending at each character.

Solution Approach

The solution uses dynamic programming, which is a method for solving complex problems by breaking them down into simpler subproblems. It utilizes additional storage to save the result of subproblems to avoid redundant calculations. Here's how the solution approach is implemented:

1. **Initial DP Array**: We create a DP (dynamic programming) array `dp` with 26 elements corresponding to the letters of the alphabet set to 0. This array will store the maximum length of the unique substrings ending with the respective alphabet character.
2. **Iterating over `p`**: We iterate over the string `p`, checking each character.
3. **Checking Continuity**: For each character `c` in `p`, we check if it forms a continuous substring with the previous character. This is done by checking the difference in ASCII values — specifically, whether `(ord(c) - ord(p[i - 1])) % 26 == 1`. This takes care of the wraparound from 'z' to 'a' by using the modulus operation.
4. **Updating Length of Substring `k`**: If the condition is true, it means the current character `c` continues the substring, and we increase our substring length counter `k`. If not, we reset `k` to 1 since the continuity is broken, and `c` itself is a valid substring of length 1.
5. **Updating DP Array**: We calculate the index of the current character in the alphabet `idx = ord(c) - ord('a')` and update the `dp` array at this index. We set `dp[idx]` to be the maximum of its current value or `k`. This step ensures we're keeping track of the longest unique substring ending with each letter without explicitly creating all substring combinations.
6. **Result by Summing DP values**: After completing the iteration over `p`, every index of the `dp` array contains the maximum length of substrings ending with the respective character that can be found in `base`. Summing up all these maximum lengths will give us the number of unique non-empty substrings of `s` in `base`.

This algorithm uses the DP array as a way to eliminate redundant checks and store only the necessary information. By keeping track of the lengths of contiguous substrings in this manner, we avoid having to store or iterate over each of the potentially very many substrings explicitly.

Example Walkthrough

Let's walk through an example to illustrate the solution approach.

Suppose our string `s` is `"abcdbef"`.

Following the steps of the solution approach:

1. **Initial DP Array**: We initiate a DP array `dp` with 26 elements set to 0. The array indices represent letters `'a'` to `'z'`.
2. **Iterating over `s`**: We begin iterating over string `s`.
3. **Checking Continuity**: As we check each character, we look for continuity from its predecessor. Since the first character 'a' has no predecessor, we skip to the second character 'b'. The ASCII difference from 'a' to 'b' is 1, thus we continue to the third character, 'c', and the same applies. However, when we get to the fourth character 'd', 'c' and 'd' are continuous, but for the fifth character 'b', 'd' and 'b' are not adjacent characters in the English alphabet.
4. **Updating Length of Substring `k`**: As we continue iterating:
   - 'a' starts a new substring with `k=1`.
   - 'b' is contiguous with 'a', so `k=2`.
   - 'c' is contiguous with 'b', so `k=3`.
   - 'd' is contiguous with 'c', so `k=4`.
   - 'b' breaks the pattern, resetting `k=1`.
   - 'e' is not contiguous with 'b', `k=1`.
   - 'f' is contiguous with 'e', so `k=2`.
5. **Updating DP Array**: Throughout the iteration, we update our `dp`. Here's how `dp` changes:
   - After 'a': `dp[0] = max(dp[0], 1) => 1`
   - After 'b': `dp[1] = max(dp[1], 2) => 2`
   - After 'c': `dp[2] = max(dp[2], 3) => 3`
   - After 'd': `dp[3] = max(dp[3], 4) => 4`
   - After reset at 'b': `dp[1] = max(dp[1], 1) => 2` (no change because 'b' already had 2 from 'ab')
   - After reset at 'e': `dp[4] = max(dp[4], 1) => 1`
   - After 'f': `dp[5] = max(dp[5], 2) => 2`
6. **Result by Summing DP Values**: Summing up the values in `dp`, we get the total count of unique non-empty substrings of `s` that can be found in `base`: `1 + 2 + 3 + 4 + 1 + 2 = 13`

So, our string `s` "abcdbef" has 13 unique non-empty substrings that fit consecutively into the infinite base string of the English alphabet cycled indefinitely.

Python Solution

```python
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int:
        # Initialize an array to keep track of the max length of substrings that end with each letter of the alphabet.
        max_length_end_with = [0] * 26

        # Initialize a variable to keep track of the current substring length.
        current_length = 0

        # Iterate through the string, character by character.
        for i in range(len(p)):
            # Check if the current character and the one before it are consecutive in the 'z-a' wraparound string.
            # The condition checks if they are in alphabetical order or if it's a 'z' followed by 'a'.
            if i > 0 and (ord(p[i]) - ord(p[i - 1])) % 26 == 1:
                # If consecutive, increment the current length.
                current_length += 1
            else:
                # Otherwise, reset the current length to 1.
                current_length = 1

            # Calculate the index in the alphabet for the current character.
            index = ord(p[i]) - ord('a')
            # Update the max length for this character if the current length is greater.
            max_length_end_with[index] = max(max_length_end_with[index], current_length)

        # Return the sum of max lengths, which gives the total number of distinct substrings.
        return sum(max_length_end_with)

```

Java Solution

```java
class Solution {
    public int findSubstringInWraproundString(String p) {
        // dp array to store the maximum length substring ending with each alphabet
        int[] maxSubstringLengths = new int[26];
        int currentLength = 0; // Length of current substring

        // Iterate through the string
        for (int i = 0; i < p.length(); ++i) {
            char currentChar = p.charAt(i);

            // If the current and previous characters are consecutive in the wraparound string
            if (i > 0 && (currentChar - p.charAt(i - 1) + 26) % 26 == 1) {
                // Increment length of current substring
                currentLength++;
            } else {
                // Restart length if not consecutive
                currentLength = 1;
            }

            // Find the index in the alphabet for the current character
            int index = currentChar - 'a';

            // Update the maximum length for the particular character if it's greater than the previous value
            maxSubstringLengths[index] = Math.max(maxSubstringLengths[index], currentLength);
        }

        int totalCount = 0; // Holds the total count of unique substrings

        // Sum up all the maximum lengths as each length contributes to the distinct substrings
        for (int maxLength : maxSubstringLengths) {
            totalCount += maxLength;
        }

        return totalCount; // Return total count of all unique substrings
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int findSubstringInWraproundString(string p) {
        // dp array to keep track of the max length of unique substrings ending with each letter.
        vector<int> maxLengthEndingWith(26, 0);

        // 'k' will be used to store the length of the current valid substring.
        int currentLength = 0;

        // Loop through all characters in string 'p'.
        for (int i = 0; i < p.size(); ++i) {
            char currentChar = p[i];

            // Check if the current character forms a consecutive sequence with the previous character.
            if (i > 0 && (currentChar - p[i - 1] + 26) % 26 == 1) {
                // If consecutive, increment the length of the current valid substring.
                ++currentLength;
            } else {
                // If not consecutive, start a new substring of length 1.
                currentLength = 1;
            }

            // Update the maximum length of substrings that end with the current character.
            int index = currentChar - 'a';
            maxLengthEndingWith[index] = max(maxLengthEndingWith[index], currentLength);
        }

        // Compute the result by summing the max lengths of unique substrings ending with each letter.
        int result = 0;
        for (int length : maxLengthEndingWith) result += length;

        // Return the total number of all unique substrings.
        return result;
    }
};

```

Typescript Solution

```typescript
function findSubstringInWraproundString(p: string): number {
    // Total length of the input string 'p'
    const length = p.length;
    // Initialize an array to store the maximum length of unique substrings that end with each alphabet letter
    const maxSubstringLengthByCh = new Array(26).fill(0);
    // Current substring length
    let currentLength = 1;

    // Set the maximum length for the first character
    maxSubstringLengthByCh[p.charCodeAt(0) - 'a'.charCodeAt(0)] = 1;

    // Iterate through the string starting from the second character
    for (let i = 1; i < length; i++) {
        // Check if the current and the previous characters are consecutive in the wraparound string
        if ((p.charCodeAt(i) - p.charCodeAt(i - 1) + 26) % 26 === 1) {
            // If they are consecutive, increment the length of the substring that includes the current character
            currentLength++;
        } else {
            // If not, reset the current substring length to 1
            currentLength = 1;
        }
        // Determine the index for the current character in the alphabet array
        const index = p.charCodeAt(i) - 'a'.charCodeAt(0);
        // Update the maximum substring length for the current character if necessary
        maxSubstringLengthByCh[index] = Math.max(maxSubstringLengthByCh[index], currentLength);
    }

    // Compute the sum of all maximum substring lengths to get the final count of distinct non-empty substrings
    return maxSubstringLengthByCh.reduce((sum, value) => sum + value);
}

```

Time and Space Complexity

Time Complexity

The given code iterates through each character of the string `p` only once with a constant time operation for each character including arithmetic operations and a maximum function. This results in a time complexity of `O(n)`, where `n` is the length of the string `p`.

Space Complexity

The space complexity is determined by the additional space used besides the input itself. Here, a constant size array `dp` of size 26 is used, which does not grow with the size of the input string `p`. Therefore, the space complexity is `O(1)`, since the space used is constant and does not depend on the input size.