1147. Longest Chunked Palindrome Decomposition

Hard | Greedy | Two Pointers | String | Dynamic Programming | Hash Function | Rolling Hash

Leetcode Link

Problem Description

The problem presents a scenario where you have a string `text`, and your task is to split it into `k` non-empty substrings such that each substring `subtext_i` equals the `subtext` at the position `k - i + 1`. This condition makes it so that the substrings are symmetrical around the center of `text`. The objective is to determine the largest possible value of `k`, which corresponds to the most such symmetrical substrings you can split `text` into. The problem requires you to take `text` and look for the longest sequences at the start and end that are identical, split them off, and then continue doing this iteratively until the entire `text` has been processed.

Intuition

The intuition behind the offered solution is to use a two-pointer approach. One pointer starts at the beginning of `text` (`i`) and the other at the end (`j`). The strategy is to look for the longest identical substring from these two starting points, which would be a candidate for `subtext_1` and `subtext_k`. If a matching pair is found, then we can consider this as two parts of the decomposition, increment the number of substrings `ans` by 2, and move the pointers inward accordingly.

This process continues, searching for the next longest match, but now starting from the new position of `i` and `j`. This is done by expanding the length of the substring we're checking (`k`) until another match is found or we run out of characters to check. If at any point, it's impossible to find a matching pair for the current positions of `i` and `j`, it essentially means that the central part of `text` does not have a symmetrical counterpart, and we can increment `ans` by 1 to account for the remaining middle substring and then terminate the process.

The reason why this approach works is due to the greedy nature of the problem: slicing off the longest possible matching substrings at each step ensures that we're getting the most significant `k` value possible, as any longer substring would have naturally included shorter matching pairs that we find at each iteration.

Solution Approach

The solution implements a straightforward greedy algorithm using a two-pointer technique, which is a common pattern for exploring intervals or sequences within arrays and strings.

Initially, the count of substrings `ans` is set to zero. Two index pointers, `i` and `j`, are initialized to point to the start and end of the string `text`, respectively. The main loop continues as long as `i` is less than or equal to `j`, ensuring that we haven't completely processed all characters in `text`.

While in the loop, we initialize a variable `k` to 1, which represents the current length of the substring we're trying to match from both ends of the text. The variable `ok` is set to `False`, indicating whether we've found a matching pair of substrings. As long as the potential matching substrings don't overlap (ensured by checking that `i + k - 1 < j - k + 1`), we compare the substrings using slicing (i.e., `text[i: i + k] == text[j - k + 1: j + 1]`). If a match is found:

- We increment `ans` by 2 as we have found a pair of matching substrings.
- We update `i` and `j` to move past the matched substrings, effectively reducing the remaining portion of `text` that needs to be processed.
- We set `ok` to `True` since we've found a match.

If the inner `while` loop finishes without finding any matching substrings (`ok` remains `False`), it implies that the remaining text cannot be split symmetrically. In that case, we increment `ans` by 1 to account for the unique central substring and break the loop.

Finally, the function returns the number of decomposed substrings `ans`.

This solution relies on Python's ability to slice strings efficiently, and no additional data structures are used outside of the basic variables to keep track of indices and counts. The algorithm's efficiency primarily comes from the greedy approach of attempting to peel off the longest matching substrings first, thereby optimizing the overall number of splits, `k`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we are given the string `text = "abbaaccbbaa"` and we want to split it into the maximum number of symmetrical substrings.

We initialize `ans` to `0`, `i` to `0`, and `j` to `len(text) - 1` which is `10`. Our text has a length of `11`, where characters at positions `0` and `10` are `'a'`.

The main loop starts, and we define `k` to be `1` inside the loop, and `ok` to be `False`. We check whether the substring at the start and end of `text` are equal, which they are (`text[0:1]` is `'a'`, and `text[10:11]` is also `'a'`), hence we increment `ans` to `2`, and update `i` to `1` and `j` to `9`.

We again set `k` to `1`. The next characters at the start and end (`i = 1` and `j = 9`) of the remaining `text` are `'b'` and `'b'`, which match. We continue this process and increment `ans` to `4`, update `i` to `2` and `j` to `8`.

At this point, `text[i:i+1]` is `'b'` and `text[j:j+1]` is `'a'`, which do not match. We increase `k` to `2` and check again. Now, `text[2:4]` is `'ba'` and `text[7:9]` is `'cb'`, so they also don't match. After increasing `k` to `3`, we find that `text[2:5]` is `'baa'` and `text[6:9]` is `'bba'`, that don't match either. We increment `k` until it's not possible to compare without overlap, it does not find any matching pairs, so it sets `ok` to `False`.

Since we have run out of characters to check for symmetrical substrings without overlap and `ok` is `False`, we increment `ans` by `1` to account for the remaining middle portion of the `text` and break out of the loop.

The central part of the text that doesn't have a symmetrical counterpart is `cca`. Therefore, we now have `5` symmetrical substrings within `text`: `'a'`, `'bb'`, `'cca'`, `'bb'`, `'a'`.

The process terminates here and returns `ans` which is `5`, corresponding to these substrings. This gives us the largest possible value of `k` for the given `text`, which is the most such symmetrical substrings we can split `text` into using the greedy algorithm approach described in the solution.

Python Solution

```python
class Solution:
    def longestDecomposition(self, text: str) -> int:
        # Initialize the count of decomposed pieces
        count = 0
        # Pointers to the start and end of the string
        start, end = 0, len(text) - 1
        # Continue decomposing as long as the start pointer is before or equal to the end pointer
        while start <= end:
            match_length = 1  # Length of the matching piece
            found_match = False  # Flag to check if we found a matching piece
            # Attempt to find the longest piece from the start equal to the end
            while start + match_length - 1 < end - match_length + 1:
                # Check if the substring from the start is equal to the substring from the end
                if text[start : start + match_length] == text[end - match_length + 1 : end + 1]:
                    count += 2  # If a match is found, increase count by two pieces
                    # Move the pointers inward after counting the matched pieces
                    start += match_length
                    end -= match_length
                    found_match = True  # Set the flag to true as we have found a match
                    break  # Break out of the loop since we only consider the longest piece
                match_length += 1  # Increase the length of the match and check again

            # If no matching piece is found, there must be a unique middle part
            if not found_match:
                count += 1  # Count the unique middle part as one piece
                break  # Break out of the loop as we are done decomposing

        # Return the total count of decomposed pieces
        return count

```

Java Solution

```java
class Solution {
    // Method to count the maximum number of non-empty parts the string can be decomposed into
    // such that the concatenation of those parts equals the original string. Also, each part
    // is a substring of the string such that the beginning and ending parts of the string are equal.
    public int longestDecomposition(String text) {
        int count = 0; // Initialize the count of decomposed parts to 0

        // Two pointers: 'start' and 'end' are used to progressively check the string from both ends.
        for (int start = 0, end = text.length() - 1; start <= end;) {
            boolean foundMatchingPart = false;

            // Try to find a matching pair starting from smallest possible parts moving to larger ones.
            for (int k = 1; start + k - 1 < end - k + 1; ++k) {

                // Check if there is a matching part at the current position.
                if (isMatchingPart(text, start, end - k + 1, k)) {
                    count += 2; // Increment count by 2 since a matching pair is found.
                    start += k; // Move the 'start' pointer forward by the length of the matched part.
                    end -= k;   // Move the 'end' pointer backward by the length of the matched part.
                    foundMatchingPart = true; // A matching part was found.
                    break; // Break the loop as we found the matching part.
                }
            }

            // If no matching part is found, it means we have the middle part or unmatched single character left.
            if (!foundMatchingPart) {
                count++; // Increment the count as this is a unique part.
                break; // Break the loop as we have covered the whole string.
            }
        }

        return count; // Return the total count of parts.
    }

    // Helper method to check if two substrings of 's' of length 'k' are equal
    private boolean isMatchingPart(String s, int start, int end, int k) {
        while (k-- > 0) {
            // Compare characters of both parts one by one.
            if (s.charAt(start++) != s.charAt(end++)) {
                // If characters do not match, the parts are not equal.
                return false;
            }
        }
        // After comparing all characters, the parts are equal.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int longestDecomposition(string text) {
        int answer = 0; // To store the count of decomposed parts.

        // Lambda function that compares substrings of length 'length'
        // starting from 'start1' and 'start2' indices for equality.
        auto isSubstringEqual = [&](int start1, int start2, int length) -> bool {
            while (length--) {
                if (text[start1++] != text[start2++]) {
                    return false;
                }
            }
            return true;
        };

        // Iterate through the string to find the maximum number of parts
        // the string can be decomposed into.
        for (int left = 0, right = text.size() - 1; left <= right;) {
            bool matched = false; // Flag to check if a match was found.

            // Try to find the longest prefix that matches the suffix
            // where the current length is 'partLength'.
            for (int partLength = 1; left + partLength - 1 < right - partLength + 1; ++partLength) {
                // If a matching part is found, increment the answer by 2
                // (since both prefix and suffix are counted) and adjust
                // the pointers to search the remaining string.
                if (isSubstringEqual(left, right - partLength + 1, partLength)) {
                    answer += 2;
                    left += partLength;
                    right -= partLength;
                    matched = true; // Found a match, so set the flag.
                    break; // Break as we are done processing this part.
                }
            }

            // If no matching parts were found, the middle part cannot be decomposed
            // further and it adds 1 to the number of decomposed parts.
            if (!matched) {
                answer += 1;
                break; // Break as no further decomposition is possible.
            }
        }
        return answer; // Return the total count of decomposed parts.
    }
};

```

Typescript Solution

```typescript
function longestDecomposition(text: string): number {
    // Get the length of the text
    const textLength: number = text.length;

    // Base case: if the text is less than 2 characters, return its length
    if (textLength < 2) {
        return textLength;
    }

    // Loop through the text, checking symmetric substrings from the start and end
    for (let i: number = 1; i <= textLength >> 1; i++) {
        // If a symmetric substring is found at the start and end of the text
        if (text.slice(0, i) === text.slice(textLength - i)) {
            // Recursively perform the decomposition on the remaining central part of the text
            // Add 2 to the count for the two decomposed parts found
            return 2 + longestDecomposition(text.slice(i, textLength - i));
        }
    }

    // If no symmetrical substrings are found, the text can't be further decomposed
    return 1;
}

```

Time and Space Complexity

The time complexity of the code is given by the nested `while` loops. The outer loop runs for at most `n/2` iterations, where `n` is the length of the input string `text`, because in each iteration at least one character is removed from each end of the `text`. The inner `while` loop could potentially run for `n/2` iterations as well in the case where the matching substrings at the ends are just one character long, but located at the center of `text`. In the worst case scenario, every character is checked against its mirror character on the other side of the string, which happens `n/2` times for half the length of the string. Therefore, the worst-case time complexity is `O(n^2)`.

The space complexity of the code is `O(1)`. This is because the algorithm uses a fixed amount of extra space regardless of the input size: variables `ans`, `i`, `j`, and `k` do not depend on the size of the input `text`.