1216. Valid Palindrome III

Hard String Dynamic Programming

Leetcode Link

Problem Description

The task is to determine if a given string `s` can be transformed into a palindrome by removing at most `k` characters. A palindrome is a sequence that reads the same backward as forward, such as "radar" or "level." A k-palindrome is an extended version of this concept, where some flexibility is allowed by permitting the removal of up to `k` characters to achieve a palindrome. To solve this problem, we need to establish a method for deciding whether or not the string, after the allowed modifications, can be considered a palindrome.

Intuition

The solution leverages a dynamic programming approach. We use a 2D array `f` with dimensions equivalent to the length of the string `s`, where `f[i][j]` represents the length of the longest palindromic subsequence within the substring `s[i..j]`. The underlying intuition is that if we can identify such a subsequence that is sufficiently long, then the characters that are not part of this subsequence are the ones we can consider removing. If the number of these characters is less than or equal to `k`, then `s` is a `k`-palindrome.

We start by recognizing that any single character is trivially a palindrome, which gives us the initial condition for our dynamic programming table (`f[i][i] = 1` for all indices `i`). From there, we move on to consider substrings of increasing length. When the characters at both ends of the current substring are the same, it contributes to a longer palindromic subsequence by simply framing any palindromic subsequence found within the interior of the substring (i.e., without the matched characters). When they do not match, we have a choice to make: we either consider the longest subsequence possible without the first character (`s[i]`) or without the last character (`s[j]`), which is embodied by taking the maximum of `f[i+1][j]` and `f[i][j-1]`.

Finally, if we find a `f[i][j]` value such that `f[i][j] + k` is greater than or equal to the length of the string `n`, we have established that `s` can be considered a k-palindrome and can return `true`. If no such pair of indices `(i, j)` yields a suitable longest palindromic subsequence, the function will ultimately return `false`.

The intuition emerges from recognizing that palindromicity is, fundamentally, about symmetry and that dynamic programming allows us to explore and remember the outcomes of smaller problems (substrings) to solve larger ones.

Solution Approach

The provided code employs a dynamic programming technique to solve the problem. The algorithm involves the following steps:

1. Initialize a 2D table `f` with dimensions `n x n`, where `n` is the length of the string `s`. This table will store the length of the longest palindromic subsequence for every substring `s[i..j]`.
2. Set `f[i][i] = 1` for all `i`, representing the fact that each individual character is a palindrome of length 1.
3. Fill the table `f` in a bottom-up manner. Start by considering all substrings of length 2 and increase the length gradually. The outer loop starts from the second to last character and moves towards the first one (in reverse order).
4. For each pair `(i, j)` where `i < j`, two cases are possible:
   - If `s[i] == s[j]`, the characters at both ends are the same, and they can contribute to the palindromic subsequence. The current cell `f[i][j]` is updated to `f[i+1][j-1] + 2`, adding the two matching characters to the length of the longest palindrome within the inner substring `s[i+1..j-1]`.
   - If `s[i] != s[j]`, the characters are different and cannot be part of the same palindrome so we must choose whether to exclude the character at the start or at the end of the string. The cell `f[i][j]` is updated to the maximum between `f[i+1][j]` and `f[i][j-1]`, effectively discarding either `s[i]` or `s[j]` and considering the longer of the two resulting subsequences.
5. As we fill the table, we check after each update if the condition `f[i][j] + k >= n` is met. This condition checks whether the length of the longest palindromic subsequence plus the maximum allowed number of removals `k` covers the entire string length `n`. If it does, we know that the string `s` can be transformed into a palindrome by removing at most `k` characters, and we can return `true`.
6. If we go through the entire table without the condition being met, we return `false`, as it is not possible to transform `s` into a palindrome within the constraints of the problem.

The use of dynamic programming here is crucial as it optimizes the solution by avoiding redundant computation of the longest palindromic subsequence through memoization in the table `f`. This optimization reduces the time complexity of the algorithm compared to checking all possible subsequences individually.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose the given string `s` is "abxcyba" and `k` is 1. We want to determine if "abxcyba" can be turned into a palindrome by removing at most one character.

We'll initialize a 2D array `f` with dimensions `7x7` (`n = 7` for our example, since the string length is 7). Initially, every entry `f[i][i]` is set to 1, which corresponds to each individual character being a palindrome of length 1.

```
1 f = [1, 0, 0, 0, 0, 0, 0]
2     [0, 1, 0, 0, 0, 0, 0]
3     [0, 0, 1, 0, 0, 0, 0]
4     [0, 0, 0, 1, 0, 0, 0]
5     [0, 0, 0, 0, 1, 0, 0]
6     [0, 0, 0, 0, 0, 1, 0]
7     [0, 0, 0, 0, 0, 0, 1]
```

We start populating `f` for substrings of length 2:

- For `i = 5` and `j = 6` (`s[i]` is 'b' and `s[j]` is 'a'), they don't match, so we take the maximum of `f[i+1][j]` and `f[i][j-1]`, both of which are 0, so `f[i][j]` remains 0.
- For `i = 4` and `j = 5` (`s[i]` is 'y' and `s[j]` is 'b'), they don't match, we do the same and `f[4][5]` remains 0.
- ... (We continue this for all substrings of length 2)

Then we move to substrings of length 3 and so on, up to the length of the whole string:

- During this process, for `i = 0` and `j = 6` (the whole string), if `s[i]` and `s[j]` (`s[0] = 'a'` and `s[6] = 'a'`) are the same, so `f[i][j]` is `f[1][5] + 2`.

In the end, we check if `f[0][n-1]` (which represents the length of the longest palindromic subsequence of the whole string) plus `k` (which is 1 in this case) is greater than or equal to the length of the string `n` (which is 7). For the example, we'll assume that the longest palindromic subsequence we found has a length of 5.

```
1 f[0][n-1] = 5
```

Since `f[0][n-1] + k = 5 + 1 = 6` is not greater than or equal to 7, we determine that we cannot transform the string "abxcyba" into a palindrome by removing at most one character, and we return `false`.

This walkthrough provides a step-by-step illustration of how the dynamic programming table is populated and how the decision is made based on the values computed.

Python Solution

```python
class Solution:
    def isValidPalindrome(self, s: str, k: int) -> bool:
        # Length of the input string
        length = len(s)

        # Initialize a 2D DP array with zeros, with the same dimensions as the string length
        dp = [[0] * length for _ in range(length)]

        # Each single character is a palindrome, so we fill the diagonal with 1's
        for i in range(length):
            dp[i][i] = 1

        # Dynamic programming to compute the longest palindromic subsequence
        # Start from the second last character and move towards the first character
        for i in range(length - 2, -1, -1):
            # Start from the next character till the end of the string
            for j in range(i + 1, length):
                # If characters match, extend the length of the palindrome by 2
                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                # If no match, take the maximum length of the palindrome without one of the characters
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

                # Check if the current palindromic subsequence plus allowed deletions covers the entire string
                if dp[i][j] + k >= length:
                    return True

        # If no valid palindrome subsequence is found that can be transformed with k deletions, return False
        return False

```

Java Solution

```java
class Solution {
    public boolean isValidPalindrome(String s, int k) {
        int stringLength = s.length();
        int[][] dp = new int[stringLength][stringLength];

        // Initialize each character as a palindrome of length 1
        for (int i = 0; i < stringLength; ++i) {
            dp[i][i] = 1;
        }

        // Build the table in a bottom-up manner
        for (int i = stringLength - 2; i >= 0; --i) {
            for (int j = i + 1; j < stringLength; ++j) {
                // If characters match, take the length from the diagonally lower cell and add 2
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
                    // If characters do not match, take the max from either side (ignoring one character)
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
                }
                // If the palindrome length plus the allowed deletions covers the whole string length, it is valid
                if (dp[i][j] + k >= stringLength) {
                    return true;
                }
            }
        }

        // If none of the substrings could be a palindrome with the given k, return false
        return false;
    }
}

```

C++ Solution

```cpp
#include <cstring> // Include necessary header for memset
#include <algorithm> // Include necessary header for max function
using namespace std;

class Solution {
public:
    bool isValidPalindrome(string s, int k) {
        int length = s.length(); // Get the length of the string
        int dp[length][length]; // Declare the dp (dynamic programming) array
        memset(dp, 0, sizeof dp); // Initialize the dp array with 0

        // Fill dp for substrings of length 1 (individual characters)
        for (int i = 0; i < length; ++i) {
            dp[i][i] = 1;
        }

        // Fill the dp array in a bottom-up manner
        for (int start = length - 2; start >= 0; --start) {
            for (int end = start + 1; end < length; ++end) {
                // If characters match, increment the count by 2
                if (s[start] == s[end]) {
                    dp[start][end] = dp[start + 1][end - 1] + 2;
                } else {
                    // Otherwise, take the maximum of excluding either character
                    dp[start][end] = max(dp[start + 1][end], dp[start][end - 1]);
                }
                // If the palindrome length plus allowed deletions cover the whole string, return true
                if (dp[start][end] + k >= length) {
                    return true;
                }
            }
        }
        // If no valid palindrome is found, return false
        return false;
    }
};
```

Typescript Solution

```typescript
/**
 * Determines if a string can become a palindrome through a maximum of 'k' character deletions.
 * @param s The input string to be evaluated.
 * @param k The maximum number of characters that can be deleted.
 * @returns A boolean indicating whether the string can be made into a palindrome.
 */
function isValidPalindrome(s: string, k: number): boolean {
    // Get the length of the input string.
    const strLength = s.length;

    // Create a 2D array to store the length of the longest palindromic subsequence.
    const longestPalindrome = Array.from({ length: strLength }, () => Array(strLength).fill(0));

    // Initialize one-letter palindromes.
    for (let i = 0; i < strLength; ++i) {
        longestPalindrome[i][i] = 1;
    }

    // Build the 2D array with lengths of the longest palindromic subsequences using dynamic programming.
    for (let i = strLength - 2; i >= 0; --i) {
        for (let j = i + 1; j < strLength; ++j) {
            // If characters are equal, add 2 to the result from the substring without these two characters.
            if (s[i] === s[j]) {
                longestPalindrome[i][j] = longestPalindrome[i + 1][j - 1] + 2;
            } else {
                // If characters are not equal, use the maximum result from ignoring either character.
                longestPalindrome[i][j] = Math.max(longestPalindrome[i + 1][j], longestPalindrome[i][j - 1]);
            }
            // Check if the current longest palindrome subsequence plus allowed deletions would cover the whole string.
            if (longestPalindrome[i][j] + k >= strLength) {
                return true;
            }
        }
    }

    // If no palindromic subsequence long enough to be valid with 'k' deletions is found, return false.
    return false;
}

```

Time and Space Complexity

The time complexity of the code is `O(n^2)`. This arises because the algorithm contains two nested for-loops each running from 0 to n-1 for the outer loop and from i+1 to n for the inner loop, which performs a number of operations proportional to the square of the length of the input string `s`.

The space complexity of the algorithm is also `O(n^2)` due to the allocation of a 2D array `f` with `n * n` elements, where `n` is the length of string `s`. This array stores the length of the longest palindrome sequence found at every start and end index pair in the input string.