1745. Palindrome Partitioning IV

Hard String Dynamic Programming Leetcode Link

Problem Description

The problem requires us to determine whether a given string `s` can be divided into three non-empty substrings that each form a palindrome. A palindrome is a string that reads the same forward and backward—for example, `racecar` or `madam`. If it is possible to split the input string into such three palindromic segments, we return `true`; otherwise, we return `false`.

Intuition

The solution approach involves first understanding the characteristics of a palindrome. A string is a palindrome if the first and last characters are the same, and if the substring between them (if any) is also a palindrome. The intuition behind the solution involves checking all possible ways of splitting the string into three substrings and testing for each split if all three resulting substrings are palindromes.

To decide whether a substring is a palindrome, we can precompute the palindromic status of all possible substrings. We create a 2D array `g` where `g[i][j]` is `True` if the substring from index `i` to index `j` is a palindrome. This precomputation uses dynamic programming. Once we have this 2D array, we can iterate through all possible splits with two nested loops, using the `g` array to check if each potential three-segment split consists of palindromes. If such a three-palindrome partitioning exists, we return `True`; if we finish our iterations without finding one, we return `False`.

The dynamic programming part to fill the `g` array checks palindromes by comparing the characters at the indices `i` and `j`, and by ensuring that the substring between them is a palindrome (by checking the entry `g[i+1][j-1]`).

By precomputing the palindrome statuses, we save time in the later stage where we're iterating over potential partition indices, allowing this solution to be efficient and effective for the problem.

Solution Approach

The solution employs dynamic programming and precomputation to determine the palindromic nature of all substrings in the string `s`. The data structure used for this is a 2-dimensional array `g` where `g[i][j]` holds a boolean value indicating whether the substring starting at index `i` and ending at index `j` is a palindrome.

Here's a step-by-step breakdown of the implementation:

1. Initialize a 2D array `g` of size `n * n` filled with `True`, where `n` is the length of the string `s`. Since a substring of length 1 is trivially a palindrome, the diagonal of the array `g` (where `i==j`) represents these substrings and is thus initialized to `True`.

2. Populate the `g` array using a nested loop. Starting from the end of the string, we move towards the beginning (decreasing `i`). For each `i`, we iterate `j` from `i+1` to the end of the string. We update `g[i][j]` to `True` only if the characters at `i` and `j` are equal and the substring `g[i+1][j-1]` is a palindrome. This check ensures that we confirm the palindromic property for a substring by comparing its outer characters and using the previously computed statuses of its inner substrings.

3. Once the `g` array is computed, we use two nested loops to try all possible partitions. The first loop iterates over the first potential cut in the string (denoted by `i`; it marks the end of the first substring), which goes from index `0` to `n-3`. The second loop goes over the second potential cut (denoted by `j`; it marks the end of the second substring), which starts one position after `i` and goes until `n-2`. These loops make sure that each of the three resulting substrings is non-empty.

4. For each pair of positions `(i, j)`, we check if the substrings `s[0...i]`, `s[i+1...j]`, and `s[j+1...end]` (end being `n-1`, inclusive) are palindromes. This is done using the `g` array: if `g[0][i]`, `g[i+1][j]`, and `g[j+1][n-1]` are all `True`, this means that the current partition forms three palindromic substrings.

5. If a valid partition is found, we immediately return `True`.

6. If no valid partition is found after going through all possible pairs of `(i, j)`, then it's not possible to split the string into three palindromic substrings, and we return `False`.

The use of dynamic programming makes this process efficient since we avoid recomputing the palindrome status of substrings when checking possible partition positions.

Example Walkthrough

Let's use the string `"abacaba"` to illustrate the solution approach.

1. We first initialize the 2D array `g` of size `7 x 7` (since "abacaba" has 7 characters) with `True` on the diagonal, as any single character is a palindrome by itself.

2. We then fill the `g` array with the palindromic status for all substrings. For example, `g[0][1]` will be `False` because `s[0]` is 'a' and `s[1]` is 'b', and "ab" is not a palindrome. We continue this for all possible substrings, ending up with `g[0][6]` being `True` because "abacaba" is a palindrome, and similarly `g[2][4]` being `True` for the substring "aca".

3. Next, we iterate over all possible pairs of indices `(i, j)` to find valid partitions. Our first loop with variable `i` will run from 0 to 4 (since we need at least two characters after the first partition for the remaining two palindromic substrings to be non-empty), and our nested loop with variable `j` will run from `i+1` to 5.

4. For each pair `(i, j)`, we check the 2D array `g` to see if `g[0][i]`, `g[i+1][j]`, and `g[j+1][6]` are all `True`.

5. When `i` is `2` (end of the first substring) and `j` is `4` (end of the second substring), we find that:

   - `g[0][2]` which represents "aba" is `True`,
   - `g[3][4]` which represents "ca" is `False`, and
   - `g[5][6]` which represents "ba" is `False`.

6. So the substring partitions "aba|ca|ba" are not palindromes collectively. We continue searching.

7. We eventually find that when `i` is `0` and `j` is `3`, we get:

   - `g[0][0]` which represents "a" is `True`,
   - `g[1][3]` which represents "bac" is `False`, and
   - `g[4][6]` which represents "aba" is `True`.

   No palindromic partition here, so we keep searching.

8. Continuing this, we finally hit a combination when `i` is `2` and `j` is `5`, resulting in the partitions "aba|cac|aba":

   - `g[0][2]` is `True` for "aba",
   - `g[3][5]` is `True` for "cac", and
   - `g[6][6]` is `True` for "a".

9. Thus, "abacaba" can be split into three palindromic substrings: "aba", "cac", and "a". At this point, we return `True`.

No further searching is needed; we have successfully found a split of the example string "abacaba" into three palindromic substrings using the outlined solution approach.

Python Solution

```python
class Solution:
    def checkPartitioning(self, s: str) -> bool:
        # Get the length of the string
        length = len(s)

        # Initialize a matrix to keep track of palindrome substrings
        palindrome = [[True] * length for _ in range(length)]

        # Fill the palindrome matrix with correct values
        # A substring s[i:j+1] is a palindrome if s[i] == s[j] and
        # the substring s[i+1:j] is also a palindrome
        for start in range(length - 1, -1, -1):
            for end in range(start + 1, length):
                palindrome[start][end] = s[start] == s[end] and (start + 1 == end or palindrome[start + 1][end - 1])

        # Attempt to partition the string into three palindromes
        # The external two loops iterate over every possible first and second partition
        for first_cut in range(length - 2):
            for second_cut in range(first_cut + 1, length - 1):
                # If the three substrings created by first_cut and second_cut are all palindromes,
                # return True, as it is possible to partition the string into three palindromes
                if palindrome[0][first_cut] and palindrome[first_cut + 1][second_cut] and palindrome[second_cut + 1][-1]:
                    return True
        # If no partitioning into three palindromes is found, return False
        return False

```

Java Solution

```java
class Solution {
    public boolean checkPartitioning(String s) {
        int length = s.length();

        // dp[i][j] will be 'true' if the string from index i to j is a palindrome.
        boolean[][] dp = new boolean[length][length];

        // Initial fill of dp array with 'true' for all single letter palindromes.
        for (boolean[] row : dp) {
            Arrays.fill(row, true);
        }

        // Fill the dp array for substrings of length 2 to n.
        for (int start = length - 1; start >= 0; --start) {
            for (int end = start + 1; end < length; ++end) {
                // Check for palindrome by comparing characters and checking if the substring
                // between them is a palindrome as well.
                dp[start][end] = s.charAt(start) == s.charAt(end) && dp[start + 1][end - 1];
            }
        }

        // Try to partition the string into 3 palindrome parts by checking all possible splits.
        for (int i = 0; i < length - 2; ++i) {
            for (int j = i + 1; j < length - 1; ++j) {
                // If the three parts [0, i], [i+1, j], [j+1, length-1] are all palindromes,
                // return true indicating the string can be partitioned into 3 palindromes.
                if (dp[0][i] && dp[i + 1][j] && dp[j + 1][length - 1]) {
                    return true;
                }
            }
        }

        // If no partitioning was found, return false.
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool checkPartitioning(string str) {
        int strSize = str.size();
        // A 2D dynamic programming matrix to store palindrome status
        vector<vector<bool>> isPalindrome(strSize, vector<bool>(strSize, true));

        // Fill the dp matrix for all substrings
        for (int start = strSize - 1; start >= 0; --start) {
            for (int end = start + 1; end < strSize; ++end) {
                // Check and set palindrome status
                isPalindrome[start][end] = (str[start] == str[end]) &&
                                           (start + 1 == end || isPalindrome[start + 1][end - 1]);
            }
        }

        // Check all possible partitions
        // The outer loop is for the first cut
        for (int firstCut = 0; firstCut < strSize - 2; ++firstCut) {
            // The inner loop is for the second cut
            for (int secondCut = firstCut + 1; secondCut < strSize - 1; ++secondCut) {
                // If the three partitions are palindromes, return true
                if (isPalindrome[0][firstCut] &&
                    isPalindrome[firstCut + 1][secondCut] &&
                    isPalindrome[secondCut + 1][strSize - 1]) {
                    return true;
                }
            }
        }

        // If no partitioning satisfies the condition, return false
        return false;
    }
};

```

Typescript Solution

```typescript
// Method to check if a string can be partitioned into three palindromic substrings
function checkPartitioning(str: string): boolean {
    const strSize = str.length;
    // A 2D array to store palindrome status
    const isPalindrome: boolean[][] = Array.from({length: strSize}, () => Array(strSize).fill(true));

    // Fill the array for checking palindrome substrings
    for (let start = strSize - 1; start >= 0; --start) {
        for (let end = start + 1; end < strSize; ++end) {
            // Check and set palindrome status
            isPalindrome[start][end] = (str[start] === str[end]) &&
                                       (start + 1 === end || isPalindrome[start + 1][end - 1]);
        }
    }

    // Try all possible partitions with two cuts
    // The outer loop is for the position of the first cut
    for (let firstCut = 0; firstCut < strSize - 2; ++firstCut) {
        // The inner loop is for the position of the second cut
        for (let secondCut = firstCut + 1; secondCut < strSize - 1; ++secondCut) {
            // Check if the partitions created by these cuts are palindromic
            if (isPalindrome[0][firstCut] &&
                isPalindrome[firstCut + 1][secondCut] &&
                isPalindrome[secondCut + 1][strSize - 1]) {
                // If all three partitions are palindromes, return true
                return true;
            }
        }
    }

    // If no partition satisfies the condition, return false
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n^3)`. Here's a breakdown:

- Building the `g` matrix involves a nested loop that compares each character to every other character, resulting in a `O(n^2)` time complexity.
- The nested loops where `i` ranges from `0` to `n-3` and `j` ranges from `i+1` to `n-2` have a combined time complexity of `O(n^2)`.
- Within the innermost loop, the code checks if three substrings (`g[0][i]`, `g[i+1][j]`, and `g[j+1][-1]`) are palindromes, which is an `O(1)` operation since the `g` matrix already contains this information.

These two processes are sequential, so we consider the larger of the two which is `O(n^3)` as the time complexity. This comes from the fact that for each potential partition point `j`, we consider each `i` and for each `i` and `j`, we check a constant-time condition.

Space Complexity

The space complexity of the code is `O(n^2)`. This stems from the space required to store a 2D list (`g`) that maintains whether each substring from `i` to `j` is a palindrome. Since `g` is an `n` by `n` matrix, the space required is proportional to the square of the length of string `s`.