2911. Minimum Changes to Make K Semi-palindromes

Hard | Two Pointers | String | Dynamic Programming — Leetcode Link

Problem Description

The problem asks for a method to partition a string `s` into `k` substrings, with the goal of turning each substring into a "semi-palindrome" while making the least number of letter changes. A semi-palindrome here is defined as a string that, when divided into equal parts of length `d`, has those parts that read the same way backward as forward when compared index-wise.

An important thing to note is that a string of length `len` is considered a semi-palindrome if a positive integer divisor `d` exists (`1 <= d < len`) such that all characters at indices that are equivalent modulo `d` form a palindrome. For instance, 'adbgad' is a semi-palindrome because if we break it down with `d=2`, we get 'ad' and 'bg' as the sub-parts, and the characters at the equivalent modulo `d` positions ('a' with 'a' and 'd' with 'd') form palindromes.

The function should return an integer that represents the minimum number of letter changes required to achieve this partitioning into semi-palindromes.

Intuition

The intuition behind the solution is predicated on dynamic programming, a method used to solve complex problems by breaking them down into simpler sub-problems. The first insight is to realize that we can preprocess a "cost" matrix that represents the cost of converting any substring of `s` into a semi-palindrome.

We define a two-dimensional array `g` where `g[i][j]` represents the minimum number of changes needed to convert the substring from `i` to `j` into a semi-palindrome. This matrix is filled out by examining all substrings and finding the number of changes needed, ensuring that we check for all possible `d` that can make the string a semi-palindrome.

Once we have this `g` matrix, our main problem now is to find the optimal way to partition the string such that the sum of costs of converting `k` substrings is minimal. For this, we define another two-dimensional dynamic programming array `f`, where `f[i][j]` represents the minimal total changes needed for the first `i` characters in the string with `j` partitions already made.

Our final answer would be `f[n][k]`, which represents the minimal total changes needed for the entire string `s` with `k` partitions.

The preprocessing step suggested is to maintain a list of divisors for each length, which reduces the computations needed during the dynamic programming step. By avoiding unnecessary division operations while filling in the `g` matrix, we improve the efficiency of the solution.

Solution Approach

To implement the solution efficiently, the code uses dynamic programming. Here's a step-by-step explanation of how the algorithm works:

1. **Preprocessing the Cost Matrix `g`**:
   - A two-dimensional array `g` of size `(n+1) x (n+1)` is initialized with `inf` (infinity), signifying the initial state of uncomputed minimum changes for each substring.
   - The goal is to calculate `g[i][j]`, the minimum number of changes required to convert the substring from index `i` to `j` into a semi-palindrome.
   - To calculate `g[i][j]`, the algorithm iterates over all substrings and, for each substring, iterates over all possible divisors `d`.
   - For each divisor `d`, it calculates how many changes are needed to make the characters at indices equivalent modulo `d` form a palindrome.
2. **Dynamic Programming Array `f`**:
   - Another two-dimensional array `f` of size `(n+1) x (k+1)` is used to keep track of the minimum changes needed.
   - Here, `f[i][j]` represents the minimum total changes needed for the first `i` characters with `j` partitions.
   - The array is initialized with `inf`, except for `f[0][0]`, which is set to `0`.
   - Then, the algorithm iterates through all positions `i` in the string and for each possible partition `j`.
   - For each position `i` and partition `j`, it checks all ways to form the previous partition. This means checking each possible end `h` of the previous partition and adding the cost of converting the substring from `h+1` to `i` into a semi-palindrome from the matrix `g`.
3. **Optimization**:
   - As suggested, an optimization could be added by precomputing a list of divisors for each length, so the algorithm doesn't perform repeated division operations while populating the `g` matrix.
   - This would involve creating a list or dictionary to store divisors for each length before entering the main `g` matrix calculation loop.

Using these data structures and steps, the algorithm ensures that it calculates the minimum number of letter changes efficiently by leveraging the power of dynamic programming and avoiding redundant computations.

Finally, the algorithm returns the value of `f[n][k]`, which represents the minimum changes required for the entire string `s` with `k` partitions.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Imagine we have the string `s = "aacbbbd"` and we want to partition it into `k = 2` substrings and make the least number of changes to turn each substring into a semi-palindrome.

We start by initializing the cost matrix `g`. In our example, this will be a 7×7 matrix because our string's length `n` is 7. Initially, all the values in `g` are set to infinity, which will later be replaced by the actual costs of changing substrings into semi-palindromes.

We fill the `g` matrix as follows:

1. For the substring "aac", `g[0][2]`, we check for divisors of length 3. The only possible divisor is `d=1`. We don't need to make any change for indices modulo `d`, so `g[0][2]` is set to 0.
2. For the substring "bb", `g[3][4]`, with only one possible divisor `d=1`, no change is needed, and `g[3][4]` is also set to 0.
3. For "acbbbd", `g[1][6]`, for `d=1`, the character at index 1 is different from the character at index 6, so at least one change is needed to make them the same. However, let's say for `d=2` we find that fewer changes are needed; then we would set `g[1][6]` with this smaller number.

After calculating the `g` matrix by reviewing all substrings and their possible divisors, we'll move on to the dynamic programming matrix `f`, sized at 8×3 (considering `n+1` and `k+1` for the dimensions). Here, we're looking for `f[7][2]`, the minimum number of changes required to split "aacbbbd" into 2 semi-palindromes.

Suppose we decide the first partition will end after "aac". Then we solve for the remaining string "bbbd":

- We know that `f[3][1]` is the minimum changes for "aac" with 1 partition, which is 0 because "aac" is already a semi-palindrome.
- Now, we need to find `f[7][2]`, considering that we have one partition already. We compare all possible ends of the first partition—which is a single point since "aac" is semi-palindromic—and the minimum cost of making the second partition semi-palindromic.
- Let's say for argument's sake that turning "bbbd" into a semi-palindrome would require one change because only the last 'd' doesn't fit; then `g[4][7]` would be 1. We add this to `f[3][1]`, and our `f[7][2]` would be 1.

After filling the `f` matrix, considering all possible partitions and substrings, the algorithm concludes that the minimum number of changes required for "aacbbbd" into 2 semi-palindromes is 1.

The key takeaways from this example are to understand the two kinds of matrices used (`g` for substring conversion costs and `f` for minimum chase scores up to a certain length and partition count) and the dynamic approach to solving the problem by subdividing it into smaller, solvable parts.

Python Solution

```python
class Solution:
    def minimumChanges(self, s: str, k: int) -> int:
        n = len(s)
        # Initialize a DP table for storing minimum change count for substrings
        min_changes_dp = [[float('inf')] * (n + 1) for _ in range(n + 1)]

        # Precompute minimum changes needed for each substring to make it periodic
        for start in range(1, n + 1):
            for end in range(start, n + 1):
                substring_length = end - start + 1

                # Check for each possible period within the substring
                for period in range(1, substring_length):
                    if substring_length % period == 0:
                        change_count = 0

                        # Compare characters in the string to determine changes
                        for idx in range(substring_length):
                            mirror_idx = ((substring_length // period - 1 - idx // period) * period + idx % period)

                            # Stop if the index reaches or exceeds the mirror index
                            if idx >= mirror_idx:
                                break

                            if s[start - 1 + idx] != s[start - 1 + mirror_idx]:
                                change_count += 1

                        # Record the minimum changes found for the current substring
                        min_changes_dp[start][end] = min(min_changes_dp[start][end], change_count)

        # Initialize a DP table for the main problem
        min_changes_total = [[float('inf')] * (k + 1) for _ in range(n + 1)]
        min_changes_total[0][0] = 0

        # Compute minimum changes needed for each split of the string into k parts
        for i in range(1, n + 1):
            for j in range(1, k + 1):

                # Check the best way to split the string into j parts
                for previous in range(i - 1):
                    min_changes_total[i][j] = min(
                        min_changes_total[i][j],
                        min_changes_total[previous][j - 1] + min_changes_dp[previous + 1][i]
                    )

        # Return the minimum changes needed to split the string into k periods
        return min_changes_total[n][k]

```

Java Solution

```java
class Solution {

    // Function to calculate the minimum number of changes needed to meet the condition for each k
    public int minimumChanges(String s, int k) {
        int n = s.length();
        // The g matrix will store the minimum cost of making the substring (i, j) beautiful
        int[][] costMatrix = new int[n + 1][n + 1];
        // The dp matrix will store the minimum number of changes to make the first i characters beautiful with j operations
        int[][] dp = new int[n + 1][k + 1];
        final int infinity = 1 << 30; // Define a large number to represent infinity

        // Initialize the g and f matrices with infinity
        for (int i = 0; i <= n; ++i) {
            Arrays.fill(costMatrix[i], infinity);
            Arrays.fill(dp[i], infinity);
        }

        // Populate the costMatrix with the actual costs
        for (int i = 1; i <= n; ++i) {
            for (int j = i; j <= n; ++j) {
                int substringLength = j - i + 1;
                // Check for each possible divisor of m (substringLength)
                for (int d = 1; d < substringLength; ++d) {
                    if (substringLength % d == 0) {
                        int count = 0;
                        // Count the changes needed to make the substring beautiful from both ends towards the middle
                        for (int l = 0; l < substringLength; ++l) {
                            int r = ((substringLength / d) - 1 - (l / d)) * d + (l % d);
                            // We break to avoid double-counting changes since we compare elements from both ends
                            if (l >= r) {
                                break;
                            }
                            // Check if at positions l and r the characters are different
                            if (s.charAt(i - 1 + l) != s.charAt(i - 1 + r)) {
                                ++count;
                            }
                        }
                        // Store the minimum count for making the substring beautiful
                        costMatrix[i][j] = Math.min(costMatrix[i][j], count);
                    }
                }
            }
        }

        // Initialize the first row of the dp matrix
        dp[0][0] = 0;

        // Populate the dp matrix using previously computed values
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= k; ++j) {
                // Try making the first i characters beautiful by splitting the substring at each possible point h
                for (int h = 0; h < i - 1; ++h) {
                    // Calculate the minimum number of changes needed up to index i with j operations allowed
                    dp[i][j] = Math.min(dp[i][j], dp[h][j - 1] + costMatrix[h + 1][i]);
                }
            }
        }

        // The result is in the last cell of the dp matrix
        return dp[n][k];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum number of changes required
    // to make `k` palindromic subsequences in the given string `s`.
    int minimumChanges(string s, int k) {
        int stringSize = s.size();
        // Declare a 2D array (dpChangeCount) to store the minimum number of changes for each substring.
        // Initialize with a large value.
        int dpChangeCount[stringSize + 1][stringSize + 1];
        // Declare a 2D array (dpMinimumChanges) to store the minimum changes to form `k` subsequences.
        // Initialize with a large value.
        int dpMinimumChanges[stringSize + 1][k + 1];

        // Initialize arrays with max possible value (signifying that the value has not yet been computed).
        memset(dpChangeCount, 0x3f, sizeof(dpChangeCount));
        memset(dpMinimumChanges, 0x3f, sizeof(dpMinimumChanges));

        // Base case: with 0 length string and 0 subsequences, the changes needed are 0.
        dpMinimumChanges[0][0] = 0;

        // Iterate through the string to compute 'g', the minimum changes for each substring.
        for (int i = 1; i <= stringSize; ++i) {
            for (int j = i; j <= stringSize; ++j) {
                int substringLength = j - i + 1;
                // We check all divisors 'd' of the substring length.
                for (int d = 1; d < substringLength; ++d) {
                    if (substringLength % d == 0) {
                        int changeCount = 0;
                        for (int l = 0; l < substringLength; ++l) {
                            // Calculate the mirror index (r) for the current index (l).
                            int r = (substringLength / d - 1 - l / d) * d + l % d;
                            if (l >= r) {
                                break;
                            }
                            // If characters don't match, increase change count.
                            if (s[i - 1 + l] != s[i - 1 + r]) {
                                ++changeCount;
                            }
                        }
                        // Update g with the minimum changeCount found.
                        dpChangeCount[i][j] = min(dpChangeCount[i][j], changeCount);
                    }
                }
            }
        }

        // Compute the minimum changes f to form `k` palindromic subsequences.
        for (int i = 1; i <= stringSize; ++i) {
            for (int j = 1; j <= k; ++j) {
                for (int h = 0; h < i; ++h) {
                    // Combine the previous sub problem with the current, by selecting a
                    // previous state (f[h][j - 1]) and adding the changes for the new subsequence.
                    dpMinimumChanges[i][j] = min(dpMinimumChanges[i][j],
                                                 dpMinimumChanges[h][j - 1] + dpChangeCount[h + 1][i]);
                }
            }
        }

        // The result is the minimum number of changes needed to create `k` subsequences in the entire string.
        return dpMinimumChanges[stringSize][k];
    }
};

```

Typescript Solution

```typescript
function minimumChanges(inputString: string, k: number): number {
    const inputLength = inputString.length;

    // g[i][j] contains the minimum number of changes to make the substring from i to j (1-indexed) a palindrome.
    const minChangesToPalindrome = Array.from({ length: inputLength + 1 },
                                              () => Array(inputLength + 1).fill(Infinity));

    // f[i][j] contains the minimum changes to split the substring ending at i into j palindromic segments.
    const minChangesForSegments = Array.from({ length: inputLength + 1 },
                                             () => Array(k + 1).fill(Infinity));
    minChangesForSegments[0][0] = 0;

    // Pre-computing the number of changes needed to make each substring a palindrome.
    for (let start = 1; start <= inputLength; ++start) {
        for (let end = start; end <= inputLength; ++end) {
            const substringLength = end - start + 1;

            // Try all divisibility lengths to minimize palindrome formation changes.
            for (let d = 1; d < substringLength; ++d) {
                if (substringLength % d === 0) {
                    let changeCount = 0;
                    for (let l = 0; l < substringLength; ++l) {
                        const mirrorIndex = (((substringLength / d) | 0) - 1 - ((l / d) | 0)) * d + (l % d);
                        if (l >= mirrorIndex) {
                            break;
                        }
                        if (inputString[start - 1 + l] !== inputString[start - 1 + mirrorIndex]) {
                            ++changeCount;
                        }
                    }
                    minChangesToPalindrome[start][end] = Math.min(minChangesToPalindrome[start][end], changeCount);
                }
            }
        }
    }

    // Calculating the minimum number of changes for the full input string considering k segments.
    for (let i = 1; i <= inputLength; ++i) {
        for (let segments = 1; segments <= k; ++segments) {
            for (let prevEnd = 0; prevEnd < i; ++prevEnd) {
                minChangesForSegments[i][segments] = Math.min(minChangesForSegments[i][segments],
                                                              minChangesForSegments[prevEnd][segments - 1] +
                                                              minChangesToPalindrome[prevEnd + 1][i]);
            }
        }
    }

    // Returning the minimum changes required for the whole string to be split into k palindromic segments.
    return minChangesForSegments[inputLength][k];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down by analyzing the nested loops and the operations performed within them.

1. Constructing the `g` matrix (grouping and counting changes):

- The first two loops iterate over all subarrays `(i, j)` of the array, resulting in `O(n^2)`.
- Inside these loops, there is another loop iterating over the length of the subarray to find divisible lengths `d`, in the worst case (when `m` equals `n`), this results in an `O(n)` loop.
- Inside the innermost loop, we iterate up to `m` times (in the worst case, this is `n`), but due to the break condition (`l >= r`), we'll process each element at most once, so in the worst case, the innermost loop contributes `O(n/2)`, which simplifies to `O(n)`.
- Combining these loops, we have an `O(n^2)` loop times an `O(n * n/2)` loop, which simplifies to `O(n^4)`.

2. Constructing the `f` matrix (computing minimum changes):

- There are three nested loops: the outer loop runs `n` times, the middle loop runs `k` times, and the inner loop runs up to `i-1` times in the worst case.
- The time complexity of this section is `O(n * k * n)`, which simplifies to `O(n^2 * k)`.

The overall time complexity combines both parts, `O(n^4)` from the `g` matrix computation and `O(n^2 * k)` from the `f` matrix computation. The dominant term is `O(n^4)`, therefore the overall time complexity is `O(n^4)`.

Space Complexity

The space complexity can be analyzed by looking at the space allocated for the matrices `g` and `f`:

- The `g` matrix is a two-dimensional array of size `(n + 1) x (n + 1)`, which contributes `O(n^2)` to the space complexity.
- The `f` matrix is also a two-dimensional array of size `(n + 1) x (k + 1)`, which contributes `O(n * k)` to the space complexity.

The overall space complexity is the sum of the space complexities for `g` and `f`, so `O(n^2) + O(n * k)`. Since `k` can be at most `n`, the space complexity simplifies to `O(n^2)`.