1246. Palindrome Removal

Hard Array Dynamic Programming Leetcode Link

Problem Description

The problem gives us an integer array `arr` and asks us to remove palindromic subarrays using as few moves as possible. A subarray is palindromic if it reads the same backward as forward, for example `[1, 2, 2, 1]`. Each time you remove such a subarray, the remaining parts of the array close in to fill the gap. The task is to determine the minimum number of these moves required to remove all elements from the array.

Intuition

The intuition behind the solution is grounded in dynamic programming. We understand that trying all possible subarrays and checking if they are palindromic would be inefficient. Therefore, to optimize our approach, we can break down the problem:

1. We can start by understanding that if a single element is always a palindrome, thus it can be removed in one move.
2. For two elements, they can be removed in one move if they are the same or two moves if they are different, which can be the basis for our dynamic programming transition.
3. For a larger subarray, the number of moves depends on whether the ends of the subarray match. If they do, the subarray could potentially be removed in one move (if the entire subarray is a palindrome), or the minimum moves could be achieved by splitting the subarray at some other point.
4. For subarrays that don't have matching ends, we know that they can't be palindromic by themselves, so we look for where to split the subarray into two parts, each of which can be further split into palindromic subarrays.
5. A dynamic programming table `f` can store the minimum number of moves required to remove a subarray `arr[i:j]`. Initially, all entries are set to zero, except for subarrays of length 1, which are set to 1.

By approaching the problem in this way, we incrementally build up the solution using information from smaller subarrays, storing and reusing these results to make the algorithm efficient.

Solution Approach

We create a 2D array `f` to use as a dynamic programming table, where `f[i][j]` represents the minimum number of moves needed to remove the subarray starting at `i` and ending at `j`. We initialize a diagonal of this table with ones since any single element can be removed in one move.

The next step is to fill in the table for larger subarrays. We consider each possible size of subarray starting from the smallest (2) and going up to the size of the entire array.

For each size, we iterate over all possible starting indices. The table is filled in a manner that for each subarray `arr[i:j]`:

- If `arr[i]` equals `arr[j]`, it means we can potentially remove the subarray in one move if the inside part `arr[i+1:j-1]` is also a palindrome (which we know from `f[i+1][j-1]`).
- If `arr[i]` is not equal to `arr[j]`, it means we need to split the array into two parts and the number of moves is the sum of moves for each part at the optimal splitting point. To find this point, we iterate through all possible ways to split the subarray `arr[i:j]` into two parts `arr[i:k]` and `arr[k+1:j]`, and keep the minimum number of moves.

Once we fill in the table, the answer to the problem is the entry that represents the number of moves to remove the entire array, which is stored in `f[0][n - 1]`, where `n` is the length of the array.

Solution Approach

The solution uses dynamic programming, which is a method where we break down a complex problem into simpler subproblems and store the results of these subproblems to avoid redundant computations.

The algorithm uses a two-dimensional array `f` with dimensions `n` by `n`, where `n` is the length of the input array `arr`. Each entry `f[i][j]` in this table will eventually contain the minimum number of moves required to remove the subarray `arr[i...j]`.

Here's the detailed process of implementing the solution:

1. **Initialization**:
   - First, we initialize the array `f` with zeros and then we fill the diagonal `f[i][i]` with `1`, because a single element is a palindrome and can be removed in one move.
2. **Filling the DP Table**:
   - We then fill in the table in a bottom-up manner. To do this, we need to iterate over the subarrays starting from the smaller ones to the larger ones.
   - First, we iterate over the possible lengths of subarrays. Then for each length, we iterate over all possible starting points `i`.
   - For each subarray `arr[i...j]`, we check:
     - If `arr[i]` and `arr[j]` are equal and we have already calculated the minimum moves for `arr[i+1...j-1]` in `f[i+1][j-1]`, then the entire subarray is potentially a palindrome.
     - If `arr[i]` and `arr[j]` are the same and the inner subarray `arr[i+1...j-1]` is a palindrome, the entire subarray can be removed in one move—hence, `f[i][j]` gets the value of `f[i+1][j-1]`.
     - If `arr[i]` and `arr[j]` are not equal, or if the inner subarray is not a palindrome, we need to split the subarray at some point. To find the optimal splitting point, we iterate through all possible `k` to split `arr[i...j]` into `arr[i...k]` and `arr[k+1...j]`. We calculate the sum of moves for these parts, and `f[i][j]` gets the minimum value among all possible splits.
   - This process continues until we fill in the entry for the entire array `f[0][n-1]`.
3. **End Result**:
   - After completly filling up the DP table, the solution to the problem—which is the minimum number of moves needed to remove all elements from the array—is found in `f[0][n-1]`.

This dynamic programming approach is more efficient than brute force because we only compute the minimum moves for each subarray once, and we use these precomputed values to calculate the moves for larger subarrays.

The implementation is a classical DP solution where overlapping subproblems are solved just once and their solutions are stored, which reduces the time complexity significantly compared to a naive recursive approach. This showcases the power of dynamic programming in optimizing problem-solving strategies for specific types of problems.

By walking through the solution implementation, we see the application of the dynamic programming pattern, specifically the use of a 2D DP table, initialization based on base cases, filling in the table based on recursive relationships, and retrieval of the solution from the filled table.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the integer array `arr = [1, 2, 1, 3, 2, 2]`.

1. **Initialization**: We create a matrix `f` of size `6x6` because our array `arr` has 6 elements. We then fill the diagonal from `f[0][0]` to `f[5][5]` with `1` because a single element is a palindrome and can be removed in one move.

2. **Filling the DP Table**:
   - For subarrays of length 2, we compare each pair of elements:
     - `arr[0]` and `arr[1]` are not equal, so `f[0][1]` will be set to 2 (1 move for each element).
     - `arr[1]` and `arr[2]` are not equal, so `f[1][2]` will also be set to 2.
     - `arr[2]` and `arr[3]` are not equal, so `f[2][3]` will be set to 2.
     - `arr[3]` and `arr[4]` are not equal, so `f[3][4]` will be set to 2.
     - `arr[4]` and `arr[5]` are equal, so `f[4][5]` will be set to 1, since `[2, 2]` is a palindrome and can be removed in one move.
   - For subarrays longer than 2, we consider their internal splits:
     - `arr[0:2]` could potentially be palindromic if `arr[0]` and `arr[2]` are equal, which they are, so we check if `f[1][1]` is a palindrome, and indeed it is (since it's a single element). Hence, `f[0][2]` is set to 1.
     - We continue this process for subarrays `arr[1:3]`, `arr[2:4]`, `arr[3:5]`, checking the ends and the internal sections of the subarrays, updating our table as we find palindromes.
3. **End Result**:
   - After filling out the table, if we want to compute `f[0][5]`, which is the minimum number of moves to remove all elements in the array, we check:
     - The array ends with `arr[0]` and `arr[5]` are not the same, so we must split the array.
     - We try all possible splits, looking for the minimum `f[i][j]` values for `i` to `k` and `k+1` to `j` subarrays.
     - We find the minimum moves for all splits, and `f[0][5]` will have the minimum of these values.

As a result, `f[0][5]` holds the minimum number of moves required to remove all elements from `arr` by removing palindromic subarrays. This process reduces the problem into smaller, manageable steps that dynamically build upon each other to find the most efficient solution.

Python Solution

```python
from typing import List

class Solution:
    def minimum_moves(self, arr: List[int]) -> int:
        # The length of the array
        n = len(arr)

        # Initialize a 2D array to store the minimum number of moves for each subarray
        dp = [[0] * n for _ in range(n)]

        # Base case: A single element requires one move to become a palindrome
        for i in range(n):
            dp[i][i] = 1

        # Start from the second to last element down to the first element
        for i in range(n-2, -1, -1):
            # For each starting position, process subarrays with different lengths
            for j in range(i+1, n):
                # If we have a subarray of length 2, check if the elements are the same
                if i + 1 == j:
                    dp[i][j] = 1 if arr[i] == arr[j] else 2
                else:
                    # If the current elements are the same, compare with the inner subarray
                    # excluding both ends
                    moves = dp[i+1][j-1] if arr[i] == arr[j] else float('inf')
                    # Try all possible partitions of the subarray, and
                    # take the minimum number of moves required
                    for k in range(i, j):
                        moves = min(moves, dp[i][k] + dp[k+1][j])
                    dp[i][j] = moves

        # Return the minimum number of moves needed to make the entire array a palindrome
        return dp[0][n-1]

# Example of usage
# sol = Solution()
# print(sol.minimum_moves([1,3,4,1,5])) # Example input to get output

```

Java Solution

```java
class Solution {
    public int minimumMoves(int[] arr) {
        // Get the length of the array.
        int length = arr.length;
        // Initialize the memoization table with dimensions of the array length.
        int[][] dpMinMoves = new int[length][length];

        // Base case: single elements require one move to create a palindrome.
        for (int i = 0; i < length; ++i) {
            dpMinMoves[i][i] = 1;
        }

        // Fill the table in reverse order to ensure that
        // all sub-problems are solved before the bigger ones.
        for (int start = length - 2; start >= 0; --start) {
            for (int end = start + 1; end < length; ++end) {
                // If we're checking a pair of adjacent elements,
                // we can make a palindrome with one move if they're equal,
                // or with two moves if they are not.
                if (start + 1 == end) {
                    dpMinMoves[start][end] = arr[start] == arr[end] ? 1 : 2;
                } else {
                     // If the elements at the start and end are equal,
                     // we can potentially remove them both with a single move.
                     // Start with an initial large value to not affect the minimum comparison.
                    int minMoves = arr[start] == arr[end] ? dpMinMoves[start + 1][end - 1] : Integer.MAX_VALUE;
                    // Sweep through the array and split at every possible point to find
                    // the minimum moves for this start and end combination.
                    for (int split = start; split < end; ++split) {
                        minMoves = Math.min(minMoves, dpMinMoves[start][split] + dpMinMoves[split + 1][end]);
                    }
                    // Record the minimum moves needed for this subarray.
                    dpMinMoves[start][end] = minMoves;
                }
            }
        }

        // Return the minimum moves needed to make the entire array a palindrome.
        return dpMinMoves[0][length - 1];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minimumMoves(vector<int>& arr) {
        int length = arr.size();

        // Create dp (Dynamic Programming) table and initialize with zeros
        vector<vector<int>> dp(length, vector<int>(length, 0));

        // Base case: single element requires only one move
        for (int i = 0; i < length; ++i) {
            dp[i][i] = 1;
        }

        // Fill up the DP table for substrings of length >= 2
        for (int start = length - 2; start >= 0; --start) {
            for (int end = start + 1; end < length; ++end) {
                if (start + 1 == end) {
                    // If there are only two elements and they are equal, 1 move is required
                    // Otherwise 2 moves are required: one for each element
                    dp[start][end] = arr[start] == arr[end] ? 1 : 2;
                } else {
                    // Set initial value high for comparison purposes
                    int moves = arr[start] == arr[end] ? dp[start + 1][end - 1] : INT_MAX;
                    // Check for the minimum moves by dividing the range at different points
                    for (int split = start; split < end; ++split) {
                        moves = min(moves, dp[start][split] + dp[split + 1][end]);
                    }
                    dp[start][end] = moves;
                }
            }
        }

        // Return the minimum number of moves to make the full array a palindrome
        return dp[0][length - 1];
    }
};

```

Typescript Solution

```typescript
// Define the array type for ease of reference
type Array2D = number[][];

// Function to calculate the minimum number of moves to make the array a palindrome
function minimumMoves(arr: number[]): number {
    const length: number = arr.length;

    // Create dp (Dynamic Programming) table and initialize with zeros
    let dp: Array2D = Array.from({ length }, () => Array(length).fill(0));

    // Base case: single element requires only one move
    for (let i = 0; i < length; ++i) {
        dp[i][i] = 1;
    }

    // Fill up the DP table for substrings of length >= 2
    for (let start = length - 2; start >= 0; --start) {
        for (let end = start + 1; end < length; ++end) {
            if (arr[start] == arr[end]) {
                // If the elements at the start and end are equal, this can potentially
                // be folded into a palindrome, so check the inner subrange for moves
                dp[start][end] = dp[start + 1][end - 1];
            } else {
                // If elements are different, it requires at least 2 moves;
                // initialize with the worst case (1 move for each element)
                dp[start][end] = 2;
                // Check for the minimum moves by dividing the range at different points
                for (let split = start; split < end; ++split) {
                    dp[start][end] = Math.min(dp[start][end], dp[start][split] + dp[split + 1][end]);
                }
            }
        }
    }

    // Return the minimum number of moves to make the full array a palindrome
    return dp[0][length - 1];
}

// Minimum number of moves for a specific array can be found by calling the function.
// Example:
// const arr: number[] = [1, 2, 3, 4, 1];
// const moves: number = minimumMoves(arr);
// console.log(moves); // Output would be the number of moves needed

```

Time and Space Complexity

Time Complexity

The given code employs dynamic programming to find the minimum number of moves to make a palindrome by merging elements in the array. The time complexity is determined by the nested loops and the operations performed within them.

1. There are two nested loops, where one loops in a backward manner from `n-2` to `0` and the other loops from `i+1` to `n`. Each of these loops has `O(n)` iterations resulting in `O(n^2)` for the nested loops combined.
2. Inside the inner loop, there is another loop that ranges from `i` to `j`. In the worst-case scenario, this loop can iterate `n` times.
3. The innermost computation, however, is just a min comparison which is `O(1)`.

Multiplying all these together, the worst-case time complexity is `O(n^3)`.

Space Complexity

The space complexity is derived from the storage used for the dynamic programming table `f`.

1. The `f` table is a 2D array with dimensions `n x n`, containing `n^2` elements.
2. None of the loops use additional significant space.

Thus, the space complexity of the code is `O(n^2)`.