131. Palindrome Partitioning

Medium String Dynamic Programming Backtracking Leetcode Link

Problem Description

The problem asks for all ways a given string `s` can be split such that each substring in the partition is a palindrome. A palindrome is a string that reads the same forwards and backwards, such as 'racecar'.

For example, if the input string is `"aab"`, then there are two ways to partition it into substrings that are all palindromes: `["aa", "b"]` and `["a", "a", "b"]`.

Intuition

The core idea behind the solution is to use depth-first search (DFS). We want to explore all possible partitions and whenever we encounter a partition where all substrings are palindromes, we add that partition to our answer.

Here, DFS is used to recursively build partitions. Starting from the beginning of the string, we check every possible end index for a substring that could be a palindrome. If a substring from the current start index to the current end index is a palindrome, then we add it to the current partition list and invoke DFS from the next index. If it's not, we skip the current index and move to the next.

To optimize the identification of palindromes, we use dynamic programming. A 2D table `f` is maintained where `f[i][j]` is `True` if the substring from index `i` to `j` is a palindrome. The elements in this table are filled in by checking if the two ends of the substring are equal and if the inside substring (from `i+1` to `j-1`) is also a palindrome.

The recursive DFS approach combined with dynamic programming allows us to efficiently compute all the ways to partition the string into substrings that are palindromes.

Solution Approach

The solution uses a DFS algorithm combined with dynamic programming to efficiently find all possible palindrome partitioning of the string `s`. Let's walk through the key parts of the implementation.

Firstly, the dynamic programming table `f` is prepared, which is a square matrix, where `f[i][j]` indicates whether the substring from `i` to `j` (inclusive) is a palindrome. This precomputation avoids redundant checks and optimizes the process of verifying palindromes during the DFS traversal.

Here's how the table `f` is populated:

- Initialize `f` as an `n x n` matrix filled with `True`, as every single character is a palindrome by itself.
- The table is then filled in a bottom-up manner. For substrings longer than one character (from `n-1` to `0` for `i` and from `i+1` to `n` for `j`), we check if the two ends are the same, `s[i] == s[j]`, and if the inside substring `f[i+1][j-1]` is a palindrome.

Once the table is prepared, we use the DFS approach to build the partitions:

- A recursive function `dfs` takes an index `i` as a parameter, indicating the current starting point of the substring being considered.
- The recursive function works as follows:
  1. If `i` equals the length of the string `n`, it means we reached the end of the string and thus, have a valid partition we can add to the answer list `ans`.
  2. For other cases, we check all possible end indices `j` (from `i` to `n-1`) and if `f[i][j]` is `True`, indicating a palindrome, we append this substring to the current partition list `t`.
  3. We then recursively call `dfs(j + 1)`, which will attempt to find palindrome partitions for the remaining substring.
  4. Afterwards, we must backtrack by removing the last substring added to `t` before moving on to the next index.

The DFS is initiated by calling `dfs(0)`, meaning it starts with an empty partition and it looks at the string from the very start. The result is accumulated in the list `ans`, which eventually contains all the possible palindrome partitionings.

Note that `ans` is a list of lists, where each sublist represents a distinct palindrome partitioning of `s`. The variable `t` represents the current state of the partition list under consideration at each step of the DFS.

By using a combination of DFS for enumeration, dynamic programming for palindrome checking, and backtracking for generating partitions, the solution efficiently enumerates all palindrome partitions of the string `s`.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the string `"aab"`.

Step 1: First, we prepare the dynamic programming table `f`. The length of the string `s` is `3`, so our table will be a 3×3 matrix. Initially, `f[i][i]` (all positions where `i == j`) are set to `True` since all characters are palindromes by themselves.

```
f = [
  [True, False, False],
  [False, True, False],
  [False, False, True]
]
```

Step 2: Now we fill the remaining `f[i][j]` values where `i != j`. This involves checking substrings of length 2 and above.

- For `i = 1` and `j = 2` ("aa"), `s[i]` is equal to `s[j]` and the inside substring (which is empty) is "trivially" a palindrome. So, `f[1][2]` is set to `True`.
- For `i = 0` and `j = 1` ("ab"), `s[i]` is **not** equal to `s[j]`, so `f[0][1]` remains `False`.
- For `i = 0` and `j = 2` ("aab"), `s[i]` is **not** equal to `s[j]`, so `f[0][2]` remains `False`.

Our DP table now looks like this:

```
f = [
  [True, False, False],
  [False, True, True],  // "aa" is a palindrome
  [False, False, True]
]
```

Step 3: We start a DFS traversal to build palindrome partitions.

- We call `dfs(0)` and look for all palindromic substrings starting at index `0`.
- For `i = 0`, we consider substring `"a"` (`f[0][0]` is `True`), and so we add `["a"]` to our current partition `t` and call `dfs(1)`.
- Inside `dfs(1)`, we again find a palindromic substring `"a"` (`f[1][1]` is `True`), so we add `["a"]` to `t` and call `dfs(2)`.
- Finally, in `dfs(2)`, we add `"b"` to `t`, since `f[2][2]` is `True`, and reach the end of the string. We now have `["a", "a", "b"]` and add this partition to our answer list `ans`.
- We backtrack to `dfs(1)` and continue the loop for `j = 2`, now `"aa"` is a palindrome as `f[1][2]` is `True`, so we add `["aa"]` to `t` and call `dfs(3)`.
- In `dfs(3)`, we've reached the end of the string again, so we now have `["a", "aa"]` which is a valid partition that we add to `ans`.

The DFS finishes, and our answer `ans` contains all possible palindrome partitioning for the input string `"aab"`:

```
ans = [
  ["a", "a", "b"],
  ["aa", "b"]
]
```

We successfully obtained all partitions where each substring is a palindrome using a combination of DFS, dynamic programming, and backtracking.

Python Solution

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        def dfs(start_index: int):
            # Base case: if start_index reaches the end of the string,
            # add a copy of the current partition to the answer list
            if start_index == length:
                result.append(current_partition[:])  # Make a shallow copy
                return

            # Iterate over the substring starting from start_index
            for end_index in range(start_index, length):
                # If the substring s[start_index:end_index+1] is a palindrome,
                # we proceed to add it to the current partition and recurse
                if palindrome_flags[start_index][end_index]:
                    current_partition.append(s[start_index:end_index + 1])
                    dfs(end_index + 1)  # Recurse with the next starting index
                    current_partition.pop()  # Backtrack

        # Get the length of the input string
        length = len(s)

        # Initialize a 2D list to store palindrome flags
        palindrome_flags = [[True] * length for _ in range(length)]

        # Fill the palindrome_flags list in reverse order
        for i in range(length - 1, -1, -1):
            for j in range(i + 1, length):
                # A substring is a palindrome if the first and last characters are the same,
                # and if the substring between them is also a palindrome
                palindrome_flags[i][j] = s[i] == s[j] and palindrome_flags[i + 1][j - 1]

        # Initialize the result list and the list to store the current partition
        result = []
        current_partition = []

        # Start the DFS traversal from index 0
        dfs(0)

        return result

```

Java Solution

```java
class Solution {
    private int stringLength;
    private String inputString;
    private boolean[][] palindromeTable;
    private List<String> currentPartition = new ArrayList<>();
    private List<List<String>> allPartitions = new ArrayList<>();

    public List<List<String>> partition(String s) {
        stringLength = s.length();
        inputString = s;
        palindromeTable = new boolean[stringLength][stringLength];

        // Initialize the palindrome table with true for all entries
        for (int i = 0; i < stringLength; ++i) {
            Arrays.fill(palindromeTable[i], true);
        }

        // Populate the palindrome table with actual palindrome information
        for (int i = stringLength - 1; i >= 0; --i) {
            for (int j = i + 1; j < stringLength; ++j) {
                palindromeTable[i][j] = (s.charAt(i) == s.charAt(j)) && palindromeTable[i + 1][j - 1];
            }
        }

        // Start the depth-first search from the beginning of the string
        performDfs(0);
        return allPartitions;
    }

    private void performDfs(int startIndex) {
        // If the current start index reaches the end of the string, we've found a complete partition
        if (startIndex == inputString.length()) {
            allPartitions.add(new ArrayList<>(currentPartition));
            return;
        }

        // Explore further partitions starting from the current index
        for (int endIndex = startIndex; endIndex < stringLength; ++endIndex) {
            // If the substring starting at startIndex and ending at endIndex is a palindrome
            if (palindromeTable[startIndex][endIndex]) {
                // Add the palindrome substring to the current partition
                currentPartition.add(inputString.substring(startIndex, endIndex + 1));

                // Continue searching for palindromes from the next index after the current palindrome
                performDfs(endIndex + 1);

                // Backtrack and remove the last added palindrome from the current partition
                currentPartition.remove(currentPartition.size() - 1);
            }
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <cstring>
#include <functional>

class Solution {
public:
    std::vector<std::vector<std::string>> partition(std::string s) {
        int length = s.size();

        // Create a table to record if the substring s[i..j] is a palindrome.
        bool palindromeTable[length][length];
        memset(palindromeTable, true, sizeof(palindromeTable));

        // Fill the palindromeTable using dynamic programming.
        for (int i = length - 1; i >= 0; --i) {
            for (int j = i + 1; j < length; ++j) {
                // A substring is a palindrome if its outer characters are equal
                // and if its inner substring is also a palindrome.
                palindromeTable[i][j] = (s[i] == s[j]) && palindromeTable[i + 1][j - 1];
            }
        }

        // Create a vector to store all palindrome partitioning results.
        std::vector<std::vector<std::string>> result;

        // Temporary vector to store current partitioning.
        std::vector<std::string> tempPartition;

        // Recursively search for palindrome partitions starting from index 0.
        std::function<void(int)> depthFirstSearch = [&](int start) {
            // If we've reached the end of the string, add the current partitioning to results.
            if (start == length) {
                result.push_back(tempPartition);
                return;
            }

            // Explore all possible partitionings.
            for (int end = start; end < length; ++end) {
                // If the substring starting from 'start' to 'end' is a palindrome
                if (palindromeTable[start][end]) {
                    // Push the current palindrome substring to the temporary partitioning.
                    tempPartition.push_back(s.substr(start, end - start + 1));

                    // Move to the next part of the string.
                    depthFirstSearch(end + 1);

                    // Backtrack to explore other partitioning possibilities.
                    tempPartition.pop_back();
                }
            }
        };

        // Start the depth-first search from the first character.
        depthFirstSearch(0);

        // Return all the palindrome partitioning found.
        return result;
    }
};

```

Typescript Solution

```typescript
function partition(s: string): string[][] {
  const length = s.length;
  // 'isValidPalindrome' table to keep track of valid palindrome substrings.
  const isValidPalindrome: boolean[][] = new Array(length)
    .fill(0)
    .map(() => new Array(length).fill(true));

  // Fill the table with the correct values, bottom-up manner.
  for (let start = length - 1; start >= 0; --start) {
    for (let end = start + 1; end < length; ++end) {
      isValidPalindrome[start][end] = (s[start] === s[end]) && isValidPalindrome[start + 1][end - 1];
    }
  }

  // This will hold all possible palindrome partitions.
  const allPartitions: string[][] = [];
  // 'currentPartition' temporarily stores one possible partition.
  const currentPartition: string[] = [];

  // Helper function to perform depth-first search for partitions.
  const dfs = (index: number) => {
    // If the entire string has been processed, save the current partition.
    if (index === length) {
      allPartitions.push(currentPartition.slice());
      return;
    }
    // Explore further partitions.
    for (let end = index; end < length; ++end) {
      // Check if the substring is a valid palindrome
      if (isValidPalindrome[index][end]) {
        // Add the palindrome substring to the current partition.
        currentPartition.push(s.slice(index, end + 1));
        // Recursively check for further palindromes from the end of the current substring.
        dfs(end + 1);
        // Backtrack to explore other possible partitions by removing the last added palindrome.
        currentPartition.pop();
      }
    }
  };

  // Start the depth-first search from the beginning of the string.
  dfs(0);
  // Return all possible palindrome partitions found.
  return allPartitions;
}

```

Time and Space Complexity

Time Complexity

The given code first prepares a 2D boolean array `f`, which is used to determine if a substring `s[i:j]` is a palindrome. This preparation takes O(n^2) time because it gradually shrinks the substring window from both sides and each cell in `f` is filled based on the result of palindrome checks on previous substrings.

The main part of the algorithm uses depth-first search (DFS) to build all possible palindrome partitions of the string. Every recursive call of `dfs` goes through the string to find palindromes starting from the current index `i` up to the end of the string. In the worst case scenario, each character could be the start of a palindrome substring, leading to a situation where the number of recursive calls is proportional to the Catalan numbers because each step can involve a choice of different ending indices for the palindrome. The nth Catalan number is given by the formula `(2*n)! / ((n+1)!*n!)` which is approximately `O(4^n / (n^(3/2)))`. This determines the number of possible partitions in the worst case and thus the recursive calls made.

Given that the length of the input string is `n`, the time complexity is O(n^2) for the initial palindrome check array preparation plus the time complexity of the DFS traversal, which is O(4^n / (n^(3/2))). Hence, the overall worst-case time complexity is `O(n^2 + 4^n / (n^(3/2)))`.

Space Complexity

The space complexity is influenced by two factors: the space taken by the 2D array `f` and the space used by the recursion call stack.

- The 2D array `f` has size `n^2`, hence it uses O(n^2) space.
- The depth of the recursion stack is at most `n` (the size of the string `s`), because that's the deepest it can go (each character can be a separate palindrome). Furthermore, at each level of recursion a string is potentially added to the current partition `t`. Thus, we need to account for the space taken by the list of current palindromes `t`, which in the worst case can store up to `n` strings.

Thus, the space complexity of the recursive calls and the space to store the current partitions is O(n).

However, we also have an `ans` list that stores all the possible palindrome partitions. In the worst case, there can be exponential (Catalan number) many partitions, and the list of partitions can grow very large, significantly larger than the stack depth.

Combining these factors, the space complexity is `O(n^2)` for the 2D array and `O(4^n / (n^(3/2)))` for the partitions, which gives us the overall space complexity of `O(n^2 + 4^n / (n^(3/2)))`.