5. Longest Palindromic Substring

Medium String Dynamic Programming

Problem Description

The goal of this problem is to find the longest palindromic substring within a given string `s`. A palindromic string is a string that reads the same backward as forward, such as 'radar' or 'level'.

To understand the problem, let's consider what makes up a palindrome:

- A single character is always a palindrome.
- Two characters are a palindrome if they are identical.
- A substring of three or more characters is a palindrome if its first and last characters are the same, and the substring obtained by removing them is also a palindrome.

Given these observations, we need an algorithm that can check for palindromic substrings efficiently and keep track of the longest one found so far.

Intuition

The solution involves Dynamic Programming (DP), an optimization technique that solves complex problems by breaking them into simpler subproblems, storing the solution to each subproblem, and reusing those solutions.

Solution 1: Dynamic Programming The idea is to use a 2D table `dp` to store whether a substring `s[i..j]` is a palindrome. We fill this table in a bottom-up manner. For every substring length (from 2 to n), we set `dp[i][j]` to `true` if the corresponding substring is a palindrome. Here's the process:

- For substrings of length 1, each is trivially a palindrome.
- For substrings of length 2, they are palindromes if both characters are the same.
- For longer substrings, we check if the first and last characters are the same and if the substring obtained by removing them (`dp[i+1][j-1]`) is a palindrome.

If `dp[i][j]` is `true`, we check if it is the longest palindrome so far. If it is, we update the starting index and maximum length of the palindrome.

Solution 2: Enumerating the Palindrome Center An alternative approach is to consider each possible center of the palindrome (which could be a character or between two characters), and expand outwards from the center to see how far the palindrome can be extended. We keep track of the length and starting point of the longest palindrome found during the process.

Implementing the DP Solution The provided Python code uses the dynamic programming approach. Here's a breakdown of its parts:

- It initializes the `dp` table (`f` in the code) to all `True` for the length 1 substrates, and then iterates backwards over the string.
- For each pair `(i, j)` it checks if `s[i]` matches `s[j]`, then it sets `f[i][j]` to whatever the value at `f[i+1][j-1]` is, effectively checking if removing the matching characters still leaves a palindrome.
- It keeps track of the start position `k` and the max length `mx` of the longest palindrome.

The time complexity of this approach is $O(n^2)$ because it examines all possible substrings, making it practical for reasonably short strings.

Solution Approach

The solution to finding the longest palindromic substring employs two main algorithms - Dynamic Programming and Center Expansion. Below is a walkthrough for both.

Dynamic Programming Solution The dynamic programming approach creates a table `dp` where each entry `dp[i][j]` records whether the substring `s[i..j]` is a palindrome.

- Initialize a `n` by `n` matrix `dp` with `True` values along the diagonal, indicating that all single characters are palindromes.
- Iterate over all possible substring lengths `L` starting from 2 to the length of the input string `n`.
- For each length `L`, iterate over all possible starting indices `i` from which a substring of length `L` can be obtained.
- Use the ending index `j = i+L-1` to cover the substring of length `L` starting at index `i`.
- Set `dp[i][j]` to `True` if and only if the end characters match (`s[i] == s[j]`) and the internal substring `s[i+1..j-1]` is a palindrome (`dp[i+1][j-1] == True`).
- Track the starting index and maximum length of the longest palindromic substring found so far.

```python
dp = [[False] * n for _ in range(n)]
for i in range(n):
    dp[i][i] = True  # Base case for one-letter palindrome

mx_len = 1
start = 0

for L in range(2, n + 1):
    for i in range(n - L + 1):
        j = i + L - 1
        if L == 2:
            dp[i][j] = (s[i] == s[j])
        else:
            dp[i][j] = (s[i] == s[j] and dp[i + 1][j - 1])

        if dp[i][j] and L > mx_len:
            mx_len = L
            start = i
```

- The time complexity of this approach is `O(n^2)` as we check all `n(n-1)/2` substrings and updating the `dp` table takes constant time.

In the code, a 2D list `f` represents the `dp` table, where we fill the table starting from the second to last row to the first (`i = n - 2` to `0`) and from left to right (`j = i + 1` to `n`) within each row. This is because `dp[i][j]` depends on `dp[i + 1][j - 1]`, which is the next diagonal element below it.

```python
f = [[True] * n for _ in range(n)]
k, mx = 0, 1
for i in range(n - 2, -1, -1):
    for j in range(i + 1, n):
        if s[i] != s[j]:
            f[i][j] = False
        else:
            if j - i == 1 or f[i + 1][j - 1]:
                f[i][j] = True
                if mx < j - i + 1:
                    k, mx = i, j - i + 1
        else:
            f[i][j] = False
```

Here, the variable `k` tracks the starting index of the longest palindrome, and `mx` tracks the length of the longest palindrome.

Center Expansion Solution The center expansion algorithm doesn't require any additional space except for keeping track of the longest palindrome found.

```python
mx_len = 1
start = 0

for i in range(n):
    # Odd Length Palindromes
    odd_len = expandFromCenter(s, n, i, i)
    if odd_len > mx_len:
        mx_len = odd_len
        start = i - mx_len // 2

    # Even Length Palindromes
    even_len = expandFromCenter(s, n, i, i + 1)
    if even_len > mx_len:
        mx_len = even_len
        start = i - (mx_len - 1) // 2

def expandFromCenter(s, n, l, r):
    while l >= 0 and r < n and s[l] == s[r]:
        l -= 1
        r += 1
    return r - l - 1
```

Here, the `expandFromCenter` function checks for the longest palindrome centered at `l` (for odd lengths) and between `l` and `r` (for even lengths), expanding its boundaries outwards as long as the characters match. The lengths of the longest palindromes for both odd and even centers are compared to update the maximum palindrome length `mx_len` and the starting index `start`.

The time complexity for this approach is also `O(n^2)` because each expansion may take `O(n)` time and there are `O(n)` potential centers to consider. However, this approach does not use extra space, making it more space-efficient than the dynamic programming approach.

Both approaches are efficient for different scenarios and can be used depending on the space-time trade-off that is more critical for the application at hand.

Example Walkthrough

Let's consider a small example to illustrate the solution approach, specifically the dynamic programming solution.

Suppose the input string is `s = "babad"`. We are interested in finding the longest palindromic substring.

Following the dynamic programming strategy:

1. Initialize the `dp` table for a string of length 5 (since `"babad"` is 5 characters long). Each cell `dp[i][i]` along the diagonal is set to `True`, representing that every single character is a palindrome by itself. Our `dp` table initially looks something like this (where `0` indicates `False` and diagonals are implicitly `True`):

b	a	b	a	d
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

2. We check two-character substrings (length `L = 2`) for palindromicity. We notice that "ba" is not a palindrome, "ab" is not a palindrome, "ba" again is not a palindrome, but "ad" is not a palindrome either. So, our `dp` table does not change.

3. Now we move on to substrings of length 3. We find "bab" is a palindrome and so is "aba". The table updates to:

b	a	b	a	d
1	0	1	0	0
0	1	0	1	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

4. We then proceed to substrings of length 4 and 5 but find no longer palindromes.

5. The dynamic programming algorithm records the longest palindrome we found, which is "bab" starting at index 0 or "aba" starting at index 1, both with a length of 3.

Here is a snippet of Python code that reflects the process for this example:

```python
s = "babad"
n = len(s)
dp = [[False] * n for _ in range(n)]
for i in range(n):
    dp[i][i] = True
mx_len = 1
start = 0

for L in range(2, n + 1):
    for i in range(n - L + 1):
        j = i + L - 1
        if L == 2:
            dp[i][j] = (s[i] == s[j])
        else:
            dp[i][j] = (s[i] == s[j] and dp[i + 1][j - 1])
        if dp[i][j] and L > mx_len:
            mx_len = L
            start = i
longest_palindrome = s[start:start+mx_len]
print(f'The longest palindrome in the string is: "{longest_palindrome}"')
```

When this code is executed, the longest palindrome "bab" or "aba" (whichever comes first in the updating process) will be printed. The 2D `dp` matrix serves as a memory that helps avoid recalculating whether a substring is a palindrome, thus saving computational time. Each step relies on the information gathered in the previous steps, making this a bottom-up dynamic programming approach.

Python Solution

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)  # Length of the string

        # Table to store the palindrome status
        # dp[i][j] will be 'True' if the string from index i to j is a palindrome.
        dp = [[True] * n for _ in range(n)]

        start_index = 0  # Start index of the longest palindrome found
        max_length = 1   # Length of the longest palindrome found, initially 1 character

        # Bottom-up dynamic programming approach.
        # Start from the end of the string and move towards the beginning.
        for i in range(n - 2, -1, -1):  # i is the start index of the substring
            for j in range(i + 1, n):  # j is the end index of the substring
                dp[i][j] = False  # Initially set to False

                # Check if the current substring is a palindrome
                if s[i] == s[j] and dp[i + 1][j - 1]:
                    dp[i][j] = True  # Update the palindrome status
                    # Check if the current palindrome is the longest found so far
                    if max_length < j - i + 1:
                        start_index = i
                        max_length = j - i + 1  # Update max_length

        # Return the longest palindrome substring
        return s[start_index : start_index + max_length]

```

Java Solution

```java
class Solution {
    public String longestPalindrome(String s) {
        int n = s.length(); // Get the length of the string.
        boolean[][] dp = new boolean[n][n]; // Create a dynamic programming (DP) table.

        // Initialize all substrings of length 1 (single character) as a palindrome.
        for (boolean[] row : dp) {
            Arrays.fill(row, true);
        }

        int startIdx = 0; // Starting index of the longest palindromic substring found.
        int maxLength = 1; // Length of the longest palindromic substring found, initialized with length 1.

        // Build the DP table in a bottom-up manner.
        for (int i = n - 2; i >= 0; --i) { // Start from the second last character and move backwards.
            for (int j = i + 1; j < n; ++j) { // Compare it with characters ahead of it.
                dp[i][j] = false; // Initialize the current substring (i, j) as not palindrome.
                if (s.charAt(i) == s.charAt(j)) { // If the characters match,
                    dp[i][j] = dp[i + 1][j - 1]; // Check if removing them gives a palindrome.
                    // Update the start position and max length if a larger palindrome is found.
                    if (dp[i][j] && maxLength < j - i + 1) {
                        maxLength = j - i + 1;
                        startIdx = i;
                    }
                }
            }
        }

        // Extract the longest palindromic substring from the string.
        return s.substring(startIdx, startIdx + maxLength);
    }
}

```

C++ Solution

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

class Solution {
public:
    // Finds the longest palindromic substring in a given string 's'
    std::string longestPalindrome(std::string s) {
        int n = s.size(); // Length of the given string

        // Create a 2D vector 'dp' to store palindrome information.
        // dp[i][j] will be true if the substring s[i..j] is a palindrome.
        std::vector<std::vector<bool>> dp(n, std::vector<bool>(n, false));

        int start_index = 0; // Starting index of the longest palindrome found
        int max_length = 1;  // Length of the longest palindrome found, at least 1 because single characters are palindromes.

        // Initialize one-character and two-character palindromes
        for (int i = 0; i < n; ++i) {
            dp[i][i] = true; // Substrings of length 1 are palindromes.
            if (i < n - 1 && s[i] == s[i + 1]) {
                dp[i][i + 1] = true; // Substrings of length 2 are palindromes if both characters are equal.
                start_index = i; // Update the starting index of the longest palindrome
                max_length = 2;  // Update the max_length to 2
            }
        }

        // Check for palindromes of length 3 and more
        for (int len = 3; len <= n; ++len) {
            for (int i = 0; i < n - len + 1; ++i) {
                int j = i + len - 1; // Calculate the end index of the current substring

                // Check if the current substring is a palindrome using previously calculated sub-problems
                if(s[i] == s[j] && dp[i + 1][j - 1]) {
                    dp[i][j] = true; // Mark as palindrome

                    // Update max_length and start_index if a bigger palindrome is found
                    if(len > max_length) {
                        start_index = i;
                        max_length = len;
                    }
                }
            }
        }

        // Extract and return the longest palindrome substring from the original string
        return s.substr(start_index, max_length);
    }
};

```

Typescript Solution

```typescript
function longestPalindrome(s: string): string {
    // Get the length of the string.
    const length = s.length;
    // Initialize a 2D array f to track palindromes, initializing all entries to true.
    const isPalindrome: boolean[][] = Array(length)
        .fill(0)
        .map(() => Array(length).fill(true));

    // Initialize a variable to store the starting index of the longest palindrome found.
    let startIndex = 0;
    // Initialize a variable to store the length of the longest palindrome found.
    let maxLength = 1;

    // Loop through the string from end to beginning, adjusting the isPalindrome matrix.
    for (let i = length - 2; i >= 0; --i) {
        for (let j = i + 1; j < length; ++j) {
            // Invalidate the current state since it's not a single character string.
            isPalindrome[i][j] = false;
            // If the characters at i and j are the same, check the inner substring.
            if (s[i] === s[j]) {
                // Set the state based on the substring inside the current bounds.
                isPalindrome[i][j] = isPalindrome[i + 1][j - 1];
                // If the substring is a palindrome and larger than the current maxLength, update maxLength and startIndex.
                if (isPalindrome[i][j] && maxLength < j - i + 1) {
                    maxLength = j - i + 1;
                    startIndex = i;
                }
            }
        }
    }

    // Get the longest palindrome substring from the given string based on startIndex and maxLength.
    return s.slice(startIndex, startIndex + maxLength);
}

```

Time and Space Complexity

The time complexity of the code provided is `O(n^2)`, as it includes two nested loops that both iterate over the string length `n`. Specifically, the outer loop counts down from `n-2` to `0`, and the inner loop counts up from `i+1` to `n`, resulting in a quadratic number of steps in terms of the length of the input string `s`.

The space complexity, however, differs from the reference answer. The code creates a 2D list `f` sized `n` by `n`, which is used to store boolean values representing whether a substring is a palindrome or not. This means the space complexity is not `O(1)`, but in fact `O(n^2)` as well, because storing these values requires a space proportional to the square of the length of the input string `s`.