# 647. Palindromic Substrings

Medium String Dynamic Programming

Leetcode Link

## Problem Description

The problem requires us to determine the number of palindromic substrings in a given string `s`. A palindromic substring is defined as a substring that reads the same backward as forward. Substrings are contiguous sequences of characters that are formed from the string. For example, in the string "abc", "a", "b", "c", "ab", "bc", and "abc" are all substrings. The solution to the problem should count all such substrings that are palindromes.

## Intuition

The intuition behind the solution lies in recognizing that a palindromic substring can be centered around either a single character or a pair of identical characters. With this understanding, we can extend outwards from each possible center and check for palindrome properties.

The solution uses an algorithm known as "Manacher's Algorithm", which is efficient for finding palindromic substrings. The key idea is to avoid redundant computations by storing and reusing information about previously identified palindromes.

### Steps underlying Manacher's algorithm:

1. **Transform the String**: Insert a dummy character (say `#`) between each pair of characters in the original string (and at the beginning and end). This transformation helps us handle even-length palindromes uniformly with odd-length palindromes. The insertion of '^' at the beginning and '$' at the end prevents index out-of-bound errors.
2. **Array Initialization**: Create an array `p` that will hold the lengths of the palindromes centered at each character of the transformed string.
3. **Main Loop**: Iterate through each character of the transformed string, treating it as the center of potential palindromes. For each center, expand outwards as long as the substring is a palindrome.
4. **Center and Right Boundary**: Keep track of the rightmost boundary (`maxRight`) of any palindrome we've found and the center of that palindrome (`pos`). This is used to optimize the algorithm by reusing computations for palindromes that overlap with this rightmost palindrome.
5. **Update and count**: During each iteration, update the `p` array with the new palindrome radius and calculate the number of unique palindromes up to that point. Due to the transformation of the string, divide the radius by 2 to get the count of actual palindromes in the original string.
6. **Result**: The sum of the values in `p` divided by 2 gives us the total count of palindromic substrings in the original string.

The efficiency of Manacher's algorithm comes from the fact that it cleverly reuses previously computed palindromic lengths and thus avoids re-computing the length of a palindrome when each new centered palindrome is considered. It runs in linear time relative to the length of the transformed string, making it much more efficient than a naive approach.

## Solution Approach

The implementation of the solution using Manacher's algorithm follows the underlying steps:

1. **String Transformation**: The given string `s` is first transformed into `t` by inserting `'#'` between each character and adding a beginning marker `'^'` and an end marker `'$'`. This transformation ensures that we consider both even and odd-length palindromes by making every palindrome center around a character in `t`.
2. **Initialize Variables**: Three main variables are initiated:
   - `pos`: The center of the palindrome that reaches furthest to the right.
   - `maxRight`: The right boundary of this palindrome.
   - `ans`: The total count of palindromic substrings (to be computed).
3. **Array `p` for Palindrome Radii**: An array `p` is created with the same length as the transformed string `t`. Each element `p[i]` will keep track of the maximum half-length of the palindrome centered at `t[i]`.
4. **Iterate Over `t`**: We iterate over each character in the transformed string (excluding the first and last characters which are the markers), checking for palindromic substrings.
5. **Calculate Current Palindrome Radius**:
   - If the current index `i` is within the bounds of a known palindrome, we initialize `p[i]` to the minimum of `maxRight - i` and the mirror palindrome length, `p[2 * pos - i]`.
   - If `i` is beyond `maxRight`, we start with a potential palindrome radius of 1.
6. **Expand Around Centers**:
   - We expand around the current center `i` by incrementing `p[i]` as long as characters at `i + p[i]` and `i - p[i]` are equal (palindromic condition).
   - This loop helps in finding the maximum palindrome centered at `i`.
7. **Update Right Boundary and Center (`pos`)**:
   - If the current palindrome goes past the `maxRight`, update `maxRight` and `pos` to the new values based on the current center `i` and its palindrome radius `p[i]`.
8. **Counting Substrings**:
   - The palindrome length for the center `i` divided by 2 is added to `ans`. This accounts for the fact that only half the length of palindromes in the transformed string is relevant due to the inserted characters (`'#'`).
9. **Return Result**: Finally, the variable `ans` holds the total count of palindromic substrings in the original string, which is then returned.

In summary, the solution effectively leverages Manacher's algorithm for the palindrome counting problem. It transforms the input string to facilitate the uniform handling of even and odd length palindromes, utilizes dynamic programming through array `p` to store palindrome lengths, and optimizes by remembering the furthest reaching palindrome to avoid redundant checks.

### Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the string `s = "abba"`.

1. **String Transformation**: Inserting `'#'` between each character of `s` and adding `'^'` at the beginning and `'$'` at the end, the transformed string `t` becomes: `^#a#b#b#a#$`.
2. **Initialize Variables**:
   - `pos` is set to 0.
   - `maxRight` is set to 0.
   - `ans` is initialized as 0, which will hold the count of palindromic substrings.
3. **Array `p` for Palindrome Radii**: We create an array `p` with length equal to `t`, which is 11. Initially, `p` is filled with zeros: `p = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`.
4. **Iterate Over `t`**: We ignore the first and last characters (`^` and `$`) and start iterating from the second character (index 1) to the second to last (index 9).
5. **Calculate Current Palindrome Radius**:
   - For the first character `#` after `^`, since `maxRight` is 0, we start with `p[1] = 1`.
   - Next, we examine `a`. Since `maxRight` is still 0, we also start `p[2] = 1`.
6. **Expand Around Centers**:
   - The palindrome at the second character `#` can't be expanded, so `p[1]` remains 1.
   - For `a` at index 3, we can expand one step outward to check the characters around it (`#` on both sides), and it's still a palindrome. So `p[3] = 2`.
7. **Update Right Boundary and Center (`pos`)**:
   - For `a` at index 2, since `2 + p[1]` (2 + 2) is greater than `maxRight` (0), we update:
     - `maxRight = 4`
     - `pos = 2`.
8. **Counting Substrings**:
   - After updating `maxRight` and `pos` for `a`, we add `p[2] / 2` to `ans`, which means we add 1 to `ans`.
   - Proceeding with the same logic, for `a` at index 3 and `b` at index 4, we'll find that they form a longer palindrome of length 4 (since `abba` is a palindrome). So `p[3] = 3`, and `p[4] = 1`, and our `ans` will be updated accordingly.
9. **Return Result**: Continuing this process for all characters in `t` and summing up `p[i] / 2` for each character, we get the total count of palindromic substrings in the original string `s` as `ans`.

For `s = "abba"`, the final result is 6 which represents the palindromic substrings: `a`, `b`, `b`, `a`, `bb`, and `abba`.

## Python Solution

```python
class Solution:
    def countSubstrings(self, s: str) -> int:
        # Preprocess the string to insert '#' between characters and '^', '$' at the ends.
        # This helps to handle palindrome of even length properly.
        transformed_string = '^#'.join('#'.join(s)) + '$'
        n = len(transformed_string)

        # Array to store the length of the palindrome centered at each character in T.
        palindrome_lengths = [0] * n

        # Current center, and the right boundary of the rightmost palindrome.
        center, right_boundary = 0, 0

        # Accumulator for the count of palindromic substrings.
        palindromic_substring_count = 0

        # Loop through the transformed string starting from the first character after '^'
        # and ending at the last character before '$'.
        for i in range(1, n - 1):
            # If the current position is within the rightmost palindrome, we can use
            # previously calculated data (palindrome_lengths[2 * center - i]).
            # to determine the minimum length for the palindrome centered at i.
            if right_boundary > i:
                palindrome_lengths[i] = min(right_boundary - i, palindrome_lengths[2 * center - i])
            else:
                palindrome_lengths[i] = 1  # Start with a palindrome of length 1 (a single character).

            # Expand around the center i, and check for palindromes.
            while transformed_string[i + palindrome_lengths[i]] == transformed_string[i - palindrome_lengths[i]]:
                palindrome_lengths[i] += 1

            # Update the rightmost palindrome's boundary and center position
            # if the palindrome centered at i expands past it.
            if i + palindrome_lengths[i] > right_boundary:
                right_boundary = i + palindrome_lengths[i]
                center = i

            # Increment the count of palindromic substrings.
            # We divide by 2 because we inserted '#' and need the actual length.
            palindromic_substring_count += palindrome_lengths[i] // 2

        return palindromic_substring_count
```

## Java Solution

```java
class Solution {
    public int countSubstrings(String s) {
        // Initialize a StringBuilder to hold the transformed string with boundary characters
        StringBuilder transformedString = new StringBuilder("^");
        // Transform the input string by inserting '#' between characters
        // This is done to handle both even and odd length palindrome uniformly
        for (char ch : s.toCharArray()) {
            transformedString.append('#').append(ch);
        }
        // Finalize the transformed string by adding a '$' at the end
        // This prevents index out of range during the while loop checks
        transformedString.append('#').append('$');
        String modifiedStr = transformedString.toString();

        int length = modifiedStr.length();
        // Array to hold the length of the palindrome at each index
        int[] palindromeLengths = new int[length];

        // Initialize center and right boundary of the current longest palindrome
        int center = 0;
        int rightBoundary = 0;
        // Variable to store the total count of palindromes
        int countPalindromes = 0;

        // Loop through the string starting from the first character index to the second last
        for (int i = 1; i < length - 1; ++i) {
            // Determine the mirror index of the current index 'i'
            int mirror = 2 * center - i;

            // Check if the current right boundary exceeds the current index
            // If so, pick the minimum between the mirrored palindrome's length and the distance to the right boundary
            palindromeLengths[i] = rightBoundary > i ? Math.min(rightBoundary - i, palindromeLengths[mirror]) : 1;

            // Expand around the center to determine the length of palindrome at index 'i'
            while (modifiedStr.charAt(i + palindromeLengths[i]) == modifiedStr.charAt(i - palindromeLengths[i])) {
                palindromeLengths[i]++;
            }

            // If the palindrome expanded past the right boundary, update the center and right boundary
            if (i + palindromeLengths[i] > rightBoundary) {
                rightBoundary = i + palindromeLengths[i];
                center = i;
            }

            // Increment countPalindromes by half the length of palindromes at i (since we insert #'s)
            // This will effectively count each character in the original string once
            countPalindromes += palindromeLengths[i] / 2;
        }

        // Return the total count of palindromic substrings
        return countPalindromes;
    }
}
```

## C++ Solution

```cpp
class Solution {
public:
    int countSubstrings(string s) {
        int totalCount = 0; // Initialize count of palindromic substrings
        int length = s.size(); // Get the size of the input string

        // Iterate over the string, considering both odd and even length palindromes
        for (int center = 0; center < length * 2 - 1; ++center) {
            // For an odd-length palindrome, both i and j start at the same index.
            // For an even-length palindrome, j starts at the next index after i.
            int left = center / 2;
            int right = center / 2 + center % 2;

            // Expand around the center as long as the characters are the same
            while (left >= 0 && right < length && s[left] == s[right]) {
                ++totalCount; // Increment the count for the current palindrome
                --left;  // Move the left pointer backward
                ++right; // Move the right pointer forward
            }
        }
        return totalCount; // Return the total count of palindromic substrings
    }
};
```

## Typescript Solution

```typescript
/**
 * Counts the number of palindromic substrings in a given string.
 *
 * @param {string} str - The string to analyze.
 * @return {number} - The count of palindromic substrings.
 */
function countSubstrings(str: string): number {
    let count: number = 0;
    const length: number = str.length;

    // Iterate through each character position in the string, expanding potential palindromes.
    for (let center = 0; center < length * 2 - 1; ++center) {
        // Calculate the starting positions of potential palindromes.
        let left: number = center / 2;
        let right: number = (center + 1) >> 1;

        // Expand around the center while the characters match and we're within the bounds of the string.
        while (left >= 0 && right < length && str[left] === str[right]) {
            // Increment the palindrome count.
            ++count;
            // Move outwards to check the next characters.
            --left;
            ++right;
        }
    }
    return count;
}
```

## Time and Space Complexity

The given code implements the Manacher's algorithm for finding the total number of palindromic substrings in a given string `s`.

The time complexity of the Manacher's algorithm is $O(n)$, where `n` is the length of the transformed string `t`. The transformation involves adding separators and boundaries to the original string, effectively doubling its length and adding two extra characters; thus, the length of `t` is roughly `2 * len(s) + 3`. The main loop of the algorithm iterates through each character of the transformed string `t` once, and the inner `while` loop that checks for palindromes does not increase the overall time complexity since the expansion of `p[i]` can only be as large as the length of `t`.

As for the space complexity, the algorithm allocates an array `p` that is the same length as `t`, so the space complexity is also $O(n)$, where `n` is the length of the transformed string `t`.

Therefore, when we relate it back to the original string `s`, the time and space complexities can be considered $O(n)$ as well, with `n = len(s)`.