Manacher's Algorithm

While dealing with strings we may come across various problems related to palindromes where some of them may be finding the longest palindromic substring, finding the number of palindrome substrings in a string efficiently.
We will start by defining what are palindromes and then devise an algorithm known as **Manacher's Algorithm** that helps in solving the above problems and subproblems of some hard problems involving palindromes efficiently.

Palindrome String

A string is said to be a palindrome that reads the same in forward and backward directions i.e s = rev(s) where rev() is a function that reverses the string.
For Example: "racecar", "madam", "saas" are palindrome strings.

Properties of palindrome string:

- Given a palindrome string S, removing the first and last characters of the s also results in a palindrome string.
- Given a palindrome string S, adding the same character to the beginning and the end of the string also results in a palindrome string.

Center of palindrome string:
The center of a palindrome string is defined as the middle of the string which divides the string into two equal halves which are symmetrical about the center.
For Example:

1. In the palindrome string "racecar", being an **odd length palindrome, the** center of the string is the **character** 'e', which divides the string into two equal symmetrical halves "rac" and "car".
2. In the palindrome string "saas", being an **even length palindrome** we can define the center of the string as the **point** which divides the string into two equal halves i.e here the point/center is between the two a's which divides the string into two equal symmetrical halves "sa" and "as".

Now let us define a problem statement that helps us understand the above concepts and Manacher's algorithm easily.

Problem Statement

Problem Statement: Given a string **S**, we are interested in finding the length of the **longest palindromic substring** in **S**.

There exists a dynamic programming solution(which we have discussed previously) to the above problem whose time complexity is O(N^2), where N is the length of the string **S**.

Let us use the concept of centers we learned which will basically be a brute force solution to the problem.

Brute Force(Expanding around the center):
The idea of the algorithm is to look for all odd and even length palindrome strings and return the maximum length.

Algorithm:

1. For **odd length palindromes**: Taking each character **S[i]**, for every index 'i' as the center, we try to expand around **S[i]** and check if S[i-1] equals S[i+1] and accordingly update the length of the palindrome found so far.
2. For **even length palindrome:** Since for an even length palindrome the center of the string is a point as defined above, so for every pair of adjacent characters **S[i]** and **S[i+1]**, we check if S[i] equals S[i+1] as the minimum length of an even length palindrome is 2, here the center will be the point between S[i] and S[i+1] and then expand around these characters i.e we check if S[i-1] equals S[i+1] and accordingly update the length of the palindrome found so far.

An alternative approach:
Instead of checking for every pair of adjacent characters in case of even length palindromes, we can **modify** our given string **S** to a string **S'** by placing a **'#' before and after** every character of **S**.
For example:
S = "apple"
S' = "#a#p#p#l#e#"

Now we don't need to consider separate cases for odd and even length palindromes as the length of the resulting string will be **always odd**(2*x + 1, where x is the length of the original string) and hence we can apply the same step as defined for 'odd-length palindromes' i.e taking each character **S'[i]**, for every index 'i' as the center, we try to expand around **S'[i]** and check if S'[i-1] equals S'[i+1] which will cover both the cases.

It can be clearly observed that:
If the current character is '#': Expanding around this character as the center will always give even length palindromes, we can think of # as the "point" that acts as the center in an even length palindrome.
If the current character is a character of the original string S: Expanding around this character as the center will always give odd length palindromes.

Let the length of the palindrome found by 'L' in S', then the length of the palindrome in the original string S will be floor(L/2).

The following is a pseudo-code using the alternative approach:

Pseudocode:

```
/*
        The function takes the input string S and returns the modified string S'
*/
function modifyString(S)
        //  Initializing len as length of string S and declaring string S'
        string S', len = S.length

        for cur = 0 to len - 1
                S' = S' + '#'
                S' = S' + S[cur]

        S' = S' + '#'
        return S'

/*
        The function takes the input string S and returns the max length of the longest
        palindromic substring.
*/
function maxPalindromeSubstring(s)

        //  Calling modifyString() to get the modifies string S'
        S' = modifyString(S)
        len = S'.length

        //  Initializing the ans to 0 denoting the length of longest palindromic substring
        ans = 0

        for cur = 0 to len - 1
                /*
                        Taking the 'cur' as center, initializing the cur_len to 1 as a string of
                        length 1 is palindrome.
                */
                cur_len = 1
                //  Initializing the left and right pointers
                left = cur-1, right = cur+1

                //  Expanding around the center 'cur'
                while(left >= 0 and right < len and S'[left] == S'[right])
                        cur_len = cur_len + 2

                //  Updating the value of ans
                ans = max(ans, cur_len)
```

```
        return ans
```

Time Complexity: O(N^2), where N is the length of the original string S as we check for all palindromic substrings for each index ‘i’ as the center.