Manacher’s Algorithm

Manacher’s algorithm can find the length of the longest palindromic substring in O(N), where N is the length of the given string **S**. The drawback of the brute force approach was to check repeatedly for all possible palindrome strings with every index ‘i’ as the center, Manacher’s algorithm tries to reuse these results which **brings down the time complexity from O(N^2) to O(N)** rather than checking for all possible palindrome strings from starting.

For the sake of simplicity, let’s modify our string **S** to **S’** which will eventually take care of both the even and odd length palindromes similar to that of the brute force solution defined earlier.

Let us define an array **LPS**, where **LPS[i]** denotes the length of one of the halves of the palindrome string with center at index ‘i’ i.e LPS[i] will denote a palindrome substring from index **i - LPS[i] to i + LPS[i]**.

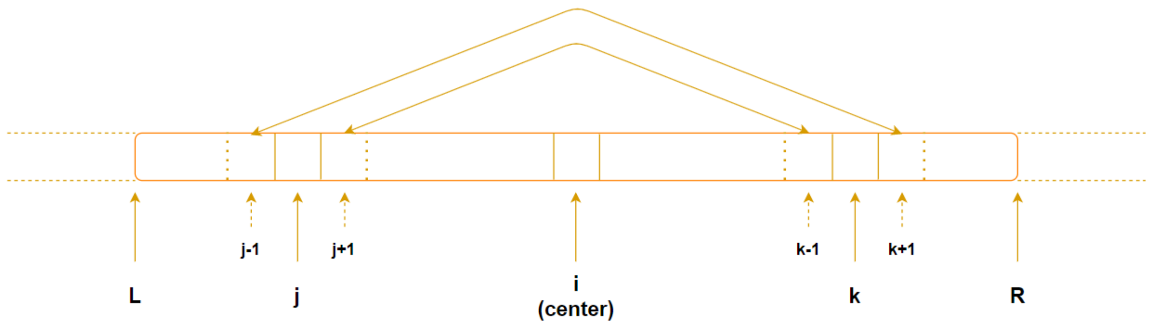
For Example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
String	#	a	#	b	#	b	#	a	#	a	#	c	#	a	#
LPS	0	1	0	1	4	1	0	1	2	1	0	3	0	1	0

In the above diagram, the original string S = “abbaaca”, so our modified string S’ = “#a#b#b#a#a#c#a#”.
LPS[4] = 4 denotes that we have a palindrome string from index (4 - 4) to (4 + 4) => 0 to 8, which also is the length(LPS[4]) of the palindrome in the original string S, with the center being the point between index 1 and index 2.

LPS[9] = 1 denotes that we have a palindrome string from 8 to 10.

How to calculate the LPS array efficiently?



As stated earlier Manacher’s algorithm tries to reuse the already calculated values, let us say we are at index **i(center)** and the value at index **i** be **LPS[i]** and the left and right borders of the palindromic substring be defined by **LPS[i]** where **L = i - LPS[i]** and **R = i + LPS[i]** and since we are at index **i**, we know the values of **LPS[j] ∀ j <= i.**

From the property of palindromes, we know that **S’[j] = S’[k]**, where j and k are indexes in the left and right halves with center at index **i** and equidistant from **L** and **R** respectively. In other words, if **L** and **R** are the left and right borders so **j - L = R - k**, hence **j = R - k + L**.

We know the value of **LPS[j]** as j <= i, we can use this value to determine the LPS value at index **k.**

Let us say the value at **LPS[j]** be **val, which** means the substring from **LPS[j]-val** to **LPS[j]+val** is a palindrome with center at **j**.
Let val = 1, so **S’[j-1] = S’[j+1]** and since the string from L to R is a palindrome, from the property of palindromes **S’[k-1] = S’[j-1]** and **S’[k+1] = S’[j+1] => S’[k-1] = S’[k+1]**, thus we can see that we are able to reuse the already calculated values of the LPS array.

So using the above approach we define two cases that will help us finding either the exact value of LPS[k] or a lower bound on LPS[k]:

- **Case 1:** If the value of **LPS[j]** does not exceed or touch the left border **L** i.e **LPS[j] < j - L** or the left border defined by LPS[j] which is equal to j-LPS[j] > L, then we can be assured that the value of LPS[k] will be equal to LPS[j], hence **LPS[k] = LPS[R - k + L]**.
- **Case 2:** If the value of **LPS[j]** exceeds or touches the left border **L** i.e **LPS[j] >= j - L**, then:
  We cannot say that LPS[k] will be equal to LPS[j] as if LPS[j] >= j - L, it denotes that we have a palindrome string with the characters present before the left border L having j as the center, but on the contrary, we have not explored the characters beyond the right border **R** which makes us unsure of the fact that there exist characters beyond R having k as the center which form a palindrome string.
  However, we can say that **LPS[k] >= (R - i) or (j - L)** as j - L = R - i and for knowing the right border R for which the string is a palindrome with the center as k, we start expanding our range as we did in brute force approach starting with checking the equality of the characters at indices {(k - LPS[k] - 1, R + 1), (k - LPS[k] - 2, R + 2)} and so on, with k being the center.

Note: We try to maximize the right border R as we calculate the LPS array as maximizing R will allow us to reuse and calculate the values of LPS array for larger indices efficiently.

Time Complexity
Observing Manacher’s algorithm makes it look like the time complexity will be O(N^2) similar to that of the brute force algorithm we discussed earlier, where N is the length of the string. However it is to be noted that the algorithm always tries to maximise the right border R and as we move iteratively through the string it either reuses the already computed values(R remains the same) or increases the right border R to calculate the value of LPS at a given index. Hence,

the time complexity is reduced from O(N^2) to **O(N)** as the right border always remains the same or increases to the right.

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

        //  Initializing the left and right borders L and R and declaring the LPS array
        L = 0, R = -1, LPS[len]

        //  Iterating over the current index to find LPS[cur]
        for cur = 0 to len - 1

                //  Declaring the length of palindrome for the current index cur
                pLen

                /*
                If cur > R, then the length pLen = 0 as we need to compute the
                LPS[cur] by expanding around the center cur.
            */
            if cur > R
                pLen = 0
            else
                /*
                    Getting the minimum of the LPS of the index in left half
                    and R - cur, so that we do not cross the boundary(Case 1
                    and Case 2)
                */
                left = L + R - cur
                pLen = min(LPS[left], R - cur)

            /*
                This expands around the center cur if L > R otherwise handles
                both cases: Case1 and Case 2 both as in Case1, the loop will simply
                terminate while in Case2 tries to expand beyond R.
            */
            while(cur - pLen - 1 >= 0 and cur + pLen + 1 < len and S'[cur-pLen-1] ==
            S'[cur+pLen+1])
                pLen = pLen + 1

            LPS[cur] = pLen

            //   Updating the value of ans
            ans = max(ans, LPS[cur])

            //   Checking if can get a larger R, as we want to maximize right border R
            if cur + pLen > R
                L = cur - pLen
                R = cur + pLen

        return ans
```

Time Complexity: O(N), where N is the length of the original string S, even if the modified string S’ has length 2*N + 1 so the overall complexity will still be O(N).