Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 3

Last Updated : 24 Mar, 2023

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.
We have seen that there are no new character comparison needed in case 1 and case 2. In case 3 and case 4, necessary new comparison are needed.
In following figure,

If at all we need a comparison, we will only compare actual characters, which are at "odd" positions like 1, 3, 5, 7, etc.
Even positions do not represent a character in string, so no comparison will be performed for even positions.
If two characters at different odd positions match, then they will increase LPS length by 2.

There are many ways to implement this depending on how even and odd positions are handled. One way would be to create a new string 1^{st} where we insert some unique character (say #, $ etc) in all even positions and then run algorithm on that (to avoid different way of even and odd position handling). Other way could be to work on given string itself but here even and odd positions should be handled appropriately.

Here we will start with given string itself. When there is a need of expansion and character comparison required, we will expand in left and right positions one by one. When odd position is found, comparison will be done and LPS Length will be incremented by ONE. When even position is found, no comparison done and LPS Length will be incremented by ONE (So overall, one odd and one even positions on both left and right side will increase LPS Length by TWO).

Implementation:

```cpp
// A C++ program to implement Manacher’s Algorithm
#include <bits/stdc++.h>
using namespace std;

void findLongestPalindromicString(string text)
{
    int N = text.length();
    if (N == 0)
        return;

    // Position count
    N = 2 * N + 1;

    // LPS Length Array
    int L[N];
    L[0] = 0;
    L[1] = 1;

    // centerPosition
    int C = 1;

    // centerRightPosition
    int R = 2;

    // currentRightPosition
    int i = 0;

    // currentLeftPosition
    int iMirror;
    int expand = -1;
    int diff = -1;
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;

    // Uncomment it to print LPS Length array
    // printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++) {
        // Get currentLeftPosition iMirror
        // for currentRightPosition i
        iMirror = 2 * C - i;

        // Reset expand - means no
        // expansion required
        expand = 0;
```

```
        diff = R - i;

        // If currentRightPosition i is
        // within centerRightPosition R
        if (diff >= 0) {

            // Case 1
            if (L[iMirror] < diff)
                L[i] = L[iMirror];

            // Case 2
            else if (L[iMirror] == diff && R == N - 1)
                L[i] = L[iMirror];

            // Case 3
            else if (L[iMirror] == diff && R < N - 1) {
                L[i] = L[iMirror];

                // Expansion required
                expand = 1;
            }

            // Case 4
            else if (L[iMirror] > diff) {
                L[i] = diff;

                // Expansion required
                expand = 1;
            }
        }
        else {
            L[i] = 0;

            // Expansion required
            expand = 1;
        }

        if (expand == 1) {

            // Attempt to expand palindrome centered
            // at currentRightPosition i. Here for odd
            // positions, we compare characters and
            // if match then increment LPS Length by ONE
            // If even position, we just increment LPS
            // by ONE without any character comparison

            while (((i + L[i]) < N && (i - L[i]) > 0)
```

```
                    && (((i + L[i] + 1) % 2 == 0)
                        || (text[(i + L[i] + 1) / 2]
                            == text[(i - L[i] - 1) / 2]))) {
                L[i]++;
            }
        }

        // Track maxLPSLength
        if (L[i] > maxLPSLength) {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        // If palindrome centered at
        // currentRightPosition i expand
        // beyond centerRightPosition R,
        // adjust centerPosition C based
        // on expanded palindrome.
        if (i + L[i] > R) {
            C = i;
            R = i + L[i];
        }

        // Uncomment it to print LPS Length array
        // System.out.print("%d ", L[i]);
    }

    start = (maxLPSCenterPosition - maxLPSLength) / 2;
    end = start + maxLPSLength - 1;

    // System.out.print("start: %d end: %d\n",
    //                  start, end);
    cout << "LPS of string is " << text << " : ";

    for (i = start; i <= end; i++)
        cout << text[i];
    cout << endl;
}

int main()
{

    string text1 = "babcbabcbaccba";
    findLongestPalindromicString(text1);

    string text2 = "abaaba";
    findLongestPalindromicString(text2);
```

```
    string text3 = "abababa";
    findLongestPalindromicString(text3);

    string text4 = "abcbabcbabcba";
    findLongestPalindromicString(text4);

    string text5 = "forgeeksskeegfor";
    findLongestPalindromicString(text5);

    string text6 = "caba";
    findLongestPalindromicString(text6);

    string text7 = "abacdfgdcaba";
    findLongestPalindromicString(text7);

    string text8 = "abacdfgdcabba";
    findLongestPalindromicString(text8);

    string text9 = "abacdedcaba";
    findLongestPalindromicString(text9);
    return 0;
}

// This code is contributed by Ishankhandelwals.
```

Output

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

Time Complexity: O(N)

Auxiliary Space: O(N)

This is the implementation based on the four cases discussed in Part 2. In Part 4, we have discussed a different way to look at these four cases and few other approaches.

Next Article
Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 4

Similar Reads

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 3

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.We have...

15+ min read

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 4

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. In Part 3, we implemented the same.Â Here...

12 min read

Manacher's Algorithm - Linear Time Longest Palindromic Substring - Part 1

Given a string, find the longest substring which is palindrome. if the given string is â€œforgeeksskeegforâ€ , the output should be â€œgeeksskeegâ€ if the given string is...

5 min read

Suffix Tree Application 6 - Longest Palindromic Substring

Given a string, find the longest substring which is palindrome.We have already discussed NaÃ¯ve [O(n3)], quadratic [O(n2)] and linear [O(n)] approaches in Set 1, Set 2 and Manacherâ€™s Algorithm.Â In this...

15+ min read

Print the longest palindromic prefix of a given string

Given a string str, the task is to find the longest palindromic prefix of the given string. Examples: Input: str = "abaac" Output: aba Explanation: The longest prefix of the given string which is palindromic is "aba"....

12 min read

Longest Palindromic Substring using Palindromic Tree | Set 3

Given a string, find the longest substring which is a palindrome. For example, if the given string is â€œforgeeksskeegforâ€ , the output should be â€œgeeksskeegâ€ . Prerequisite : Palindromic Tree |...

15+ min read

Largest palindromic number by permuting digits

Given N(very large), the task is to print the largest palindromic number obtained by permuting the digits of N. If it is not possible to make a palindromic number, then print an appropriate message. Examples :...

12 min read

Longest Palindromic Substring using hashing in O(nlogn)

Given a string S, The task is to find the longest substring which is a palindrome using hashing in O(N log N) time. Input: S: â€ forgeeksskeegforâ€ , Output: â€œgeeksskeegâ€ Input: S: â€ Geeksâ€ ,...

11 min read

Longest palindromic string possible after removal of a substring

Given a string str, the task is to find the longest palindromic string that can be obtained from it after removing a substring. Examples: Input: str = "abcdefghiedcba" Output: "abcdeiedcba" Explanation:...

11 min read

Rearrange string to obtain Longest Palindromic Substring

Given string str, the task is to rearrange the given string to obtain the longest palindromic substring. Examples: Input: str = â€œgeeksforgeeksâ€ Output: eegksfskgeeorExplanation: eegksfskgee is the...

9 min read