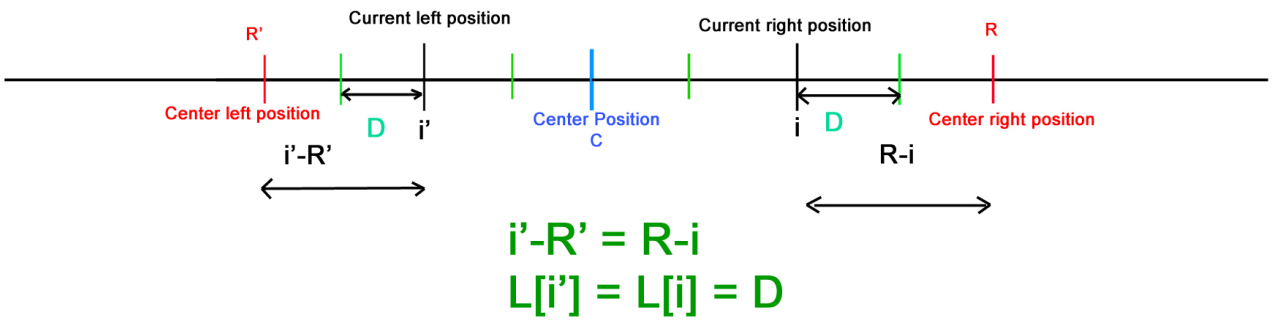
Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 4

Last Updated : 03 Apr, 2024

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.
Here we will review the four cases again and try to see it differently and implement the same.
All four cases depends on LPS length value at currentLeftPosition (L[iMirror]) and value of (centerRightPosition – currentRightPosition), i.e. (R – i). These two information are known before which helps us to reuse previous available information and avoid unnecessary character comparison.



*If we look at all four cases, we will see that we 1st set minimum of **L[iMirror]** and **R-i** to **L[i]** and then we try to expand the palindrome in whichever case it can expand.*
Above observation may look more intuitive, easier to understand and implement, given that one understands LPS length array, position, index, symmetry property etc.

Implementation:

C++ Java Python3 C# JavaScript

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>
```

```
char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -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;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //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]++;
        }

        if(L[i] > maxLPSLength) // Track 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
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

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)

Other Approaches:

We have discussed two approaches here. One in Part 3 and other in current article. In both approaches, we worked on given string. Here we had to handle even and odd positions differently while comparing characters for expansion (because even positions do not represent any character in string).

To avoid this different handling of even and odd positions, we need to make even positions also to represent some character (actually all even positions should represent SAME character because they MUST match while character comparison). One way to do this is to set some character at all even positions by modifying given string or create a new copy of given string. For example, if input string is “abcb”, new string should be “#a#b#c#b#” if we add # as unique character at even positions.

The two approaches discussed already can be modified a bit to work on modified string where different handling of even and odd positions will not be needed.

We may also add two DIFFERENT characters (not yet used anywhere in string at even and odd positions) at start and end of string as sentinels to avoid bound check. With these changes string “abcb” will look like “^#a#b#c#b#$” where ^ and $ are sentinels.

This implementation may look cleaner with the cost of more memory.

We are not implementing these here as it’s a simple change in given implementations.

Next Article
Ukkonen's Suffix Tree Construction - Part 2

Similar Reads

What is Pattern Searching ?

Pattern searching in Data Structures and Algorithms (DSA) is a fundamental concept that involves searching for a specific pattern or sequence of elements within a given data structure. This technique is...

5 min read

Introduction to Pattern Searching - Data Structure and Algorithm Tutorial

Pattern searching is an algorithm that involves searching for patterns such as strings, words, images, etc. We use certain algorithms to do the search process. The complexity of pattern searching varies from...

15+ min read

Naive algorithm for Pattern Searching

Given text string with length n and a pattern with length m, the task is to prints all occurrences of pattern in text. Note: You may assume that n > m. Examples:Â Input: Â text = "THIS IS A TEST TEXT", pattern =...

6 min read

Rabin-Karp Algorithm for Pattern Searching

Given a text T[0. . .n-1] and a pattern P[0. . .m-1], write a function search(char P[], char T[]) that prints all occurrences of P[] present in T[] using Rabin Karp algorithm. You may assume that n > m. Examples:...

15 min read

KMP Algorithm for Pattern Searching

Given two strings txt and pat, the task is to return all indices of occurrences of pat within txt. Examples: Input: txt = "abcab", pat = "ab"Output: [0, 3]Explanation: The string "ab" occurs twice in txt, first...

14 min read

Z algorithm (Linear time pattern searching Algorithm)

This algorithm efficiently locates all instances of a specific pattern within a text in linear time. If the length of the text is "n" and the length of the pattern is "m," then the total time taken is O(m + n), with a linear...

13 min read

Finite Automata algorithm for Pattern Searching

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.Examples: Input: txt[] = "THIS IS A TEST...

13 min read

Boyer Moore Algorithm for Pattern Searching

Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search...

15+ min read

Aho-Corasick Algorithm for Pattern Searching

Given an input text and an array of k words, arr[], find all occurrences of all words in the input text. Let n be the length of text and m be the total number of characters in all words, i.e. m = length(arr[0]) +...

15+ min read

ÂÂkasaiâ€™s Algorithm for Construction of LCP array from Suffix Array

Background Suffix Array : A suffix array is a sorted array of all suffixes of a given string. Let the given string be "banana". 0 banana 5 a1 anana Sort the Suffixes 3 ana2 nana ----------> 1 anana 3 ana...

15+ min read