Longest palindromic substring

From Wikipedia, the free encyclopedia

In computer science, the longest palindromic substring or longest symmetric factor problem is the problem of finding a maximum-length contiguous substring of a given string that is also a palindrome. For example, the longest palindromic substring of "bananas" is "anana". The longest palindromic substring is not guaranteed to be unique; for example, in the string "abracadabra", there is no palindromic substring with length greater than three, but there are two palindromic substrings with length three, namely, "aca" and "ada". In some applications it may be necessary to return all maximal palindromic substrings (that is, all substrings that are themselves palindromes and cannot be extended to larger palindromic substrings) rather than returning only one substring or returning the maximum length of a palindromic substring.

Manacher (1975) found a linear time algorithm for listing all the palindromes that appear at the start of a given string. However, as observed e.g., by Apostolico, Breslauer & Galil (1995), the same algorithm can also be used to find all maximal palindromic substrings anywhere within the input string, again in linear time. Therefore, it provides a linear time solution to the longest palindromic substring problem. Alternative linear time solutions were provided by Jeuring (1994), and by Gusfield (1997), who described a solution based on suffix trees. Efficient parallel algorithms are also known for the problem.[1]

The longest palindromic substring problem should not be confused with the different problem of finding the longest palindromic subsequence.

Contents

Manacher's algorithm

To find in linear time a longest palindrome in a string, an algorithm may take advantage of the following characteristics or observations about a palindrome and a sub-palindrome:

1. The left side of a palindrome is a mirror image of its right side.
2. (Case 1) A third palindrome whose center is within the right side of a first palindrome will have exactly the same length as that of a second palindrome anchored at the mirror center on the left side, if the second palindrome is within the bounds of the first palindrome by at least one character (not meeting the left bound of the first palindrome).
3. (Case 2) If the second palindrome meets or extends beyond the left bound of the first palindrome, then the third palindrome is guaranteed to have at least the length from its own center to the right outermost character of the first palindrome. This length is the

same from the center of the second palindrome to the left outermost character of the first palindrome.

4. To find the length of the third palindrome under Case 2, the next character after the right outermost character of the first palindrome would then be compared with its mirror character about the center of the third palindrome, until there is no match or no more characters to compare.
5. (Case 3) Neither the first nor second palindrome provides information to help determine the palindromic length of a fourth palindrome whose center is outside the right side of the first palindrome.
6. It is therefore desirable to have a palindrome as a reference (i.e., the role of the first palindrome) that possesses characters farthest to the right in a string when determining from left to right the palindromic length of a substring in the string (and consequently, the third palindrome in Case 2 and the fourth palindrome in Case 3 could replace the first palindrome to become the new reference).
7. Regarding the time complexity of palindromic length determination for each character in a string: there is no character comparison for Case 1, while for Cases 2 and 3 only the characters in the string beyond the right outermost character of the reference palindrome are candidates for comparison (and consequently Case 3 always results in a new reference palindrome while Case 2 does so only if the third palindrome is actually longer than its guaranteed minimum length).
8. For even-length palindromes, the center is at the boundary of the two characters in the middle.

Implementation

Let:

- s be a string of N characters
- s2 be a derived string of s, comprising N * 2 + 1 elements, with each element corresponding to one of the following: the N characters in s, the N-1 boundaries among characters, and the boundaries before and after the first and last character respectively
- A boundary in s2 is equal to any other boundary in s2 with respect to element matching in palindromic length determination
- p be an array of palindromic span for each element in s2, from center to either outermost element, where each boundary is counted towards the length of a palindrome (e.g. a palindrome that is three elements long has a palindromic span of 1)
- c be the position of the center of the palindrome currently known to include a boundary closest to the right end of s2 (i.e., the length of the palindrome = p[c]*2+1)
- r be the position of the right-most boundary of this palindrome (i.e., r = c + p[c])
- i be the position of an element (i.e., a character or boundary) in s2 whose palindromic span is being determined, with i always to the right of c
- i2 be the mirrored position of i around c (e.g., {i, i2} = {6, 4}, {7, 3}, {8, 2},⋯ when c = 5 (i.e., i2 = c * 2 - i)

```java
import java.util.Arrays;

public class ManachersAlgorithm {

    public static String findLongestPalindrome(String s) {
        if (s==null || s.length()==0)
            return "";

        char[] s2 = addBoundaries(s.toCharArray());
        int[] p = new int[s2.length];
        int c = 0, r = 0; // Here the first element in s2 has been processed.
        int m = 0, n = 0; // The walking indices to compare if two elements are the same
        for (int i = 1; i<s2.length; i++) {
            if (i>r) {
                p[i] = 0; m = i-1; n = i+1;
            } else {
                int i2 = c*2-i;
                if (p[i2]<(r-i-1)) {
                    p[i] = p[i2];
                    m = -1; // This signals bypassing the while loop below.
                } else {
                    p[i] = r-i;
                    n = r+1; m = i*2-n;
                }
            }
            while (m>=0 && n<s2.length && s2[m]==s2[n]) {
                p[i]++; m--; n++;
            }
            if ((i+p[i])>r) {
                c = i; r = i+p[i];
            }
        }
        int len = 0; c = 0;
        for (int i = 1; i<s2.length; i++) {
            if (len<p[i]) {
                len = p[i]; c = i;
            }
        }
        char[] ss = Arrays.copyOfRange(s2, c-len, c+len+1);
        return String.valueOf(removeBoundaries(ss));
    }

    private static char[] addBoundaries(char[] cs) {
        if (cs==null || cs.length==0)
            return "||".toCharArray();

        char[] cs2 = new char[cs.length*2+1];
        for (int i = 0; i<(cs2.length-1); i = i+2) {
            cs2[i] = '|';
            cs2[i+1] = cs[i/2];
        }
        cs2[cs2.length-1] = '|';
        return cs2;
    }

    private static char[] removeBoundaries(char[] cs) {
        if (cs==null || cs.length<3)
            return "".toCharArray();

        char[] cs2 = new char[(cs.length-1)/2];
        for (int i = 0; i<cs2.length; i++) {
            cs2[i] = cs[i*2+1];
        }
        return cs2;
    }
}
```

Notes

1. Crochemore & Rytter (1991), Apostolico, Breslauer & Galil (1995).

References

- Apostolico, Alberto; Breslauer, Dany; Galil, Zvi (1995), "Parallel detection of all palindromes in a string", *Theoretical Computer Science*, 141 (1 – 2): 163 – 173, doi:10.1016/0304-3975(94)00083-U (https://doi.org/10.1016%2F0304-3975%2894%2900083-U).
- Crochemore, Maxime; Rytter, Wojciech (1991), "Usefulness of the Karp – Miller – Rosenberg algorithm in parallel computations on strings and arrays", *Theoretical Computer Science*, 88 (1): 59 – 82, doi:10.1016/0304-3975(91)90073-B (https://doi.org/10.1016%2F0304-3975%2891%2990073-B), MR 1130372 (https://www.ams.org/mathscinet-getitem?mr=1130372).
- Crochemore, Maxime; Rytter, Wojciech (2003), "8.1 Searching for symmetric words", *Jewels of Stringology: Text Algorithms*, World Scientific, pp. 111 – 114, ISBN 978-981-02-4897-0.
- Gusfield, Dan (1997), "9.2 Finding all maximal palindromes in linear time", *Algorithms on Strings, Trees, and Sequences*, Cambridge: Cambridge University Press, pp. 197 – 199, doi:10.1017/CBO9780511574931 (https://doi.org/10.1017%2FCBO9780511574931), ISBN 0-521-58519-8, MR 1460730 (https://www.ams.org/mathscinet-getitem?mr=1460730).
- Jeuring, Johan (1994), "The derivation of on-line algorithms, with an application to finding palindromes", *Algorithmica*, 11 (2): 146 – 184, doi:10.1007/BF01182773 (https://doi.org/10.1007%2FBF01182773), MR 1272521 (https://www.ams.org/mathscinet-getitem?mr=1272521).
- Manacher, Glenn (1975), "A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string", *Journal of the ACM*, 22 (3): 346 – 351, doi:10.1145/321892.321896 (https://doi.org/10.1145%2F321892.321896).

External links

- *Longest Palindromic Substring Part II.* (http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html), 2011-11-20. A description of Manacher’s algorithm for finding the longest palindromic substring in linear time.
- Akalin, Fred (2007-11-28), *Finding the longest palindromic substring in linear time* (https://www.akalin.com/longest-palindrome-linear-time), retrieved 2016-10-01. An explanation and Python implementation of Manacher's linear-time algorithm.
- Jeuring, Johan (2007 – 2010), *Palindromes* (http://www.staff.science.uu.nl/~jeuri101/homepage/palindromes/index.html), retrieved 2011-11-22. Haskell implementation of Jeuring's linear-time algorithm.
- *Palindromes* (https://github.com/vvikas/palindromes/blob/master/src/LongestPalindromicSubString.java). Java implementation of Manacher's linear-time algorithm.

Categories: Problems on strings | Palindromes
