The longest palindrome

Larry LIU Xinyu

September 20, 2014

1 The problem

A palindrome is a symmetric sequence of elements. It doesn't change when gets reversed. For example, the English word 'madam' is a palindrome. If S is a palindrome, we have $S = reverse(S)$. Palindrome can be a number, a string, a piece of DNA genome, or even music. A palindromic sub-string is part of the string that is a palindrome. For example 'issi' is a palindromic sub-string in 'Mississippi'. Given a sequence, there can be multiple palindromic sub-strings. The problem is to find the longest one. Take 'Mississippi' for example, the longest plindromic sub-string is 'ississi'.

2 Manacher's algorithm

There are several methods to solve this problem. The brute-force solution enumerates all the sub-strings, filters the palindromic ones, and picks the longest. There are $n(n+1)/2$ sub-strings where n is the length of the string (Empty string is ignored). The performance of the brute-force method is quadratic.

Another method is to use suffix tree. If w is a palindromic sub-string in S, it must be sub-string of $reverse(S)$ as well. For example, "issi" is a palindromic sub-string of "Mississippi" and its reversed form "ippississiM".

Based on this fact, we can find the longest palindrome by searching the longest common sub-string in S and $reverse(S)$.

$$LCS(T_{\text{suffix}}(S + reverse(S))) \tag{1}$$

Where function LCS finds the longest common sub-string in the suffix tree in linear time.

The key point is to construct the suffix tree efficiently. There are some good algorithms achieve the linear time performance, like Ukkonen's algorithm[1]. This ensures the overall performace of generalised suffix tree method to be linear.

Suffix array provides an easier solution than suffix tree. But it downgrades the performance to $O(n \lg n)$.

We'll explain a linear time algorithm found by Glenn K. Manacher in 1975 [2]. This method scans the string in one round. It reuses the information gained during the scan which leads to an efficient solution.

For given string $S = \{s_1, s_2, ..., s_n\}$, let $P = \{p_1, p_2, ...p_n\}$ be a table. p_i is defined as the following.

$$p_i = max\{d | \{s_{i-d}, ..., s_{i+d}\} \text{ is palindrome}, d = 0, 1, 2, ...\} \tag{2}$$

We call P the palindrome table. p_i tells us how long we can extend from the i-th element in S to left and right to form a palindrome. In other words the sub-string $\{s_{i-p_i}, s_{i-p_i+1}, ..., s_i, ..., s_{i+p_i}\}$ is the longest palindromic sub-string at the center of i. We donote this sub-string as $S_{(i,p_i)}$. Below table shows an example for string 'eneven'.

S	e	n	e	v	e	n
P	0	1	0	2	0	0
$S_{(i,p_i)}$	e	ene	e	neven	e	n

The first value $p_1 = 0$, the palindrome at the center of the first element contains only one character "e". The second value $p_2 = 1$, it means we can extend from the second element 1 character to both sides. It gives the palindrome "ene". The fourth value $p_4 = 2$, we can extend 2 characters to get the palindrome "neven". We can think p_i as the 'radius' of the palindrome at center i.

However, there is a problem in the definition of p_i, in equation (2). The length of the palindrome is forced to be odd number. It doesn't work for even length. For example, the P table for string $S = ...issi...$ is as below.

S	...	i	s	s	i	...
P	...	0	0	0	0	...

Acutally, we need it to be something like this:

S	...	i	s		s	i	...
P	...			2			...

One solution is to insert a special delimiter $\# \notin S$ between all elements in S to turn it into another string S'.

$$S' = \{\#, s_1, \#, s_2, \#, ..., \#, s_n, \#\} \tag{3}$$

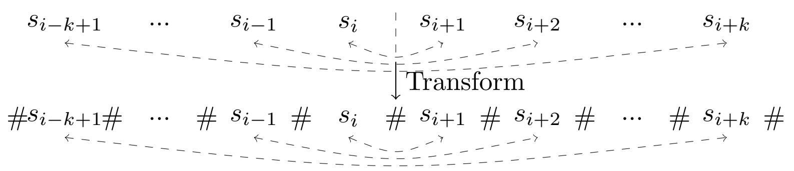
There are two cases.

- For the odd length palindrome, Let the length be $L = 2k + 1$.

  $$\begin{array}{c} s_{i-k} \quad \cdots \quad s_{i-1} \quad s_i \quad s_{i+1} \quad \cdots \quad s_{i+k} \\ \big\downarrow \text{Transform} \\ \# \; s_{i-k} \; \# \; \cdots \; \# \; s_{i-1} \; \# \; s_i \; \# \; s_{i+1} \; \# \; \cdots \; \# \; s_{i+k} \; \# \end{array}$$

  After the transformation, the center of the new palindrome is still s_i. In the new palindrome talbe P', the value corresponds to s_i is $p'_j = 2k + 1$.

- For the even length palindrome, Let the length be $L = 2k$.


s_{i-k+1} ... s_{i-1} s_i s_{i+1} s_{i+2} ... s_{i+k}
Transform
$\# s_{i-k+1} \#$ ... $\#$ s_{i-1} $\#$ s_i $\#$ s_{i+1} $\#$ s_{i+2} $\#$ ... $\#$ s_{i+k} $\#$

After the transformation, the center of the new palindrome becomes a delimiter #. In the new palindrome talbe P', the value corresponds to the new center is $p'_j = 2k$.

In both cases, we have the relationship $L = p'_j$. If we construct a palindrome table for the transformed string, each value in the table that does not correspond to the delimiter is the length of the original palindromic sub-string in S. Because the the maximum value in the palindrome table is bound to the longest one, the problem can be solved with the following strategy:

1. Transform to a new string with speical delimiter interspersed;
2. Construct the palindrome table effeciently;
3. The longest palindromic sub-string can be located by finding the maximum value in the palindrome table.

3 The brute-force scan

The brute-force method scans the transformed string from left to right. For each position i, it checks if $s_{i+k} = s_{i-k}$ for $k = 0, 1, 2, ...$ can be satisfied.

```
function PALINDROME($S$)
    $S' \leftarrow$ empty
    for $i \leftarrow 1$ to $|S|$ do
        $S' \leftarrow S' + \{\#, s_i\}$
    $S' \leftarrow S' + \{\#\}$
    $P \leftarrow \{0, 0, ..., 0\}$                                  ▷ length of $|S'|$
    for $i \leftarrow 1$ to $|S'|$ do
        while $1 \leq i \pm p_i \leq |S'|$ and $s'_{i-p_i} = s'_{i+p_i}$ do
            $p_i \leftarrow p_i + 1$
        $p_i \leftarrow p_i - 1$
    return $max\{p_i\}$
```

The worst case happens when all the characters are same, for instance "aaa...a", that there are n same characters 'a'. The brute-force method is quadratic ($O(n^2)$).

4 Manacher's method

The key to Manacher's method is information reusing. In order to construct the palindrome table P effeciently, we need *reuse* so far gained result $p_1 \sim p_{i-1}$

when caculate p_i.

Observe the brute-force solution. It compares s_{i+1} with s_{i-1}, s_{i+2} with s_{i-2}, ... The element to the right of s_i, may have been examined and it may belong to some previously found palindromic sub-string. In other words there may be some $j \in [1 \sim i-1]$ that satisfies $i \leq j + p_j$. This is illustrated in figure 1. We can record the right most position that the so far found palindromic sub-strings have riched:

$$r_m = max\{j + p_j, j < i\} \tag{4}$$

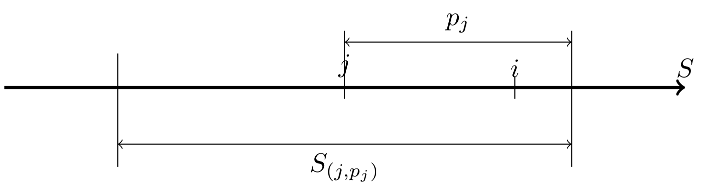


Figure 1: Element at i belongs to palindrome $S_{(j,p_j)}$. The palindrome extends to the right of i.

When scan to the i-th position from left, we firstly compare i with r_m to check if this position has been examined before, so that we can reuse the $p_1 \sim p_{i-1}$ information. There are two cases:

Case 1, $i > r_m$

Because $i > max\{j + p_j, j < i\}$, s_i doesn't belong to any known palindromes. What we can do is as same as the brute-force method. Scan and compare the symmetric elements s_{i+k} and s_{i-k} until we find an unmatch. The farest k is the value of p_i.

```
1: **if** $i > r_m$ **then**
2:     $p_i \leftarrow 0$
3:     **while** $s_{i+p_i} = s_{i-p_i}$ **do**
4:         $p_i \leftarrow p_i + 1$
5:     $p_i \leftarrow p_i - 1$
```

Case 2, $i \leq r_m$

Because $i \leq max\{j + p_j, j < i\}$, s_i belongs to some known palindrome. There are three sub-cases:

Sub-case 1

As shown in figure 2, Denote the j leading to r_m as j_m. The palindrome at the center of j_m is $A = S_{(j_m, p_{j_m})}$. The two elements out of the bound of A are different. They are represented as x and y, $x \neq y$. Because element at i belongs to the palindrome at the center of j_m, its symmetric position to j_m is $2j_m - i$. This is because $\frac{i+(2j_m-i)}{2} = j_m$. Denote the palindrome at the center of this position as B. In sub-case 1, the left bound of B is to the left of A.

$$left(B) < left(A)$$

This tells us that $x \in B$. According to the definition of palindrome, there must be a symmetric element x' in B, that $x' = x$. Because $x' \in A$, again, there must be a symmetric element x'' in palindrome A, so that $x'' = x$. Since $x \neq y$, we have $x'' \neq y$, this means that the 'radius' of the palindrome at the center of i is equal to the distance between points a and b.

$$r = |a - b| = j_m + p_{j_m} - i$$

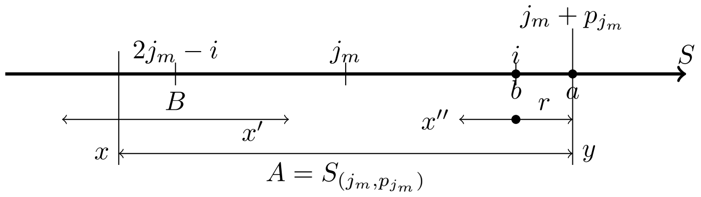


Figure 2: $left(B) < left(A)$

The result for sub-case 1 is that $p_i = j_m + p_{j_m} - i$.

Sub-case 2

In sub-case 2, palindrome B is within palindrome A as shown in figure 3. Denote the two elements next to the bounds of B as x and y, $x \neq y$. As illustrated in the figure, both $x, y \in A$. Their symmetric elements in A are x' and y', $x' \neq y'$. The palindrome at the center of i are as same as the palindrome at the center of $2j_m - i$.

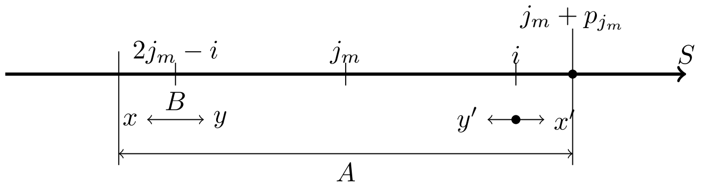


Figure 3: $left(A) < left(B)$

This fact indicates the result of sub-case 2 is $p_i = p_{2j_m - i}$.

Sub-case 3

In sub-case 3, palindrome A and B are left aligned as shown in figure 4. Denote the two different elements next to the bounds of A as x and y, $x \neq y$. The elements next to the right bound of B as $y' \neq x$. Because $y' \in A$, its symmetric element in A is $y'' = y'$. But we can't determine if y is equal to y''. This tells us that p_i is at least $p_{2j_m - i}$. We need further scan on both sides as what the brute-force method does.

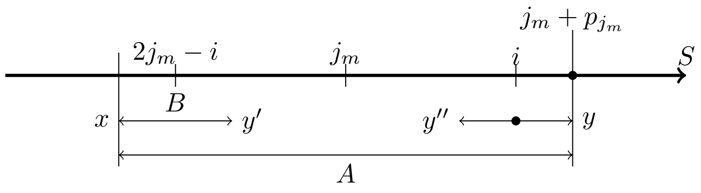


Figure 4: A and B are left aligned.

```
1: $p_i \leftarrow p_{2j_m - i}$
2: while $1 \leq i \pm p_i \leq |S|$ and $s_{i+p_i} = s_{i-p_i}$ do
3:     $p_i \leftarrow p_i + 1$
4: $p_i \leftarrow p_i - 1$
```

Summarize the above three sub-cases, we have the following result.

```
1: $p_i \leftarrow min\{p_{2j_m - i}, j_m + p_{j_m} - i\}$
2: while $1 \leq i \pm p_i \leq |S|$ and $s_{i+p_i} = s_{i-p_i}$ do
3:     $p_i \leftarrow p_i + 1$
4: $p_i \leftarrow p_i - 1$
```

5 Manacher's algorithm

In order to find the length of the longest panlindrome, we intersperse a special delimiter in the string. Then scan the string to construct the palindrome table. The maximum value in P gives the final result.

```
1: function MANACHER-PALINDROME($S$)
2:     for $c \in S$ do
3:         $S' \leftarrow S' + \{\#, c\}$
4:     $S' \leftarrow S' + \{\#\}$            ▷ Add extra terminator in ANSI C like language
5:     $P \leftarrow \{0, 0, ..., 0\}$                                   ▷ length of $|S'|$
6:     $j \leftarrow 0, m \leftarrow -\infty$
7:     for $i \leftarrow 1$ to $|S'|$ do
8:         if $i \leq j + p_j$ then
9:             $p_i \leftarrow min\{p_{2j-i}, j + p_j - i\}$
10:        while $1 \leq i \pm p_i \leq |S'|$ and $s'_{i+p_i} = s'_{i-p_i}$ do
11:            $p_i \leftarrow p_i + 1$
```

```
12:         $p_i \leftarrow p_i - 1$
13:         **if** $i + p_i > j + p_j$ **then**
14:             $j \leftarrow i$
15:         $m \leftarrow max\{p_i, m\}$
16:     **return** $m$
```

The palindrome table stores enough information about the palindromic substrings. Given p_j, The palindrome in the original string S can be expressed as:

$$S[\frac{j - p_j + 1}{2}, \frac{j + p_j - 1}{2}] \tag{5}$$

When implement this algorithm in programming languages that use left-include- right-exclude index representation, ANSI C for example, we need adjust the definition of p_i to simplify the boundary condition. In such environment, index starts from 0, and end with $n-1$, or expressed as `s[0..n)`. It's convenient to use the same idea for the palindrome table. We define `p[i]` as the minimum integer that satisifies `s[i+p[i]]` $\neq$ `s[i-p[i]]`. This value is one element longer than what we defined previously. We needn't decrease it by one after the while-loop. But we need change the condition when compare i with r_m. The length of the original palindrome changes to `p[i] - 1` respectively. The following Python example program implements the Manacher's algorithm.

```
DELIMITER = "#"
def manacher_palindrome(s):
    s = DELIMITER + DELIMITER.join(s) + DELIMITER
    n = len(s)
    p = [0] * n
    j = 0
    m = (-1, 0)  # (max length so far, position)
    for i in xrange(n):
        p[i] = min(p[2*j-i], j + p[j] - i) if i < j + p[j] else 1
        while 0 <= i - p[i] and i + p[i] < n and s[i-p[i]] == s[i+p[i]]:
            p[i] = p[i] + 1
        if j + p[j] < i + p[i]:
            j = i
        m = max(m, (p[i] - 1, i))
    return m
```

It can report both the length and the content of the longest palindrome like below.

```
(n, i) = manacher_palindrome(s)
print "The longest:", s[(i-n+1)/2 : (i+n-1)/2+1], " len:", n
```

Manacher's algorithm performs in linear time $O(n)$. This is because for any element s_i, after its first time successful comparison with some other s_k, $k < i$, it would never be compared succesfully with any s_j, $j < i$. The overal unsuccessful comparison is at most n.

References

[1] Esko Ukkonen. "On-line construction of suffix trees". Algorithmica 14 (3): 249–260. doi:10.1007/BF01206331. http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[2] 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): 346C351, doi:10.1145/321892.321896

[3] Longest palindromic substring. Wikipedia. http://en.wikipedia.org/wiki/Longest_palindromic_substring