01.05 最长回文子串

题目描述

给定一个字符串，求它的最长回文子串的长度。

分析与解法

最容易想到的办法是枚举所有的子串，分别判断其是否为回文。这个思路初看起来是正确的，但却做了很多无用功，如果一个长的子串包含另一个短一些的子串，那么对子串的回文判断其实是不需要的。

解法一

那么如何高效的进行判断呢？我们想想，如果一段字符串是回文，那么以某个字符为中心的前缀和后缀都是相同的，例如以一段回文串“aba”为例，以b为中心，它的前缀和后缀都是相同的，都是a。

那么，我们是否可以可以枚举中心位置，然后再在该位置上用扩展法，记录并更新得到的最长的回文长度呢？答案是肯定的，参考代码如下：

```
int LongestPalindrome(const char *s, int n)
{
        int i, j, max,c;
        if (s == 0 || n < 1)
                return 0;
        max = 0;

        for (i = 0; i < n; ++i) { // i is the middle point of the palindrome
                for (j = 0; (i - j >= 0) && (i + j < n); ++j){ // if the length of
the palindrome is odd
                        if (s[i - j] != s[i + j])
                                break;
                        c = j * 2 + 1;
                }
                if (c > max)
                        max = c;
                for (j = 0; (i - j >= 0) && (i + j + 1 < n); ++j){ // for the even
case
                        if (s[i - j] != s[i + j + 1])
                                break;
                        c = j * 2 + 2;
                }
                if (c > max)
                        max = c;
        }
        return max;
}
```

代码稍微难懂一点的地方就是内层的两个 for 循环，它们分别对于以 i 为中心的，长度为奇数和偶数的两种情况，整个代码遍历中心位置 i 并以之扩展，找出最长的回文。

解法二、**O(N)**解法

在上文的解法一：枚举中心位置中，我们需要特别考虑字符串的长度是奇数还是偶数，所以导致我们在编写代码实现的时候要把奇数和偶数的情况分开编写，是否有一种方法，可以不用管长度是奇数还是偶数，而统一处理呢？比如是否能把所有的情况全部转换为奇数处理？

答案还是肯定的。这就是下面我们将要看到的Manacher算法，且这个算法求最长回文子串的时间复杂度是线性O(N)的。

首先通过在每个字符的两边都插入一个特殊的符号，将所有可能的奇数或偶数长度的回文子串都转换成了奇数长度。比如 abba 变成 #a#b#b#a#， aba变成 #a#b#a#。

此外，为了进一步减少编码的复杂度，可以在字符串的开始加入另一个特殊字符，这样就不用特殊处理越界问题，比如$#a#b#a#。

以字符串12212321为例，插入#和$这两个特殊符号，变成了 S[] = "$#1#2#2#1#2#3#2#1#"，然后用一个数组 P[i] 来记录以字符S[i]为中心的最长回文子串向左或向右扩张的长度（包括S[i]）。

比如S和P的对应关系：

- S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
- P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1

可以看出，P[i]-1正好是原字符串中最长回文串的总长度，为5。

接下来怎么计算P[i]呢？Manacher算法增加两个辅助变量id和mx，其中id表示最大回文子串中心的位置，mx则为id+P[id]，也就是最大回文子串的边界。得到一个很重要的结论：

- 如果mx > i，那么P[i] >= Min(P[2 * id - i], mx - i)

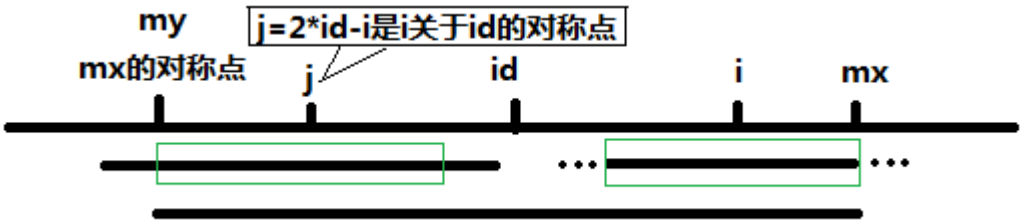
C代码如下：

```
//mx > i，那么P[i] >= MIN(P[2 * id - i], mx - i)
//故谁小取谁
if (mx - i > P[2*id - i])
    P[i] = P[2*id - i];
else  //mx-i <= P[2*id - i]
    P[i] = mx - i;
```

下面，令j = 2*id - i，也就是说j是i关于id的对称点。

当 mx - i > P[j] 的时候，以S[j]为中心的回文子串包含在以S[id]为中心的回文子串中，由于i和j对称，以S[i]为中心的回文子串必然包含在以S[id]为中心的回文子串中，所以必有P[i] = P[j]；

当 P[j] >= mx - i 的时候，以S[j]为中心的回文子串不一定完全包含于以S[id]为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以S[i]为中心的回文子串，其向右至少会扩张到mx的位置，也就是说 P[i] >= mx - i。至于mx之后的部分是否对称，再具体匹配。



此外，对于 mx <= i 的情况，因为无法对 P[i]做更多的假设，只能让P[i] = 1，然后再去匹配。

综上，关键代码如下：

```
//输入，并处理得到字符串s
int p[1000], mx = 0, id = 0;
memset(p, 0, sizeof(p));
for (i = 1; s[i] != '\0'; i++)
{
        p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
        while (s[i + p[i]] == s[i - p[i]])
                p[i]++;
        if (i + p[i] > mx)
        {
                mx = i + p[i];
                id = i;
        }
}
//找出p[i]中最大的
```

此Manacher算法使用id、mx做配合，可以在每次循环中，直接对P[i]的快速赋值，从而在计算以i为中心的回文子串的过程中，不必每次都从1开始比较，减少了比较次数，最终使得求解最长回文子串的长度达到线性O(N)的时间复杂度。

参考：http://www.felix021.com/blog/read.php?2040 。另外，这篇文章也不错：http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html 。