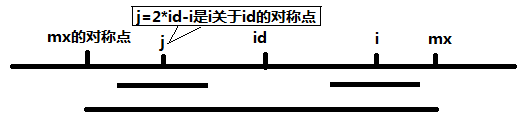
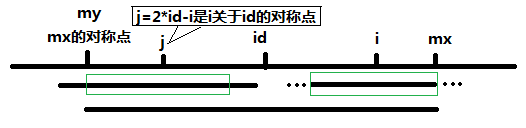
**[Manacher's ALGORITHM: O(n)时间求字符串的最长回文子串](http://www.felix021.com/blog/read.php?2040)**

源于这两篇文章：   
<http://blog.csdn.net/ggggiqnypgjg/article/details/6645824>  
<http://zhuhongcheng.wordpress.com/2009/08/02/a-simple-linear-time-algorithm-for-finding-longest-palindrome-sub-string/>  
  
这个算法看了三天，终于理解了，在这里记录一下自己的思路，免得以后忘了又要想很久- -.  
  
首先用一个非常巧妙的方式，将所有可能的奇数/偶数长度的回文子串都转换成了奇数长度：在每个字符的两边都插入一个特殊的符号。比如 abba 变成 #a#b#b#a#， aba变成 #a#b#a#。 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#。  
  
下面以字符串12212321为例，经过上一步，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/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  
(p.s. 可以看出，P[i]-1正好是原字符串中回文串的总长度）

那么怎么计算P[i]呢？该算法增加两个辅助变量（其实一个就够了，两个更清晰）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。  
(**mx:之前求得的最远右界**)  
然后可以得到一个非常神奇的结论，这个算法的关键点就在这里了：如果mx > i，那么P[i] >= MIN(P[2 \* id - i], mx - i)。就是这个串卡了我非常久。实际上如果把它写得复杂一点，理解起来会简单很多：

//记j = 2 \* id - i，也就是说 j 是 i 关于 id 的对称点。  
**if** (mx - i > P[j])   
    P[i] = P[j];  
**else** /\* P[j] >= mx - i \*/  
    P[i] = mx - i; // P[i] >= mx - i，取最小值，之后再匹配更新。

当然光看代码还是不够清晰，还是借助图来理解比较容易。  
  
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]，见下图。  
[](http://www.felix021.com/blog/attachment.php?fid=447)  
  
当 P[j] > mx - i 的时候，以S[j]为中心的回文子串不完全包含于以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，就只能老老实实去匹配了。  
[](http://www.felix021.com/blog/attachment.php?fid=448)  
  
对于 mx <= i 的情况，无法对 P[i]做更多的假设，只能P[i] = 1，然后再去匹配了。  
  
于是代码如下：

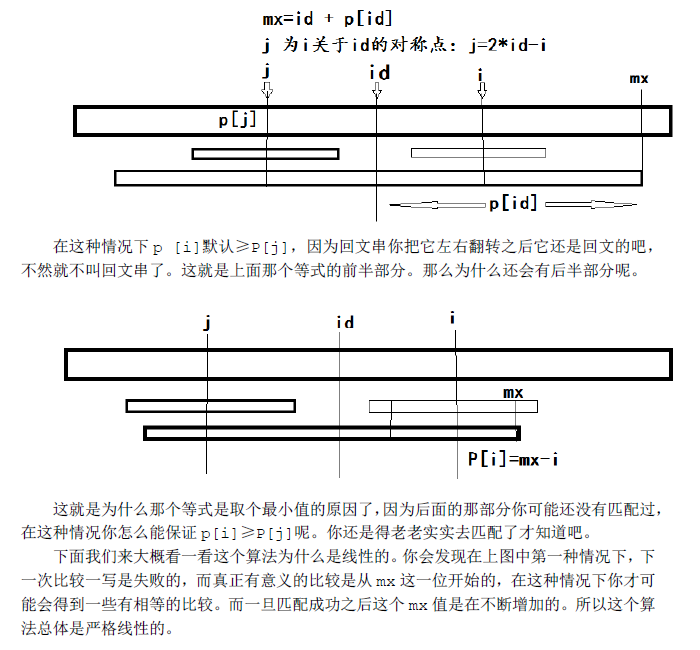
//输入，并处理得到字符串s  
**int** p[1000], mx = 0, id = 0;  
memset(p, 0, **sizeof**(p));  
**for** (i = 1; s[i] != '\0'; i++) {  
    p[i] = mx > i ? min(p[2\*id-i], mx-i) : 1;  
    **while** (s[i + p[i]] == s[i - p[i]]) p[i]++;  
    **if** (i + p[i] > mx) {  
        mx = i + p[i];  
        id = i;  
  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

补充：

如何在O(n)时间内处理字符串以每个位置为中心的最长回文。这里转载一个Manacher算法的论文翻译。经常有一些题目围绕回文子串进行讨论，比如POJ3974最长回文，求最长回文子串的长度。朴素算法是依次以每一个字符为中心向两侧进行扩展，显然这个复杂度是O(N^2)的，关于字符串的题目常用的算法有KMP、后缀数组、AC 自动机，这道题目利用扩展KMP可以解答，其时间复杂度也很快O(N\*logN)。但是，今天笔者介绍一个专门针对回文子串的算法，其时间复杂度为O(n)，这就是manacher 算法。

其实原文说得是比较清楚的，只是英文的，我这里写一份中文的吧。   
     首先：大家都知道什么叫回文串吧，这个算法要解决的就是一个字符串中最长的回文子串有多长。这个算法可以在O（n）的时间复杂度内既线性时间复杂度的情况下，求出以每个字符为中心的最长回文有多长，   
     这个算法有一个很巧妙的地方，它把奇数的回文串和偶数的回文串统一起来考虑了。这一点一直是在做回文串问题中时比较烦的地方。这个算法还有一个很好的地方就是充分利用了字符匹配的特殊性，避免了大量不必要的重复匹配。   
     算法大致过程是这样。先在每两个相邻字符中间插入一个分隔符，当然这个分隔符要在原串中没有出现过。一般可以用‘#’分隔。这样就非常巧妙的将奇数长度回文串与偶数长度回文串统一起来考虑了（见下面的一个例子，回文串长度全为奇数了），然后用一个辅助数组P记录以每个字符为中心的最长回文串的信息。P［id］记录的是以字符str［id］为中心的最长回文串，当以str［id］为第一个字符，这个最长回文串向右延伸了P［id］个字符。   
     原串：    w aa bwsw f d    
     新串：   # w # a # a # b # w # s # w # f # d #   
辅助数组P：  1 2 1 2 3 2 1 2 1 2 1 4 1 2 1 2 1 2 1   
     这里有一个很好的性质，P［id］-1就是该回文子串在原串中的长度（包括‘#’）。如果这里不是特别清楚，可以自己拿出纸来画一画，自己体会体会。当然这里可能每个人写法不尽相同，不过我想大致思路应该是一样的吧。   
     好，我们继续。现在的关键问题就在于怎么在O（n）时间复杂度内求出P数组了。只要把这个P数组求出来，最长回文子串就可以直接扫一遍得出来了。   
     由于这个算法是线性从前往后扫的。那么当我们准备求P［i］的时候，i以前的P［j］我们是已经得到了的。我们用mx记在i之前的回文串中，延伸至最右端的位置。同时用id这个变量记下取得这个最优mx时的id值。（注：为了防止字符比较的时候越界，我在这个加了‘#’的字符串之前还加了另一个特殊字符‘$’，故我的新串下标是从1开始的）   
好，到这里，我们可以先贴一份代码了。   
复制代码void pk()   
{   
     int i;   
     int mx = 0;   
     int id;   
     for(i=1; i<n; i++)   
     {   
         if( mx > i )   
             p[i] = MIN( p[2\*id-i], mx-i );           
         else   
             p[i] = 1;   
         for(; str[i+p[i]] == str[i-p[i]]; p[i]++)   
             ;   
         if( p[i] + i > mx )   
         {   
             mx = p[i] + i;   
             id = i;   
         }   
     }   
}   
     
    
    
     代码是不是很短啊，而且相当好写。很方便吧，还记得我上面说的这个算法避免了很多不必要的重复匹配吧。这是什么意思呢，其实这就是一句代码。

if( mx > i )   
     p[i] = MIN( p[2\*id-i], mx-i );

就是当前面比较的最远长度mx>i的时候，P［i］有一个最小值。这个算法的核心思想就在这里，为什么P数组满足这样一个性质呢?   
    （下面的部分为图片形式）   
  

以上都是翻译部分，这里在贴一个poj3974的AC代码，果然是瞬秒啊。   
#include <cstdio>   
#include <algorithm>   
#include <cstring>   
#include <iostream>   
using namespace std;   
const int MAXN=1000010;   
char str[MAXN],s[MAXN\*2];   
int n,p[MAXN\*2],ans,cases=1;   
void rebuild()   
{   
     s[0]='$',s[1]='#';   
     n=strlen(str);   
     for(int i=0;i<n;++i)   
     s[2\*i+2]=str[i],s[2\*i+3]='#';   
     s[n=n\*2+2]=0;   
}   
void solve()   
{   
     int mx=0,id,ans=1;   
     for(int i=0;i<n;++i)   
     {   
         if(mx>i)p[i]=min(p[2\*id-i],mx-i);   
         else p[i]=1;   
         for(;s[i-p[i]]==s[i+p[i]];p[i]++);   
         if(p[i]+i>mx)mx=p[i]+i,id=i;   
         ans=max(ans,p[i]);   
     }   
     printf("Case %d: %d\n",cases++,ans-1);   
}   
int main()   
{   
     while(scanf("%s",str),str[0]-'E')   
     {   
         rebuild();   
         solve();   
     }   
     return 0;   
}

参考资料

<http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html>

<http://www.akalin.cx/longest-palindrome-linear-time>