Manacher's Algorithm 马拉车算法

继 KMP 算法之后顺便总结一下马拉车算法，总结与此，仅供自身复习与知识分享所用。

这个马拉车算法Manacher's Algorithm是用来查找一个字符串的最长回文子串的线性方法，由一个叫Manacher的人在1975年发明的，这个方法的最大贡献是在于将时间复杂度提升到了线性，这是非常了不起的。

对于回文串想必大家都不陌生，就是正读反读都一样的字符串，比如 "bob", "level", "noon" 等等，那么如何在一个字符串中找出最长回文子串呢，可以以每一个字符为中心，向两边寻找回文子串，在遍历完整个数组后，就可以找到最长的回文子串。但是这个方法的时间复杂度为O(n*n)，并不是很高效，下面我们来看 **时间复杂度为O(n)** 的马拉车算法。

1. 插入特殊字符

由于回文串的长度可奇可偶，比如"bob"是奇数形式的回文，"noon"就是偶数形式的回文，马拉车算法的第一步是预处理，做法是在每一个字符的左右都加上一个特殊字符，比如加上'#'，那么

bob --> #b#o#b#

noon --> #n#o#o#n#

这样做的好处是不论原字符串是奇数还是偶数个，处理之后得到的字符串的个数都是奇数个，这样就不用分情况讨论了，而可以一起搞定。接下来我们还需要和处理后的字符串t等长的数组 p，其中 **p[i]表示以 t[i] 字符为中心的回文子串的半径**，若p[i] = 1，则该回文子串就是t[i]本身，那么我们来看一个简单的例子：

1 # 2 # 2 # 1 # 2 # 2 # 1 2 1 2 5 2 1 6 1 2 3 2 1

2. 确定回文子串半径

为啥我们关心回文子串的半径呢？看上面那个例子，以中间的 '1' 为中心的回文子串 "#2#2#1#2#2#" 的半径是6，而未添加#号的回文子串为 "22122"，长度是5，为半径减1。这是个普遍的规律么？

- 我们再看看之前的那个 "#b#o#b#"，我们很容易看出来以中间的 'o' 为中心的回文串的半径是4，而 "bob"的长度是3，符合规律。
- 再来看偶数个的情况"noon"，添加#号后的回文串为 "#n#o#o#n#"，以最中间的 '#' 为中心的回文串的半径是5，而 "noon" 的长度是4，完美符合规律。

所以我们只要找到了最大的半径，就知道最长的回文子串的字符个数了：原最长回文子串长度 = 插入符号后最长回文子串半径 - 1

只知道长度无法定位子串，我们还需要知道子串的起始位置。

3. 确定回文子串起始位置

我们还是先来看中间的 '1' 在字符串 "#1#2#2#1#2#2#" 中的位置是7，而半径是6，貌似7-6=1，刚好就是回文子串 "22122" 在原串 "122122" 中的起始位置1。 那么我们再来验证下 "bob"，"o" 在 "#b#o#b#" 中的位置是3，但是半径是4，这一减成负的了，肯定不对。**所以我们应该至少把中心位置向后移动一位，才能为0啊，那么我们就需要在前面增加一个字符**，这个字符不能是#号，也不能是s中可能出现的字符，所以我们暂且就用美元号吧，毕竟是博主最爱的东西嘛。这样都不相同的话就不会改变p值了，那么末尾要不要对应的也添加呢，其实不用的，不用加的原因是字符串的结尾标识为'\0'，等于默认加过了。那此时 "o" 在 "$#b#o#b#" 中的位置是4，半径是4，一减就是0了，貌似没啥问题。

- 我们再来验证一下那个数字串，中间的 '1' 在字符串 "$#1#2#2#1#2#2#" 中的位置是8，而半径是6，这一减就是2了，而我们需要的是1，所以我们要除以2。
- 之前的 "bob" 因为相减已经是0了，除以2还是0，没有问题。
- 再来验证一下 "noon"，中间的 '#' 在字符串 "$#n#o#o#n#" 中的位置是5，半径也是5，相减并除以2还是0，

完美。可以任意试试其他的例子，都是符合这个规律的，**最长子串的长度是半径减1，起始位置是中间位置减去半径再除以2。**

那么下面我们就来看如何求p数组，**需要新增两个辅助变量 mx 和 id，其中 id 为能延伸到最右端的位置的那个回文子串的中心点位置，mx 是回文串能延伸到的最右端的位置，p[i] 表示以 t[i] 字符为中心的回文子串的半径**，这个算法的最核心的一行如下：

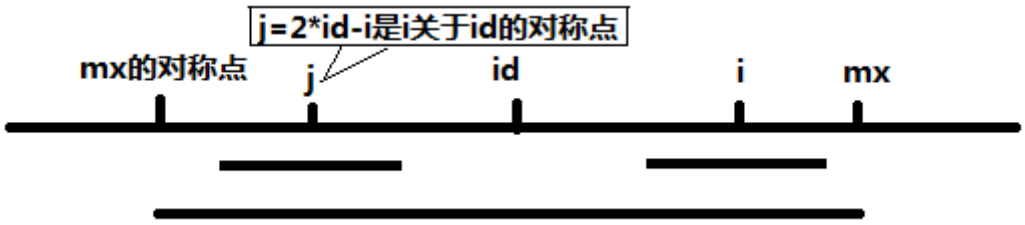
```
p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
```

可以这么说，这行要是理解了，那么马拉车算法基本上就没啥问题了，那么这一行代码拆开来看就是

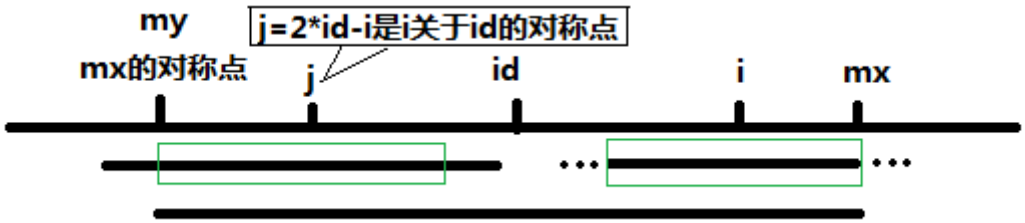
如果 mx > i, 则 `p[i] = min( p[2 * id - i] , mx - i )`

否则，p[i] = 1

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]，其中 `j = 2*id - i`，因为 j 到 id 之间到距离等于 id 到 i 之间到距离，为 i - id，所以 `j = id - (i - id) = 2*id - i`，参见下图。



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，就只能老老实实去匹配了，这就是后面紧跟到while循环的作用。



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。

参见如下实现代码：

```
#include <vector>
#include <iostream>
#include <string>

using namespace std;

string Manacher(string s) {
```

```
    // Insert '#'
    string t = "$#";
    for (int i = 0; i < s.size(); ++i) {
        t += s[i];
        t += "#";
    }
    // Process t
    vector<int> p(t.size(), 0);
    int mx = 0, id = 0, resLen = 0, resCenter = 0;
    for (int i = 1; i < t.size(); ++i) {
        p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
        while (t[i + p[i]] == t[i - p[i]]) ++p[i];
        if (mx < i + p[i]) {
            mx = i + p[i];
            id = i;
        }
        if (resLen < p[i]) {
            resLen = p[i];
            resCenter = i;
        }
    }
    return s.substr((resCenter - resLen) / 2, resLen - 1);
}

int main() {
    string s1 = "12212";
    cout << Manacher(s1) << endl;
    string s2 = "122122";
    cout << Manacher(s2) << endl;
    string s = "waabwswfd";
    cout << Manacher(s) << endl;
}
```

参考文献

[1] Manacher's Algorithm 马拉车算法