最大回文子串

宁华

POJ3974 Palindrome

- 给定字符串，求它的最长回文子串的长度是多少。

- 输入格式
- 输入将包含最多 30 个测试用例，每个测试用例占一行，以最多 1000000 个小写字符的形式给出。
- 输入以一个以字符串 END 开头的行表示输入终止。

- 输出格式
- 对于输入中的每个测试用例，输出测试用例编号和最大回文子串的长度（参考样例格式）。
- 每个输出占一行。

POJ3974 Palindrome

- Sample Input
- abcbabcbabcba
- abacacbaaaab
- END

- Sample Output
- Case 1: 13
- Case 2: 6

头脑风暴

- 求最大回文子串

- 提出任何你能想到的方法~

方法1：暴力

方法1.1：纯暴力

- 枚举所有的子串：
- 枚举起点和终点。。。从两边往中间依次对比两端字符

- 复杂度 O(n^3)

方法1.2：中心扩展

- 枚举中心点，从中间往两边依次扩展
- （1）奇回文
- （2）偶回文

- （1+2）经过处理，统一变成奇回文处理

- 复杂度 O(n^2)

方法1.3：

- 先将串逆置，再与原串求最长公共子序列(LCS)

- 复杂度 O(n^2)

方法1.4：dp

- 暴力改进，辅助数组 ok[i][j] 记录从i到j是否回文
- 枚举长度

方法1.4：dp

```
// 长度1
for(int i=0; i<n; i++)
        ok[i][i] = true;
// 长度2
for(int i=0; i+1<n; i++)
{
        if(c[i]==c[i+1])
        {
                ok[i][i+1] = true;
                mi = i;
                mx = i+1;
        }
}
// 长度3
for(int i=0; i+2<n; i++)
{
        if(c[i] == c[i+2])
        {
                ok[i][i+2] = true;
                mi = i;
                mx = i+2;
        }
}
```

```
// 长度为4及以上的区间
        for(int k=4; k<=n; k++)
        {
                for(int i=0; i+k-1<n; i++)
                {
                        int j = i+k-1;
                        if(c[i]==c[j] && ok[i+1][j-1])
                        {
                                ok[i][j] = true;
                                mi = i;
                                mx = j;
                        }
                }
        }
```

方法二：字符串哈希 + 二分

- 例如:
- “abcbcabcbacb”
- 哈希为:
- 123231232132

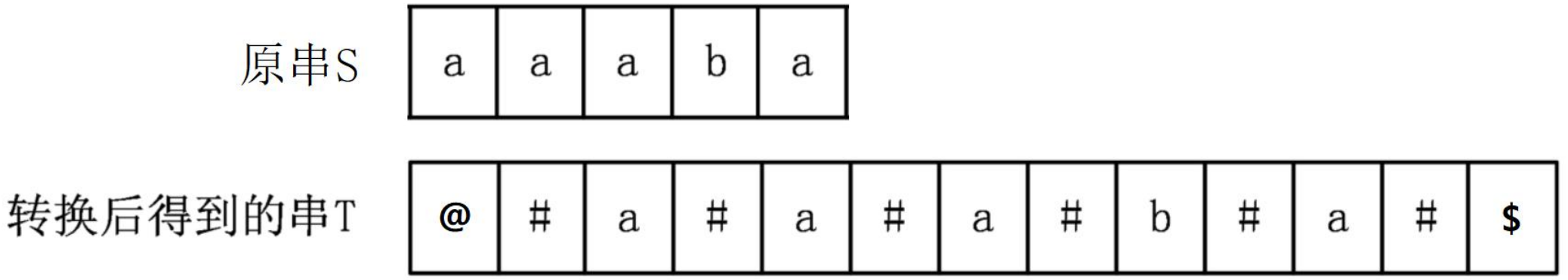
- 复杂度 O(nlogn)

方法三：Manacher算法

- 复杂度 O(n)

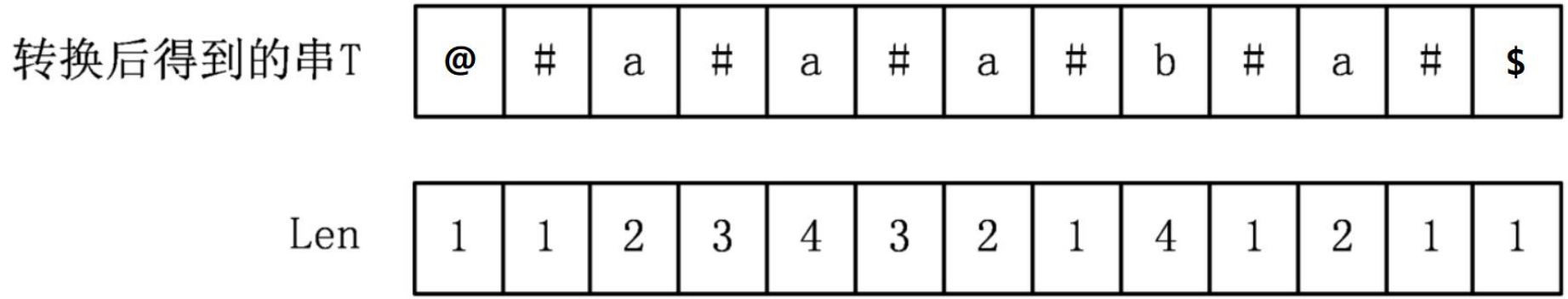
1、预处理

- 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：在字符中间插入一个特殊的符号。
- 为了进一步减少编码的复杂度，可以在字符串的两端加入另外两个不同的特殊字符，这样就不用特殊处理越界问题。比如：



2、半径数组Len[]

- Len[i] : 以字符 T[i] 为中心的最长回文子串向左/右扩张的长度（包括T[i]本身），比如：



- (p.s. 可以看出，Len[i] - 1 正好是原字符串中回文串的总长度)
- 证明：
- 首先在转换得到的字符串T中，所有的回文字串的长度都为奇数，那 么对于以T[i]为中心的最长回文字串，其长度就为2*Len[i]-1,经过观察可知，T中所有的回文子串，其中分隔符的数量一定比其他字符的数量多 1，也就是有Len[i]个分隔符，剩下Len[i]-1个字符来自原字符串，所以该回文串在原字符串中的长度就为Len[i]-1。

3、我们的目标

- 只要计算出 Len[] 就万事大吉了~

- ans = max (Len[i])

- 如何计算 Len[i] ?

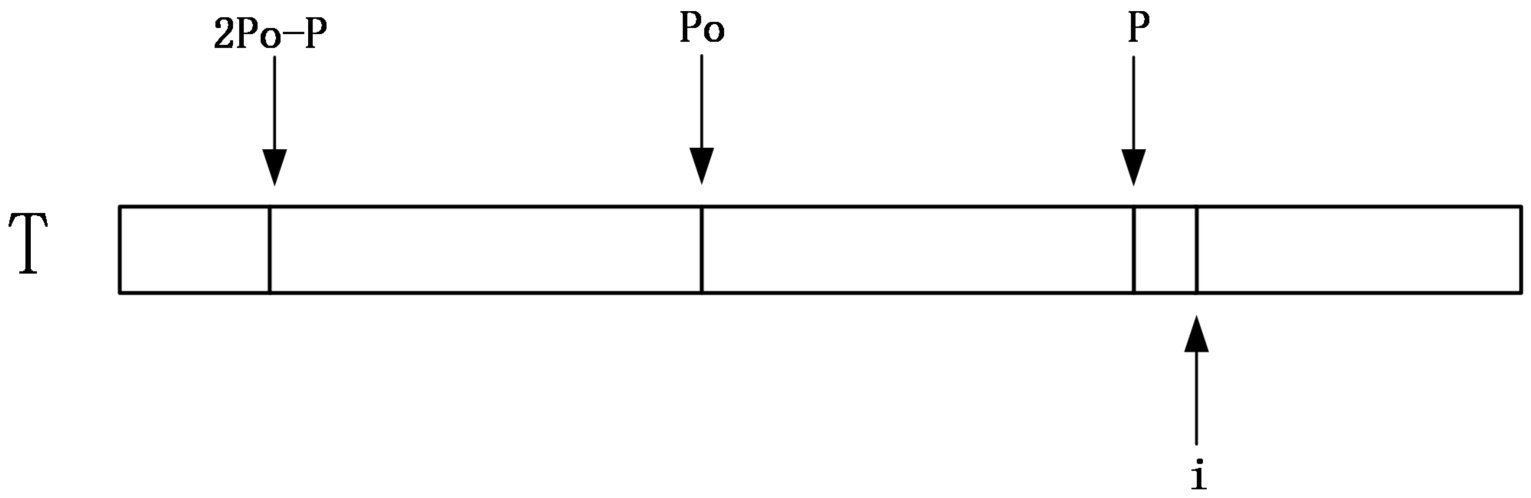
- 从左到右依次计算 Len[i]
- 当计算Len[i]时，对于所有的 0 <= j < i , Len[j] 已经计算出来了

4、具体展开

- 引入两个值：
- P：表示 i 之前计算回文所扩展的最远的点的下标
- Po：表示 MaxP 所对应的中心点的下标

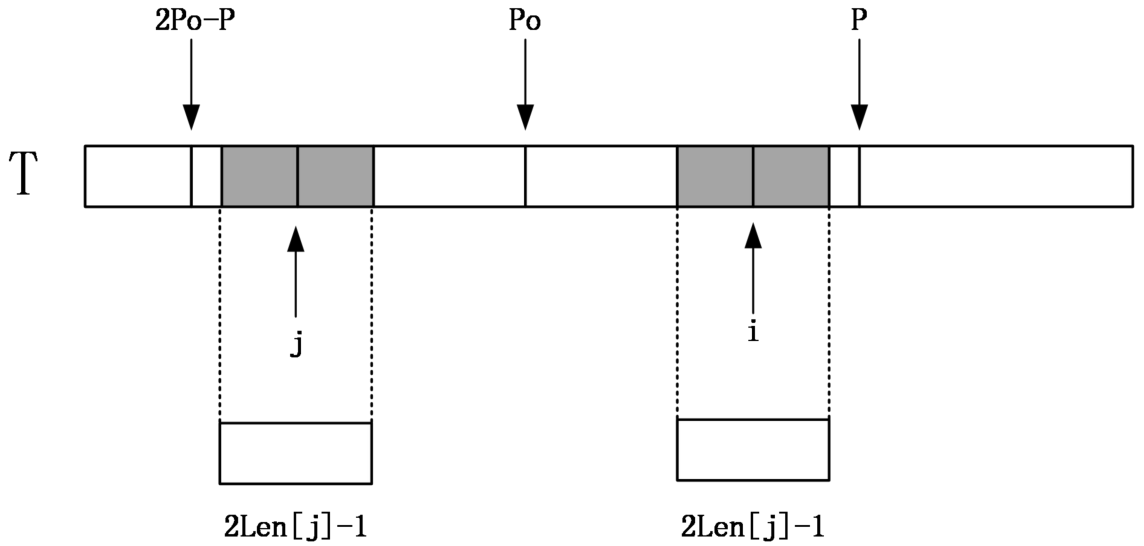
- 现在我们要计算 Len[i] , 有以下两种情况：

（一） i > P



- 如果 i 比 P 还要大，说明对于中点为 i 的回文串还一点都没有匹配，这个时候，就只能老老实实地一个一个匹配了，匹配完成后要更新 P 的位置和对应的Po以及 Len[i]。

（二）i <= P



- 找到 i 相对于po的对称位置 2*Po - i，设为 j，那么
- （1）如果Len[j] < P-i，说明以 j 为中心的回文串一定在以 Po 为中心的回文串的内部，由于 j 和 i 关于位置 Po 对称，所以以 i 为中心的回文串一定在以 Po 为中心的回文串的内部，即Len[i]==Len[j]。
- （2）如果Len[j]>=P-i, 由对称性，以 i 为中心的回文串至少扩展到P，还可能会延伸到P之外，而大于P的部分我们还没有进行匹配，所以要从P+1位置开始一个一个进行匹配，直到发生失配，从而更新P和对应的Po以及Len[i]。

核心代码参考

```
for (int i = 1; i < len; i++)
{
    if (i < P)
        Len[i] = min(Len[2 * Po - i], P - i);  //  i 关于 Po 的对称点 j = 2*Po-i
    else
        Len[i] = 1;

    while (s_new[i - Len[i]] == s_new[i + Len[i]])  // 不需边界判断，因为左有 @，右有 $
        Len[i]++;

    // 我们每走一步 i，都要和 P 比较，我们希望 P 尽可能的远，
    // 这样才能更有机会执行 if (i < P)这句代码，从而提高效率
    if (P < i + Len[i])
    {
        Po = i;
        P = i + Len[i];
    }

    max_len = max(max_len, Len[i] - 1);
}
```