文 最长回文子串——Manacher 算法

python 算法 字符串处理

0. 问题定义

最长回文子串问题：给定一个字符串，求它的最长回文子串长度。

如果一个字符串正着读和反着读是一样的，那它就是回文串。下面是一些回文串的实例：

```
12321 a aba abba aaaa tattarrattat（牛津英语词典中最长的回文单词）
```

1. Brute-force 解法

对于最长回文子串问题，最简单粗暴的办法是：找到字符串的所有子串，遍历每一个子串以验证它们是否为回文串。一个子串由子串的起点和终点确定，因此对于一个长度为n的字符串，共有n^2个子串。这些子串的平均长度大约是n/2，因此这个解法的时间复杂度是O(n^3)。

2. 改进的方法

显然所有的回文串都是对称的。长度为奇数回文串以最中间字符的位置为对称轴左右对称，而长度为偶数的回文串的对称轴在中间两个字符之间的空隙。可否利用这种对称性来提高算法效率呢？答案是肯定的。我们知道整个字符串中的所有字符，以及字符间的空隙，都可能是某个回文子串的对称轴位置。可以遍历这些位置，在每个位置上同时向左和向右扩展，直到左右两边的字符不同，或者达到边界。对于一个长度为n的字符串，这样的位置一共有n+n-1=2n-1个，在每个位置上平均大约要进行n/4次字符比较，于是此算法的时间复杂度是O(n^2)。

3. Manacher 算法

对于一个比较长的字符串，O(n^2)的时间复杂度是难以接受的。Can we do better?

先来看看解法**2**存在的缺陷。

1) 由于回文串长度的奇偶性造成了不同性质的对称轴位置，解法**2**要对两种情况分别处理；
2) 很多子串被重复多次访问，造成较差的时间效率。

缺陷2）可以通过这个直观的小□体现：

```
char: a b a b a
  i : 0 1 2 3 4
```

当i==1，和i==2时，左边的子串aba分别被遍历了一次。

如果我们能改善解法**2**的不足，就很有希望能提高算法的效率。Manacher正是针对这些问题改进算法。

(1) 解决长度奇偶性带来的对称轴位置问题

Manacher算法首先对字符串做一个预处理，在所有的空隙位置(包括首尾)插入同样的符号，要求这个符号是不会在原串中出现的。这样会使得所有的串都是奇数长度的。以插入#号为例：

```
aba  ——>  #a#b#a#
abba ——>  #a#b#b#a#
```

插入的是同样的符号，且符号不存在于原串，因此子串的回文性不受影响，原来是回文的串，插完之后还是回文的，原来不是回文的，依然不会是回文。

(2) 解决重复访问的问题

我们把一个回文串中最左或最右位置的字符与其对称轴的距离称为回文半径。Manacher定义了一个回文半径数组RL，用RL[i]表示以第i个字符为对称轴的回文串的回文半径。我们一般对字符串从左往右处理，因此这里定义RL[i]为第i个字符为对称轴的回文串的最右一个字符与字符i的距离。对于上面插入分隔符之后的两个串，可以得到RL数组：

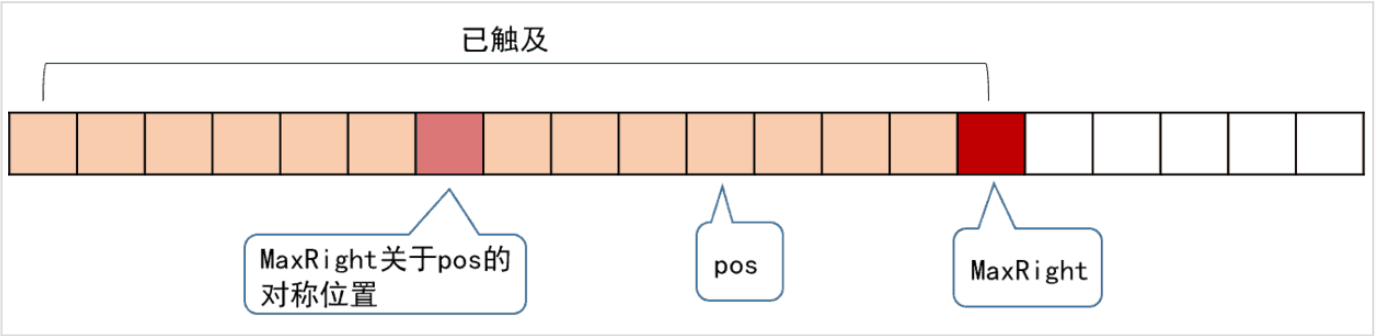
```
char:    # a # b # a #
 RL :    1 2 1 4 1 2 1
RL-1:    0 1 0 3 0 1 0
  i :    0 1 2 3 4 5 6

char:    # a # b # b # a #
 RL :    1 2 1 2 5 2 1 2 1
RL-1:    0 1 0 1 4 1 0 1 0
  i :    0 1 2 3 4 5 6 7 8
```

上面我们还求了一下RL[i]-1。通过观察可以发现，RL[i]-1的值，正是在原本那个没有插入过分隔符的串中，以位置i为对称轴的最长回文串的长度。那么只要我们求出了RL数组，就能得到最长回文子串的长度。

于是问题变成了，怎样高效地求的**RL**数组。基本思路是利用回文串的对称性，扩展回文串。

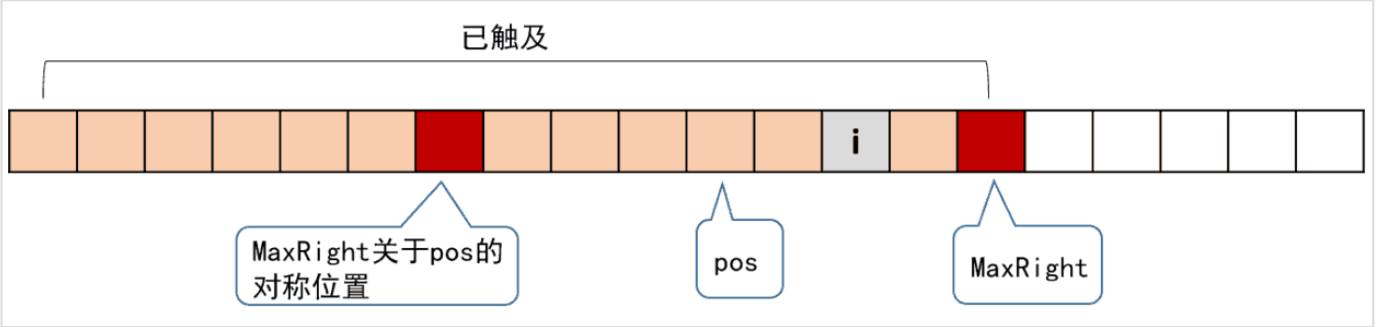
我们再引入一个辅助变量`MaxRight`，表示当前访问到的所有回文子串，所能触及的最右一个字符的位置。另外还要记录下`MaxRight`对应的回文串的对称轴所在的位置，记为`pos`，它们的位置关系如下。



我们从左往右地访问字符串来求RL，假设当前访问到的位置为`i`，即要求RL[i]，在对应上图，`i`必然是在`po`右边的(obviously)。但我们更关注的是，`i`是在`MaxRight`的左边还是右边。我们分情况来讨论。

1）当`i`在`MaxRight`的左边

情况1)可以用下图来刻画：

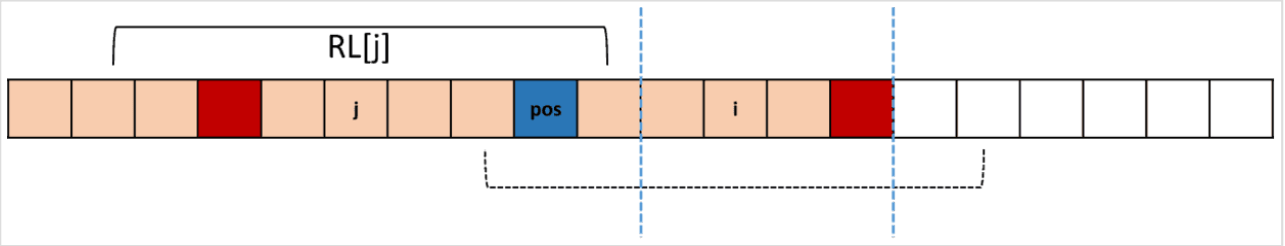


我们知道，图中两个红色块之间（包括红色块）的串是回文的；并且以`i`为对称轴的回文串，是与红色块间的回文串有所重叠的。我们找到`i`关于`pos`的对称位置`j`，这个`j`对应的`RL[j]`我们是已经算过的。根据回文串的对称性，以`i`为对称轴的回文串和以`j`为对称轴的回文串，有一部分是相同的。这里又有两种细分的情况。

1. 以`j`为对称轴的回文串比较短，短到像下图这样。

这时我们知道RL[i]至少不会小于RL[j]，并且已经知道了部分的以`i`为中心的回文串，于是可以令`RL[i]=RL[j]`。但是以`i`为对称轴的回文串可能实际上更长，因此我们试着以`i`为对称轴，继续往左右两边扩展，直到左右两边字符不同，或者到达边界。

1. 以`j`为对称轴的回文串很长，这么长：



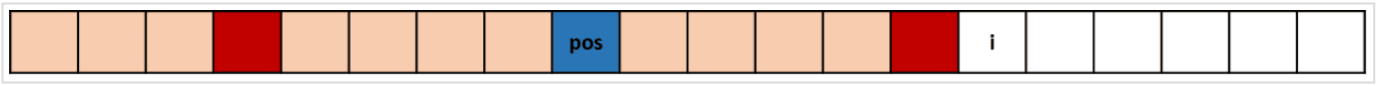
这时，我们只能确定，两条蓝线之间的部分（即不超过MaxRight的部分）是回文的，于是从这个长度开始，尝试以`i`为中心向左右两边扩展，，直到左右两边字符不同，或者到达边界。

不论以上哪种情况，之后都要尝试更新`MaxRight`和`pos`，因为有可能得到更大的MaxRight。

具体操作如下：

```
step 1: 令RL[i]=min(RL[2*pos-i], MaxRight-i)
step 2: 以i为中心扩展回文串，直到左右两边字符不同，或者到达边界。
step 3: 更新MaxRight和pos
```

2）当`i`在`MaxRight`的右边



遇到这种情况，说明以`i`为对称轴的回文串还没有任何一个部分被访问过，于是只能从`i`的左右两边开始尝试扩展了，当左右两边字符不同，或者到达字符串边界时停止。然后更新`MaxRight`和`pos`。

(3) 算法实现

```python
#Python
def manacher(s):
    #预处理
    s='#'+'#'.join(s)+'#'

    RL=[0]*len(s)
    MaxRight=0
    pos=0
    MaxLen=0
    for i in range(len(s)):
        if i<MaxRight:
            RL[i]=min(RL[2*pos-i], MaxRight-i)
        else:
            RL[i]=1
        #尝试扩展，注意处理边界
        while i-RL[i]>=0 and i+RL[i]<len(s) and s[i-RL[i]]==s[i+RL[i]]:
            RL[i]+=1
        #更新MaxRight,pos
        if RL[i]+i-1>MaxRight:
            MaxRight=RL[i]+i-1
            pos=i
        #更新最长回文串的长度
        MaxLen=max(MaxLen, RL[i])
    return MaxLen-1
```

(4) 复杂度分析

空间复杂度：插入分隔符形成新串，占用了线性的空间大小；RL数组也占用线性大小的空间，因此空间复杂度是线性的。
时间复杂度：尽管代码里面有两层循环，通过amortized analysis我们可以得出，Manacher的时间复杂度是线性的。由于内层的循环只对尚未匹配的部分进行，因此对于每一个字符而言，只会进行一次，因此时间复杂度是**O(n)**。

4. 更多关于回文串的 fun facts（参考自维基百科）

4.1 人们在一座名为赫库兰尼姆的古城遗迹中，找到了一个好玩的拉丁语回文串：`sator arepo tenet opera rotas`。翻译成中文大概就是`一个叫做Arepo的播种者，他用力地扶（把）着车轮。这个串的每个单词首字母刚好组成了第一个单词，每个单词的第二个字母刚好组成了第二个单词...于是乎，如果写出酱紫，你会发现上下左右四个方向读起来是一样的。这个串被称为 Sator Square.

4.2 本文开头给出的单词`tattarrattat`，出现在爱尔兰作家詹姆斯·乔伊斯的小说《尤利西斯》，是敲门的意思。吉尼斯纪录的最长回文英文单词是`detartrated`，是个化学术语。另外，还有些已出版的英文回文小说（你们歪果仁真会玩），比如*Satire: Veritas*，*Dr Awkward & Olson in Oslo*等。

2015.11.9 更新。

可以采用动态规划，列举回文串的起点或者终点来解最长回文串问题，无需讨论串长度的奇偶性。看下面的扎瓦代码，容易理解。

```
public int longestPalindrome(String s) {
 int n=s.length();
 boolean[][] pal=new boolean[n][n];
 //pal[i][j] 表示s[i...j]是否是回文串
 int maxLen=0;
 for (int i=0;i<n;i++){  // i作为终点
     int j=i;    //j作为起点
     while (j>=0){
         if (s.charAt(j)==s.charAt(i)&&(i-j<2||pal[j+1][i-1])){
             pal[j][i]=true;
            maxLen=Math.max(maxLen, i-j+1);
         }
         j--;
     }
 }
 return maxLen;
}
```