后 缀 数 组

安徽省芜湖市第一中学　许智磊

【摘要】

本文介绍后缀数组的基本概念、方法以及应用。

首先介绍 O(nlogn)复杂度构造后缀数组的倍增算法，接着介绍了配合后缀数组的最长公共前缀 LCP（Longest Common Prefix）的计算方法，并给出一个线性时间内计算 height 数组（记录跨度为 1 的 LCP 值的数组）的算法。为了让读者对如何运用后缀数组有一个感性认识，还介绍了两个应用后缀数组的例子：多模式串的模式匹配（给出每次匹配 O(m+logn)时间复杂度的算法）以及求最长回文子串（给出 O(nlogn)时间复杂度的算法）。最后对后缀数组和后缀树作了一番比较。

【关键字】

字符串　后缀　k-前缀比较关系
后缀数组　名次数组　后缀树　倍增算法　基数排序
最长公共前缀　RMQ 问题　模式匹配　回文串　最长回文子串

【正文】

在字符串处理当中，后缀树和后缀数组都是非常有力的工具，其中后缀树大家了解得比较多，关于后缀数组则很少见于国内的资料。其实后缀数组是后缀树的一个非常精巧的替代品，它比后缀树容易编程实现，能够实现后缀树的很多功能而时间复杂度也不太逊色，并且，它比后缀树所占用的空间小很多。可以说，在信息学竞赛中后缀数组比后缀树要更为实用。因此在本文中笔者想介绍一下后缀数组的基本概念、构造方法，以及配合后缀数组的最长公共前缀数组的构造方法，最后结合一些例子谈谈后缀数组的应用。

基本概念

首先明确一些必要的定义：

字符集　一个字符集Σ是一个建立了全序关系的集合，也就是说，Σ中的任意两个不同的元素 α 和 β 都可以比较大小，要么 α<β，要么 β<α（也就是 α>β）。字符集Σ中的元素称为字符。

字符串　一个字符串 S 是将 n 个字符顺次排列形成的数组，n 称为 S 的长度，表示为 len(S)。S 的第 i 个字符表示为 S[i]。

子串　字符串 S 的子串 S[i..j]，i≤j，表示 S 串中从 i 到 j 这一段，也就是顺次排列 S[i],S[i+1],...,S[j]形成的字符串。

后缀　后缀是指从某个位置 i 开始到整个串末尾结束的一个特殊子串。字符串 S 的从 i 开头的后缀表示为 Suffix(S,i)，也就是 Suffix(S,i)=S[i..len(S)]。

关于字符串的大小比较，是指通常所说的“字典顺序”比较，也就是对于两个字符串 u、v，令 i 从 1 开始顺次比较 u[i]和 v[i]，如果 u[i]=v[i]则令 i 加 1，否则若 u[i]<v[i]则认为 u<v，u[i]>v[i]则认为 u>v（也就是 v<u），比较结束。如果 i>len(u)或者 i>len(v)仍比较出结果，那么若 len(u)<len(v)则认为 u<v，若 len(u)=len(v)则认为 u=v，若 len(u)>len(v)则 u>v。

从字符串的大小比较的定义来看，S 的两个开头位置不同的后缀 u 和 v 进行比较的结果不可能是相等，因为 u=v 的必要条件 len(u)=len(v)在这里不可能满足。

下面我们约定一个字符集Σ和一个字符串 S，设 len(S)=n，且 S[n]='$'，也就是说 S 以一个特殊字符'$'结尾，并且'$'小于Σ中的任何一个字符。除了 S[n]之外，S 中的其他字符都属于Σ。对于约定的字符串 S，从位置 i 开头的后缀直接写成 Suffix(i)，省去参数 S。

后缀数组　后缀数组 SA 是一个一维数组，它保存 1..n 的某个排列 SA[1],SA[2],...SA[n]，并且保证 Suffix(SA[i])<Suffix(SA[i+1]),1≤i<n。也就是将 S 的 n 个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入 SA 中。

名次数组　名次数组 $Rank=SA^{-1}$，也就是说若 SA[i]=j，则 Rank[j]=i，不难看出 Rank[i]保存的是 Suffix(i)在所有后缀中从小到大排列的“名次”。

构造方法

如何构造后缀数组呢？最直接最简单的方法当然是把 S 的后缀都看作一些普通的字符串，按照一般字符串排序的方法对它们从小到大进行排序。

不难看出，这种做法是很笨拙的，因为它没有利用到各个后缀之间的有机联系，所以它的效率不可能很高。即使采用字符串排序中比较高效的 Multi-key Quick Sort，最坏情况的时间复杂度仍然是 $O(n^2)$的，不能满足我们的需要。

下面介绍倍增算法(Doubling Algorithm)，它正是充分利用了各个后缀之间的联系，将构造后缀数组的最坏时间复杂度成功降至 O(nlogn)。

对一个字符串 u，我们定义 u 的 k-前缀

$$u^k = \begin{cases} u[1..k] & , len(u) \geq k \\ u & , len(u) < k \end{cases}$$

定义 k-前缀比较关系$<_k$、$=_k$和$\leq_k$：

设两个字符串 u 和 v，

$u<_k v$　当且仅当　$u^k<v^k$

$u=_k v$　当且仅当　$u^k=v^k$

$u\leq_k v$　当且仅当　$u^k\leq v^k$

直观地看这些加了一个下标 k 的比较符号的意义就是对两个字符串的前 k 个字符进行字典序比较，特别的一点就是在作大于和小于的比较时如果某个字符串的长度不到 k 也没有关系，只要能够在 k 个字符比较结束之前得到第一个字符串大于或者小于第二个字符串就可以了。

根据前缀比较符的性质我们可以得到以下的非常重要的性质：

性质 1.1　对 k≥n，Suffix(i)$<_k$Suffix(j)　等价于　Suffix(i)<Suffix(j)。

性质 1.2　Suffix(i)=$_{2k}$Suffix(j)等价于

Suffix(i)=$_k$Suffix(j) 且 Suffix(i+k)=$_k$Suffix(j+k)。

性质 1.3　Suffix(i)$<_{2k}$Suffix(j) 等价于

Suffix(i)$<_k$S(j) 或 (Suffix(i)=$_k$Suffix(j) 且 Suffix(i+k)$<_k$Suffix(j+k))。

这里有一个问题，当 i+k>n 或者 j+k>n 的时候 Suffix(i+k)或 Suffix(j+k)是无明确定义的表达式，但实际上不需要考虑这个问题，因为此时 Suffix(i)或者 Suffix(j)的长度不超过 k，也就是说它们的 k-前缀以'$'结尾，于是 k-前缀比较的结果不可能相等，也就是说前 k 个字符已经能够比出大小，后面的表达式自然可以忽略，这也就看出我们规定 S 以'$'结尾的特殊用处了。

定义 **k-后缀数组** SA_k 保存 1..n 的某个排列 $SA_k[1],SA_k[2],\ldots SA_k[n]$使得 Suffix($SA_k$[i]) $\leq_k$Suffix(SA_k[i+1]),1≤i<n。也就是说对所有的后缀在 k-前缀比较关系下从小到大排序，并且把排序后的后缀的开头位置顺次放入数组 SA_k 中。

定义 **k-名次数组** $Rank_k$，$Rank_k$[i]代表 Suffix(i)在 k-前缀关系下从小到大的“名次”，也就是 1 加上满足 Suffix(j)$<_k$Suffix(i)的 j 的个数。通过 SA_k 很容易在 O(n)的时间内求出 $Rank_k$。

假设我们已经求出了 SA_k 和 $Rank_k$，那么我们可以很方便地求出 SA_{2k} 和 $Rank_{2k}$，因为根据**性质 1.2** 和 **1.3**，2k-前缀比较关系可以由常数个 k-前缀比较关系组合起来等价地表达，而 $Rank_k$ 数组实际上给出了在常数时间内进行$<_k$ 和$=_k$比较的方法，即：

Suffix(i)$<_k$Suffix(j)　当且仅当　$Rank_k$[i]<$Rank_k$[j]

Suffix(i)=$_k$Suffix(j)　当且仅当　$Rank_k$[i]=$Rank_k$[j]

因此，比较 Suffix(i)和 Suffix(j)在 k-前缀比较关系下的大小可以在常数时间内完成，于是对所有的后缀在$\leq_k$ 关系下进行排序也就和一般的排序没有什么区别了，它实际上就相当于每个 Suffix(i)有一个主关键字 $Rank_k$[i]和一个次关键字 $Rank_k$[i+k]。如果采用快速排序之类 O(nlogn)的排序，那么从 SA_k 和 $Rank_k$ 构造

出 SA_{2k} 的复杂度就是 O(nlogn)。更聪明的方法是采用基数排序，复杂度为 O(n)。

求出 SA_{2k} 之后就可以在 O(n)的时间内根据 SA_{2k} 构造出 $Rank_{2k}$。因此，从 SA_k 和 $Rank_k$ 推出 SA_{2k} 和 $Rank_{2k}$ 可以在 O(n)时间内完成。

下面只有一个问题需要解决：如何构造出 SA_1 和 $Rank_1$。这个问题非常简单：因为 $<_1$，$=_1$ 和 $\leq_1$ 这些运算符实际上就是对字符串的第一个字符进行比较，所以只要把每个后缀按照它的第一个字符进行排序就可以求出 SA_1，不妨就采用快速排序，复杂度为 O(nlogn)。

于是，可以在 O(nlogn)的时间内求出 SA_1 和 $Rank_1$。

求出了 SA_1 和 $Rank_1$，我们可以在 O(n)的时间内求出 SA_2 和 $Rank_2$，同样，我们可以再用 O(n)的时间求出 SA_4 和 $Rank_4$，这样，我们依次求出：

SA_2 和 $Rank_2$，SA_4 和 $Rank_4$，SA_8 和 $Rank_8$，……直到 SA_m 和 $Rank_m$，其中 $m=2^k$ 且 $m\geq n$。而根据**性质 1.1**，SA_m 和 SA 是等价的。这样一共需要进行 logn 次 O(n)的过程，因此

可以在 O(nlogn)的时间内计算出后缀数组 SA 和名次数组 Rank。

最长公共前缀

现在一个字符串 S 的后缀数组 SA 可以在 O(nlogn)的时间内计算出来。利用 SA 我们已经可以做很多事情，比如在 O(mlogn)的时间内进行模式匹配，其中 m,n 分别为模式串和待匹配串的长度。但是要想更充分地发挥后缀数组的威力，我们还需要计算一个辅助的工具——**最长公共前缀**（Longest Common Prefix）。

对两个字符串 u,v 定义函数 $lcp(u,v)=max\{i|u=_i v\}$，也就是从头开始顺次比较 u 和 v 的对应字符，对应字符持续相等的最大位置，称为这两个字符串的**最长公共前缀**。

对正整数 i,j 定义 LCP(i,j)=lcp(Suffix(SA[i]),Suffix(SA[j])，其中 i,j 均为 1 至 n 的整数。LCP(i,j)也就是后缀数组中第 i 个和第 j 个后缀的最长公共前缀的长度。

关于 LCP 有两个显而易见的性质：

性质 2.1　LCP(i,j)=LCP(j,i)

性质 2.2　LCP(i,i)=len(Suffix(SA[i]))=n-SA[i]+1

这两个性质的用处在于，我们计算 LCP(i,j)时只需要考虑 i<j 的情况，因为 i>j 时可交换 i,j，i=j 时可以直接输出结果 n-SA[i]+1。

直接根据定义，用顺次比较对应字符的方法来计算 LCP(i,j)显然是很低效的，时间复杂度为 O(n)，所以我们必须进行适当的预处理以降低每次计算 LCP 的复杂度。

经过仔细分析，我们发现 LCP 函数有一个非常好的性质：

设 i<j，则 LCP(i,j)=min{LCP(k-1,k)|i+1≤k≤j}　（**LCP Theorem**）

要证明 **LCP Theorem**，首先证明 **LCP Lemma**:

对任意 1≤i<j<k≤n，LCP(i,k)=min{LCP(i,j),LCP(j,k)}

证明：设 p=min{LCP(i,j),LCP(j,k)}，则有 LCP(i,j)≥p,LCP(j,k)≥p。

设 Suffix(SA[i])=u,Suffix(SA[j])=v,Suffix(SA[k])=w。
由 $u=_{LCP(i,j)}v$ 得 $u=_p v$；同理 $v=_p w$。
于是 $Suffix(SA[i])=_p Suffix(SA[k])$，即 $LCP(i,k)\geq p$。　**(1)**

又设 $LCP(i,k)=q>p$，则
u[1]=w[1],u[2]=w[2],...u[q]=w[q]。
而 $\min\{LCP(i,j),LCP(j,k)\}=p$ 说明 $u[p+1]\neq v[p+1]$或 $v[p+1]\neq w[q+1]$，
设 u[p+1]=x,v[p+1]=y,w[p+1]=z，显然有 $x\leq y\leq z$，又由 $p<q$ 得 $p+1\leq q$，应该有 x=z，也就是 x=y=z，这与 $u[p+1]\neq v[p+1]$或 $v[p+1]\neq w[q+1]$矛盾。
于是，$q>p$ 不成立，即 $LCP(i,k)\leq p$。　**(2)**
综合**(1)**,**(2)**知 $LCP(i,k)=p=\min\{LCP(i,j),LCP(j,k)\}$，**LCP Lemma** 得证。

于是 **LCP Theorem** 可以证明如下：
当 j-i=1 和 j-i=2 时，显然成立。
设 j-i=m 时 **LCP Theorem** 成立，当 j-i=m+1 时，
由 **LCP Lemma** 知 $LCP(i,j)=\min\{LCP(i,i+1),LCP(i+1,j)\}$，
因 $j-(i+1)\leq m$，$LCP(i+1,j)=\min\{LCP(k-1,k)|i+2\leq k\leq j\}$，故当 j-i=m+1 时，仍有
$LCP(i,j)=\min\{LCP(i,i+1),\min\{LCP(k-1,k)|i+2\leq k\leq j\}\}=\min\{LCP(k-1,k\}|i+1\leq k\leq j)$
根据数学归纳法，**LCP Theorem** 成立。

根据 **LCP Theorem** 得出必然的一个推论：
LCP Corollary　对 $i\leq j<k$，$LCP(j,k)\geq LCP(i,k)$。

定义一维数组 height，令 $height[i]=LCP(i-1,i)$，$1<i\leq n$，并设 height[1]=0。
由 **LCP Theorem**，$LCP(i,j)=\min\{height[k]|i+1\leq k\leq j\}$，也就是说，计算 LCP(i,j)等同于询问一维数组 height 中下标在 i+1 到 j 范围内的所有元素的最小值。如果 height 数组是固定的，这就是非常经典的 RMQ（Range Minimum Query）问题。

RMQ 问题可以用线段树或静态排序树在 O(nlogn)时间内进行预处理，之后每次询问花费时间 O(logn)，更好的方法是 RMQ 标准算法，可以在 O(n)时间内进行预处理，每次询问可以在常数时间内完成。

对于一个固定的字符串 S，其 height 数组显然是固定的，只要我们能高效地求出 height 数组，那么运用 RMQ 方法进行预处理之后，每次计算 LCP(i,j)的时间复杂度就是常数级了。于是只有一个问题——如何尽量高效地算出 height 数组。

根据计算后缀数组的经验，我们不应该把 n 个后缀看作互不相关的普通字符串，而应该尽量利用它们之间的联系，下面证明一个非常有用的性质：

为了描述方便，设 h[i]=height[Rank[i]]，即 height[i]=h[SA[i]]。h 数组满足一个性质：

性质 3　对于 i>1 且 Rank[i]>1，一定有 $h[i]\geq h[i-1]-1$。

为了证明**性质 3**，我们有必要明确两个事实：

设 i<n,j<n，Suffix(i)和 Suffix(j)满足 lcp(Suffix(i),Suffix(j)>1，则成立以下两点：

Fact 1　Suffix(i)<Suffix(j)　等价于　Suffix(i+1)<Suffix(j+1)。

Fact 2　一定有 lcp(Suffix(i+1),Suffix(j+1))=lcp(Suffix(i),Suffix(j))-1。

看起来很神奇，但其实很自然：lcp(Suffix(i),Suffix(j))>1 说明 Suffix(i)和 Suffix(j)的第一个字符是相同的，设它为 α，则 Suffix(i)相当于 α 后连接 Suffix(i+1)，Suffix(j)相当于 α 后连接 Suffix(j+1)。比较 Suffix(i)和 Suffix(j)时，第一个字符 α 是一定相等的，于是后面就等价于比较 Suffix(i)和 Suffix(j)，因此 **Fact 1** 成立。**Fact 2** 可类似证明。

于是可以证明**性质 3**：

当 h[i-1]≤1 时，结论显然成立，因 h[i]≥0≥h[i-1]-1。

当 h[i-1]>1 时，也即 height[Rank[i-1]]>1，可见 Rank[i-1]>1，因 height[1]=0。

令 j=i-1,k=SA[Rank[j]-1]。显然有 Suffix(k)<Suffix(j)。

根据 h[i-1]=lcp(Suffix(k),Suffix(j))>1 和 Suffix(k)<Suffix(j)：

由 **Fact 2** 知 lcp(Suffix(k+1),Suffix(i))=h[i-1]-1。

由 **Fact 1** 知 Rank[k+1]<Rank[i]，也就是 Rank[k+1]≤Rank[i]-1。

于是根据 **LCP Corollary**，有

LCP(Rank[i]-1,Rank[i])≥LCP(Rank[k+1],Rank[i])
=lcp(Suffix(k+1),Suffix(i))
=h[i-1]-1

由于 h[i]=height[Rank[i]]=LCP(Rank[i]-1,Rank[i])，最终得到 h[i]≥h[i-1]-1。

根据**性质 3**，可以令 i 从 1 循环到 n 按照如下方法依次算出 h[i]：

若 Rank[i]=1，则 h[i]=0。字符比较次数为 0。

若 i=1 或者 h[i-1]≤1，则直接将 Suffix(i)和 Suffix(Rank[i]-1)从第一个字符开始依次比较直到有字符不相同，由此计算出 h[i]。字符比较次数为 h[i]+1，不超过 h[i]-h[i-1]+2。

否则，说明 i>1，Rank[i]>1，h[i-1]>1，根据**性质 3**，Suffix(i)和 Suffix(Rank[i]-1)至少有前 h[i-1]-1 个字符是相同的，于是字符比较可以从 h[i-1]开始，直到某个字符不相同，由此计算出 h[i]。字符比较次数为 h[i]-h[i-1]+2。

设 SA[1]=p，那么不难看出总的字符比较次数不超过

$$h[1]+1+\sum_{i=2}^{p-1}(h[i]-h[i-1]+2)+h[p+1]+\sum_{i=p+1}^{n}(h[i]-h[i-1]+2)$$

$$\le 1+h[p-1]+2(p-2)+h[n]+2+2(n-p)\le 4n$$

也就是说，整个算法的复杂度为 O(n)。

求出了 h 数组，根据关系式 height[i]=h[SA[i]]可以在 O(n)时间内求出 height 数组，于是

可以在 O(n)时间内求出 height 数组。

结合 RMQ 方法，在 O(n)时间和空间进行预处理之后就能做到在常数时间内计算出对任意(i,j)计算出 LCP(i,j)。

因为 lcp(Suffix(i),Suffix(j))=LCP(Rank[i],Rank[j])，所以我们也就可以在常数时间内求出 S 的任何两个后缀之间的最长公共前缀。这正是后缀数组能强有力地处理很多字符串问题的重要原因之一。

后缀数组的应用

下面结合两个例子谈谈如何运用后缀数组。

例一　多模式串的模式匹配问题

给定一个固定待匹配串 S，长度为 n，然后每次输入一个模式串 P，长度为 m，要求返回 P 在 S 中的一个匹配或者返回匹配失败。所谓匹配指某个位置 i 满足 $1\le i\le n-m+1$ 使得 S[i..(i+m-1)]=P，也即 $Suffix(i)=_m P$。

我们知道，如果只有一个模式串，最好的算法就是 KMP 算法，时间复杂度为 O(n+m)，但是如果有多个模式串，我们就要考虑做适当的预处理使得对每个模式串进行匹配所花的时间小一些。

最简单的预处理莫过于建立 S 的后缀数组（先在 S 的后面添加'$'），然后每次寻找匹配转化为用二分查找法在 SA 中找到和 P 的公共前缀最长的一个后缀，判断这个最长的公共前缀是否等于 m。

这样，每次比较 P 和一个后缀的复杂度为 O(m)，因为最坏情况下可能比较了 m 个字符。二分查找需要调用比较的次数为 O(logn)，因此总复杂度为 O(mlogn)，于是每次匹配的复杂度从 O(n+m)变为 O(mlogn)，可以说改进了不少。

可是这样仍然不能令我们满足。前面提到 LCP 可以增加后缀数组的威力，我们来试试用在这个问题上。

我们分析原始的二分查找算法，大体有以下几步：

Step 1　令 left=1,right=n,max_match=0。

Step 2　令 mid=(left+right)/2（这里“/”表示取整除法）。

Step 3　顺次比较 Suffix(SA[mid])和 P 的对应字符，找到两者的最长公共前缀 r，并判断出它们的大小关系。若 r>max_match 则令 max_match=r,ans=mid。

Step 4　若 Suffix(SA[mid])<P 则令 left=mid+1，若 Suffix(SA[mid])>P 则令 right=mid-1，若 Suffix(SA[mid])=P 则转至 Step 6。

Step 5　若 left<right 则转至 Step 2，否则至 Step 6。

Step 6　若 max_match=m 则输出 ans，否则输出“无匹配”。

注意力很快集中在 Step 3，如果能够避免每次都从头开始比较 Suffix(SA[mid])和 P 的对应字符，也许复杂度就可以进一步降低。

类似于前面求 height 数组，我们考虑利用以前求得的最长公共前缀作为比较的“基础”，避免冗余的字符比较。

在比较 Suffix(SA[mid])和 P 之前，我们先用常数时间计算 LCP(mid,ans)，然后比较 LCP(mid,ans)和 max_match：

情况一：LCP(mid,ans)<max_match，则说明 Suffix(SA[mid])和 P 的最长公共前缀就是 LCP(mid,ans)，即直接可以确定 Step 3 中的 r=LCP(mid,ans)，所以可以直接比较两者的第 r+1 个字符（结果一定不会是相等）就可以确定 Suffix(SA[mid])和 P 的大小。这种情况下，字符比较次数为 1 次。

情况二：LCP(mid,ans)≥max_match，则说明 Suffix(SA[mid])和 Suffix(SA[ans])的前 max_match 个字符一定是相同的，于是 Suffix(SA[mid])和 P 的前 max_match 个字符也是相同的，于是比较两者的对应字符可以从第 max_match+1 个开始，最后求出的 r 一定大于等于原先的 max_match，字符比较的次数为 r-max_match+1，不难看出 Step 3 执行过后 max_match 将等于 r。

设每次 Step 3 执行之后 max_match 值增加的量为Δmax。在情况一中，Δmax=0，字符比较次数为 1=Δmax+1；在情况二中，Δmax=r-max_match，字符比较次数为 r-max_match+1，也是Δmax+1。综上所述，每次 Step 3 进行字符比较的次数为Δmax+1。

总共的字符比较次数为所有的Δmax 累加起来再加上 Step 3 执行的次数。所有Δmax 累加的结果显然就是最后的 max_match 值，不会超过 len(P)=m，而 Step 3 执行的次数为 O(logn)，因此总共的字符比较次数为 O(m+logn)。而整个算法的复杂度显然和字符比较次数同阶，为 O(m+logn)。

至此，问题得到圆满解决，通过 O(nlogn)的时间进行预处理（构造后缀数组、名词数组，计算 height 数组，RMQ 预处理），之后就可以在 O(m+logn)的时间内对一个长度为 m 的模式串 P 进行模式匹配，这仅仅是在读入 P 的复杂度上附加了 logn 的复杂度，是非常优秀的。

例二　最长回文子串问题

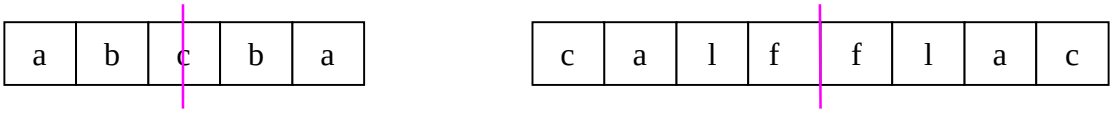
一个回文串是指满足如下性质的字符串 u：

u[i]=u[len(u)-i+1]，对所有的 1≤i≤len(u)。

也就是说，回文串 u 是关于 u 的中间位置“对称”的。

按照回文串的长度的奇偶性把回文串分为两类：长度为奇数的回文串称为奇回文串，长度为偶数的回文串称为偶回文串。

设想我们在回文串 u 的“中心位置”划一条直线，显然，对于奇回文串，这条线划在中间一个字符（u[(len(u)+1)/2]）上，而对于偶回文串，这条线划在中间的两个字符（u[len(u)/2]和 u[len(u)/2+1]）之间。以下是两个例子：



回文串里的字符是关于这条中心线对称分布的。中心线左边的字符串颠倒过来等于右边的字符串，我们称之为“反射相等”。

字符串 T 的回文子串指的是 T 的子串中同时又是回文串的那些字符串。T 的回文子串中长度最大的称为最长回文子串。类似地定义奇（偶）回文子串和最长奇（偶）回文子串。

最长回文子串问题是给定一个字符串 T，求 T 的最长回文子串，简便起见，只要求给出最长回文子串的长度即可。

下面我们分析求最长奇回文串的方法，最长偶回文串的求法是类似的。

因为每个奇回文子串一定有一个中心位置（是整个回文串的中间一个字符），这个中心位置是 T 串的某个字符，所以我们首先枚举定下它的中心位置。对于一个固定的中心位置 i，可能存在多个以 T[i]为中心的奇回文子串，但是它们满足一个性质：奇回文串在 T[i]左边的部分和右边的部分长度相等，而且关于 T[i]对称，即跟 T[i]距离相同的左右两个字符对应相等。

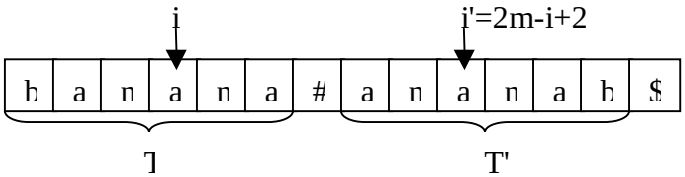
那么任何一个以 T[i]为中心的奇回文子串都可以表示为 T[i-r..i+r],r≥0。可以看出 r 越大这个以 T[i]为中心的奇回文串也就越长。因此只要能够找到最大的一个 r 使得 T[i-r..i-1]和 T[i+1..i+r]关于 T[i]对称，也就求出了以 T[i]为中心的奇回文子串中最长的一个（可以看出，也一定只有一个）。如果枚举 i 从 1 到 len(T)，求出以每个 T[i]为中心的奇回文子串中的最长者，这些最长者里面长度最大的一个也就是整个 T 串的最长奇回文子串。

如何求以 T[i]为中心的奇回文子串中的最长者呢？首先，当 r=0 时，T[i]单个字符本身构成一个奇回文子串，它的长度为 1。下面我们考虑将 r 不断增加，假设当 r=k 时 T[i-r..i+r]构成奇回文子串，也就是说 T[i-r..i-1]和 T[i+1..i+r]反射相等，当 r=k+1 时，我们比较 T[i-(k+1)]与 T[i+k+1]这两个字符，若相等则说明 T[i-(k+1)..i-1]与 T[i+1..i+k+1]是关于 T[i]对称的（因为根据假设 T[i-k..i-1]与 T[i+1..i+k]已经关于 T[i]对称），于是 r 可以扩展到 k+1。如果 T[i-(k+1)]和 T[i+k+1]不同，则说明 T[i-(k+1)..i-1]和 T[i+1..i+k+1]不对称，并且对任何 r'>k+1，T[i-r'..i-1]和 T[i+1..i+r']也不可能关于 T[i]对称了，所以 r 最大只能到 k。

我们把 r 递增的过程称为向两边扩展，扩展一次就可以把以 T[i]为中心的奇回文子串的长度加 2。最后 r 扩展到的最大值决定了以 T[i]为中心的奇回文子串中的最长者的长度（为 2r+1）。

设 len(T)=m，如果用依次比较对应字符的方法来求向两边扩展的最大值，则最多可能比较 m-1 个字符。由于要枚举每个位置作为中心向两边扩展，所以最坏情况下总的复杂度可以达到 $O(m^2)$，不很理想。下面优化算法的核心部分——以一个位置为中心求向两边扩展的最大值。

在 T 串的末尾添加一个特殊字符'#'，规定它不等于 T 的任何一个字符，然后把 T 串颠倒，接在'#'后，在 T'串后再添加特殊字符'$'，规定它小于前面的任何一个字符，拼接后形成的串称为 S 串。



不难看出 T 串中任何一个字符都可在 T'中对称地找到一个相同的字符。如果都用 S 里的字符来表示，S[1..m]是 T 串，S[m+2..2m+1]是 T'串，则每个 S[i](1≤i≤m)关于'#'对称的字符是 S[2m-i+2]。这样原先 T 串里面的一个子串 S[i..j](1≤i≤j≤m)关于'#'也可以对称地找到一个反射相等的子串 S[2m-j+2..2m-i+2]。

现在我们定下 T 串的某个位置 S[i]为中心，假设向两边扩展到了 i-r 和 i+r，那么 S[i-r..i-1]和 S[i+1..i+r]是反射相等的，S[i]可以在 T'中找到对称的字符 S[2m-i+2]，设 i'=2m-i+2，则 S[i-r..i-1]也可以在 T'中找到对称的子串 S[i'+1..i'+r]，

那么 S[i+1..i+r]和 S[i'+1..i'+r]同时与 S[i-r..i-1]反射相等，也就是说，S[i+1..i+r]=S[i'+1..i'+r]。又因为 S[i]=S[i']，故 S[i..i+r]=S[i'..i'+r]。也就是说，Suffix(i)=$_{r+1}$Suffix(i')。现在要求 r 尽量大，也就是求 max{r|Suffix(i)=$_{r+1}$Suffix(i')}，不难看出，这里 r=LCP(i,i')-1。

上面的推理还存在一个问题，即求出的 LCP(i,i')-1 还只能看作 r 的一个上界，还不能当成 r 的最大值，因为还需要证明给出 Suffix(i)和 Suffix(i')的最长公共前缀，一定可以反过来在 T 串中找到相应的以 i 为中心的回文串，这个证明与前面的推理类似，只是需要注意一点：这里利用到了'#'这个特殊字符避免了潜在的 LCP(i,i')超过实际的 r 最大值的危险。这个证明留给读者自行完成。

总之，我们已经确定求以 T[i]为中心向两边扩展的最大值等价于求 LCP(i,i')，根据前面后缀数组和 LCP 的相关内容这一步操作可以在常数时间内完成，只要我们预先花费 O(nlogn)的复杂度计算后缀数组、height 数组和进行预处理。其中 n=len(S)=2m+2。

现在每次求以一个位置 T[i]为中心的回文子串中的最长者的长度可以在常数时间内完成，我们枚举 i 从 1 到 m，依次求出所有的这些最长者，记录其中最大的一个的长度，就是所要求的最长奇回文子串的长度。由于对每个中心花费时间为常数，所以总的复杂度为 O(m)。

因此整个算法的复杂度是 O(nlogn+m)=O(2mlog(2m)+m)=O(mlogm)，是非常优秀的算法，比之前的平方级算法大为改进。

后缀数组与后缀树的比较

通过上面的两个例子相信读者已经对后缀数组的强大功能有所了解，另一种数据结构——后缀树，也可以用在这些问题中，那么后缀数组和后缀树有什么区别和联系呢？我们来比较一下：

首先，后缀数组比较容易理解，也易于编程实现，而且不像后缀树那样需要涉及到指针操作，所以调试起来比较方便。

第二，后缀数组占用的空间比后缀树要小，刚才分析中我们并没有提到空间复杂度的问题，这里简单说一下：后缀数组 SA 和名词数组 Rank 都只需要 n 个整数的空间，而在由 $Rank_k$ 计算出 SA_{2k} 的过程中需要用两个一维数组来辅助完成，各占 n 个整数的空间，滚动地进行操作，整个算法只需要这四个一维数组和常数个辅助变量，因此总的空间占用为 4n 个整数。

而后缀树通常有 2n 个以上节点，通常每个节点要两个整数（即使采用一些技巧，至少还是要保存一个整数），每个节点要有两个指针（假设采用儿子-兄弟表示方法），因此总共的空间占用至少是 4n 个指针和 2n 个整数（至少是 n 个整数）。如果采用其他方法表示树状结构，需要的空间更大。

可以看出后缀数组的空间需求比后缀树小。

最后比较它们的复杂度：

首先按照字符总数|Σ|把字符集 Σ 分为三种类型：

若|Σ|是一个常数，则称 Σ 为 **Constant** Alphabet，

若|Σ|的大小是关于 S 的长度 n 的多项式函数，则称 Σ 为 **Integer** Alphabet，

若|Σ|没有大小上的限制，则称 Σ 为 **General** Alphabet。

显然 Constant Alphbet 属于 Integer Alphabet 的一种，而 Integer Alphabet 是 General Alphabet 的一种。

构造后缀数组的复杂度与字符集无关，因为它是直接针对 General Alphabet 的算法。

对于普通方法构造后缀树，如果用儿子-兄弟方式表达树状结构，时间复杂度达到 O(n*|Σ|)，显然对于 Integer Alphabet 和 General Alphabet 都很低效，对|Σ|较大的 Constant Alphabet 也不适用。

解决的方法是用平衡二叉树来保存指向儿子的指针，这样复杂度变为 O(n*log|Σ|)。

可见后缀树在某些情况下相对后缀数组有速度上的优势，但是并不明显。

对于|Σ|很小的字符串，后缀树相比后缀数组的速度优势还是比较可观的。尤其是对于很常见的 0-1 串。

后缀数组实际上可以看作后缀树的所有叶结点按照从左到右的次序排列放入数组中形成的，所以后缀数组的用途不可能超出后缀树的范围。甚至可以说，如果不配合 LCP，后缀数组的应用范围是很狭窄的。但是 LCP 函数配合下的后缀数组就非常强大，可以完成大多数后缀树所能完成的任务，因为 LCP 函数实际上给出了任意两个叶子结点的最近公共祖先，这方面的内容大家可以自行研究。

后缀树和后缀数组都是字符串处理中非常优秀的数据结构，不能说一个肯定优于另一个，对于不同场合、不同条件的问题，我们应该灵活应用，精心选择地选择其中较为适合的一个。算法和数据结构都是死的，而运用它们的人，才是真正的主角，对经典的算法和数据结构熟练掌握并适当地运用以发挥它们最大的力量，这才是信息学研究和竞赛中最大的智慧，也是信息学竞赛的魅力所在。