信息学奥林匹克

China Nation Olympiad in Informatics

国家集训队论文

题　　目： 后缀数组——处理字符串的有力工具

作　　者： 罗穗骞

指导教师： 张学东

学　　校： 华南师范大学附属中学

完成时间： ２００９年１月

目录

后 缀 数 组

----处理字符串的有力工具

【摘要】

后缀数组是处理字符串的有力工具。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并不逊色，而且它比后缀树所占用的内存空间小很多。可以说，在信息学竞赛中后缀数组比后缀树要更为实用。本文分两部分。第一部分介绍两种构造后缀数组的方法，重点介绍如何用简洁高效的代码实现，并对两种算法进行了比较。第二部分介绍后缀数组在各种类型题目中的具体应用。

【关键字】

字符串 后缀 后缀数组 名次数组 基数排序

【正文】

一、后缀数组的实现

本节主要介绍后缀数组的两种实现方法：**倍增算法和 DC3 算法**，并对两种算法进行了比较。可能有的读者会认为这两种算法难以理解，即使理解了也难以用程序实现。本节针对这个问题，在介绍这两种算法的基础上，还给出了简洁高效的代码。其中倍增算法只有 25 行，DC3 算法只有 40 行。

1．1 基本定义

子串：字符串 S 的子串 r[i..j]，i≤j，表示 r 串中从 i 到 j 这一段，也就是顺次排列 r[i],r[i+1],...,r[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 r 的从第 i 个字符开始的后缀表示为 Suffix(i)，也就是 Suffix(i)=r[i..len(r)]。

大小比较：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)仍比较不出结果，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。

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[i]保存的是 Suffix(i)在所有后缀中从小到大排列的“名次”。

简单的说，后缀数组是“排第几的是谁？”，名次数组是“你排第几？”。容易看出，后缀数组和名次数组为互逆运算。如图 1 所示。

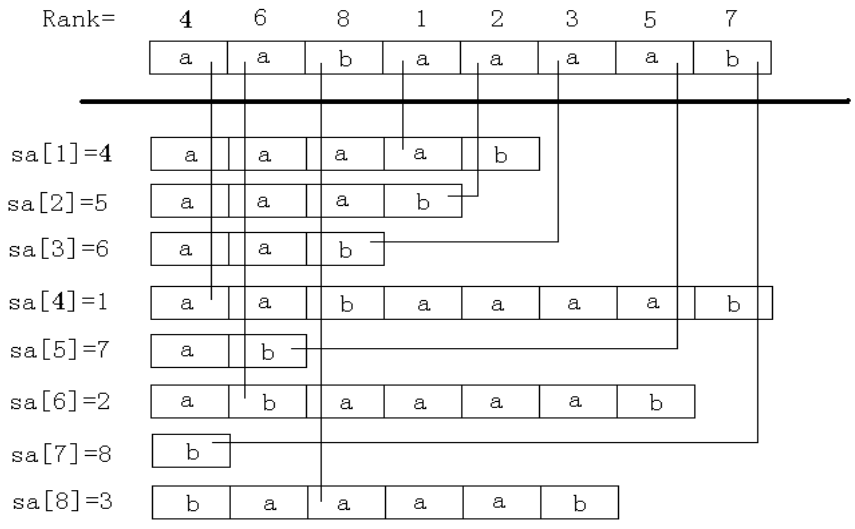

图 1

设字符串的长度为 n。为了方便比较大小，可以在字符串后面添加一个字符，这个字符没有在前面的字符中出现过，而且比前面的字符都要小。在求出名次数组后，可以仅用 O(1)的时间比较任意两个后缀的大小。在求出后缀数组或名次数组中的其中一个以后，便可以用 O(n)的时间求出另外一个。任意两个后缀如果直接比较大小，最多需要比较字符 n 次，也就是说最迟在比较第 n 个字符时一定能分出“胜负”。

１．２倍增算法

倍增算法的主要思路是：用倍增的方法对每个字符开始的长度为 2^k 的子字符串进行排序，求出排名，即 rank 值。k 从 0 开始，每次加 1，当 2^k 大于 n 以后，每个字符开始的长度为 2^k 的子字符串便相当于所有的后缀。并且这些子字符串都一定已经比较出大小，即 rank 值中没有相同的值，那么此时的 rank 值就是最后的结果。每一次排序都利用上次长度为 2^{k-1} 的字符串的 rank 值，那么长度为 2^k 的字符串就可以用两个长度为 2^{k-1} 的字符串的排名作为关键字表示，然后进行基数排序，便得出了长度为 2^k 的字符串的 rank 值。以字符串“aabaaaab”为例，整个过程如图 2 所示。其中 x、y 是表示长度为 2^k 的字符串的两个关键字。

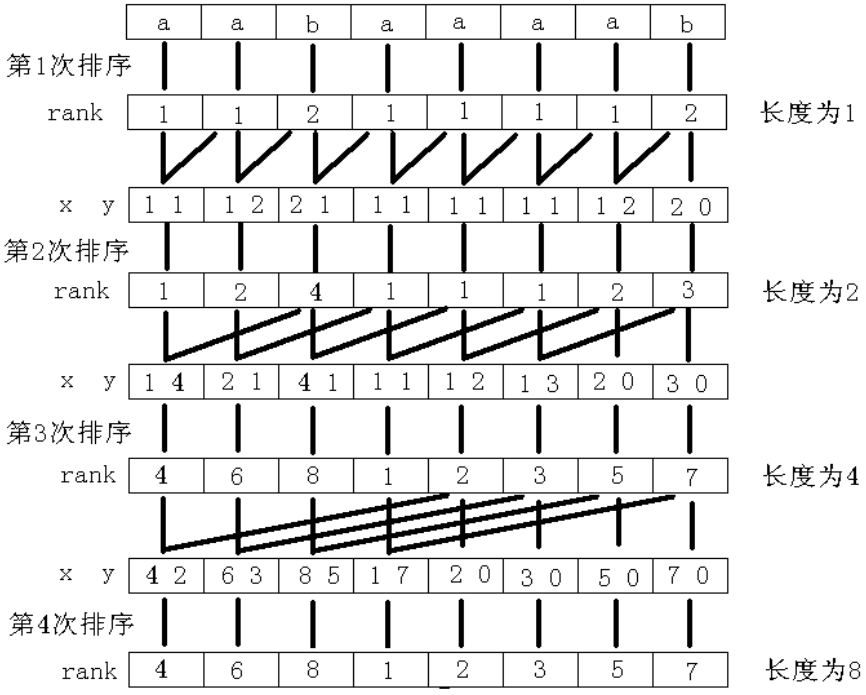

图2

具体实现：

```
int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
int cmp(int *r,int a,int b,int l)
{return r[a]==r[b]&&r[a+l]==r[b+l];}
void da(int *r,int *sa,int n,int m)
{
     int i,j,p,*x=wa,*y=wb,*t;
     for(i=0;i<m;i++) ws[i]=0;
     for(i=0;i<n;i++) ws[x[i]=r[i]]++;
     for(i=1;i<m;i++) ws[i]+=ws[i-1];
     for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
     for(j=1,p=1;p<n;j*=2,m=p)
     {
       for(p=0,i=n-j;i<n;i++) y[p++]=i;
       for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
       for(i=0;i<n;i++) wv[i]=x[y[i]];
       for(i=0;i<m;i++) ws[i]=0;
       for(i=0;i<n;i++) ws[wv[i]]++;
       for(i=1;i<m;i++) ws[i]+=ws[i-1];
       for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
       for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
       x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
     }
     return;
}
```

待排序的字符串放在 r 数组中，从 r[0]到 r[n-1]，长度为 n，且最大值小于 m。为了函数操作的方便，约定除 r[n-1]外所有的 r[i]都大于 0, r[n-1]=0。函数结束后，结果放在 sa 数组中，从 sa[0]到 sa[n-1]。

函数的第一步，要对长度为 1 的字符串进行排序。一般来说，在字符串的题目中，r 的最大值不会很大，所以这里使用了基数排序。如果 r 的最大值很大，那么把这段代码改成快速排序。代码：

```
for(i=0;i<m;i++) ws[i]=0;
for(i=0;i<n;i++) ws[x[i]=r[i]]++;
for(i=1;i<m;i++) ws[i]+=ws[i-1];
for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
```

这里 x 数组保存的值相当于是 rank 值。下面的操作只是用 x 数组来比较字符的大小，所以没有必要求出当前真实的 rank 值。

接下来进行若干次基数排序，在实现的时候，这里有一个小优化。基数排序要分两次，第一次是对第二关键字排序，第二次是对第一关键字排序。对第二关键字排序的结果实际上可以利用上一次求得的 sa 直接算出，没有必要再算一次。代码：

```
for(p=0,i=n-j;i<n;i++) y[p++]=i;
for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
```

其中变量 j 是当前字符串的长度，数组 y 保存的是对第二关键字排序的结果。然后要对第一关键字进行排序，代码：

```
for(i=0;i<n;i++) wv[i]=x[y[i]];
for(i=0;i<m;i++) ws[i]=0;
for(i=0;i<n;i++) ws[wv[i]]++;
for(i=1;i<m;i++) ws[i]+=ws[i-1];
for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
```

这样便求出了新的 sa 值。在求出 sa 后，下一步是计算 rank 值。这里要注意的是，可能有多个字符串的 rank 值是相同的，所以必须比较两个字符串是否完全相同，y 数组的值已经没有必要保存，为了节省空间，这里用 y 数组保存 rank 值。这里又有一个小优化，将 x 和 y 定义为指针类型，复制整个数组的操作可以用交换指针的值代替，不必将数组中值一个一个的复制。代码：

```
for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
```

其中 cmp 函数的代码是：

```
int cmp(int *r,int a,int b,int l)
{return r[a]==r[b]&&r[a+l]==r[b+l];}
```

这里可以看到规定 r[n-1]=0 的好处，如果 r[a]=r[b]，说明以 r[a]或 r[b]开头的长度为 l 的字符串肯定不包括字符 r[n-1],所以调用变量 r[a+l]和 r[b+l]不会导致数组下标越界，这样就不需要做特殊判断。执行完上面的代码后，rank 值保存在 x 数组中，而变量 p 的结果实际上就是不同的字符串的个数。这里可以加一个小优化，如果 p 等于 n，那么函数可以结束。因为在当前长度的字符串中，已经没有相同的字符串，接下来的排序不会改变 rank 值。例如图 1 中的第四次排序，实际上是没有必要的。对上面的两段代码，循环的初始赋值和终止条件可以这样写：

```
for(j=1,p=1;p<n;j*=2,m=p) {……}
```

在第一次排序以后，rank 数组中的最大值小于 p，所以让 m=p。

整个倍增算法基本写好，代码大约 25 行。

算法分析：

倍增算法的时间复杂度比较容易分析。每次基数排序的时间复杂度为 O(n)，排序的次数决定于最长公共子串的长度，最坏情况下，排序次数为 logn 次，所以总的时间复杂度为 O(nlogn)。

1．3DC3 算法

DC3 算法分 3 步：

（1）、先将后缀分成两部分，然后对第一部分的后缀排序。

将后缀分成两部分，第一部分是后缀 k（k 模 3 不等于 0），第二部分是后缀 k（k 模 3 等于 0）。先对所有起始位置模 3 不等于 0 的后缀进行排序，即对 suffix(1), suffix(2), suffix(4), suffix(5), suffix(7)……进行排序。做法是将 suffix(1)和 suffix(2)连接，如果这两个后缀的长度不是 3 的倍数，那先各自在末尾添 0 使得长度都变成 3 的倍数。然后每 3 个字符为一组，进行基数排序，将每组字符“合并”成一个新的字符。然后用递归的方法求这个新的字符串的后缀数组。如图 3 所示。在得到新的字符串的 sa 后，便可以计算出原字符

串所有起始位置模 3 不等于 0 的后缀的 sa。要注意的是，原字符串必须以一个最小的且前面没有出现过的字符结尾，这样才能保证结果正确（请读者思考为什么）。

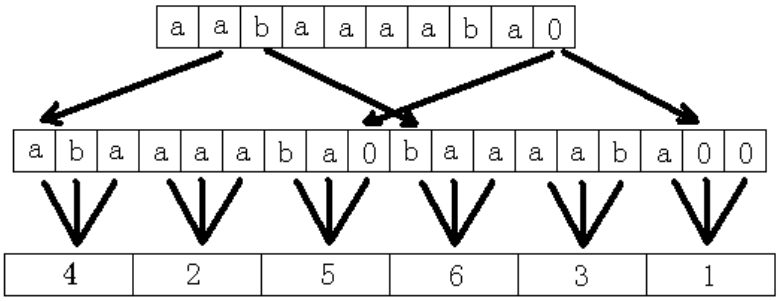

图3

（2）、利用（1）的结果，对第二部分的后缀排序。

剩下的后缀是起始位置模 3 等于 0 的后缀，而这些后缀都可以看成是一个字符加上一个在（1）中已经求出 rank 的后缀，所以只要一次基数排序便可以求出剩下的后缀的 sa。

（3）、将（1）和（2）的结果合并，即完成对所有后缀排序。

这个合并操作跟合并排序中的合并操作一样。每次需要比较两个后缀的大小。分两种情况考虑，第一种情况是 suffix(3*i)和 suffix(3*j+1)的比较，可以把 suffix(3*i)和 suffix(3*j+1)表示成：

suffix(3*i) = r[3*i] + suffix(3*i+1)

suffix(3*j+1) = r[3*j+1] + suffix(3*j+2)

其中 suffix(3*i+1)和 suffix(3*j+2)的比较可以利用(2)的结果快速得到。第二种情况是 suffix(3*i)和 suffix(3*j+2)的比较，可以把 suffix(3*i)和 suffix(3*j+2)表示成：

suffix(3*i) = r[3*i] + r[3*i+1] + suffix(3*i+2)

suffix(3*j+2) = r[3*j+2] + r[3*j+3] + suffix(3*(j+1)+1)

同样的道理，suffix(3*i+2)和 suffix(3*(j+1)+1) 的比较可以利用（2）的结果快速得到。所以每次的比较都可以高效的完成，这也是之前要每 3 个字符合并，而不是每 2 个字符合并的原因。

具体实现：

```
#define F(x) ((x)/3+((x)%3==1?0:tb))
#define G(x) ((x)<tb?(x)*3+1:((x)-tb)*3+2)
int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
int c0(int *r,int a,int b)
{return r[a]==r[b]&&r[a+1]==r[b+1]&&r[a+2]==r[b+2];}
int c12(int k,int *r,int a,int b)
{if(k==2) return r[a]<r[b]||r[a]==r[b]&&c12(1,r,a+1,b+1);
else return r[a]<r[b]||r[a]==r[b]&&wv[a+1]<wv[b+1];}
void sort(int *r,int *a,int *b,int n,int m)
{
    int i;
    for(i=0;i<n;i++) wv[i]=r[a[i]];
    for(i=0;i<m;i++) ws[i]=0;
    for(i=0;i<n;i++) ws[wv[i]]++;
    for(i=1;i<m;i++) ws[i]+=ws[i-1];
    for(i=n-1;i>=0;i--) b[--ws[wv[i]]]=a[i];
    return;
}
void dc3(int *r,int *sa,int n,int m)
{
    int i,j,*rn=r+n,*san=sa+n,ta=0,tb=(n+1)/3,tbc=0,p;
    r[n]=r[n+1]=0;
    for(i=0;i<n;i++) if(i%3!=0) wa[tbc++]=i;
    sort(r+2,wa,wb,tbc,m);
    sort(r+1,wb,wa,tbc,m);
    sort(r,wa,wb,tbc,m);
    for(p=1,rn[F(wb[0])]=0,i=1;i<tbc;i++)
    rn[F(wb[i])]=c0(r,wb[i-1],wb[i])?p-1:p++;
    if(p<tbc) dc3(rn,san,tbc,p);
```

```
    else for(i=0;i<tbc;i++) san[rn[i]]=i;
    for(i=0;i<tbc;i++) if(san[i]<tb) wb[ta++]=san[i]*3;
    if(n%3==1) wb[ta++]=n-1;
    sort(r,wb,wa,ta,m);
    for(i=0;i<tbc;i++) wv[wb[i]=G(san[i])]=i;
    for(i=0,j=0,p=0;i<ta && j<tbc;p++)
    sa[p]=c12(wb[j]%3,r,wa[i],wb[j])?wa[i++]:wb[j++];
    for(;i<ta;p++) sa[p]=wa[i++];
    for(;j<tbc;p++) sa[p]=wb[j++];
    return;
}
```

各个参数的作用和前面的倍增算法一样，不同的地方是 r 数组和 sa 数组的大小都要是 3*n，这为了方便下面的递归处理,不用每次都申请新的内存空间。函数中用到的变量：

```
int i,j,*rn=r+n,*san=sa+n,ta=0,tb=(n+1)/3,tbc=0,p;
```

rn 数组保存的是（1）中要递归处理的新字符串，san 数组是新字符串的 sa。变量 ta 表示起始位置模 3 为 0 的后缀个数，变量 tb 表示起始位置模 3 为 1 的后缀个数，已经直接算出。变量 tbc 表示起始位置模 3 为 1 或 2 的后缀个数。先按（1）中所说的用基数排序把 3 个字符“合并”成一个新的字符。为了方便操作，先将 r[n]和 r[n+1]赋值为 0。

代码：

```
r[n]=r[n+1]=0;
for(i=0;i<n;i++) if(i%3!=0) wa[tbc++]=i;
sort(r+2,wa,wb,tbc,m);
sort(r+1,wb,wa,tbc,m);
sort(r,wa,wb,tbc,m);
```

其中 sort 函数的作用是进行基数排序。代码：

```
void sort(int *r,int *a,int *b,int n,int m)
```

```
{
    int i;
    for(i=0;i<n;i++) wv[i]=r[a[i]];
    for(i=0;i<m;i++) ws[i]=0;
    for(i=0;i<n;i++) ws[wv[i]]++;
    for(i=1;i<m;i++) ws[i]+=ws[i-1];
    for(i=n-1;i>=0;i--) b[--ws[wv[i]]]=a[i];
    return;
}
```

基数排序结束后，新的字符的排名保存在 wb 数组中。

跟倍增算法一样，在基数排序以后，求新的字符串时要判断两个字符组是否完全相同。代码：

```
for(p=1,rn[F(wb[0])]=0,i=1;i<tbc;i++)
rn[F(wb[i])]=c0(r,wb[i-1],wb[i])?p-1:p++;
```

其中 F(x)是计算出原字符串的 suffix(x)在新的字符串中的起始位置，c0 函数是比较是否完全相同，在开头加一段代码：

```
#define F(x) ((x)/3+((x)%3==1?0:tb))
inline int c0(int *r,int a,int b)
{return r[a]==r[b]&&r[a+1]==r[b+1]&&r[a+2]==r[b+2];}
```

接下来是递归处理新的字符串，这里和倍增算法一样，可以加一个小优化，如果 p 等于 tbc，那么说明在新的字符串中没有相同的字符，这样可以直接求出 san 数组，并不用递归处理。代码：

```
if(p<tbc) dc3(rn,san,tbc,p);
else for(i=0;i<tbc;i++) san[rn[i]]=i;
```

然后是第（2）步，将所有起始位置模 3 等于 0 的后缀进行排序。其中对第二关键字的排序结果可以由新字符串的 sa 直接计算得到，没有必要再排一次。代码：

```
for(i=0;i<tbc;i++) if(san[i]<tb) wb[ta++]=san[i]*3;
if(n%3==1) wb[ta++]=n-1;
```

```
sort(r,wb,wa,ta,m);
for(i=0;i<tbc;i++) wv[wb[i]=G(san[i])]=i;
```

要注意的是，如果 n%3==1，要特殊处理 suffix(n-1)，因为在 san 数组里并没有 suffix(n)。G(x)是计算新字符串的 suffix(x)在原字符串中的位置，和 F(x)为互逆运算。在开头加一段：

```
#define G(x) ((x)<tb?(x)*3+1:((x)-tb)*3+2)。
```

最后是第（3）步，合并所有后缀的排序结果，保存在 sa 数组中。代码：

```
for(i=0,j=0,p=0;i<ta && j<tbc;p++)
sa[p]=c12(wb[j]%3,r,wa[i],wb[j])?wa[i++]:wb[j++];
for(;i<ta;p++) sa[p]=wa[i++];
for(;j<tbc;p++) sa[p]=wb[j++];
```

其中 c12 函数是按（3）中所说的比较后缀大小的函数，k=1 是第一种情况，k=2 是第二种情况。代码：

```
int c12(int k,int *r,int a,int b)
{if(k==2) return r[a]<r[b]||r[a]==r[b]&&c12(1,r,a+1,b+1);
else return r[a]<r[b]||r[a]==r[b]&&wv[a+1]<wv[b+1];}
```

整个 DC3 算法基本写好，代码大约 40 行。

算法分析：

假设这个算法的时间复杂度为 f(n)。容易看出第（1）步排序的时间为 O(n)（一般来说，m 比较小，这里忽略不计），新的字符串的长度不超过 2n/3，求新字符串的 sa 的时间为 f(2n/3)，第（2）和第（3）步的时间都是 O(n)。所以

$$f(n) = O(n) + f(2n/3)$$

$$f(n) \leqslant c\times n + f(2n/3)$$

$$f(n) \leqslant c\times n + c\times(2n/3) + c\times(4n/9) + c\times(8n/27) + \cdots\cdots \leqslant 3c\times n$$

$$\text{所以} \quad f(n) = O(n)$$

由此看出，DC3 算法是一个优秀的线性算法。

1．4 倍增算法与 DC3 算法的比较

从时间复杂度、空间复杂度、编程复杂度和实际效率等方面对倍增算法与

DC3 算法进行比较。

时间复杂度：

倍增算法的时间复杂度为 O(nlogn)，DC3 算法的时间复杂度为 O(n)。从常数上看，DC3 算法的常数要比倍增算法大。

空间复杂度：

倍增算法和 DC3 算法的空间复杂度都是 O(n)。按前面所讲的实现方法，倍增算法所需数组总大小为 6n，DC3 算法所需数组总大小为 10n。

编程复杂度：

倍增算法的源程序长度为 25 行，DC3 算法的源程序长度为 40 行。

实际效率：

测试环境：NOI-linux Pentium(R) 4 CPU 2.80GHz

N	倍增算法	DC3 算法
200000	192	140
300000	367	244
500000	750	499
1000000	1693	1248

（不包括读入和输出的时间，单位：ms）

从表中可以看出，DC3 算法在实际效率上还是有一定优势的。倍增算法容易实现，DC3 算法效率比较高，但是实现起来比倍增算法复杂一些。对于不同的题目，应当根据数据规模的大小决定使用哪个算法。

二、后缀数组的应用

本节主要介绍后缀数组在各种类型的字符串问题中的应用。各题的原题请见附件二，参考代码请见附件三。

２．１最长公共前缀

这里先介绍后缀数组的一些性质。

height 数组：定义 height[i]=suffix(sa[i-1])和 suffix(sa[i])的最长公共前缀，也就是排名相邻的两个后缀的最长公共前缀。那么对于 j 和 k，不妨设 rank[j]<rank[k],则有以下性质：

suffix(j)和 suffix(k)的最长公共前缀为 height[rank[j]+1], height[rank[j]+2], height[rank[j]+3], … ,height[rank[k]]中的最小值。

例如，字符串为“aabaaaab"，求后缀“abaaaab”和后缀“aaab”的最长公共前缀，如图 4 所示：

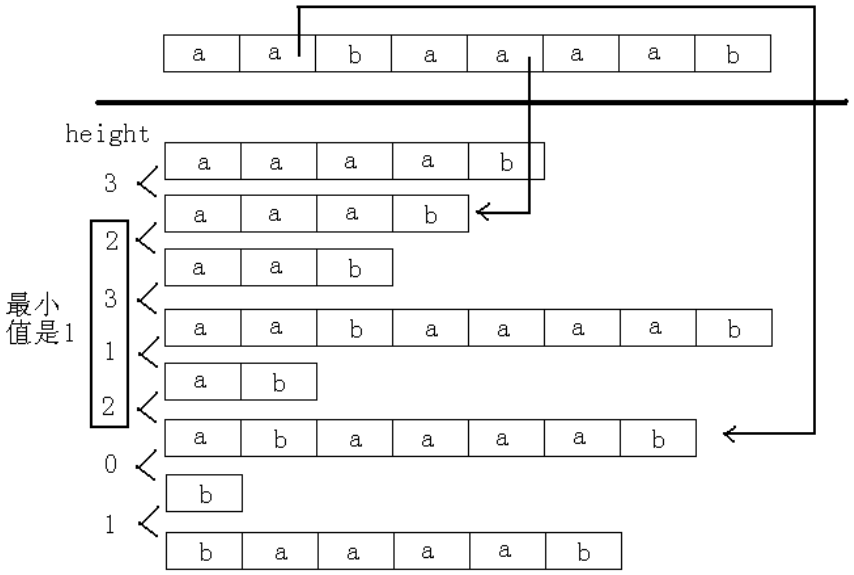

图4

那么应该如何高效的求出 height 值呢？

如果按 height[2]，height[3]，……，height[n]的顺序计算，最坏情况下时间复杂度为 $O(n^2)$。这样做并没有利用字符串的性质。定义 h[i]=height[rank[i]]，也就是 suffix(i)和在它前一名的后缀的最长公共前缀。

h 数组有以下性质：

$$h[i] \geqslant h[i-1]-1$$

证明：

设 suffix(k)是排在 suffix(i-1)前一名的后缀，则它们的最长公共前缀是 h[i-1]。那么 suffix(k+1)将排在 suffix(i)的前面（这里要求 h[i-1]>1，如果 h[i-1]≤1，原式显然成立）并且 suffix(k+1)和 suffix(i)的最长公共前缀是

h[i-1]-1，所以 suffix(i)和在它前一名的后缀的最长公共前缀至少是 h[i-1]-1。按照 h[1],h[2],……,h[n]的顺序计算,并利用 h 数组的性质，时间复杂度可以降为 O(n)。

具体实现：

实现的时候其实没有必要保存 h 数组，只须按照 h[1],h[2],……,h[n]的顺序计算即可。代码：

```
int rank[maxn],height[maxn];
void calheight(int *r,int *sa,int n)
{
    int i,j,k=0;
    for(i=1;i<=n;i++) rank[sa[i]]=i;
    for(i=0;i<n;height[rank[i++]]=k)
    for(k?k--:0,j=sa[rank[i]-1];r[i+k]==r[j+k];k++);
    return;
}
```

例 1：最长公共前缀

给定一个字符串，询问某两个后缀的最长公共前缀。

算法分析：

按照上面所说的做法，求两个后缀的最长公共前缀可以转化为求某个区间上的最小值。对于这个 RMQ 问题（如果对 RMQ 问题不熟悉，请阅读其他相关资料），可以用 O(nlogn)的时间先预处理，以后每次回答询问的时间为 O(1)。所以对于本问题，预处理时间为 O(nlogn)，每次回答询问的时间为 O(1)。如果 RMQ 问题用 O(n)的时间预处理，那么本问题预处理的时间可以做到 O(n)。

２．２单个字符串的相关问题

这类问题的一个常用做法是先求后缀数组和 height 数组，然后利用 height 数组进行求解。

２．２．１重复子串

重复子串：字符串 R 在字符串 L 中至少出现两次，则称 R 是 L 的重复子串。

例 2：可重叠最长重复子串

给定一个字符串，求最长重复子串，这两个子串可以重叠。

算法分析：

这道题是后缀数组的一个简单应用。做法比较简单，只需要求 height 数组里的最大值即可。首先求最长重复子串，等价于求两个后缀的最长公共前缀的最大值。因为任意两个后缀的最长公共前缀都是 height 数组里某一段的最小值，那么这个值一定不大于 height 数组里的最大值。所以最长重复子串的长度就是 height 数组里的最大值。这个做法的时间复杂度为 O(n)。

例 3：不可重叠最长重复子串（pku1743）

给定一个字符串，求最长重复子串，这两个子串不能重叠。

算法分析：

这题比上一题稍复杂一点。先二分答案，把题目变成判定性问题：判断是否存在两个长度为 k 的子串是相同的，且不重叠。解决这个问题的关键还是利用 height 数组。把排序后的后缀分成若干组，其中每组的后缀之间的 height 值都不小于 k。例如，字符串为“aabaaaab”，当 k=2 时，后缀分成了 4 组，如图 5 所示。

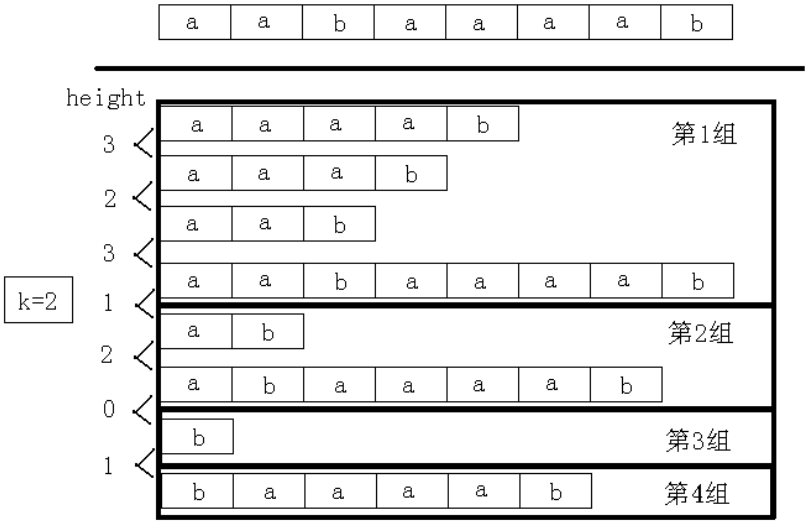

图5

容易看出，有希望成为最长公共前缀不小于 k 的两个后缀一定在同一组。然后对于每组后缀，只须判断每个后缀的 sa 值的最大值和最小值之差是否不小于 k。如果有一组满足，则说明存在，否则不存在。整个做法的时间复杂度为 O(nlogn)。本题中利用 height 值对后缀进行分组的方法很常用，请读者认真体会。

例 4：可重叠的 k 次最长重复子串（pku3261）

给定一个字符串，求至少出现 k 次的最长重复子串，这 k 个子串可以重叠。

算法分析：

这题的做法和上一题差不多，也是先二分答案，然后将后缀分成若干组。不同的是，这里要判断的是有没有一个组的后缀个数不小于 k。如果有，那么存在 k 个相同的子串满足条件，否则不存在。这个做法的时间复杂度为 O(nlogn)。

２．２．２子串的个数

例 5：不相同的子串的个数（spoj694,spoj705）

给定一个字符串，求不相同的子串的个数。

算法分析：

每个子串一定是某个后缀的前缀，那么原问题等价于求所有后缀之间的不相同的前缀的个数。如果所有的后缀按照 suffix(sa[1]), suffix(sa[2]), suffix(sa[3]), …… ,suffix(sa[n])的顺序计算，不难发现，对于每一次新加进来的后缀 suffix(sa[k]),它将产生 n-sa[k]+1 个新的前缀。但是其中有 height[k]个是和前面的字符串的前缀是相同的。所以 suffix(sa[k])将“贡献”出 n-sa[k]+1- height[k]个不同的子串。累加后便是原问题的答案。这个做法的时间复杂度为 O(n)。

２．２．３回文子串

回文子串：如果将字符串 L 的某个子字符串 R 反过来写后和原来的字符串 R 一样，则称字符串 R 是字符串 L 的回文子串。

例 6：最长回文子串（ural1297）

给定一个字符串，求最长回文子串。

算法分析：

穷举每一位，然后计算以这个字符为中心的最长回文子串。注意这里要分两种情况，一是回文子串的长度为奇数，二是长度为偶数。两种情况都可以转化为求一个后缀和一个反过来写的后缀的最长公共前缀。具体的做法是：将整个字符串反过来写在原字符串后面，中间用一个特殊的字符隔开。这样就把问题变为了求这个新的字符串的某两个后缀的最长公共前缀。如图 6 所示。

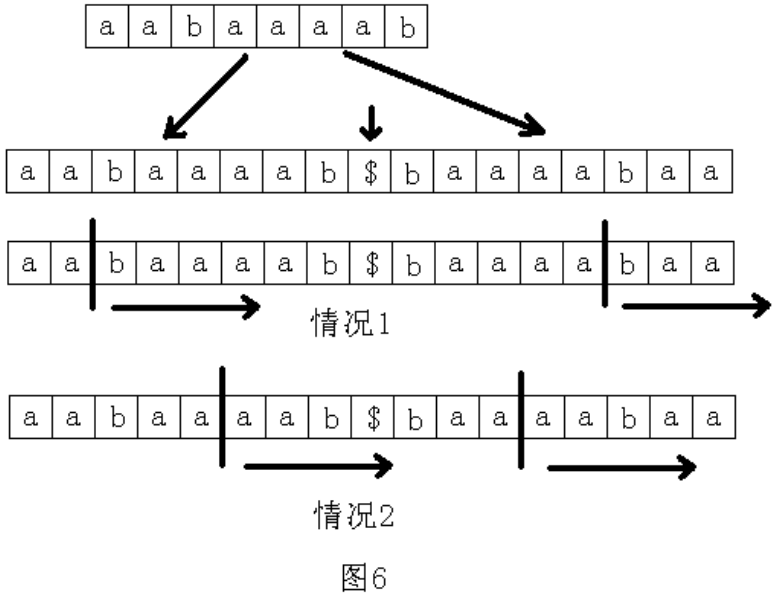

图6

这个做法的时间复杂度为 O(nlogn)。如果 RMQ 问题用时间为 O(n)的方法预处理，那么本题的时间复杂度可以降为 O(n)。

２．２．４ 连续重复子串

连续重复串：如果一个字符串 L 是由某个字符串 S 重复 R 次而得到的，则称 L 是一个连续重复串。R 是这个字符串的重复次数。

例 7：连续重复子串(pku2406)

给定一个字符串 L，已知这个字符串是由某个字符串 S 重复 R 次而得到的，求 R 的最大值。

算法分析：

做法比较简单，穷举字符串 S 的长度 k，然后判断是否满足。判断的时候，先看字符串 L 的长度能否被 k 整除，再看 suffix(1)和 suffix(k+1)的最长公共前缀是否等于 n-k。在询问最长公共前缀的时候，suffix(1)是固定的，所以 RMQ 问题没有必要做所有的预处理，只需求出 height 数组中的每一个数到 height[rank[1]]之间的最小值即可。整个做法的时间复杂度为 O(n)。

例 8：重复次数最多的连续重复子串(spoj687,pku3693)

给定一个字符串，求重复次数最多的连续重复子串。

算法分析：

先穷举长度 L，然后求长度为 L 的子串最多能连续出现几次。首先连续出现 1 次是肯定可以的，所以这里只考虑至少 2 次的情况。假设在原字符串中连续出现 2 次，记这个子字符串为 S，那么 S 肯定包括了字符 r[0], r[L], r[L*2], r[L*3], ……中的某相邻的两个。所以只须看字符 r[L*i]和 r[L*(i+1)]往前和往后各能匹配到多远，记这个总长度为 K，那么这里连续出现了 K/L+1 次。最后看最大值是多少。如图 7 所示。

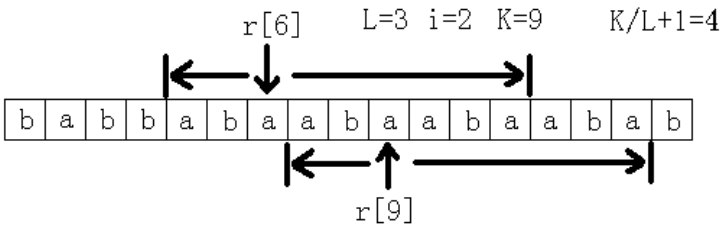

图7

穷举长度 L 的时间是 n，每次计算的时间是 n/L。所以整个做法的时间复杂度是 O(n/1+n/2+n/3+……+n/n)=O(nlogn)。

２．３两个字符串的相关问题

这类问题的一个常用做法是，先连接这两个字符串，然后求后缀数组和 height 数组，再利用 height 数组进行求解。

2.3.1 公共子串

公共子串:如果字符串 L 同时出现在字符串 A 和字符串 B 中，则称字符串 L 是字符串 A 和字符串 B 的公共子串。

例 9：最长公共子串(pku2774,ural1517)

给定两个字符串 A 和 B，求最长公共子串。

算法分析:

字符串的任何一个子串都是这个字符串的某个后缀的前缀。求 A 和 B 的最长公共子串等价于求 A 的后缀和 B 的后缀的最长公共前缀的最大值。如果枚举 A 和 B 的所有的后缀，那么这样做显然效率低下。由于要计算 A 的后缀和 B 的后缀的最长公共前缀，所以先将第二个字符串写在第一个字符串后面，中间用一个没有出现过的字符隔开，再求这个新的字符串的后缀数组。观察一下，看看能不能从这个新的字符串的后缀数组中找到一些规律。以 A=“aaaba”，B=“abaa”为例，如图 8 所示。

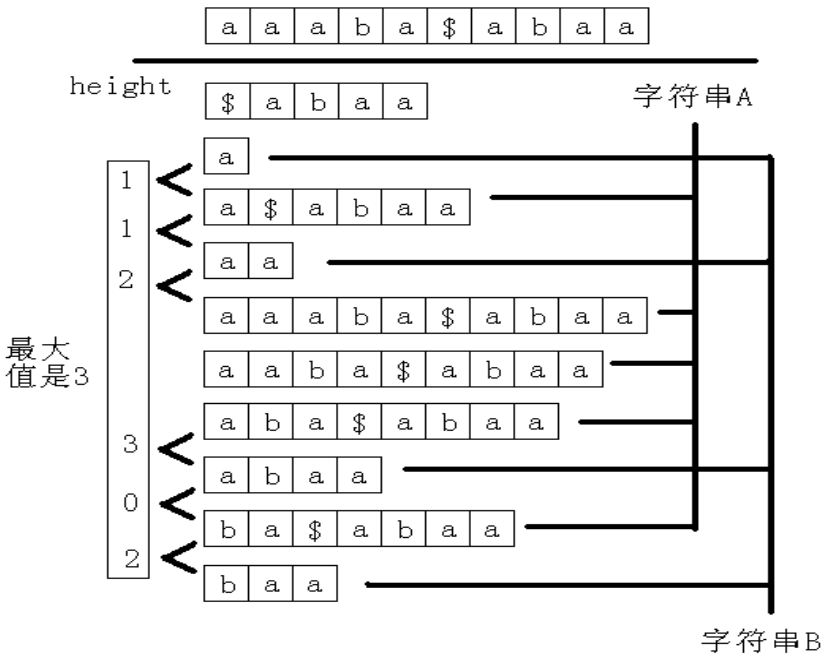

图8

那么是不是所有的 height 值中的最大值就是答案呢？不一定！有可能这两个后缀是在同一个字符串中的，所以实际上只有当 suffix(sa[i-1])和

suffix(sa[i])不是同一个字符串中的两个后缀时，height[i]才是满足条件的。而这其中的最大值就是答案。记字符串 A 和字符串 B 的长度分别为|A|和|B|。求新的字符串的后缀数组和 height 数组的时间是 O(|A|+|B|)，然后求排名相邻但原来不在同一个字符串中的两个后缀的 height 值的最大值，时间也是 O(|A|+|B|)，所以整个做法的时间复杂度为 O(|A|+|B|)。时间复杂度已经取到下限，由此看出，这是一个非常优秀的算法。

2.3.2 子串的个数

例 10:长度不小于 k 的公共子串的个数(pku3415)

给定两个字符串 A 和 B，求长度不小于 k 的公共子串的个数（可以相同）。

样例 1:

A=“xx”，B=“xx”，k=1，长度不小于 k 的公共子串的个数是 5。

样例 2:

A=“aababaa”，B=“abaabaa”，k=2，长度不小于 k 的公共子串的个数是22。

算法分析:

基本思路是计算 A 的所有后缀和 B 的所有后缀之间的最长公共前缀的长度，把最长公共前缀长度不小于 k 的部分全部加起来。先将两个字符串连起来，中间用一个没有出现过的字符隔开。按 height 值分组后，接下来的工作便是快速的统计每组中后缀之间的最长公共前缀之和。扫描一遍，每遇到一个 B 的后缀就统计与前面的 A 的后缀能产生多少个长度不小于 k 的公共子串，这里 A 的后缀需要用一个单调的栈来高效的维护。然后对 A 也这样做一次。具体的细节留给读者思考。

2.4 多个字符串的相关问题

这类问题的一个常用做法是，先将所有的字符串连接起来，然后求后缀数组和 height 数组，再利用 height 数组进行求解。这中间可能需要二分答案。

例 11:不小于 k 个字符串中的最长子串(pku3294)

给定 n 个字符串，求出现在不小于 k 个字符串中的最长子串。

算法分析:

将 n 个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，用和例 3 同样的方法将后缀分成若干组，判断每组的后缀是否出现在不小于 k 个的原串中。这个做法的时间复杂度为 O(nlogn)。

例 12:每个字符串至少出现两次且不重叠的最长子串(spoj220)

给定 n 个字符串，求在每个字符串中至少出现两次且不重叠的最长子串。

算法分析:

做法和上题大同小异，也是先将 n 个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，再将后缀分组。判断的时候，要看是否有一组后缀在每个原来的字符串中至少出现两次，并且在每个原来的字符串中，后缀的起始位置的最大值与最小值之差是否不小于当前答案（判断能否做到不重叠，如果题目中没有不重叠的要求，那么不用做此判断）。这个做法的时间复杂度为 O(nlogn)。

例 13:出现或反转后出现在每个字符串中的最长子串(PKU3294)

给定 n 个字符串，求出现或反转后出现在每个字符串中的最长子串。

算法分析:

这题不同的地方在于要判断是否在反转后的字符串中出现。其实这并没有加大题目的难度。只需要先将每个字符串都反过来写一遍，中间用一个互不相同的且没有出现在字符串中的字符隔开，再将 n 个字符串全部连起来，中间也是用一个互不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，再将后缀分组。判断的时候，要看是否有一组后缀在每个原来的字符串或反转后的字符串中出现。这个做法的时间复杂度为 O(nlogn)。

三、结束语

后缀数组是字符串处理中非常优秀的数据结构，是一种处理字符串的有力工

具，在不同类型的字符串问题中有广泛的应用。我们应该掌握好后缀数组这种数据结构，并且能在不同类型的题目中灵活、高效的运用。本文希望能为各位读者提供一点关于后缀数组的参考资料。对于前面所写的实现方法和各题的解答，如果读者有更好的做法，也欢迎读者与作者联系。

参考文献

[1] 刘汝佳，《算法艺术与信息学竞赛》，北京：清华大学出版社，2004
[2] 许智磊，IOI2004 国家集训队论文《后缀数组》
[3] 周源，2005 年信息学国家集训队作业《线性后缀排序算法》

致谢

感谢 CCF 给我提供一个与大家交流的平台
感谢清华大学的唐文斌教练对我的指导
感谢张学东老师在我写这篇论文时对我的帮助
感谢广州市第二中学的林盛华老师对我的指导和启发
感谢中山市第一中学的方展鹏、中山纪念中学的姜碧野同学提供关于后缀数组的资料