后缀数组及其应用

吉林大学附属中学高中部 吴一凡

December 4, 2014

将一个字符串的所有**后缀**按照字典序从小到大排序,我们记$sa[i]$表示排名第i的后缀在原串的开头位置。(均从0开始标号)

将一个字符串的所有**后缀**按照字典序从小到大排序,我们记$sa[i]$表示排名第i的后缀在原串的开头位置。(均从0开始标号)
例如说串$banana$,它的后缀集合为$\{banana, anana, nana, ana, na, a\}$.

将一个字符串的所有**后缀**按照字典序从小到大排序,我们记$sa[i]$表示排名第i的后缀在原串的开头位置。(均从0开始标号)
例如说串$banana$,它的后缀集合为$\{banana, anana, nana, ana, na, a\}$.
那么按照字典序从小到大排序后为$\{a, ana, anana, banana, na, nana\}$.

将一个字符串的所有**后缀**按照字典序从小到大排序,我们记$sa[i]$表示排名第i的后缀在原串的开头位置。(均从0开始标号)
例如说串$banana$,它的后缀集合为$\{banana, anana, nana, ana, na, a\}$.
那么按照字典序从小到大排序后为$\{a, ana, anana, banana, na, nana\}$.
因此，我们求出的sa数组为:$\{5, 3, 1, 0, 4, 2\}$.

后缀数组是处理字符串问题的一种有效工具。
能够发掘出字符串最深处的性质。
至于到底能干什么之后再讲。

我们不难想到直接利用*sort*给所有后缀排序。

我们不难想到直接利用*sort*给所有后缀排序。
然而比较两个后缀的字典序的复杂度可能达到$O(n)$.

我们不难想到直接利用$sort$给所有后缀排序。
然而比较两个后缀的字典序的复杂度可能达到$O(n)$.
因此,在最坏情况下,总时间复杂度很可能达到$O(n^2 logn)$.

我们不难想到直接利用$sort$给所有后缀排序。
然而比较两个后缀的字典序的复杂度可能达到$O(n)$.
因此,在最坏情况下,总时间复杂度很可能达到$O(n^2 logn)$.
在理论上讲,这种算法显然是无法应对长度动辄10^5的字符串的.

我们不难想到直接利用$sort$给所有后缀排序。
然而比较两个后缀的字典序的复杂度可能达到$O(n)$.
因此,在最坏情况下,总时间复杂度很可能达到$O(n^2 logn)$.
在理论上讲,这种算法显然是无法应对长度动辄10^5的字符串的.
不过如果串是随机的话,期望比较次数为$O(1)$,这时候时间复杂度变为$O(nlogn)$.所以实在没有时间的话就可以这么做。

算法1之所以低效是由于我们每次比较的信息过多。
那么我们就要尽可能减小每一次比较的复杂度。

算法1之所以低效是由于我们每次比较的信息过多。
那么我们就要尽可能减小每一次比较的复杂度。
我们考虑如下的倍增算法:我们进行k次排序,对于第k次排序,我们比较的是n个后缀**仅考虑前2^k个字符时的顺序**.
我们的任务就是高效的求解这个过程.

算法1之所以低效是由于我们每次比较的信息过多。
那么我们就要尽可能减小每一次比较的复杂度。
我们考虑如下的倍增算法:我们进行k次排序,对于第k次排序,我们比较的是n个后缀**仅考虑前2^k个字符时的顺序**.
我们的任务就是高效的求解这个过程.
考虑第$k-1$次排序，我们已经知道了n个后缀仅考虑前2^{k-1}个字符时的顺序，那么也就意味着我们把每连续2^{k-1}个字符都排好了序，那么前2^k个字符显然可以看成是一个包含有两个长度为2^{k-1}的子串的二元组！从而,我们只需对这些二元组排序即可！

算法1之所以低效是由于我们每次比较的信息过多。
那么我们就要尽可能减小每一次比较的复杂度。
我们考虑如下的倍增算法:我们进行k次排序,对于第k次排序,我们比较的是n个后缀**仅考虑前2^k个字符时的顺序**.
我们的任务就是高效的求解这个过程.
考虑第$k-1$次排序，我们已经知道了n个后缀仅考虑前2^{k-1}个字符时的顺序，那么也就意味着我们把每连续2^{k-1}个字符都排好了序，那么前2^k个字符显然可以看成是一个包含有两个长度为2^{k-1}的子串的二元组！从而,我们只需对这些二元组排序即可！
排好序之后我们要对这些二元组进行**重标号**,作为前2^k个字符的顺序，同时以便于下一次的排序。

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.
第0次排序后的参数值为$\{1,0,13,0,13,0\}$.现在的有效长度为$2^0=1$,其实也就是在比较首字母。

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.
第0次排序后的参数值为$\{1,0,13,0,13,0\}$.现在的有效长度为$2^0=1$,其实也就是在比较首字母。
第一次排序后，我们根据上一次的结果搞出一系列二元组，就是$\{(1,0),(0,13),(13,0),(0,13),(13,0),(0,-1)\}$.

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.
第0次排序后的参数值为$\{1,0,13,0,13,0\}$.现在的有效长度为$2^0=1$,其实也就是在比较首字母。
第一次排序后，我们根据上一次的结果搞出一系列二元组，就是$\{(1,0),(0,13),(13,0),(0,13),(13,0),(0,-1)\}$.
-1这货是从什么地方出来的?因为最后一个0的后面已经没有东西了，字典序必然最小，为了依旧满足是一个二元组因此来占位。

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.
第0次排序后的参数值为$\{1,0,13,0,13,0\}$.现在的有效长度为$2^0=1$,其实也就是在比较首字母。
第一次排序后，我们根据上一次的结果搞出一系列二元组，就是$\{(1,0),(0,13),(13,0),(0,13),(13,0),(0,-1)\}$.
-1这货是从什么地方出来的?因为最后一个0的后面已经没有东西了，字典序必然最小，为了依旧满足是一个二元组因此来占位。
排序后的二元组顺序为$\{(0,-1),(0,13),(1,0),(13,0)\}$.
注意，重复的只留下一个。

这个算法到底还是不那么容易理解的,我们不妨再举一个例子。
比如说还是字符串*banana*.
第0次排序后的参数值为$\{1,0,13,0,13,0\}$.现在的有效长度为$2^0=1$,其实也就是在比较首字母。
第一次排序后，我们根据上一次的结果搞出一系列二元组，就是$\{(1,0),(0,13),(13,0),(0,13),(13,0),(0,-1)\}$.
-1这货是从什么地方出来的?因为最后一个0的后面已经没有东西了，字典序必然最小，为了依旧满足是一个二元组因此来占位。
排序后的二元组顺序为$\{(0,-1),(0,13),(1,0),(13,0)\}$.
注意，重复的只留下一个。
现在排序后重标号后变成了:$\{2,1,3,1,3,0\}$.

现在没有完全分开，于是我们再来一次，现在比较前4位。
注意现在的二元组是:$\{(2,3),(1,1),(3,3),(1,0),(3,-1),(0,-1)\}$.

现在没有完全分开，于是我们再来一次，现在比较前4位。
注意现在的二元组是:$\{(2,3),(1,1),(3,3),(1,0),(3,-1),(0,-1)\}$.
然后现在排序:$\{(0,-1),(1,0),(1,1),(2,3),(3,-1),(3,3)\}$.
随后重标号：$\{3,2,5,1,4,0\}$.
此时我们发现标号已经完全分开了，因此就没有必要继续排序了。
根据sa数组的定义，我们得到最终的sa数组:$\{5,3,1,0,4,2\}$.

现在没有完全分开，于是我们再来一次，现在比较前4位。
注意现在的二元组是:$\{(2,3),(1,1),(3,3),(1,0),(3,-1),(0,-1)\}$.
然后现在排序:$\{(0,-1),(1,0),(1,1),(2,3),(3,-1),(3,3)\}$.
随后重标号：$\{3,2,5,1,4,0\}$.
此时我们发现标号已经完全分开了，因此就没有必要继续排序了。
根据sa数组的定义，我们得到最终的sa数组:$\{5,3,1,0,4,2\}$.
重标号是很容易在$O(n)$内完成的，重要的就是如何对二元组排序。下面介绍两种实现。

由倍增算法我们不难发现最多仅需要$O(logn)$次排序。

由倍增算法我们不难发现最多仅需要$O(logn)$次排序。
我们处理出这些二元组之后，利用一次快速排序直接将它们排序。随后进行重标号。

由倍增算法我们不难发现最多仅需要$O(logn)$次排序。
我们处理出这些二元组之后，利用一次快速排序直接将它们排序。随后进行重标号。
我们来分析一下时间复杂度:首先要进行$O(logn)$次排序，每一次排序我们要对n个二元组进行快速排序，比较次数可以认为是$O(1)$,因此每一次排序的复杂度$O(nlogn)$,总的时间复杂度为$O(nlog^2n)$.

由倍增算法我们不难发现最多仅需要$O(logn)$次排序。
我们处理出这些二元组之后，利用一次快速排序直接将它们排序。随后进行重标号。
我们来分析一下时间复杂度:首先要进行$O(logn)$次排序，每一次排序我们要对n个二元组进行快速排序，比较次数可以认为是$O(1)$,因此每一次排序的复杂度$O(nlogn)$,总的时间复杂度为$O(nlog^2n)$.
这个算法尚且可以应付大多数的题目，不过并不比下面将要介绍的一种时间复杂度更低的实现易于实现。
代码实现参照$sa_sort.cpp$.

基数排序是什么高大上的东西?这是一种号称**线性排序**的神犇算法。然而，这并不是绝对的，在此暂且不表。
假设我们要对n个结构体排序，每个结构体有m个参数.
假设这些参数是按照优先级从高到低的。
同时，第$i(1 \leq i \leq m)$个参数被限制在$[1, p_i]$之间。
那么基数排序的复杂度即为$O(m * (n + max_{i=1}^{m}\{p_i\}))$.

[1]例如$O(n)$的DC3算法

基数排序是什么高大上的东西?这是一种号称**线性排序**的神犇算法。然而，这并不是绝对的，在此暂且不表。
假设我们要对n个结构体排序，每个结构体有m个参数.
假设这些参数是按照优先级从高到低的。
同时，第$i(1 \leq i \leq m)$个参数被限制在$[1, p_i]$之间。
那么基数排序的复杂度即为$O(m * (n + max_{i=1}^{m}\{p_i\}))$.
考虑将基数排序利用到二元组的排序中，容易发现二元组的值域是$O(n)$级别的，因此每次基数排序的复杂度为$O(n)$,总的时间复杂度为$O(nlogn)$,是比实现1更加优秀的算法，而且实现并不复杂。
由于基数排序与主题无关，就不再赘述。但实际中，虽存在比$O(nlogn)$更加优秀的算法[1]，但由于常数及代码复杂度等原因，我们一般选用这种算法。代码实现参照*sa_radixsort.cpp*.

[1]例如$O(n)$的DC3算法

对于有强迫症的某些人（咳咳，说你呢），这样有诸多性质的问题没有一个达到时间复杂度下限的算法是说不通的。
于是就有了时间复杂度为$O(n)$的求解后缀数组的DC3算法。
事实上与后缀数组有关的题目很少有线性解法(除了一些裸题)，也就是说事实上这个算法在实际应用中并不比倍增法更有价值。不过为了满足强迫症就简单讲讲思想。

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。
然后将头位置为1的后缀和头位置为2的后缀接到一起，如果两个后缀中的某个长度不为3的倍数，则在最后补0.

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。
然后将头位置为1的后缀和头位置为2的后缀接到一起，如果两个后缀中的某个长度不为3的倍数，则在最后补0.
这样一个大长串的长度必定是3的倍数，我们将它的每连续三个字符取出来进行基数排序重标号。

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。
然后将头位置为1的后缀和头位置为2的后缀接到一起，如果两个后缀中的某个长度不为3的倍数，则在最后补0.
这样一个大长串的长度必定是3的倍数，我们将它的每连续三个字符取出来进行基数排序重标号。
这样就会变成一个长度为$\frac{2n}{3}$的新串，我们递归求它的后缀数组。
这样那些头位置不是3的倍数的后缀的字典序关系就很明显了。

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。
然后将头位置为1的后缀和头位置为2的后缀接到一起，如果两个后缀中的某个长度不为3的倍数，则在最后补0.
这样一个大长串的长度必定是3的倍数，我们将它的每连续三个字符取出来进行基数排序重标号。
这样就会变成一个长度为$\frac{2n}{3}$的新串，我们递归求它的后缀数组。
这样那些头位置不是3的倍数的后缀的字典序关系就很明显了。
现在考虑那些头位置是3的倍数的后缀,我们将这些后缀排序，注意到可以利用刚刚求出的后缀数组判定除首字符之外的后缀的关系，再加之首字符，事实上相当于给若干个二元组排序，利用一次基数排序即可。

假设长度为n的字符串标号范围为$[0, n-1]$,我们将后缀分为两个部分，头位置是3的倍数的后缀，以及头位置不是3的倍数的后缀。
然后将头位置为1的后缀和头位置为2的后缀接到一起，如果两个后缀中的某个长度不为3的倍数，则在最后补0.
这样一个大长串的长度必定是3的倍数，我们将它的每连续三个字符取出来进行基数排序重标号。
这样就会变成一个长度为$\frac{2n}{3}$的新串，我们递归求它的后缀数组。
这样那些头位置不是3的倍数的后缀的字典序关系就很明显了。
现在考虑那些头位置是3的倍数的后缀,我们将这些后缀排序，注意到可以利用刚刚求出的后缀数组判定除首字符之外的后缀的关系，再加之首字符，事实上相当于给若干个二元组排序，利用一次基数排序即可。
最终用一次归并操作合并所有的后缀，比较两个后缀无非分为两种情况，分类讨论即可。

不难发现除递归外，其余的操作均为$O(n)$.

不难发现除递归外，其余的操作均为$O(n)$.
让我们分析一下复杂度：令$f(n)$表示调用一次规模为n的求后缀数组过程，则有：

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

$$f(n) = O(n) + O(2n/3) + O(4n/9) + ...$$

$$f(n) = O(\frac{1 - \frac{2}{3}^{\infty}}{1 - \frac{2}{3}} n) < O(3 * n)$$

这样就神奇的达到了$O(n)$啦。

不难发现除递归外，其余的操作均为$O(n)$.
让我们分析一下复杂度：令$f(n)$表示调用一次规模为n的求后缀数组过程，则有：

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

$$f(n) = O(n) + O(2n/3) + O(4n/9) + ...$$

$$f(n) = O(\frac{1 - \frac{2}{3}^{\infty}}{1 - \frac{2}{3}} n) < O(3 * n)$$

这样就神奇的达到了$O(n)$啦。
如果真想学就去参考我的模板。网上的模板（包括某些论文里的）都是存在bug的。

接下来终于要开始讲后缀数组有什么应用啦。
那么就看一些题目吧。
(我会告诉你这一页其实是水经验的吗？233)

题目大意：给定一个字符串，将它的n个循环串按照字典序从小到大排序，依次输出这些串的第一位。$n \leq 10^5$.

题目大意：给定一个字符串，将它的n个循环串按照字典序从小到大排序，依次输出这些串的第一位。$n \leq 10^5$.
这是一道后缀数组的经典裸题。
只需将原串接在后面，随后对新串求出后缀数组，随后按照字典序输出前n个后缀的首字符即可。具体是为什么就不用解释了吧？

一般在实际应用中，只求出sa数组是不够用的。
我们再求出以下两个辅助数组来解决问题。

一般在实际应用中，只求出sa数组是不够用的。
我们再求出以下两个辅助数组来解决问题。
$rank$数组。$rank[i]$表示开头位置为i的后缀的排名。
这个只需要进行简单映射即可求出.时间复杂度$O(n)$.

一般在实际应用中，只求出sa数组是不够用的。
我们再求出以下两个辅助数组来解决问题。
$rank$数组。$rank[i]$表示开头位置为i的后缀的排名。
这个只需要进行简单映射即可求出.时间复杂度$O(n)$.
$height$数组.$height[i](1 \leq i < n)$表示排名为i的后缀和排名为$i-1$的后缀的LCP(最长公共前缀)的长度。

一般在实际应用中，只求出sa数组是不够用的。
我们再求出以下两个辅助数组来解决问题。
$rank$数组。$rank[i]$表示开头位置为i的后缀的排名。
这个只需要进行简单映射即可求出.时间复杂度$O(n)$.
$height$数组.$height[i](1 \leq i < n)$表示排名为i的后缀和排名为$i-1$的后缀的LCP(最长公共前缀)的长度。
$height$数组的用途。我们可以证明，对于两个后缀j, k,不妨令$rank[j] < rank[k]$,则两个后缀的LCP等于$\min_{i=rank[j]+1}^{rank[k]}\{height[i]\}$.而这个可以利用预处理$RMQ$快速得到。

一般在实际应用中，只求出sa数组是不够用的。
我们再求出以下两个辅助数组来解决问题。
$rank$数组。$rank[i]$表示开头位置为i的后缀的排名。
这个只需要进行简单映射即可求出.时间复杂度$O(n)$.
$height$数组.$height[i](1 \leq i < n)$表示排名为i的后缀和排名为$i-1$的后缀的LCP(最长公共前缀)的长度。
$height$数组的用途。我们可以证明，对于两个后缀j, k,不妨令$rank[j] < rank[k]$,则两个后缀的LCP等于$\min_{i=rank[j]+1}^{rank[k]}\{height[i]\}$.而这个可以利用预处理$RMQ$快速得到。
$height$数组的求法。当然不能把每两个串拿出来暴力啦！这样最坏情况下依然是$O(n^2)$的。

事实上，$height$数组有一个性质:$height[rank[i]] \geq height[rank[i-1]] - 1$

事实上，$height$数组有一个性质:$height[rank[i]] \geq height[rank[i-1]] - 1$
证明？
假设排在后缀$i-1$上一个的后缀为后缀k，则$height[rank[i-1]]$表示这两个后缀之间的LCP.
那么同时去掉一个字符，后缀$k+1$也必然排在后缀i前面，那么两者的LCP显然不小于$height[rank[i-1]] - 1$.
那么就是说这一段的$height$值的最小值不小于$height[rank[i-1]] - 1$.从而我们得到$height[rank[i]] \geq height[rank[i-1]] - 1$.

事实上，$height$数组有一个性质:$height[rank[i]] \geq height[rank[i-1]] - 1$
证明？
假设排在后缀$i-1$上一个的后缀为后缀k，则$height[rank[i-1]]$表示这两个后缀之间的LCP.
那么同时去掉一个字符，后缀$k+1$也必然排在后缀i前面，那么两者的LCP显然不小于$height[rank[i-1]] - 1$.
那么就是说这一段的$height$值的最小值不小于$height[rank[i-1]] - 1$.从而我们得到$height[rank[i]] \geq height[rank[i-1]] - 1$.
这样的话我们利用单调性扫描一遍便可以在$O(n)$的时间复杂度内求出$height$数组。

题目大意：给定一个字符串，若干次询问，每次求这个字符串的字典序第k小的子串。 $n \leq 90000, Q \leq 500$.

题目大意：给定一个字符串，若干次询问，每次求这个字符串的字典序第k小的子串。 $n \leq 90000, Q \leq 500$.
思路：SPOJ常数反人类，$O(nQ)$都被卡掉了我会乱说？
考虑利用后缀数组求解。
还是以串$banana$举例。
现在后缀的大小顺序是$a, ana, anana, banana, na, nana$.
然后$height$数组是$\{0, 1, 3, 0, 0, 2\}$.
然后把每个后缀的**长度大于对应**$height$**值的前缀**按照顺序写在一起：
$(a)(an, ana)(anan, anana)(b, ba, ban, bana, banan, banana)(n, na)(nan, nana)$
可以发现这就是按照字典序排好的所有子串。所以我们预处理前缀和，对于每次询问二分就可以找到这个子串的位置。
这样时间复杂度就是$O(n + Qlogn)$.

题目大意：求k个字符串的最长公共连续子串.$k \leq 5$,每个字符串的长度$1 \leq len \leq 2000$.

题目大意：求k个字符串的最长公共连续子串.$k \leq 5$,每个字符串的长度$1 \leq len \leq 2000$.
思路：将所有的串接在一起，并且中间接上分隔符。
求出后缀数组随后二分答案mid，在$height$数组中找出不小于mid的连续块，若连续块中出现了属于所有字符串的后缀，则证明存在长度不小于mid的连续公共子串。
这样我们就得到了一个$O(nlogn)$的算法。

题目大意：一首长度为n的乐曲，包含有n个整数，你需要找到一个最长的子序列，使得你可以找到一个与该序列不相交的序列，使得这两个序列对应两个数之间的差值均相同。求这个序列的最长长度。

题目大意：一首长度为n的乐曲，包含有n个整数，你需要找到一个最长的子序列，使得你可以找到一个与该序列不相交的序列，使得这两个序列对应两个数之间的差值均相同。求这个序列的最长长度。
思路：直接将这个序列差分，转化为求新序列的最长不相交重复子串问题。
首先求出新序列的后缀数组，然后二分答案mid,同样是在$height$找到不小于mid的连续块，同时，我们记下在这个连续块中后缀出现位置的最大值和最小值，若这两个后缀的的长度为mid的前缀能够不相交，那么证明存在一个长度至少为mid的不相交重复子串。时间复杂度$O(nlogn)$.

题目大意：求一个字符串的最长回文子串。

题目大意：求一个字符串的最长回文子串。
思路：将字符串倒过来接在原串的后面，随后二分答案，看一下在$height$连续块中是否同时有正串和反串的后缀，若有则证明存在长度不小于mid的回文子串。时间复杂度$O(nlogn)$.

题目大意：给定一个字符串，对于它的每个回文子串，权值等于这个回文子串的长度乘以这个回文子串在这个字符串中的出现次数。求这个字符串的的每个回文子串的最大权值。串长≤ 300000.

题目大意：给定一个字符串，对于它的每个回文子串，权值等于这个回文子串的长度乘以这个回文子串在这个字符串中的出现次数。求这个字符串的的每个回文子串的最大权值。串长≤ 300000.
思路：首先利用*Manacher*算法在$O(n)$的时间求出所有本质不同的回文子串。
容易证明这是$O(n)$级别的。由于与今天的主题无关就不再赘述。
接下来要对于每一个回文子串求出其出现的次数，这可以利用后缀数组来解决。
对于一个子串，我们知道的是它的头位置以及它的长度，于是我们找到以他的头位置为开头的后缀，然后再找到这个后缀所在的*height*值不小于长度的连续块。块中的元素数目即为这个子串出现的次数。
于是我们预处理*height*数组的*RMQ*,再上下分别二分即可。
这样我们便可以在$O(nlogn)$的复杂度内解决此题。

后缀数组是处理字符串问题的一种灵活的数据结构。
今天我们介绍了构造后缀数组的几种算法，其中推荐使用的是基于基数排序的倍增算法或是DC3算法。
处理出后缀数组后我们经常会一起处理出辅助的$rank$, $height$数组来解决更多问题。
寥寥几道题目不能囊括所有后缀数组的性质，这里仅是抛砖引玉，更需要多多理解、发掘。