On String Suffixes

Yanheng Wang

Let $s[1\ldots n]$ be a string, and $s_i := s[i\ldots n]$ be its suffix starting from position i. We want to sort $s_1,\ldots,s_n$ in lexicographic order. For example, if $s = \text{aacab}$ then we should output

$$\overset{s_1}{\text{aacab}} < \overset{s_4}{\text{ab}} < \overset{s_2}{\text{acab}} < \overset{s_5}{\text{b}} < \overset{s_3}{\text{cab}}$$

or succinctly 1,4,2,5,3. This order has important applications in string-related problems, as we will see later.

We employ a divide-and-conquer paradigm. Denote for convenience $\text{type}(i) := i \bmod 3$.

(i) sort suffixes s_i such that $\text{type}(i) \in \{1,2\}$;

(ii) sort suffixes s_i such that $\text{type}(i) = 0$;

(iii) merge the two sequences.

Of the three steps only (i) will be solved recursively. The other two can be done in linear time given what (i) has computed. Specifically for step (ii), we equate each s_i with a pair $(s[i], s_{i+1})$. Since $\text{type}(i+1) = 1$, the lexicographic order of s_{i+1}'s were computed in (i) already. So we can sort the pairs with one pass of RadixSort, which costs linear time.

In step (iii), we merge two suffix sequences $a_1 < \cdots < a_{2n/3}$ and $b_1 < \cdots < b_{n/3}$ as in MergeSort. That is, we slide two pointers $p \in [2n/3]$ and $q \in [n/3]$ from left to right. At each time we compare a_p with b_q. If $a_p < b_q$ then we a_p is the next smallest and we increment p. Otherwise b_q is the next smallest and we increment q.

How do we compare a_p with b_q? Say $a_p = s_i$ and $b_q = s_j$. By definition $t := \text{type}(i) \in \{1,2\}$ and $\text{type}(j) = 0$. We first compare $s[i\ldots i+t-1]$ with $s[j\ldots j+t-1]$, which takes constant time. If they are different then the comparison completes. Otherwise we proceed to compare s_{i+t} with s_{j+t}. Since $\text{type}(i+t) \equiv 2t \not\equiv 0$ and $\text{type}(j+t) = t \neq 0$, the result can be read from (i) directly.

It remains to implement step (i). Assume without loss of generality that $\mathrm{type}(n)=2$. Construct a new string $s' := s[1\ldots n]\blacklozenge s[2\ldots n]\blacklozenge\blacklozenge$, whose length is a multiple of three. Break it into substrings of three characters, then obtain their lexicographic order r by RadixSort. See below for an example.

s	s'	r
aacab	(aac)(ab$\blacklozenge$)(aca)(b$\blacklozenge\blacklozenge$)	1,2,3,4
aaaab	(aaa)(ab$\blacklozenge$)(aaa)(b$\blacklozenge\blacklozenge$)	1,2,1,3

Call the first position in each substring a *leader*. Leaders always correspond to type-1 and -2 positions in s, and vice versa. So for every $i : \mathrm{type}(i) \in \{1,2\}$, we can recover the suffix s_i by reading s' from some leader $\ell(i)$ till $\blacklozenge$.

In fact, the lexicographic order of the suffixes is preserved even if we read till the end. More precisely, $s_i < s_j$ if and only if $s'_{\ell(i)} < s'_{\ell(j)}$. So the problem reduces to sorting $s'_{\ell(1)}, \ldots, s'_{\ell(n)}$. It is equivalent to sorting $r_1, \ldots, r_{2n/3}$ where we interpret r as a string and $r_i := r[i\ldots 2n/3]$. This last task can be solved recursively.

The analysis of running time $T(n)$ is straightforward. We have the recursion

$$T(n) = T(2n/3) + O(n),$$

thus $T(n) = O(n) \cdot \sum_{i=1}^{\infty} (2/3)^i = O(n)$.

Longest common prefix. Now that we have ordered the suffixes $s_{\pi(1)} < \cdots < s_{\pi(n)}$, it's time for applications. For convenience we write $\rho(i) := \pi^{-1}(i)$, representing the rank of s_i.

For strings a and b we define

$$\ell(a,b) := \max\{\ell : a[1\ldots\ell] = b[1\ldots\ell]\},$$

that is the length of the longest common prefix of a and b. Our ultimate goal is to pre-compute a data structure that allows answering $\ell(s_i, s_j)$ for arbitrary i, j in sublinear time. Without loss of generality we assume $\rho(i) < \rho(j)$.

Observe that, for any strings $a \leqslant b \leqslant c$,

$$\ell(a,c) = \min\{\ell(a,b), \ell(b,c)\}. \tag{1}$$

So inductively we have

$$\ell(s_i, s_j) = \min\{\ell(s_{\pi(r-1)}, s_{\pi(r)}) : \rho(i) < r \leqslant \rho(j)\}. \tag{2}$$

This motivates us to study $\Lambda(r) := \ell(s_{\pi(r-1)}, s_{\pi(r)})$ for $r = 2, \ldots, n$. After we collect these values, we build a segment tree on top, so later we can answer query via (2) in $O(\log n)$ time. It is known that query time can be reduced to $O(1)$, but we shall not go into that.

How do we compute $\Lambda(r)$? The key property is that

$$\Lambda(\rho(i+1)) \geqslant \Lambda(\rho(i)) - 1 \text{ for all } i.$$

Despite its appalling look, the proof is rather simple. In $\Lambda(\rho(i))$ we care about the longest common prefix of s_i and its predecessor s_j. ($j = \pi(\rho(i) - 1)$ if you insist.) Deleting their first character, we obtain exactly s_{i+1} and s_{j+1}, with the common prefix length dropping by one. In other words,

$$\ell(s_{j+1}, s_{i+1}) = \Lambda(\rho(i)) - 1.$$

Now we look at $\Lambda(\rho(i+1))$, which cares about the longest common prefix of s_{i+1} and its predecessor s_k. Since $s_j < s_i$, we have $s_{j+1} \leqslant s_{i+1}$; consequently $s_{j+1} \leqslant s_k$ by definition of "predecessor". Therefore,

$$\Lambda(\rho(i+1)) \geqslant \ell(s_{j+1}, s_{i+1}) = \Lambda(\rho(i)) - 1$$

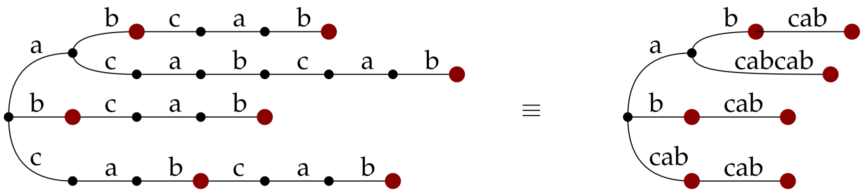
where the inequality is due to (1).

The proof immediately leads to the following algorithm.

```
compute $\Lambda(\rho(1))$
$h := \Lambda(\rho(1))$
**for** $i = 1, \ldots, n-1$ **do**
    $k := \pi(\rho(i+1) - 1)$    {*predecessor of* $i+1$}
    {$s_{i+1}$ *and* $s_k$ *must agree on the first* $h-1$ *characters*}
    **while** $s_{i+1}[h] = s_k[h]$ **do**
        $h := h + 1$
    $h := h - 1$
    $\Lambda(\rho(i+1)) := h$
```

The initialisation of $\Lambda(\rho(1))$ takes linear time. The cost of the loop is the number m of increments plus the number m' of decrements. Trivially that $m' \leqslant n - 1$. Moreover, at any point of time we have $h \leqslant n$, so $m - m' \leqslant n$ for sure. This implies $m + m' < 3n$.

Suffix tree. The *suffix tree* of s is a rooted tree where each edge stores a substring. Denote $\mathrm{path}(v)$ as the concatenation of substrings on the path root$\rightsquigarrow v$. We require that, for each suffix s_i, there is a unique node v such that $\mathrm{path}(v) = s_i$. It is called the *terminal* of s_i. The picture below shows the suffix tree of $s = \mathrm{acabcab}$.



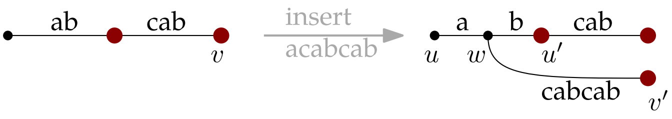
We can readily obtain the lexicographic order of suffixes via a depth-first traversal of the tree. At the same time, we can compute $\Lambda(r)$ for all r.

The backward conversion is also possible. We start with a rooted tree with only one edge that stores $s_{\pi(1)}$. The idea is to add other suffixes to the tree in order. Suppose we have added $s_{\pi(1)}, \dots, s_{\pi(r-1)}$ and are about to add $s_{\pi(r)}$. Let v be the terminal of $s_{\pi(r-1)}$, and write for simplicity $\Lambda := \Lambda(s_{\pi(r-1)}, s_{\pi(r)})$. We travel from v towards the root until reaching a node u such that

$$|\mathrm{path}(u)| \leqslant \Lambda.$$

So $\mathrm{path}(u)$ is a common prefix of $s_{\pi(r-1)}$ and $s_{\pi(r)}$. Moreover, the substring t stored in the last visited edge uu' must contain the position where $s_{\pi(r-1)}$ and $s_{\pi(r)}$ start to differ. The longest common prefix of the two is exactly $\mathrm{path}(u) \circ t[1 \dots \Lambda - |\mathrm{path}(u)|]$.

Hence, we subdivide the edge into uw and wu' by creating a new node w. They store the substrings of t before and after position $\Lambda - |\mathrm{path}(u)|$, respectively. Then we branch an edge wv' that stores $s_{\pi(r)}[\Lambda + 1 \dots]$. The vertex v' is the terminal of $s_{\pi(r)}$.



What is the time complexity? In some iteration we might need to spend linear time until we find u. But this shall forbid a large portion of the tree from later access, so the amortised cost is low. More formally, we say the iteration *kills* the nodes on path $v \rightsquigarrow u'$. The cost of the iteration is just the number of killed nodes plus $O(1)$. Because the killed path is never accessed later, each node can be killed at most once. Therefore, the total cost is at most the number of nodes in the suffix tree, plus $O(n)$.

How many nodes are there? We have n terminals for sure. Let n' count the number of other nodes. All other nodes (except root) have degree at least three, so

$$2\,(n+n'-1)=\sum_{v}\deg(v)\geqslant n+3\,(n'-1).$$

Solving it gives $n'\leqslant n+1$. Hence $n+n'\leqslant 2n+1=O(n)$.

String matching. We want to decide if the string s contains a given pattern p as substring. Note that if p appear in s then it must appear in some suffix. So we can detect it by walking from the root and always moving to the branch that corresponds to the next character in p. If we get stuck before p runs out, then no occurrence is found.

We can interpret the process as running parallel comparison threads, where thread i tries to match p with $s[i\ldots i+|p|-1]$, or equivalently, decide if p is a prefix of s_i.

Actually we can retrieve loads of information in case that p appears in s. Let v be the node that we end up with; so $\mathrm{path}(v)=p$. The number of occurrences of p is captured by the number of terminals in the subtree rooted at v. Moreover, from these terminals we can recover the starting positions of the occurrences.