String Mergesort

Standard comparison based sorting algorithms are not optimal for sorting strings because of an imbalance between effort and result in a string comparison: it can take a lot of time but the result is only a bit or a trit of useful information.

String quicksort solves this problem by using symbol comparisons where the constant time is in balance with the information value of the result.

String mergesort takes the opposite approach. It replaces a standard string comparison with the operation LcpCompare(A, B, k):

- The return value is the pair (x, ℓ), where $x \in \{<, =, >\}$ indicates the order, and $\ell = lcp(A, B)$, the length of the longest common prefix of strings A and B.

- The input value k is the length of a known common prefix, i.e., a lower bound on $lcp(A, B)$. The comparison can skip the first k characters.

Any extra time spent in the comparison is balanced by the extra information obtained in the form of the lcp value.

The following result show how we can use the information from past comparisons to obtain a lower bound or even the exact value for an lcp.

Lemma 1.27: Let A, B and C be strings.

(a) $lcp(A, C) \geq \min\{lcp(A, B), lcp(B, C)\}$.

(b) If $A \leq B \leq C$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.

Proof. Assume $\ell = lcp(A, B) \leq lcp(B, C)$. The opposite case $lcp(A, B) \geq lcp(B, C)$ is symmetric.

(a) Now $A[0..\ell) = B[0..\ell) = C[0..\ell)$ and thus $lcp(A, C) \geq \ell$.

(b) Either $|A| = \ell$ or $A[\ell] < B[\ell] \leq C[\ell]$. In either case, $lcp(A, C) = \ell$.

$\square$

It can also be possible to determine the order of two strings without comparing them directly.

Lemma 1.28: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$.

(a) If $lcp(A, B) > lcp(A, B')$, then $B < B'$.

(b) If $lcp(B, C) > lcp(B', C)$, then $B > B'$.

Proof. We show (a); (b) is symmetric. Assume to the contrary that $B \geq B'$. Then by Lemma 1.27, $lcp(A, B) = \min\{lcp(A, B'), lcp(B', B)\} \leq lcp(A, B')$, which is a contradiction. $\square$

String mergesort has the same structure as the standard mergesort: sort the first half and the second half separately, and then merge the results.

Algorithm 1.29: StringMergesort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with lcp information.
(1) if $|\mathcal{R}| = 1$ then return $\{(S_1, 0)\}$
(2) $m \leftarrow \lfloor n/2 \rfloor$
(3) $\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_m\}$)
(4) $\mathcal{Q} \leftarrow$ StringMergesort($\{S_{m+1}, S_{m+2}, \ldots, S_n\}$)
(5) return StringMerge($\mathcal{P}, \mathcal{Q}$)

The output is of the form

$$\{(T_1, \ell_1), (T_2, \ell_2), \ldots, (T_n, \ell_n)\}$$

where $\ell_i = lcp(T_i, T_{i-1})$ for $i > 1$ and $\ell_1 = 0$. In other words, $\ell_i = LCP_{\mathcal{R}}[i]$.

Thus we get not only the order of the strings but also a lot of information about their common prefixes. The procedure StringMerge uses this information effectively.

Algorithm 1.30: StringMerge($\mathcal{P}$,$\mathcal{Q}$)
Input: Sequences $\mathcal{P} = \big((S_1, k_1), \ldots, (S_m, k_m)\big)$ and $\mathcal{Q} = \big((T_1, \ell_1), \ldots, (T_n, \ell_n)\big)$
Output: Merged sequence $\mathcal{R}$

$\mathcal{R} \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$
while $i \leq m$ and $j \leq n$ do
    if $k_i > \ell_j$ then append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
    else if $\ell_j > k_i$ then append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
    else   // $k_i = \ell_j$
        $(x, h) \leftarrow$ LcpCompare(S_i, T_j, k_i)
        if $x =$ "$<$" then
            append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
            $\ell_j \leftarrow h$
        else
            append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
            $k_i \leftarrow h$
while $i \leq m$ do append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
while $j \leq n$ do append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
return $\mathcal{R}$

Lemma 1.31: StringMerge performs the merging correctly.

Proof. We will show that the following invariant holds at the beginning of each round in the loop on lines (2)–(12):

> Let X be the last string appended to $\mathcal{R}$ (or ε if $\mathcal{R} = \emptyset$). Then $k_i = lcp(X, S_i)$ and $\ell_j = lcp(X, T_j)$.

The invariant is clearly true in the beginning. We will show that the invariant is maintained and the smaller string is chosen in each round of the loop.

- If $k_i > \ell_j$, then $lcp(X, S_i) > lcp(X, T_j)$ and thus
  - $S_i < T_j$ by Lemma 1.28.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because by Lemma 1.27 $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  Hence, the algorithm chooses the smaller string and maintains the invariant. The case $\ell_j > k_i$ is symmetric.

- If $k_i = \ell_j$, then clearly $lcp(S_i, T_j) \geq k_i$ and the call to LcpCompare is safe, and the smaller string is chosen. The update $\ell_j \leftarrow h$ or $k_i \leftarrow h$ maintains the invariant. □

Theorem 1.32: String mergesort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(L(\mathcal{R}) + n \log n)$ time.

Proof. If the calls to LcpCompare took constant time, the time complexity would be $\mathcal{O}(n \log n)$ by the same argument as with the standard mergesort.

Whenever LcpCompare makes more than one, say $1 + t$ symbol comparisons, one of the lcp values stored with the strings increases by t. Since the sum of the final lcp values is exactly $L(\mathcal{R})$, the extra time spent in LcpCompare is bounded by $\mathcal{O}(L(\mathcal{R}))$.

$\square$

- Other comparison based sorting algorithms, for example heapsort and insertion sort, can be adapted for strings using the lcp comparison technique.

String Binary Search

An ordered array is a simple static data structure supporting queries in $\mathcal{O}(\log n)$ time using binary search.

Algorithm 1.33: Binary search
Input: Ordered set $R = \{k_1, k_2, \ldots, k_n\}$, query value x.
Output: The number of elements in R that are smaller than x.
(1) $left \leftarrow 0$; $right \leftarrow n + 1$ // final answer is in the range $[left..right)$
(2) while $right - left > 1$ do
(3) $\quad mid \leftarrow left + \lfloor (right - left)/2 \rfloor$
(4) $\quad$ if $k_{mid} < x$ then $left \leftarrow mid$
(5) $\quad$ else $right \leftarrow mid$
(6) return $left$

With strings as elements, however, the query time is

- $\mathcal{O}(m \log n)$ in the worst case for a query string of length m
- $\mathcal{O}(m + \log n \log_\sigma n)$ on average for a random set of strings.

We can use the lcp comparison technique to improve binary search for strings. The following is a key result.

Lemma 1.34: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$. Then $lcp(B, B') \geq lcp(A, C)$.

Proof. Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 1.27,

$$\begin{aligned} lcp(A, C) &= \min(lcp(A, B_{max}), lcp(B_{max}, C)) \\ &\leq lcp(A, B_{max}) = \min(lcp(A, B_{min}), lcp(B_{min}, B_{max})) \\ &\leq lcp(B_{min}, B_{max}) = lcp(B, B') \end{aligned}$$

$\square$

During the binary search of P in $\{S_1, S_2, \ldots, S_n\}$, the basic situation is the following:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know that $S_{left} \leq P, S_{mid} \leq S_{right}$.
- If we are using LcpCompare, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

By Lemmas 1.27 and 1.34,

$$lcp(P, S_{mid}) \geq lcp(S_{left}, S_{right}) = \min\{lcp(S_{left}, P), lcp(P, S_{right})\}$$

Thus we can skip $\min\{lcp(S_{left}, P), lcp(P, S_{right})\}$ first characters when comparing P and S_{mid}.

Algorithm 1.35: String binary search (without precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

$left \leftarrow 0$; $right \leftarrow n + 1$
$llcp \leftarrow 0$; $rlcp \leftarrow 0$
while $right - left > 1$ do
    $mid \leftarrow left + \lfloor (right - left)/2 \rfloor$
    $mlcp \leftarrow \min\{llcp, rlcp\}$
    $(x, mlcp) \leftarrow \mathrm{LcpCompare}(S_{mid}, P, mlcp)$
    if $x =$ "$<$" then $left \leftarrow mid$; $llcp \leftarrow mclp$
    else $right \leftarrow mid$; $rlcp \leftarrow mclp$
return $left$

- The average case query time is now $\mathcal{O}(m + \log n)$.
- The worst case query time is still $\mathcal{O}(m \log n)$.

We can further improve string binary search using precomputed information about the lcp's between the strings in $\mathcal{R}$.

Consider again the basic situation during string binary search:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

The values $left$ and $right$ depend only on mid. In particular, they do not depend on P. Thus, we can precompute and store the values

$$LLCP[mid] = lcp(S_{left}, S_{mid})$$
$$RLCP[mid] = lcp(S_{mid}, S_{right})$$

Now we know all lcp values between P, S_{left}, S_{mid}, S_{right} except $lcp(P, S_{mid})$. The following lemma shows how to utilize this.

Lemma 1.36: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$.

(a) If $lcp(A, B) > lcp(A, B')$, then $B < B'$ and $lcp(B, B') = lcp(A, B')$.

(b) If $lcp(A, B) < lcp(A, B')$, then $B > B'$ and $lcp(B, B') = lcp(A, B)$.

(c) If $lcp(B, C) > lcp(B', C)$, then $B > B'$ and $lcp(B, B') = lcp(B', C)$.

(d) If $lcp(B, C) < lcp(B', C)$, then $B < B'$ and $lcp(B, B') = lcp(B, C)$.

(e) If $lcp(A, B) = lcp(A, B')$ and $lcp(B, C) = lcp(B', C)$, then $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$.

Proof. Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows from Lemma 1.28. Then by Lemma 1.27, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 1.27:

$$lcp(B, B') \geq \min\{lcp(A, B), lcp(A, B')\} = lcp(A, B)$$
$$lcp(B, B') \geq \min\{lcp(B, C), lcp(B', C)\} = lcp(B, C)$$

Thus $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$. $\square$

Algorithm 1.37: String binary search (with precomputed lcps)

Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, arrays LLCP and RLCP, query string P.

Output: The number of strings in $\mathcal{R}$ that are smaller than P.

$left \leftarrow 0$; $right \leftarrow n + 1$
$llcp \leftarrow 0$; $rlcp \leftarrow 0$
while $right - left > 1$ **do**
    $mid \leftarrow left + \lfloor (right - left)/2 \rfloor$
    if $LLCP[mid] > llcp$ **then** $left \leftarrow mid$
    else if $LLCP[mid] < llcp$ **then** $right \leftarrow mid$; $rlcp \leftarrow LLCP[mid]$
    else if $RLCP[mid] > rlcp$ **then** $right \leftarrow mid$
    else if $RLCP[mid] < rlcp$ **then** $left \leftarrow mid$; $llcp \leftarrow RLCP[mid]$
    else
        $mlcp \leftarrow \max\{llcp, rlcp\}$
        $(x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
        if $x = \text{“}<\text{”}$ **then** $left \leftarrow mid$; $llcp \leftarrow mclp$
        else $right \leftarrow mid$; $rlcp \leftarrow mclp$
return $left$

Theorem 1.38: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(L(\mathcal{R}))$ time and $\mathcal{O}(n)$ space so that a binary search with a query string P can be executed in $\mathcal{O}(|P| + \log n)$ time.

Proof. The values $LLCP[mid]$ and $RLCP[mid]$ can be computed in $\mathcal{O}(lcp(S_{mid}, \mathcal{R} \setminus \{S_{mid}\})$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(lcp(\mathcal{R})) = \mathcal{O}(L(\mathcal{R}))$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 1.37 is executed $\mathcal{O}(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

If a given LcpCompare call performs $t + 1$ symbol comparisons, $mclp$ increases by t on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increases by at least t, since $mlcp$ was $\max\{llcp, rlcp\}$ before LcpCompare. Since $llcp$ and $rlcp$ never decrease and never grow larger than $|P|$, the total number of extra symbol comparisons in LcpCompare during the binary search is $\mathcal{O}(|P|)$. $\square$