Merging String Sequences by Longest Common Prefixes

WAIHONG NG†1 and KATSUHIKO KAKEHI†2

We present LCP Merge, a novel merging algorithm for merging two ordered sequences of strings. LCP Merge substitutes string comparisons with integer comparisons whenever possible to reduce the number of character-wise comparisons as well as the number of key accesses by utilizing the longest common prefixes (LCP) between the strings. As one of the applications of LCP Merge, we built a string merge sort based on recursive merge sort by replacing the merging algorithm with LCP Merge and we call it LCP Merge sort. In case of sorting strings, the computational complexity of recursive merge sort tends to be greater than $O(n \log n)$ because string comparisons are generally not constant time and depend on the properties of the strings. However, LCP Merge sort improves recursive merge sort to the extent that its computational complexity remains $O(n \log n)$ on average. We performed a number of experiments to compare LCP Merge sort with other string sorting algorithms to evaluate its practical performance and the experimental results showed that LCP Merge sort is efficient even in the real-world.

1. Introduction

Merging is a fundamental computational process which combines two or more ordered sequences of objects into a single ordered sequence of the objects and each object has a key which governs the ordering of the object. Merging has many applications, for instance, merge sort, external sorting, etc. Classical merging algorithms (e.g., Section 5.2.4 Algorithm M, two-way merge, in Ref. 8)) assume the comparison operation runs in constant time. However, when the keys are strings, which are widely used in practice, the computational complexity of the comparison operation is not constant but depends on the properties of the strings. In this paper, we present Longest Common Prefix Merge or LCP Merge for short, a novel merging algorithm for merging two ordered sequences of strings in which their keys are the strings themselves. LCP Merge has a fewer number of key accesses and character-wise comparisons than those of classical merging algorithms. The reductions in the number of key accesses and the number of character-wise comparisons are achieved by utilizing the LCPs between the strings. Approaches utilizing LCP information are well-known for constructing suffix trees or suffix arrays3,7,9. But to the best of our knowledge, there is no published LCP based merging algorithm which solves the problem discussed in this paper. We applied LCP Merge to build a string merge sort (we call it LCP Merge sort) and compared it with recursive merge sort to evaluate its improvements. In addition, LCP Merge sort is compared with four other string sorting algorithms: Multikey Quicksort1, Burstsort13, MSD radix sort10 and CRadix sort11, in a number of experiments to evaluate its performance.

The primal form of LCP Merge first appeared in our previous work4 and was called Length of Longest Common Prefix Merge (LLCP Merge). But we renamed LLCP Merge to LCP Merge, taking into account that the word LCP is very common and it connotes ‘length’.

2. Preliminaries

We define some terminology for the discussions in this paper.

Definition 2.1 An alphabet is a finite totally ordered set of symbols and the elements of the alphabet are called characters. For example, {a, b, …, z} is the alphabet of all lower-case English letters.

Without ambiguity, an alphabet is assumed where appropriate in our discussion without being mentioned.

Definition 2.2 A string is a list of characters. Given a string $s = s[1]s[2] \ldots s[a]$, $s[i]$ (1 ≤ i ≤ a) is the i-th character of $s$. The number of characters in $s$, denoted |s|, is the length of $s$. The empty or null string is the string which has zero length and is denoted ε.

Definition 2.3 Concatenation of two strings is defined as the juxtaposition of them.
A string sequence is said to be **ascending** if $s_i \leq s_{i+1}$ for $i : 1 \leq i < n$.

### 3. LCP Merging Algorithm

The classical merging algorithm (Section 5.2.4 Algorithm M (two-way merge) in Ref.8) is based on comparison operation which is assumed to be run in constant time. When the keys are strings, a general algorithm for the comparison operation will be similar to

```c
int strcmp(const char *s1, const char *s2)
{
    char c1, c2;
    int r;
    do {
        c1 = *s1++; c2 = *s2++;
        r = (unsigned char)c1 - (unsigned char)c2;
    } while ((r==0) & & (c1!='\0'));
    return r;
}
```

The above algorithm (program) has the same functionality as the standard C library function `strcmp`. It may be the simplest one as it repeatedly compares the strings given in the arguments character by character until a different one is found or a null character is encountered, and there is no way to reduce those character-wise comparisons without prior knowledge of the input strings. However, it is obvious that the running time of the algorithm is not constant since the number of character-wise comparisons required depends on the strings being compared. Further, despite that this algorithm explicitly gives us the relative order and implicitly the LCP of the strings (the number of iterations of the while loop minus one), known merging algorithms do not use their LCP. Therefore, there arises the question whether we can utilize the LLCP information to ease our computational process in some context. In the next section, we describe our merging algorithm, Longest Common Prefix Merge or LCP Merge for short, which utilizes LLCP information between the strings, for merging two ordered string sequences.

#### 3.1 LCP Merge

LCP Merge is built on the concept of annotated strings. It utilizes the LCP information between the strings to substitutes string comparisons with integer comparisons whenever applicable, which in turn reduces the running time of the comparison operation in such cases. As a
result, a significant amount of key accesses and character-wise comparisons are reduced when compared with those of classical merging algorithms.

3.1.1 Annotated String

For a string $s$ with a reference string $r$, we annotate $\lambda(r,s)$ to $s$ to form an annotated string. We interchangeably write $r\cdot s$ to denote $(\lambda(r,s),s)$. For a sequence of annotated strings, $A = r_1\cdot s_1, r_2\cdot s_2, \ldots, r_a\cdot s_a$ and a reference string $r$, if $r_1 = r$ and $s_i = s_{i-1}$ for all $i:1 \leq i \leq a$, we call $A$ an annotated string sequence with an external reference $r$ or simply an annotated string sequence (note that $\lambda(r_2,s_2), \lambda(r_3,s_3), \ldots, \lambda(r_a,s_a)$ form a sequence similar in notion to the height array mentioned in Ref. 7). The ordering on annotated strings is defined as the ordering on their particular strings being annotated.

3.1.2 Operations on Annotated Strings

Let $x$, $y$ and $z$ be strings where $z \preceq x$ and $z \preceq y$. The following theorem enables us to select the smaller annotated string from $z\cdot x$ and $z\cdot y$ and obtain $\lambda(x,y)$ without character-wise comparisons provided that $\lambda(x,z) \neq \lambda(y,z)$.

**Theorem 3.1**

**Case 1.** If $\lambda(z,x) < \lambda(z,y)$ then $x \succ y$ and $\lambda(x,y) = \lambda(z,x)$.

**Case 2.** If $\lambda(z,x) > \lambda(z,y)$ then $x \preceq y$ and $\lambda(x,y) = \lambda(z,y)$.

**Proof.** For case 1, since $\lambda(z,x) < \lambda(z,y)$, $z$ is not a prefix of $x$ and $z \neq x$. If $p$ is the LCP of $z$ and $x$, $p$ must also be a prefix of $y$ because $|p| = \lambda(z,x) < \lambda(z,y)$. However, $z < x$ and $z$ is not a prefix of $x$ imply that there exists $x[\lambda(z,x)+1]$ and $z[\lambda(z,x)+1]$ in which $x[\lambda(z,x)+1] \neq z[\lambda(z,x)+1]$ but $z[\lambda(z,x)+1] = y[\lambda(z,x)+1]$. So, $p$ is not only a prefix but also the LCP of $y$ and $x$. Thus $\lambda(y,x) = |p| = \lambda(z,x)$. Combined with $z < x$, we have $x[\lambda(y,x)+1] > z[\lambda(y,x)+1] = y[\lambda(y,x)+1]$. Therefore $x \succ y$. Similar arguments hold for case 2.

Theorem 3.1 does not handle the case $\lambda(z,y) = \lambda(z,x)$ in which $\lambda(z,x) \leq \lambda(x,y)$. In this case, we determine the relative order of $x$ and $y$ by character-wise comparing them but skip $x[1] \ldots x[\lambda(z,x)]$ and $y[1] \ldots y[\lambda(z,y)]$ since they are known to be identical.

3.1.3 Merging of Annotated String Sequences

Let $X = x_1, x_2, \ldots, x_a$ and $Y = y_1, y_2, \ldots, y_b$ be ascending string sequences. Classical merging merges $X$ and $Y$ to form a single ascending string sequence $Z = z_1, z_2, \ldots, z_{a+b}$ so that $Z$ is a permutation of $X$ and $Y$. Instead of merging $X$ and $Y$ directly, we investigate merging them in their annotated string sequence forms. In addition, we use the symbol $\leftarrow$ to denote the assignment operation. For example, $u \leftarrow v$ means the value of variable $u$ is replaced by the value of variable $v$.

Let $\alpha_i$ and $\beta_j$ be strings. Set $\alpha_i \leftarrow \epsilon$, $\beta_j \leftarrow \epsilon$ and $\alpha_i \leftarrow x_{i-1}$ for $i:2 \leq i \leq a$ and $\beta_j \leftarrow y_{j-1}$ for $j:2 \leq j \leq b$ so that $X = \alpha_1\cdot x_1, \alpha_2\cdot x_2, \ldots, \alpha_a\cdot x_a$ and $Y = \beta_1\cdot y_1, \beta_2\cdot y_2, \ldots, \beta_b\cdot y_b$ respectively form the ascending annotated string sequence equivalences of $X$ and $Y$ with $\epsilon$ as their external reference strings. The LCP Merge algorithm described below merges $X$ and $Y$ into a single ascending annotated string sequence $Z = \gamma_1\cdot z_1, \gamma_2\cdot z_2, \ldots, \gamma_{a+b}\cdot z_{a+b}$ where $\gamma_1 = \epsilon$ and $\gamma_k = z_{k-1}$ for $k:2 \leq k \leq a+b$.

1) Set $i \leftarrow 1$, $j \leftarrow 1$ and $k \leftarrow 1$

2) Compare $\lambda(\alpha_i,x_j)$ and $\lambda(\beta_j,y_j)$. If $\lambda(\alpha_i,x_j) \neq \lambda(\beta_j,y_j)$, select the smaller one of $\alpha_i\cdot x_j$ and $\beta_j\cdot y_j$ by theorem 3.1 otherwise character-wise compare $x_i$ and $y_j$ starting at the $\lambda(\alpha_i,x_j) + 1$th characters of $x_i$ and $y_j$. The character-wise comparisons terminate when unmatched characters are found or when $x_i$ or (non-exclusive) $y_j$ is exhausted.

3) From the result of step 2, if $\alpha_i \cdot x_j \preceq \beta_j \cdot y_j$, set $\gamma_k \cdot z_k \leftarrow \alpha_i\cdot x_i$, set $\beta_j \leftarrow z_k$ to reannotate $y_j$ by $\lambda(x_i,y_j)$ and set $i \leftarrow i+1$ otherwise set $\gamma_k \cdot z_k \leftarrow \beta_j\cdot y_j$, set $\alpha_i \leftarrow z_k$ to reannotate $x_i$ by $\lambda(y_j,x_i)$ and set $j \leftarrow j+1$

4) Set $k \leftarrow k+1$

5) If $i \leq a$ and $j \leq b$ return to step 2

6) If $i > a$ set $\gamma_k \cdot z_k \leftarrow \beta_j\cdot y_j$, $\gamma_k+1\cdot z_{k+1} \leftarrow \beta_j+1\cdot y_j+1$, $\ldots$, $\gamma_{a+b}\cdot z_{a+b} \leftarrow \beta_{b}\cdot y_b$ otherwise set $\gamma_k \cdot z_k \leftarrow \alpha_i\cdot x_i$, $\gamma_k+1\cdot z_{k+1} \leftarrow \alpha_i+1\cdot x_{i+1}$, $\ldots$, $\gamma_{a+b}\cdot z_{a+b} \leftarrow \alpha_a\cdot x_a$

Since $z_{k-1} \prec x_i$ and $z_{k-1} \preceq y_j$ are obvious, the essence of the algorithm is the apposite application of Theorem 3.1 to maintain the invariant $\alpha_i = \beta_j = z_{k-1}$ at step 2. Initially, $i$, $j$ and $k$ are set to 1 in step 1 that the invariant holds by assuming $z_0 = \epsilon$ since $\alpha_1 = \beta_1 = \epsilon$. Step 2 through step 5 constitutes a loop which increases $i$, $j$ and $k$, repeatedly set $\gamma_k \cdot z_k$ to the smaller annotated string of $\alpha_i\cdot x_i$ and $\beta_j\cdot y_j$ until all elements of either $X$ or $Y$ has been
scanned (when \( i > a \) or \( j > b \)). In step 2, we apply Theorem 3.1 to select the smaller annotated string from \( \alpha_i \cdot x_i \) and \( \beta_j \cdot y_j \) and obtain \( \lambda(x_i, y_j) \) without comparing \( x_i \) and \( y_j \) if \( \lambda(\alpha_i, x_i) \neq \lambda(\beta_j, y_j) \), otherwise a breakdown occurs that forces us to issue a string comparison on \( x_i \) and \( y_j \) (key accesses occur) to make the selection. The comparison skips the first \( \lambda(\alpha_i, x_i) \) character-wise identical characters of \( x_i \) and \( y_j \) and then character-wise compares \( x_i \) and \( y_j \) by starting at their \( \lambda(\alpha_i, x_i) + 1 \)th character. It terminates when character-wise unmatched characters of \( x_i \) and \( y_j \) are found or when \( x_i \) or (non-exclusive) \( y_j \) is exhausted and \( \lambda(x_i, y_j) \) is obtained as a by-product. In step 3, if \( \alpha_i \cdot x_i \leq \beta_j \cdot y_j \), we set \( \gamma_k \cdot z_k \leftarrow \alpha_i \cdot x_i \), \( \beta_j \leftarrow z_k \) to reannotate \( y_j \) by \( \lambda(x_i, y_j) \) and then increase \( i \) by 1. Since \( X \) is an annotated string sequence, \( \alpha_i = x_{i-1} \) such that after the operations of step 3, \( \alpha_i = \beta_j = z_k = x_{i-1} \). In step 4, \( k \) is increased by 1. Thus the invariant \( \alpha_i = \beta_j = z_{k-1} \) holds again and is ready for the next iteration after the bound checks in step 5. Similar arguments hold for the case \( \alpha_i \cdot x_i > \beta_j \cdot y_j \). In step 6, the remaining elements of either \( X \) or \( Y \) are concatenated to \( Z \).

It is clear that \( z_k \leq z_{k+1} \) for \( k : 1 \leq k < a + b \). In order to prove \( Z \) forms an annotated string sequence, we have to show \( \gamma_k = z_{k-1} \) for \( k : 2 \leq k \leq a + b \). In the operations of the loop constituted by step 2 through step 5, the smaller one of \( \alpha_i \cdot x_i \) and \( \beta_j \cdot y_j \) is selected at step 2 and assigned to \( \gamma_k \cdot z_k \) at step 3. Since the invariant \( \alpha_i = \beta_j = z_{k-1} \) holds, \( \gamma_k \) is actually assigned \( z_{k-1} \). Thus we have \( \gamma_k = z_{k-1} \) for the elements we assigned to \( Z \) by the loop. At step 6, the remaining elements of \( X \) or \( Y \) form an annotated string sequence with \( z_{k-1} \) as its external reference because \( \alpha_i = z_{k-1} \) or \( \beta_j = z_{k-1} \). Therefore, \( Z \) is an annotated string sequence and in particular, it is an ascending annotated string sequence with an external reference \( \epsilon \) because \( \gamma_1 \) is assigned \( \epsilon \) as \( \alpha_1 = \beta_1 = \epsilon \).

Actually, LCP Merge differs from classical merging mainly on the comparison operation. Rather than compare the strings directly as in the case of classical merging, LCP merge compares in two phases: (phase A) compulsory compares the LCPs annotated to strings. If they are equal, i.e., when a breakdown occurs, the operations in phase B is carried out; (phase B) accesses the keys (strings) and compares them starting at the next characters after their LCP. Since phase A is obligatory, the number of comparison operations remains the same as classical merging. However, the number of character-wise comparisons is reduced since no character-wise comparison occurs in phase A and a fewer number of character-wise comparisons is required in phase B by not comparing their LCP.

4. Application of LCP Merge to String Sorting

As one of the application of the merging algorithm, we applied LCP Merge to build a merge sort, LCP Merge sort, for sorting strings. LCP Merge sort is basically a recursive merge sort (henceforth, we call recursive merge sort Merge sort) but differs internally in that it recursively applies LCP merge to perform merging on annotated string sequences. This requires LCP Merge sort to transform the input strings to annotated strings but this can be achieved effortlessly by annotating a zero to every string obtained at the end of the recursion of dividing the input string sequence. In other words, the strings are transformed to annotated string sequences with \( \epsilon \) as their external references.

In the following Sections 4.1, 4.2 and 4.3, we present theoretical analyses on LCP Merge sort and theoretically compare it with two other sorting algorithms. In the analyses, we assume the strings of the inputs to the algorithms are distinct without losing generality.

4.1 Computational Complexity of LCP

Merge Sort

In this section, we study the average computational complexity of LCP Merge sort for sorting \( n \) strings \( s_1, s_2, \ldots, s_n \). As in other comparison-based sorting algorithms, the computational complexity of LCP Merge sort is taken as the number of comparison operations required to sort \( n \) strings. However, the comparison operation of LCP Merge is divided into two phases as mentioned in Section 3.1.3 so that its computational complexity should be measured in terms of the number of comparisons of the LCPs annotated to the strings (phase A) and the number of character-wise comparisons of the strings (phase B). Obviously, phase A executes exactly only one integer (LCP) comparison. Thus, almost \( n \lg n \) integer comparisons are performed in the sorting process on average. On the other hand, phase B is triggered only when a breakdown occurs in phase A and the number of character-wise comparisons carried out in phase B depends on the properties
of the strings being compared. Thus, we focus on the analysis of phase B.

Let \( s_i \) \( (1 \leq i \leq n) \) be one of the \( n \) strings to be sorted, \( l_i \) be the LLCP annotated to \( s_i \) and \( \mu_i \) be the length of distinguishing prefix of \( s_i \). In the sorting process, comparisons of \( s_i \) only occur in phase B (i.e., when the comparisons of \( \langle l_i, s_i \rangle \) incur breakdowns). Each comparison of \( s_i \) starts at \( s_i[l_i + 1] \) and may cause an increment in the value of \( l_i \) at the end. The value of \( l_i \) is 0 initially and may be increased or remains unchanged between successive comparisons of \( s_i \), but eventually reaches \( \mu_i - 1 \) because \( s_i \) has to be inspected up to \( s_i[\mu_i] \) to be distinguished from the other \( n - 1 \) strings. Suppose at the end of an execution of phase B, \( s_i \) is compared up to \( s_i[k + 1] \) \((0 \leq k < \mu_i)\) so that the value of \( l_i \) is increased to \( k \). The next time when the comparison of \( \langle l_i, s_i \rangle \) incurs breakdown, \( s_i \) is compared starting at \( s_i[k + 1] \). Thus \( s_i[k + 1] \) is compared twice. The change of value in \( l_i \) is only caused by the comparison of \( s_i \), therefore, if \( \omega_i \) is the number of breakdowns encountered in sorting \( s_i \), the number of character-wise comparisons required to sort \( s_i \) is

\[
\mu_i + \omega_i - 1. \tag{1}
\]

Hence, the total number of character-wise comparisons required to sort \( n \) strings by LCP Merge sort is

\[
\sum_{i=1}^{i=n} (\mu_i + \omega_i - 1) = \sum_{i=1}^{i=n} \mu_i + \sum_{i=1}^{i=n} \omega_i - n. \tag{2}
\]

Since we have to compare the annotated strings about \( n \lg n \) times to sort them, if we let \( P_\omega \) be the probability of breakdown and \( \mu_a \) be the average length of distinguishing prefixes (ALDP) of the \( n \) strings, then by Eq. (2) the number of character-wise comparisons of LCP Merge sort for sorting \( n \) strings is about

\[
n \mu_a + P_\omega n \lg n - n = n(\mu_a - 1) + P_\omega n \lg n. \tag{3}
\]

Adding the number of comparisons of phase A to Eq. (3), the total number of comparisons of LCP Merge sort is almost

\[
n \lg n + n(\mu_a - 1) + P_\omega n \lg n. \tag{4}
\]

Assuming the input has alphabet size \( m \) and is uniformly random, about \( \log m n = \lg n / \lg m \) characters are required to differentiate the \( n \) strings from each other (Section 6.3 in Ref. 8). Hence, \( \mu_a = \lg n / \lg m \). Thus, Eq. (4) can be rewritten as

\[
n \lg n \left( 1 + P_\omega + \frac{1}{\lg m} \right) - n. \tag{5}
\]

Therefore, providing \( m > 1 \), which is very common in practice, the computational complexity of LCP Merge sort on average is \( O(n \lg n) \).

4.2 Computational Complexity of Merge Sort

We analyze the average computational complexity of Merge sort in this section.

Clearly the total number of comparisons is equal to the number of character-wise comparisons in the case of Merge sort. Let the input to Merge sort contain \( n \) strings and have an alphabet size of \( m \). Assuming both \( n \) and \( m \) are powers of 2 for simplicity, when sorting the \( n \) strings, the recursion of Merge sort starts at recursion level 1 and ends at recursion level \( \lg n \). At recursion level \( \lg n \), no merging occurs because the inputs at that recursion level contain only one string each. Thus, if \( \mu_d \) is the ALDP of the inputs at recursion level \( d \) \((1 \leq d \leq \lg n - 1)\), the total number of character-wise comparisons of Merge sort is

\[
C_M = n(\mu_1 + \mu_2 + \cdots + \mu_{\lg n - 1}). \tag{6}
\]

Assuming the inputs at recursion level \( d \) contain \( n_d \) strings each and are uniformly random, we can express \( \mu_d \) as \( \lg n_d / \lg m \) and rewrite Eq. (6) to get

\[
C_M = n \left( \frac{\lg n_1}{\lg m} + \frac{\lg n_2}{\lg m} + \cdots + \frac{\lg n_{\lg n - 1}}{\lg m} \right). \tag{7}
\]

However, \( \mu_d \) has to be at least one character by definition 2.6, but the value of \( \lg n_d / \lg m \) is smaller than 1 when \( m > n_d > 1 \), so we replace the value of \( \lg n_d / \lg m \) with 1 in those cases. Then substituting \( n_d = \frac{m}{\lg m} \) into Eq. (7) yields

\[
C_M = n \left( \frac{\lg n}{\lg m} + \frac{\lg n}{\lg m} + \cdots + \frac{\lg m}{\lg m} + 1 + \cdots + 1 \right). \tag{8}
\]

Replacing \( \lg n / \lg m \) with \( \mu_a \) and because there are \( \lg m - 1 \) levels where \( m > n_d > 1 \), we have

\[
C_M = n \left( \mu_a + \left( \mu_a - \frac{1}{\lg m} \right) + \left( \mu_a - \frac{2}{\lg m} \right) + \cdots + \left( \mu_a - \left( \frac{\lg n - \lg m}{\lg m} \right) \right) \right)\]
\[
\begin{align*}
\frac{1}{\ln m} + \frac{2}{\ln m} + \cdots + \frac{\ln n - \ln m}{\ln m} + \frac{1}{\ln m} & = n \left( \mu_a (\ln n - \ln m + 1) - \frac{1}{\ln m} - \frac{2}{\ln m} - \cdots - \frac{\ln n - \ln m}{\ln m} \right) + \frac{\ln m - 1}{\ln m} \\
& = n \ln n \left( \frac{\mu_a^2 + 1}{2\mu_a} + \frac{\mu_a - 1}{2\mu_a \ln m} \right). \quad (9)
\end{align*}
\]

From Eq. (9), we observe that the number of character-wise comparisons of Merge sort is proportional to the ALDP of the input and we also notice that Merge sort performs worse with a small alphabet size. In fact, to ensure theoretically that LCP Merge sort performs better than Merge sort, the input should have an ALDP of about 4 characters if we anticipate \( P_\omega = 1 \), i.e., the worse case of LCP Merge sort.

### 4.3 Computational Complexity of CRadix Sort

In this section, the computational complexity of CRadix sort in the average case is discussed. The reason we chose CRadix sort is simply that it is the fastest among other sorting algorithms in the experiments conducted in section 5 and theoretically efficient since it has the same computational complexity of that of MSD radix sort.

CRadix sort does not compare but groups the strings to sort the input, so we roughly compare LCP Merge sort and CRadix sort by comparing the total number of comparisons of LCP Merge sort and the number of characters inspected by CRadix sort. In theory, CRadix sort scans the strings two times (one for calculating the number of strings in each group and another one for calculating the starting memory address of each group) to group them in each phase. Assuming the input has \( n \) strings and the ALDP of them is \( \mu_a \), the average number of character-wise comparisons (average number of characters inspected) is

\[
C_R = 2n \mu_a
\]

Thus, by Eq. (4), the difference in the total number of comparisons between LCP Merge sort and CRadix sort is

\[
|n(\mu_a - (P_\omega + 1) \ln n + 1)|. \quad (11)
\]

Hence, if \( (\mu_a + 1) > (P_\omega + 1) \ln n \), LCP Merge sort has a fewer total number of comparisons. Also, by Eq. (11), LCP Merge sort can be said to be more suitable than CRadix sort when the input has a long ALDP.

### 5. Experiments

We discuss in this section the results of our experiments in the real-world performances of LCP Merge sort and other string sorting algorithms. The interface of LCP Merge sort is declared as

\[
\text{void LCPMergesort(char **strs, size_t n)}
\]

so that it is the same interface as other sorting algorithms being compared in the experiments performed in this paper for fairness. Thus, LCP Merge sort takes an array of \( n \) string pointers, sorts the strings and then returns the pointers to the sorted strings through the same array.

#### 5.1 Test Bed and Test Data

Our experiments were conducted on the following environment.

- **Model:** IBM PC Compatible
- **CPU:** AMD Athlon XP 2500+
- **Main memory:** 1.5 GB
- **1st level cache:** 128 kb
- **2nd level cache:** 512 kb
- **OS:** Windows XP Pro SP2
- **Compiler:** Visual C++ 6.0 sp6

All sorting programs were compiled with the sources obtained from their authors with the ‘maximize speed’ option and we used the four sets of test data listed below in our experiments.

- **Dataset 1 — Random String:** Strings randomly generated from the characters drawn from the uniform distribution in the range of ASCII code 33 to 126. The length of the strings is random which varies uniformly from 0 (empty string) to 19 characters.
- **Dataset 2 — URL:** Web page addresses (URL) extracted in order of occurrence from the documents of the large web track in the TREC project \(^5\),\(^6\). ‘http://’s from the start of the URLs were stripped. The average length of the URLs is 32 characters long and there are large numbers of duplicates.
- **Dataset 3 — Web Page Word:** Distinct alphabetic strings separated by non-alphabetic characters extracted in order of first occurrence from web pages excluding tags, images, and other non-textual information. The web pages are from the same source as dataset 2.
- **Dataset 4 — Genome:** DNA sequence fragments. Each fragment is 9 characters in length. There are lots of duplicates.
Table 1  Average length of the distinguish prefixes (ALDP) of the datasets.

|      | ALDP   |
|------|--------|
| dataset 1 | 3.911  |
| dataset 2 | 31.89  |
| dataset 3 | 9.279  |
| dataset 4 | 10.00  |

The test datasets are taken from Ref. 13) because they represent real world applications well. The strings in the datasets are null terminated strings, thus the actual length of the strings is 1 character longer than stated. For example, the length of the strings in dataset 4 is actually 10 characters long if the null character is considered while an empty string has a length of 1 character. In addition, there were only 10 M strings in dataset 2 originally, but we tripled it by concatenating the original data itself to get 30 M strings in order to be consistent in size with other datasets. This is not as harmful as it sounds because there were already a large number of duplicates inside original dataset 2. Table 1 shows the ALDP of the datasets.

5.2 Comparing LCP Merge sort with Merge Sort

We compared LCP Merge sort and Merge sort on the four datasets in four categories: number of character-wise comparisons, number of key accesses, total number of comparisons (the sum of the number of LLCP comparisons in phase A and the number of character-wise comparisons in phase B in case of LCP Merge sort whereas the number of character-wise comparisons in case of Merge sort) and running times. Figure 1 shows the numbers of character-wise comparisons of LCP Merge sort and Merge sort on four datasets. We observed that even though the ALDPs of the datasets vary within a factor of about 8.15, the numbers of character-wise comparisons of LCP Merge sort on datasets 1, 3 and 4 are almost the same (their lines overlapped in Fig.1) and that on dataset 2 is just about 2 times the others while the numbers of character-wise comparisons of Merge sort vary about a factor of 7.23 among the datasets. The results agree with Eq. (3) and demonstrated that LCP Merge sort is not sensitive to the ALDP of its input.

In the experiments of total number of comparisons (Fig. 2), LCP Merge sort was slightly better than Merge sort on dataset 1 but had greater improvements on other datasets. These experimental results are consistent with our theoretical results discussed in Section 4.1 and Section 4.2. For instance, in the case of dataset 1 (when \( n = 30 \text{M} \)), the number of comparison operations of LCP Merge sort estimated by Eq. (4) is almost \( 1.893n \lg n \) \( (P_c \approx 0.776) \) and that of Merge sort estimated by Eq. (9) is about \( 2.140n \lg n \) \( (m = 95) \). These two estimations fairly agree with our experimental results shown in Fig.2 and indicate LCP Merge sort only makes a little (about 13%) improvement to Merge sort in the number of character-wise comparisons on dataset 1.
From Fig. 3, we notice that the running times of LCP Merge sort are shorter than those of Merge sort on all four datasets but not to the degree of improvement that the total number of comparisons implies regardless that the total number of comparisons is a common measure of performance in sorting algorithms. For instance, LCP Merge sort is only about 2 times faster than Merge sort on dataset 2 despite that the total number of comparisons of LCP Merge sort is roughly 1/6 that of Merge sort and similar behavior is observed in the experiments on the other datasets as well. A close look at the running times on dataset 3 and dataset 4 reveals the fact that even though the total number of comparisons on both datasets are nearly the same, LCP Merge sort ran about 13% faster (when $n = 30M$) on dataset 4 than on dataset 3. Moreover, we also observed that the number of key accesses (Fig. 4) of LCP Merge sort on dataset 4 is smaller than that on dataset 3. These observations suggest that the number of key accesses has larger impact than the total number of comparisons on the running time of LCP Merge sort. The main reasons are considered to be that when the input is large, key accesses may frequently incur cache misses that have great impact on real-world performance as we reported in Ref. 11) and the cost of LLCP/character-wise comparisons is supposed to be cheaper than the cost of cache miss penalties.

5.3 Comparing LCP Merge sort with Other Sorts

In this section, we compare the running times of LCP Merge sort and four other fast string sorting algorithms known: Mulitkey Quicksort$^1$, Burstsort$^{13}$, MSD radix sort$^{10}$ and CRadix sort$^{11}$. Multikey Quicksort is a well known string sorting algorithm and is sometimes deployed as a minor sort of the main sorting algorithms for sorting some kinds of input$^{13}$. Burstsort is a cache efficient string sorting algorithm based on burst trie which is reported to be generally two times faster than Multikey Quicksort and has a computational complexity of $O(n)$. MSD radix sort is a fast radix sort for sorting strings and CRadix sort is a cache efficient variant of MSD radix sort. There are several versions of burstsort and we chose the generally faster burstsortA for our experiments. In addition, CRadix sort needs a runtime parameter (key buffer size) to be tuned for the input data to have maximum performance and we adjusted the parameter according to Ref. 11) to give the best performance on each dataset.

Figures 5a–5d show the comparisons in running time of LCP Merge sort with CRadix sort, Burstsort, MSD radix sort and Multikey Quicksort on the four datasets respectively. It can be observed that LCP Merge sort ran faster than Multikey Quicksort on all datasets (Fig. 5d). We also notice that LCP Merge sort is the winner on dataset 2. However, LCP Merge sort indeed did not run faster but just the other sorts ran much slower on dataset 2 as the figures show that the running times of the sorts fluctuated by large factors among the four datasets. For example, when $n = 30M$, the running times of LCP Merge sort on the four datasets are within a factor of not more
than 1.315 but those of CRadix sort, Burstsort, MSD radix sort and Multikey Quicksort varied about by factors of 6.326, 5.622, 8.390 and 3.093 respectively. This agrees with the theoretical analysis in Section 4.1 and are also consistent with the experimental observations in Section 5.2 which infer LCP Merge sort is not sensitive to the ALDP of its input. Moreover, we observed that LCP Merge sort and CRadix sort are the two sorts the most relatively close to linear in performance. It is surprising that Burstsort and MSD radix sort, supposed to be $O(n)$ algorithms behaved non-linearly apparently, and in particular, MSD radix sort just could not reach CRadix sort in terms of linearity even though CRadix sort is merely a cache efficient version of MSD radix sort. This is consistent again with our hypothesis that cache misses immensely affect real-world performance.

6. Conclusions

We introduced the concept of annotated string and built LCP Merge based on it. LCP Merge requires considerably less key accesses and character-wise comparisons than merging the string sequences by classical merging algorithms. However, extra spaces are required to store the LLCPs. For a non memory-rich environment, one can apply the concept of annotated string to a less memory intensive merging algorithm such as Ref. 2), although the trade off is execution speed.

We built LCP Merge sort by using LCP Merge and demonstrated its effectiveness by conducting a number of experiments. The ex-
Experimental results are consistent with theoretical analyses and showed that LCP Merge sort is practical and robust on various kinds of test data. In addition, it is observed that the number of key accesses immensely affects real-world performances of LCP Merge sort.

Moreover, LCP Merge sort is expected to be effective in suffix sorting since many typical texts have long average LLCPs as reported in Ref. 9). Application of LCP Merge sort to multifield sorting is considered to be advantageous as well. We will investigate such kinds of new applications of LCP Merge and deepen our theoretical analysis on the behavior of LCP Merge sort.

References

1) Bentley, J.L. and Sedgewick, R.: Fast Algorithms for Sorting and Searching Strings, Proc. 8th Annual ACM-SIAM Symp. Discrete Algorithms, pp.360–369 (1997).
2) Dvorak, S. and Durian, B.: Stable Linear Time Sublinear Space Merging, Computer Journal, Vol.30, No.4, pp.372–375 (1987).
3) Farach, M.: Optimal Suffix Tree Construction with Large Alphabets, Proc. 38th Symp. on Foundations of Comp. Sci. ’97, pp.137–143 (1997).
4) Futamura, Y., Futamura, N. and Ng, W.H.: Leaves Optimal Adaptive Sort and LLCP Merge, JSSST Conference, Vol.1D-2 (2004).
5) Harman, D.: Overview of the Second Text Retrieval Conference (TREC-2), Inform. Process. Mgmt., Vol.31, No.3, pp.271–289 (1995).
6) Hawking, D., Craswell, N., Thistlewaite, P. and Harman, D.: Results and Challenges in Web Search Evaluation, Proc. 8th Intl. Conf. WWW, Toronto, Canada, pp.1321–1330 (1999).
7) Kasai, T., Lee, G., Arimura, H., Arikawa, S. and Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, Proc. 12th Symp. on CPM, LNCS 2089, pp.181–192 (2001).
8) Knuth, D.E.: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison Wesley, 2nd edition (1998).
9) Manzini, G. and Ferragina, P.: Engineering a Lightweight Suffix Array Construction Algorithm, Proc. 10th Euro. Symp. on Algorithms, LNCS 2461, pp.698–710 (2002).
10) McIlroy, P.M., Bostic, K. and McIlroy, M.D.: Engineering Radix Sort, Comput. Syst., Vol.6, No.1, pp.5–27 (1993).
11) Ng, W.H. and Kakehi, K.: Cache Efficient Radix Sort for String Sorting, IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, Vol.E 90-A, No.2, pp.457–464 (2007).
12) Nilsson, S.: Radix Sorting and Searching, Ph.D. Thesis, Dept. Comput. Sci., Lund University, Lund, Sweden (1996).
13) Sinha, R. and Zobel, J.: Efficient Trie-based Sorting of Large Sets of Strings, 26th Australasian Comput. Sci. Conf. (ACSC), pp.11–18 (2003).

(Received July 3, 2007)  
(Accepted November 6, 2007)  
(Released February 6, 2008)

Waihong Ng is a doctoral student of the Department of Computer Science, Waseda University and obtained his B.Eng. and MICSc degrees both from Waseda University in 1998 and 2000 respectively. He has been researching sorting algorithms since he was an undergraduate student. Student member of IEICE, IPSJ and JSSST. Research interest: algorithms and data structures, algorithm engineering and system evaluation.

Katsuhiko Kakehi has been a Professor in the Department of Computer Science, Waseda University since 1991. 1968 Bs.Eng. the University of Tokyo, 1970 Ms.Eng. the University of Tokyo in Applied Mathematics. Assistant Prof. (1974), then Associate Prof. (1976) of Rikkyo University Math. Dept. Prof. of Waseda University Math. Dept (1986). IPSJ fellow, member of JSSST, ACM and MSJ. Research area: programming languages, formalization and implementation.