Simpler and Faster Lempel Ziv Factorization

Keisuke Goto and Hideo Bannai
Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
{keisuke.gotou,bannai}@inf.kyushu-u.ac.jp

Abstract

We present a new, simple, and efficient approach for computing the Lempel-Ziv (LZ77) factorization of a string in linear time, based on suffix arrays. Computational experiments on various data sets show that our approach constantly outperforms the fastest previous algorithm LZ_OG (Ohlebusch and Gog 2011), and can be up to 2 to 3 times faster in the processing after obtaining the suffix array, while requiring the same or a little more space.

1 Introduction

The LZ77 factorization [1] of a string captures important properties concerning repeated occurrences of substrings in the string, and has obvious applications in the field of data compression, as well as being the key component to various efficient algorithms on strings [2, 3]. Consequently, many algorithms for its efficient calculation have been proposed. The LZ77 factorization of a string S is a factorization $S = f_1 \cdots f_n$ where each factor f_k is either (1) a single character if that character does not occur in $f_1 \cdots f_{k-1}$, or, (2) the longest prefix of the rest of the string which occurs at least twice in $f_1 \cdots f_k$.

A naïve algorithm that computes the longest common prefix with each of the $O(N)$ previous positions only requires $O(1)$ working space (excluding the output), but can take $O(N^2)$ time, where N is the length of the string. Using string indicies such as suffix trees [4] and on-line algorithms to construct them [5], the LZ factorization can be computed in an on-line manner in $O(N \log |\Sigma|)$ time and $O(N)$ space, where $|\Sigma|$ is the size of the alphabet.

Most recent efficient linear time algorithms are off-line, running in $O(N)$ time for integer alphabets using $O(N)$ space (See Table 1). They first construct the suffix array [6] of the string, and compute an array called the Longest Previous Factor (LPF) array from which the LZ factorization can be easily computed [7, 8, 9, 10, 11]. Many algorithms of this family first compute the longest common prefix (LCP) array prior to the computation of the LPF array. However, the computation of the LCP array is also costly. The algorithm CI1 (COMPUTE_LPF) of [12], and the algorithm LZ_OG [10] cleverly avoids its computation and directly computes the LPF array.

An important observation here is that the LPF is actually more information than is required for the computation of the LZ factorization, i.e., if our objective is the LZ factorization, we only use a subset of the entries in the LPF . However, the above algorithms focus

Table 1. Fast Linear time LZ-Factorization Algorithms based on Suffix Arrays

algorithm	worst case time	SA	LCP	LPF, PrevOcc
CI1 [12]	$\Theta(N)$	✓		✓
CI2 [12]	$\Theta(N)$	✓	✓	✓
CPS1 [8]	$\Theta(N)$	✓	✓	
CIS [7]	$\Theta(N)$	✓	✓	✓
CII [9]	$\Theta(N)$	✓	✓	✓
LZ_OG [10]	$\Theta(N)$	✓		✓
this work, [14]	$\Theta(N)$	✓		
Naive	$\Theta(N^2)$			

on computing the entire LPF array, perhaps since it is difficult to determine beforehand, which entries of LPF are actually required. Although some algorithms such as a variant of CPS1 [8] or CPS2 in [8] avoid computation of LPF, they either require the LCP array, or do not run in linear worst case time and are not as efficient. (See [11] for a survey.)

In this paper, we propose a new approach to avoid the computation of LCP and LPF arrays altogether, by combining the ideas of the naïve algorithm with those of CI1 and LZ_OG, and still achieve worst case linear time. The resulting algorithm is surprisingly both simple and efficient.

Computational experiments on various data sets shows that our algorithm constantly outperforms LZ_OG [10], and can be up to 2 to 3 times faster in the processing after obtaining the suffix array, while requiring the same or a little more space.

Although our algorithm might be considered as a simple combination of ideas appearing in previous works, this paper is one of the first to propose, implement and evaluate this combination. We note that algorithms that avoid the computation of LCP and LPF based on similar ideas as in this paper were developed independently and almost simultaneously by Kempa and Puglisi [13] and Kärkkäinen et al. [14]. Since we did not have knowledge of their work until very recently, we have not made comparisons between them. The worst case time complexity of [13] is not independent of alphabet size, but is fast and space efficient. In the more recent manuscript [14], two new linear time algorithms which outperform all previous algorithms (including ours) in terms of time and space are proposed, asserting the potential of this approach.

2 Preliminaries

Let $\mathcal{N}$ be the set of non-negative integers. Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The set of prefixes of T is denoted by $prefix(T)$. The *longest common prefix* of strings X, Y, denoted $lcp(X, Y)$, is the longest string in $prefix(X) \cap prefix(Y)$.

The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i..j]$ for

$1 \leq i \leq j \leq |T|$. For convenience, let $T[i..j] = \varepsilon$ if $j < i$, and $T[|T| + 1] = \$$ where $\$$ is a special delimiter character that does not occur elsewhere in the string.

2.1 Suffix Arrays

The suffix array [6] SA of any string T is an array of length $|T|$ such that for any $1 \leq i \leq |T|$, $SA[i] = j$ indicates that $T[j : |T|]$ is the i-th lexicographically smallest suffix of T. For convenience, assume that $SA[0] = |T| + 1$. The inverse array SA^{-1} of SA is an array of length $|T|$ such that $SA^{-1}[SA[i]] = i$. As in [15], let Φ be an array of length $|T|$ such that $\Phi[SA[1]] = |T|$ and $\Phi[SA[i]] = SA[i-1]$ for $2 \leq i \leq |T|$, i.e., for any suffix $j = SA[i]$, $\Phi[j] = SA[i-1]$ is the immediately preceding suffix in the suffix array. The suffix array SA for any string of length $|T|$ can be constructed in $O(|T|)$ time regardless of the alphabet size, assuming an integer alphabet (e.g. [16]). All our algorithms will assume that the SA is already computed. Given SA, arrays SA^{-1} and Φ can easily be computed in linear time by a simple scan.

2.2 LZ Encodings

LZ encodings are dynamic dictionary based encodings with many variants. The variant we consider is also known as the s-factorization [17].

Definition 1 (LZ77-factorization) *The s-factorization of a string T is the factorization $T = f_1 \cdots f_n$ where each s-factor $f_k \in \Sigma^+$ $(k = 1, \ldots, n)$ is defined inductively as follows: $f_1 = T[1]$. For $k \geq 2$: if $T[|f_1 \cdots f_{k-1}| + 1] = c \in \Sigma$ does not occur in $f_1 \cdots f_{k-1}$, then $f_k = c$. Otherwise, f_k is the longest prefix of $f_k \cdots f_n$ that occurs at least twice in $f_1 \cdots f_k$.*

Note that each LZ factor can be represented in constant space, i.e., a pair of integers where the first and second elements respectively represent the length and position of a previous occurrence of the factor. If the factor is a new character and the length of its previous occurrence is 0, the second element will encode the new character instead of the position. For example the s-factorization of the string $T = \texttt{abaababababaaaaabbabab}$ is `a`, `b`, `a`, `aba`, `baba`, `aaaa`, `b`, `babab`. This can be represented as $(0, \texttt{a})$, $(0, \texttt{b})$, $(1, 1)$, $(3, 1)$, $(4, 5)$, $(4, 10)$, $(1, 2)$, $(5, 5)$.

We define two functions LPF and $PrevOcc$ below. For any $1 \leq i \leq N$, $LPF(i)$ is the longest length of longest common prefix between $T[i : N]$ and $T[j : N]$ for any $1 \leq j < i$, and $PrevOcc(i)$ is a position j which achieves gives $LPF(i)$[1]. More precisely,

$$LPF(i) = \max(\{0\} \cup \{lcp(T[i : N], T[j : N]) \mid 1 \leq j < i\})$$

and

$$PrevOcc(i) = \begin{cases} -1 & \text{if } LPF(i) = 0 \\ j & \text{otherwise} \end{cases}$$

[1]There can be multiple choices of j, but here, it suffices to fix one.

Algorithm 1: LZ Factorization from LPF and $PrevOcc$ arrays

Input : String T, LPF, $PrevOcc$

```
1 p ← 1;
2 while p ≤ N do
3     if LPF[p] = 0 then Output: (1, T[p])
4     else Output: (LPF[p], PrevOcc[p])
5     p ← p + max(1, LPF[p]);
```

where j satisfies $1 \leq j < i$, and $T[i : i + LPF(i) - 1] = T[j : j + LPF(i) - 1]$. Let $p_k = |f_1 \cdots f_{k-1}| + 1$. Then, f_k can be represented as a pair $(LPF(p_k), PrevOcc(p_k))$ if $LPF(p_k) > 0$, and $(0, T[p_k])$ otherwise.

Most recent fast linear time algorithms for computing the LZ factorization calculate LPF and $PrevOcc$ for all positions $1 \leq i \leq N$ of the text and store the values in an array, and then use these values as in Algorithm 1 to output the LZ factorization.

3 Algorithm

We first describe the naïve algorithm for calculating the LZ factorization of a string, and analyze its time complexity The naïve algorithm does not compute all values of LPF and $PrevOcc$ as explicit arrays, but only the values required to represent each factor. The procedure is shown in Algorithm 2. For a factor starting at position p, the algorithm computes $LPF(p)$ and $PrevOcc(p)$ by simply looking at each of its $p - 1$ previous positions, and naively computes the longest common prefix (lcp) between each previous suffix and the suffix starting at position p, and outputs the factor accordingly. At first glance, this algorithm looks like an $O(N^3)$ time algorithm since there are 3 nested loops. However, the total time can be bounded by $O(N^2)$, since the total length of the longest lcp's found for each p in the algorithm, i.e., the total length of the LZ factors found, is N. More precisely, let the LZ factorization of string T of length N be $f_1 \cdots f_n$, and $p_k = |f_1 \cdots f_{k-1}| + 1$ as before. Then, the number of character comparisons executed in Line 6 of Algorithm 2 when calculating f_k is at most $(p_k - 1)|f_k + 1|$, and the total can be bounded: $\sum_{k=1}^{n}(p_k - 1)|f_k + 1| \leq N \sum_{k=1}^{n} |f_k + 1| = O(N^2)$. An important observation here is that if we can somehow reduce the number of previous candidate positions for naïvely computing lcp's (i.e. the choice of j in Line 4 of Algorithm 2) from $O(N)$ to $O(1)$ positions, this would result in a $O(N)$ time algorithm. This very simple observation is the first key to the linear running times of our new algorithms.

To accomplish this, our algorithm utilizes yet another simple but key observation made in [12]. Since suffixes in the suffix arrays are lexicographically sorted, if we fix a suffix $SA[i]$ in the suffix array, we know that suffixes appearing closer in the suffix array will have longer longest common prefixes with suffix $SA[i]$.

For any position $1 \leq i \leq N$ of the suffix array, let

$$\begin{aligned} PSV_{lex}[i] &= \max(\{0\} \cup \{1 \leq j < i \mid SA[j] < SA[i]\}) \\ NSV_{lex}[i] &= \min(\{0\} \cup \{N \geq j > i \mid SA[j] < SA[i]\}) \end{aligned}$$

Algorithm 2: Naïve Algorithm for Calculating LZ factorization

```
Input : String T
p ← 1;
while p ≤ |T| do
    LPF ← 0;
    for j ← 1, ..., p − 1 do
        l ← 0;
        while T[j + l] = T[p + l] do l ← l + 1;    // l ← lcp(T[j : N], T[p : N])
        if l > LPF then LPF ← l; PrevOcc ← j;
    if LPF > 0 then Output: (LPF, PrevOcc)
    else Output: (0, T[p])
    p ← p + max(1, LPF);
```

i.e., for the suffix starting at text position $SA[i]$, the values $PSV_{lex}[i]$ and $NSV_{lex}[i]$ represent the lexicographic rank of the suffixes that start before it in the string and are lexicographically closest (previous and next) to it, or 0 if such a suffix does not exist. From the above arguments, we have that for any text position $1 \leq p \leq N$,

$$\begin{aligned} LPF(p) = \max(&lcp(T[SA[PSV_{lex}[SA^{-1}[p]]] : N], T[p : N]), \\ &lcp(T[SA[NSV_{lex}[SA^{-1}[p]]] : N], T[p : N])). \end{aligned}$$

The above observation or its variant has been used as the basis for calculating $LPF(i)$ for all $1 \leq i \leq N$ in linear time in practically all previous linear time algorithms for LZ factorization based on the suffix array. In [10], they consider (implicitly) the arrays in text order rather than lexicographic order. In this case,

$$\begin{aligned} PSV_{text}[SA[i]] &= SA[PSV_{lex}[i]] \\ NSV_{text}[SA[i]] &= SA[NSV_{lex}[i]] \end{aligned}$$

and therefore

$$LPF(p) = \max(lcp(T[PSV_{text}[p]] : N], T[p : N]), lcp(T[NSV_{text}[p]] : N], T[p : N])).$$

While [12] and [10] utilize this observation to compute all entries of LPF in linear time, we utilize it in a slightly different way as mentioned previously, and use it to reduce the candidate positions for calculating $PrevOcc(i)$ (i.e. the choice of j in Algorithm 2) to only 2 positions. The key idea of our approach is in the combination of the above observation with the amortized analysis of the naïve algorithm, suggesting that we can defer the computation of the values of LPF until we actually require them for the LZ factorization and still achieve linear worst case time. If $PSV_{lex}[i]$ and $NSV_{lex}[i]$ (or $PSV_{text}[i]$ and $NSV_{text}[i]$) are known for all $1 \leq i \leq N$, the linear running time of the algorithm follows from the previous arguments. The basic structure of our algorithm is shown in Algorithm 3 when using PSV_{lex} and NSV_{lex}. Note that it is easy to replace them with PSV_{text} and NSV_{text}, and in such case, SA and SA^{-1} are not necessary once we have PSV_{text} and NSV_{text}.

What remains is how to compute $PSV_{lex}[i]$ and $NSV_{lex}[i]$, or, $PSV_{text}[i]$ and $NSV_{text}[i]$ for all $1 \leq i \leq N$. This can be done in several ways. We consider 3 variations.

The first is a computation of $PSV_{lex}[i]$, $NSV_{lex}[i]$ using a simple linear time scan of the suffix array with the help of a stack. The procedure is shown in Algorithm 4. This variant requires the text, and the arrays SA, SA^{-1}, PSV_{lex}, NSV_{lex} and a stack. The total space complexity is $17N + 4S_{max}$ bytes assuming that an integer occupies 4 bytes, where S_{max} is the maximum size of the stack during the execution of the algorithm and can be $\Theta(n)$ in the worstcase. We will call this variant BGS.

The other two is a process called *peak elimination*, which is very briefly described in [12] for lexicographic order (Shown in Algorithms 5 and 6), and in [10] for text order (Shown in Algorithms 7 and 8). In peak elimination, each suffix i and its lexicographically preceding suffix j ($SA^{-1}[j] + 1 = SA^{-1}[i]$) is examined in some order of i (lexicographic or text order). For simplicity, we only briefly explain the approach for text order. If $i > j$, this means that $PSV_{text}[i] = j$ and if $i < j$, $NSV_{text}[j] = i$. When both values of $PSV_{text}[i]$ and $NSV_{text}[i]$ are determined, i is identified as a peak. Given a peak i, it is possible to *eliminate* it, and determine the value of either $NSV_{text}[PSV_{text}[i]]$ (which will be $NSV_{text}[i]$ if $PSV_{text}[i] > NSV_{text}[i]$) or $PSV_{text}[NSV_{text}[i]]$ (which will be $PSV_{text}[i]$ if if $PSV_{text}[i] < NSV_{text}[i]$)), and this process is repeated. The algorithm runs in linear time since each position can be eliminated only once. The procedure for lexicographic order is a bit simpler since the lexicographic order of calculation implies that $PSV_{lex}[i]$ will always be determined before $NSV_{lex}[i]$.

The algorithm of [10] actually computes the arrays LPF and $PrevOcc$ directly without computing PSV_{text} and NSV_{text}. The algorithm we show is actually a simplification, deferring the computation of LPF and $PrevOcc$, computing PSV_{text} and NSV_{text} instead.

For lexicographic order, we need the text and the arrays SA, SA^{-1}, PSV_{lex}, NSV_{lex} and no stack, giving an algorithm with $17N$ bytes of working space. We will call this variant BGL. For text order, although the Φ array is introduced instead of the SA^{-1} array, the suffix array is not required after its computation. Therefore, by reusing the space of SA for PSV_{text}, the total space complexity can be reduced to $13N$ bytes. We will call this variant BGT. Note that although $peakElim_{lex}$ and $peakElim_{text}$ are shown as recursive functions for simplicity, they are tail recursive and thus can be optimized as loops and will not require extra space on the call stack.

3.1 Interleaving PSV and NSV

Since accesses to PSV and NSV occur at the same or close indices, it is possible to improve the memory locality of accesses by interleaving the values of PSV and NSV, maintaining them in a single array as follows. Let $PNSV$ be an array of length $2N$, and for each position $1 \leq i \leq 2N$, $PNSV[i] = PSV[j]$ if $i \mod 2 \equiv 0$, $NSV[j]$ otherwise, where $j = \lfloor i/2 \rfloor$. Naturally, for any $1 \leq i \leq N$, PSV and NSV can be accessed as $PSV[i] = PNSV[2i]$ and $NSV[i] = PNSV[2i + 1]$. This interleaving can be done for both lexicographic order and text order. We will call the variants of our algorithms that incorporate this optimization, iBGS, iBGL, iBGT.

Algorithm 3: Basic Structure of our Algorithms.

Input : String T

Calculate $PSV_{lex}[i]$ and $NSV_{lex}[i]$ for all $i = 1...N$;
$p \leftarrow 1$;
while $p \leq N$ **do**
    $LPF \leftarrow 0$;
    for $j \in \{SA[PSV_{lex}[SA^{-1}[p]]], SA[NSV_{lex}[SA^{-1}[p]]]\}$ **do**
        $l \leftarrow 0$;
        while $T[j+l] = T[p+l]$ **do** $l \leftarrow l+1$; // $l \leftarrow lcp(T[j:N], T[p:N])$
        if $l > LPF$ **then** $LPF \leftarrow l$; $PrevOcc \leftarrow j$;
    if $LPF > 0$ **then Output**: $(LPF, PrevOcc)$
    else Output: $(0, T[p])$
    $p \leftarrow p + \max(1, LPF)$;

Algorithm 4: Calculating PSV_{lex} and NSV_{lex} from SA

Input : Suffix array SA

Output: PSV_{lex}, NSV_{lex}

Let S be an empty stack;
for $i \leftarrow 1$ *to* N **do**
    $x \leftarrow SA[i]$;
    while (**not** $S.empty()$) **and** $(SA[S.top()] > x)$ **do**
        $NSV_{lex}[S.top()] \leftarrow i$; $S.pop()$;
    $PSV_{lex}[i] \leftarrow$ **if** $S.empty()$ **then** 0 **else** $S.top()$;
    $S.push(i)$;
while not $S.empty()$ **do**
    $NSV_{lex}[S.top()] \leftarrow 0$; $S.pop()$;

4 Computational Experiments

We implement and compare our algorithms with LZ_OG since it has been shown to be the most time efficient in the experiments of [10]. We also implement a variant LZ_iOG which incorporates the interleaving optimization for LPF and $PrevOcc$ arrays. We have made the source codes publicly available at `http://code.google.com/p/lzbg/`.

All computations were conducted on a Mac Xserve (Early 2009) with 2 x 2.93GHz Quad Core Xeon processors and 24GB Memory, only utilizing a single process/thread at once. The programs were compiled using the GNU C++ compiler (`g++`) 4.2.1 with the `-fast` option for optimization. The running times are measured in seconds, starting from after the suffix array is built, and the average of 10 runs is reported.

We use the data of `http://www.cas.mcmaster.ca/~bill/strings/`, used in previous work. Table 2 shows running times of the algorithms, as well as some statistics of the dataset. The running times of the fastest algorithm for each data is shown in bold.

Algorithm 5: Calculating PSV_{lex} and NSV_{lex} from SA by Peak Elimination.

Input : Suffix array SA

for $i \leftarrow 1$ *to* N **do** $NSV_{lex}[i] \leftarrow 0$;

$PSV_{lex}[1] \leftarrow 0$;

for $i \leftarrow 2$ *to* N **do** $peakElim_{lex}(i-1, i)$;

Algorithm 6: Peak Elimination $peakElim_{lex}(j, i)$ in Lexicographic Order.

if $j = 0$ **or** $SA[j] < SA[i]$ **then**

  $PSV_{lex}[i] \leftarrow j$;

else // $j \geq 1$ **and** $SA[j] > SA[i]$

  $NSV_{lex}[j] \leftarrow i$;

  $peakElim_{lex}(PSV_{lex}[j], i)$; // j was peak.

The fastest running times for the variant that uses only $13N$ bytes is prefixed with '▷'.

The results show that all the variants of our algorithms constantly outperform LZ_OG and even LZ_iOG for all data tested, and in some cases can be up to 2 to 3 times faster. We can see that iBGS is fastest when the data is not extremely repetitive, and the average length of the factor is not so large, while iBGT is fastest for such highly repetitive data. iBGT is also the fastest when we restrict our attention to the algorithms that use only $13N$ bytes of work space.

Acknowledgments: We thank Dr. Simon Gog for sharing his implementation of LZ_OG, and Dr. Simon Puglisi for sending us the manuscripts [13, 14].

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. IT-23, no. 3, pp. 337–343, 1977.

[2] R. Kolpakov and G. Kucherov, "Finding maximal repetitions in a word in linear time," in *Proc. FOCS 1999*, 1999, pp. 596–604.

[3] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre, "Linear-time computation of local periods," *Theoretical Computer Science*, vol. 326, no. 1-3, pp. 229–240, 2004.

[4] P. Weiner, "Linear pattern-matching algorithms," in *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*. Institute of Electrical Electronics Engineers, New York, 1973, pp. 1–11.

[5] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[6] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Computing*, vol. 22, no. 5, pp. 935–948, 1993.

Algorithm 7: Calculating PSV_{text} and NSV_{text} from SA using Φ.

Input : Suffix array SA

$\Phi[SA[1]] \leftarrow N$;
for $i \leftarrow 2$ *to* N **do** $\Phi[SA[i]] \leftarrow SA[i-1]$;
for $i \leftarrow 1$ *to* N **do**
    $PSV_{text}[i] \leftarrow \perp$; $NSV_{text}[i] \leftarrow \perp$;
for $i \leftarrow 1$ *to* N **do** $peakElim_{text}(\Phi[i], i)$;

Algorithm 8: Peak Elimination $peakElim_{text}(j, i)$

if $j < i$ **then**
    $PSV_{text}[i] \leftarrow j$;
    if $NSV_{text}[i] \neq \perp$ **then** $peakElim_{text}(j, NSV_{text}[i])$; `// i was peak.`
else `//` $j > i$
    $NSV_{text}[j] \leftarrow i$;
    if $PSV_{text}[j] \neq \perp$ **then** $peakElim_{text}(PSV_{text}[j], i)$; `// j was peak.`

[7] M. Crochemore, L. Ilie, and W. F. Smyth, "A simple algorithm for computing the Lempel Ziv factorization," in *Proc. DCC 2008*, 2008, pp. 482–488.

[8] G. Chen, S. Puglisi, and W. Smyth, "Lempel-Ziv factorization using less time & space," *Mathematics in Computer Science*, vol. 1, no. 4, pp. 605–623, 2008.

[9] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń, "LPF computation revisited," in *Proc. IWOCA 2009*, 2009, pp. 158–169.

[10] E. Ohlebusch and S. Gog, "Lempel-Ziv factorization revisited," in *Proc. CPM'11*, 2011, pp. 15–26.

[11] A. Al-Hafeedh, M. Crochemore, L. Ilie, J. Kopylov, W. Smyth, G. Tischler, and M. Yusufu, "A comparison of index-based Lempel-Ziv LZ77 factorization algorithms," *ACM Computing Surveys*, in press.

[12] M. Crochemore and L. Ilie, "Computing longest previous factor in linear time and applications," *Information Processing Letters*, vol. 106, no. 2, pp. 75–80, 2008.

[13] D. Kempa and S. J. Puglisi, "Lempel-Ziv factorization: Simple, fast, practical," in *Proc. ALENEX'13*, 2013.

[14] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Linear time Lempel-Ziv factorization: Simple, fast, small," 2012, arXiv:1212.2952.

[15] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *CPM*, 2009, pp. 181–192.

[16] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *Proc. ICALP 2003*, 2003, pp. 943–955.

[17] M. Crochemore, "Linear searching for a square in a word," *Bulletin of the European Association of Theoretical Computer Science*, vol. 24, pp. 66–72, 1984.

Table 2. Running times (seconds) of algorithms and various statistics for the data set of `http://www.cas.mcmaster.ca/~bill/strings/`.

File Name	LZ_OG $13N$	LZ_iOG $13N$	BGS $17N+4S_{max}$	iBGS $17N+4S_{max}$	BGL $17N$	iBGL $17N$	BGT $13N$	iBGT $13N$	$\|\Sigma\|$	Text Size N	# of LZ factors	Average Length of Factor	S_{max}
E.coli	0.64	0.58	0.26	**0.23**	0.33	0.29	0.45	▷ 0.37	4	4638690	432791	10.72	36
bible	0.37	0.34	0.20	**0.19**	0.25	0.22	0.27	▷ 0.24	63	4047392	337558	11.99	42
chr19.dna4	10.05	9.25	4.40	**4.00**	5.33	4.71	7.64	▷ 6.54	4	63811651	4411679	14.46	58
chr22.dna4	5.37	4.91	2.27	**2.06**	2.77	2.44	4.09	▷ 3.45	4	34553758	2554184	13.53	43
fib_s2178309	0.06	0.06	0.05	0.06	0.06	0.05	0.05	▷ **0.05**	2	2178309	31	70268.00	16
fib_s3524578	0.11	0.11	0.10	0.10	0.10	0.10	0.10	▷ **0.09**	2	3524578	32	110143.00	16
fib_s5702887	0.18	0.18	0.15	0.16	0.16	0.15	0.15	▷ **0.14**	2	5702887	33	172815.00	17
fib_s9227465	0.30	0.30	0.26	0.27	0.27	0.26	0.26	▷ **0.24**	2	9227465	34	271396.00	17
fib_s14930352	0.50	0.49	0.43	0.44	0.44	0.43	0.42	▷ **0.39**	2	14930352	35	426581.00	18
fss9	0.09	0.08	0.08	0.08	0.08	0.08	0.07	▷ **0.07**	2	2851443	40	71286.10	22
fss10	0.40	0.39	0.36	0.37	0.36	0.35	0.34	▷ **0.32**	2	12078908	44	274521.00	24
howto	4.20	3.91	2.30	**2.15**	2.79	2.51	3.28	▷ 2.91	197	39422105	3063929	12.87	616
mozilla	5.30	4.95	3.19	**3.13**	3.91	3.65	4.31	▷ 3.86	256	51220480	6898100	7.43	3964
p1Mb	0.08	0.07	**0.05**	0.05	0.06	0.06	0.05	▷ 0.05	23	1048576	216146	4.85	38
p2Mb	0.23	0.21	**0.11**	0.12	0.15	0.15	0.17	▷ 0.14	23	2097152	406188	5.16	40
p4Mb	0.58	0.52	0.26	**0.26**	0.35	0.33	0.43	▷ 0.35	23	4194304	791583	5.30	42
p8Mb	1.27	1.15	0.55	**0.55**	0.73	0.70	0.94	▷ 0.78	23	8388608	1487419	5.64	898
p16Mb	2.70	2.43	1.18	**1.16**	1.52	1.46	2.08	▷ 1.74	23	16777216	2751022	6.10	898
p32Mb	5.58	5.02	2.47	**2.44**	3.14	3.03	4.43	▷ 3.74	24	33554432	5040051	6.66	898
rndA2_4Mb	0.49	0.45	0.20	**0.18**	0.24	0.20	0.35	▷ 0.28	2	4194304	201910	20.77	36
rndA2_8Mb	1.08	0.99	0.42	**0.38**	0.50	0.43	0.77	▷ 0.63	2	8388608	385232	21.78	37
rndA21_4Mb	0.64	0.58	0.28	**0.28**	0.38	0.37	0.47	▷ 0.37	21	4194304	970256	4.32	34
rndA21_8Mb	1.43	1.28	0.61	**0.60**	0.83	0.79	1.05	▷ 0.85	21	8388608	1835235	4.57	37
rndA255_4Mb	0.65	0.58	**0.38**	0.39	0.51	0.47	0.49	▷ 0.40	255	4194304	2005584	2.09	35
rndA255_8Mb	1.43	1.27	0.84	**0.84**	1.12	1.04	1.10	▷ 0.92	255	8388608	3817588	2.20	38