Lempel-Ziv Factorization: LZ77 without Window

Enno Ohlebusch

May 13, 2016

1 Suffix arrays

To construct the suffix array of a string S boils down to sorting all suffixes of S in lexicographic order (also known as alphabetical order, dictionary order, or lexical order). This order is induced by an order on the alphabet Σ. In this manuscript, Σ is an ordered alphabet of constant size σ. It is sometimes convenient to regard Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] < \Sigma[2] < \cdots < \Sigma[\sigma]$. Conversely, each character in Σ is mapped to a number in $\{1, \ldots, \sigma\}$. The smallest character is mapped to 1, the second smallest character is mapped to 2, and so on. In this way, we can use a character as an index for an array.

Definition 1.1 Let $<$ be a total order on the alphabet Σ. This order induces the *lexicographic order* on Σ^* (which we again denote by $<$) as follows: For $s, t \in \Sigma^*$, define $s < t$ if and only if either s is a proper prefix of t or there are strings $u, v, w \in \Sigma^*$ and characters $a, b \in \Sigma$ with $a < b$ so that $s = uav$ and $t = ubw$.

To determine the lexicographic order of two strings, their first characters are compared. If they differ, then the string whose first character comes earlier in the alphabet is the one which comes first in lexicographic order. If the first characters are the same, then the second characters are compared, and so on. If a position is reached where one string has no more characters to compare while the other does, then the shorter string comes first in lexicographic order.

In algorithms that need to determine the lexicographic order of two suffixes of the same string S, a cumbersome distinction between "has more characters" and "has no more characters" can be avoided by appending the special symbol \$ (called *sentinel character*) to S. In the following, we assume that \$ is smaller than all other elements of the alphabet Σ. If \$ occurs nowhere else in S and the lexicographic order of two suffixes of S is determined as described above, then it cannot happen that one suffix has no more characters to compare. As we shall see later, there are other situations in which it is convenient to append the special symbol \$ to a string.

i	SA	ISA	$S_{\mathsf{SA}[i]}$
1	3	5	$aataatg$
2	6	7	$aatg$
3	4	1	$ataatg$
4	7	3	atg
5	1	8	$ctaataatg$
6	9	2	g
7	2	4	$taataatg$
8	5	9	$taatg$
9	8	6	tg

Figure 1: Suffix array and inverse suffix array of the string $S = ctaataatg$.

Definition 1.2 Let S be a string of length n. For every i, $1 \leq i \leq n$, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic order of the n suffixes of the string S. That is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$.

The *inverse suffix array* ISA is an array of size n so that for any k with $1 \leq k \leq n$ the equality $\mathsf{ISA}[\mathsf{SA}[k]] = k$ holds.

The inverse suffix array is sometimes also called *rank*-array because $\mathsf{ISA}[i]$ specifies the rank of the i-th suffix among the lexicographically ordered suffixes. More precisely, if $j = \mathsf{ISA}[i]$, then suffix S_i is the j-th lexicographically smallest suffix. Obviously, the inverse suffix array can be computed in linear time from the suffix array. Figure 1 shows the suffix array and the inverse suffix array of the string $S = ctaataatg$.

The suffix array was devised by Manber and Myers [MM93] and independently by Gonnet et al. [GBYS92] under the name PAT array. Ten years later, it was shown independently and contemporaneously by Kärkkäinen and Sanders [KS03], Kim et al. [KSPP03], Ko and Aluru [KA03], and Hon et al. [HSS03] that a direct linear-time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms (SACAs) are known [PST07].

2 Lempel-Ziv factorization

For 30 years the Lempel-Ziv factorization [ZL77] of a string has played an important role in data compression (e.g. it is used in `gzip`), and more recently it was used as the basis of linear-time algorithms for the detection of all maximal repetitions (runs) in a string.

Definition 2.1 Let S be a string of length n on an alphabet Σ. The *Lempel-Ziv factorization* (or LZ-factorization for short) of S is a factorization $S = s_1 s_2 \cdots s_m$ so that each factor s_j, $1 \leq j \leq m$, is either

Algorithm 1 Reconstruction of the string S, given its LZ-factorization $(p_1, \ell_1), \ldots, (p_m, \ell_m)$.

```
i ← 1
for j ← 1 to m do
  if ℓ_j = 0 then
    S[i] ← p_j
    i ← i + 1
  else
    for k ← 0 to ℓ_j − 1 do
      S[i] ← S[p_j + k]
      i ← i + 1
```

(a) a letter $c \in \Sigma$ that does not occur in $s_1 s_2 \cdots s_{j-1}$, or

(b) the longest substring of S that occurs at least twice in $s_1 s_2 \cdots s_j$.

The Lempel-Ziv factorization can be represented by a sequence of pairs $(p_1, \ell_1), \ldots, (p_m, \ell_m)$, where in case (a) $p_j = c$ and $\ell_j = 0$, and in case (b) p_j is a position in $s_1 s_2 \cdots s_{j-1}$ at which an occurrence of s_j starts and $\ell_j = |s_j|$.

For example, the LZ-factorization of $S = acaaacatat$ is $s_1 = a$, $s_2 = c$, $s_3 = a$, $s_4 = aa$, $s_5 = ca$, $s_6 = t$, $s_7 = at$. This LZ-factorization can be represented by $(a, 0), (c, 0), (1, 1), (3, 2), (2, 2), (t, 0), (7, 2)$.

To appreciate the full value of this compression method, consider the string a^n that consists solely of a's. Its LZ-factorization is $(a, 0), (1, n-1)$.

In this section, we will present efficient algorithms that compress a string by computing its LZ-factorization. These are the result of recent research efforts [CI08, OG11, GB13, KKP13]. The corresponding decompression algorithm is very simple, it is given in Algorithm 1.

We start with an intuitive explanation of how the next LZ-factor starting at a position k in S can be computed. Consider the string $S = acaaacatat$ and $k = 4$; see Figure 2. In the suffix array SA of S, we start at the index $i = \mathsf{ISA}[k]$ and in an upwards scan we search for first entry i_{psv} for which $\mathsf{SA}[i_{psv}] < k$; let $psv = \mathsf{SA}[i_{psv}]$. Similarly, in a downwards scan of the suffix array, starting at the index $i = \mathsf{ISA}[k]$, we search for first entry i_{nsv} for which $\mathsf{SA}[i_{nsv}] < k$ and define $nsv = \mathsf{SA}[i_{nsv}]$. In the example of Figure 2, for $k = 4$ we have $i = 2$, $psv = 3$, and $nsv = 1$. Then, we compute $s_{psv} = \mathsf{lcp}(S_{psv}, S_k)$ and $s_{nsv} = \mathsf{lcp}(S_{nsv}, S_k)$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix of the strings u and v. If $|s_{psv}| > |s_{nsv}|$, then s_{psv} is the next LZ-factor and we output its representation $(psv, |s_{psv}|)$. Otherwise s_{nsv} is the next LZ-factor and we output its representation $(nsv, |s_{nsv}|)$. In our example, $\mathsf{lcp}(S_{psv}, S_k) = \mathsf{lcp}(S_3, S_4) = aa$ and $\mathsf{lcp}(S_{nsv}, S_k) = \mathsf{lcp}(S_1, S_4) = a$; so aa is the next LZ-factor starting at a position 4 in S and $(3, 2)$ is output.

Exercise 2.2 Prove the correctness of the method sketched above.

i	SA	ISA	$S_{\mathsf{SA}[i]}$	$\mathsf{PSV}_{lex}[i]$	$\mathsf{NSV}_{lex}[i]$
0	0		ε		
1	$psv = 3$	3	$aaacatat$	0	3
2	$k = 4$	7	$aacatat$	1	3
3	$nsv = 1$	1	$acaaacatat$	0	11
4	5	2	$acatat$	3	7
5	9	4	at	4	6
6	7	8	$atat$	4	7
7	2	6	$caaacatat$	3	11
8	6	10	$catat$	7	11
9	10	5	t	8	10
10	8	9	tat	8	11
11	0	0	ε		

Figure 2: The suffix array (and other arrays) of the string $S = acaaacatat$.

Algorithm 2 Procedure `LZ_Factor`(k, psv, nsv)

$\ell_{psv} \leftarrow |\mathsf{lcp}(S_{psv}, S_k)|$
$\ell_{nsv} \leftarrow |\mathsf{lcp}(S_{nsv}, S_k)|$
if $\ell_{psv} > \ell_{nsv}$ **then**
  $(p, \ell) \leftarrow (psv, \ell_{psv})$
else
  $(p, \ell) \leftarrow (nsv, \ell_{nsv})$
if $\ell = 0$ **then**
  $p \leftarrow S[k]$
output factor (p, ℓ)
return $k + \max\{\ell, 1\}$

Algorithm 2 outputs the LZ-factor starting at position k in S, based on the values psv and nsv. Additionally, it returns the position for which the next LZ-factor in the LZ-factorization of S must be computed.

Exercise 2.3 Slightly improve Algorithm 2 by computing $|\mathsf{lcp}(S_{psv}, S_{nsv})|$ first.

Algorithm 3 computes the LZ-factorization of a string S. However, its worst-case time complexity is $O(n^2)$ because of the repeated scans of the suffix array (provide a string for which the worst case occurs). To obtain a linear-time algorithm, we must be able to compute psv and nsv in constant time. This is possible with the arrays PSV_{lex} and NSV_{lex}; see Figure 2 for an example. To deal with boundary cases, we introduce the following artificial entries in the suffix array: $\mathsf{SA}[0] = 0$ and $\mathsf{SA}[n+1] = 0$. Furthermore, we define $S_0 = \varepsilon$, so that $S_{\mathsf{SA}[0]}$ and $S_{\mathsf{SA}[n+1]}$ are the empty string.

Algorithm 3 Computation of the LZ-factorization in $O(n^2)$ time

```
compute SA in O(n) time
for i ← 1 to n do          /* compute ISA in O(n) time */
  ISA[SA[i]] ← i           /* stream SA */
k ← 1
while k ≤ n do             /* stream ISA */
  i ← ISA[k]
  compute psv by an upwards scan of SA starting at index i
  compute nsv by a downwards scan of SA starting at index i
  k ← LZ_Factor(k, psv, nsv)
```

Definition 2.4 For any index i with $1 \leq i \leq n$, we define

$$\mathsf{PSV}_{lex}[i] = \max\{j : 0 \leq j < i \text{ and } \mathsf{SA}[j] < \mathsf{SA}[i]\}$$

and

$$\mathsf{NSV}_{lex}[i] = \min\{j : i < j \leq n+1 \text{ and } \mathsf{SA}[j] < \mathsf{SA}[i]\}$$

Algorithm 4 computes the arrays PSV_{lex} and NSV_{lex} in linear time. After iteration i of the for-loop, the following two properties hold true:

- for all k with $1 \leq k \leq \min\{i, n\}$, the entry $\mathsf{PSV}_{lex}[k]$ is correctly filled in
- for all k with $1 \leq k \leq i-1$ and $\mathsf{NSV}_{lex}[k] \leq i$, the entry $\mathsf{NSV}_{lex}[k]$ is correctly filled in

Algorithm 4 Given SA, this procedure computes PSV_{lex} and NSV_{lex}.

```
for i ← 1 to n+1 do        /* stream SA */
  j ← i − 1
  while SA[i] < SA[j] do
    NSV_lex[j] ← i
    j ← PSV_lex[j]
  PSV_lex[i] ← j
```

Exercise 2.5 Prove that the two properties above are indeed invariants of the for-loop of Algorithm 4.

Exercise 2.6 Devise an alternative linear-time algorithm that computes the arrays PSV_{lex} and NSV_{lex} using a stack.

Now we have all the ingredients for a linear-time LZ-factorization; see Algorithm 5.

Algorithm 5 Computation of the LZ-factorization in $O(n)$ time

```
compute SA, ISA, PSV_lex, and NSV_lex in O(n) time
k ← 1
while k ≤ n do          /* stream ISA */
  psv ← SA[PSV_lex[ISA[k]]]
  nsv ← SA[NSV_lex[ISA[k]]]
  k ← LZ_Factor(k, psv, nsv)
```

Exercise 2.7 Show that Algorithm 5 runs in $O(n)$ time.
Hint: Give an upper bound for the overall number of character comparisons employed in the calls to procedure `LZ_Factor`.

We can get rid of the inverse suffix array in Algorithm 5 because it is possible to directly compute the arrays PSV_{text} and NSV_{text}, defined as follows:

$$\begin{aligned} \mathsf{PSV}_{text}[k] &= \mathsf{SA}[\mathsf{PSV}_{lex}[\mathsf{ISA}[k]]] \\ \mathsf{NSV}_{text}[k] &= \mathsf{SA}[\mathsf{NSV}_{lex}[\mathsf{ISA}[k]]] \end{aligned}$$

Figure 3 shows an example.

k	1	2	3	4	5	6	7	8	9	10
$S[k]$	a	c	a	a	a	c	a	t	a	t
$\mathsf{PSV}_{text}[k]$	0	1	0	3	1	2	5	6	5	6
$\mathsf{NSV}_{text}[k]$	0	0	1	1	2	0	2	0	7	8

Figure 3: The arrays PSV_{text} and NSV_{text} of the string $S = acaaacatat$.

Pseudo-code for the computation of PSV_{text} and NSV_{text} can be found in Algorithm 6. In essence, the correctness of this algorithm follows from the correctness of Algorithm 4.

Algorithm 6 Given SA, this procedure computes PSV_{text} and NSV_{text}.

```
for i ← 1 to n + 1 do          /* stream SA */
  j ← SA[i − 1]
  k ← SA[i]
  while k < j do          /* store PSV_text and NSV_text interleaved */
    NSV_text[j] ← k
    j ← PSV_text[j]
  PSV_text[k] ← j
```

Algorithm 7 shows the pseudo-code of our final algorithm for computing the LZ-factorization of a string S.

Algorithm 7 Computation of the LZ-factorization in $O(n)$ time

compute SA, PSV_{text}, and NSV_{text}
$k \leftarrow 1$
while $k \leq n$ **do** /* stream PSV_{text} and NSV_{text} */
  $k \leftarrow \mathtt{LZ\_Factor}(k, \mathsf{PSV}_{text}[k], \mathsf{NSV}_{text}[k])$

References

[CI08] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

[GB13] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. 23rd Data Compression Conference*, pages 133–142. IEEE Computer Society, 2013.

[GBYS92] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[HSS03] W.K. Hon, K. Sadakane, and W.K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2003.

[KKP13] J. Kärkkäinen, D. Kempa, and S.J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2013.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[KSPP03] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, 2003.

[MM93] U. Manber and E.W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[OG11] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2011.

[PST07] S.J. Puglisi, W.F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.