Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets*

Keisuke Goto

Fujitsu Laboratories Ltd., Kawasaki, Japan, goto.keisuke@fujitsu.com

Abstract. Suffix arrays and LCP arrays are one of the most fundamental data structures widely used for various kinds of string processing. We consider two problems for a read-only string of length N over an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$, the string contains σ distinct characters, the construction of the suffix array, and a simultaneous construction of both the suffix array and LCP array. For the word RAM model, we propose algorithms to solve both of the problems in $O(N)$ time by using $O(1)$ extra words, which are optimal in time and space. Extra words means the required space except for the space of the input string and output suffix array and LCP array. Our contribution improves the previous most efficient algorithms, $O(N)$ time using $\sigma + O(1)$ extra words by [Nong, TOIS 2013] and $O(N \log N)$ time using $O(1)$ extra words by [Franceschini and Muthukrishnan, ICALP 2007], for constructing suffix arrays, and it improves the previous most efficient solution that runs in $O(N)$ time using $\sigma + O(1)$ extra words for constructing both suffix arrays and LCP arrays through a combination of [Nong, TOIS 2013] and [Manzini, SWAT 2004].

Keywords: suffix array, longest common prefix array, in-place algorithm

1 Introduction

Suffix arrays [21] are data structures that store all suffix positions of a given string sorted in lexicographical order according to their corresponding suffixes. They were proposed as a space efficient alternative to suffix trees, which are one of the most fundamental and powerful tools used for various kinds of string processing. LCP arrays [21] are auxiliary data structures that store the lengths of the longest common prefixes between adjacent suffixes stored in suffix arrays. Suffix arrays with LCP arrays are sometimes called *enhanced suffix arrays* [1], and they can simulate various operations of suffix trees. Suffix arrays or enhanced suffix arrays can be used for efficiently solving problems in various research areas, such as pattern matching [21,23], genome analysis [1, 19], text compression [3, 5, 11], and data mining [9, 12]. In these applications, one of the main computational bottlenecks is the time and space needed to construct suffix arrays and LCP arrays.

In this paper, we consider two problems that are for a given read-only string: constructing suffix arrays and constructing both suffix arrays and LCP arrays. For both problems, we propose optimal time and space algorithms. We assume that an input string of length N is read only, consists of an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$, and contains σ distinct characters [1]. We assume that the word RAM model with a word size of $w = \lceil \log N \rceil$ bits and that basic arithmetic and bit operations on constant number of words take constant time. We say that an algorithm runs *in-place*

* The full paper is available at https://arxiv.org/abs/1703.01009.

[1] As we will describe later, this is a slightly stronger assumption than commonly used in previous research.

Keisuke Goto: Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets, pp. 111–125.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

and runs in optimal space if the algorithm requires constant extra words, which is the space except for the input string, output suffix array, and LCP array.

Suffix array construction. The first linear time (optimal time) algorithms for constructing suffix arrays for a string over an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$ were proposed at the same time by several authors [14, 16, 18], and they require at least N extra words. Nong [24] proposed a linear time and space efficient algorithm that requires $\sigma + O(1)$ extra words, but it still requires about N extra words in the worst case since σ can be N. An in-place algorithm that runs in $O(N \log N)$ time was proposed by Franceschini and Muthukrishnan [10] [2]. It has been an open problem whether there exists a suffix array construction algorithm that runs in linear time and in-place.

We propose an in-place linear time algorithm for a string over an integer alphabet $[1, \ldots, \sigma]$ that consists of σ distinct characters. Our algorithm is based on the induced sorting framework [10, 18, 24, 25], which splits all suffixes into L- and S-suffixes, sorts either of which first, and then sorts the other. The induced sorting framework uses two arrays: (1) a bit array of N bits to store each type of suffix, and (2) an integer array of σ words, for each character t, to store a pointer to the next insertion position of a suffix starting with t in the suffix array, so this framework requires $\sigma + N/\log N + O(1)$ extra words in a naive way. Our algorithm runs in almost the same way as the previous ones [10,18,24,25], but it stores these two arrays in the space of the output suffix array. Therefore, our algorithm runs in linear time and in-place. As a minor contribution, we also propose a simple space saving technique for the induced sorting framework. The framework has to store the beginning and ending positions of sub-arrays in recursive steps, which requires $O(\log N)$ words in total in a naive way. Franceschini and Muthukrishnan [10] proposed a method for storing them in-place and obtained each value in $O(\log N)$ time. We propose a simpler one for storing them in-place and obtain each value in $O(1)$ time (see the full paper for details).

Our assumption is slightly stronger than those of previous research in that all characters of an alphabet must appear in the input string [3]. However, if an input string can be writable not read-only, our algorithm still runs in linear time and in-place also for the same problem setting to previous research which an input string is over an integer $[1, \ldots, \sigma]$ and some characters may not appear in the string. Because we can transform the string to a string over an integer alphabet $[1, \ldots, \sigma']$ that consists of $\sigma' \leq \sigma$ distinct characters by the counting sort [17] that uses the space of output suffix arrays before our algorithm runs.

Recent and independent works for suffix array construction. Recently and independently, some in-place suffix array construction algorithms were proposed.

Li et al. [20] proposed an in-place linear time algorithm for a read-only string over an integer alphabet $[1, \ldots, O(N)]$ whose assumption is more general, and the result is stronger than ours. Though Li et al.'s algorithm is also based on the induced sorting framework, the details are different from ours. Both their and our algorithm look simple according to the framework, but this simplicity comes from using complex data structures and algorithms as tools. Li et al.'s algorithm uses two complex tools, in-place stable merge sort algorithm [4] and succinct data structures supporting select queries in constant time [13] which is used for storing pointers in compressed space.

[2] They assume that an input string is over a general alphabet, i.e., only comparison of any two characters is allowed, which can be done in $O(1)$ time. Their time and space complexities are optimal for general alphabets but not for integer alphabets.

[3] The same problem setting also appeared in [2].

On the other hand, our algorithm uses only the former one and store pointers in a normal array. Using complex tools tend to increase the runtime in practice, e.g., according to [7], select query times on a succinct data structure are several time slower than normal array accesses. In this perspective, our algorithm is simpler than theirs, and our work may contribute to develop practically faster in-place linear time suffix array construction algorithms in future.

Prezza [26] studied a similar problem that, for a writable input string, sorts suffixes of a size-b subset instead of all suffixes and constructs a *sparse* suffix array and *sparse* LCP array. His algorithm is based on a longest common extension data structure that, for two given positions i and j, efficiently computes the length of the longest common prefix between two suffixes starting at i and j, and it runs in $O(N+b\log^2 N)$ expected time and in-place.

Suffix array and LCP array construction. Most previous research focused on a setting that computes LCP arrays from a given string and its suffix array. Kasai et al. [15] proposed the first linear time (optimal time) algorithm that computes the inverse suffix array and then uses it and computes the LCP array. Since it stores the inverse suffix array in extra space, it requires $N + O(1)$ extra words. Manzini [22] proposed a more space efficient linear time algorithm. The algorithm constructs the ψ array, which is similar to the inverse suffix array, in the output space of the LCP array and then rewrites it to the LCP array. The rewriting process runs in-place, but constructing the ψ array requires $\sigma+O(1)$ extra words, so the algorithm runs in linear time using $\sigma + O(1)$ words in total. Suffix arrays and LCP arrays can be computed in the same time and space by computing the suffix array with Nong's algorithm [24] and then by computing the LCP array with Manzini's algorithm [22]. The problem for constant alphabets with $\sigma \in O(1)$ has been studied in [6, 8], and the algorithms in [6, 8] are very competitive in practice for constant alphabets.

Our proposed linear time in-place algorithm constructs the suffix array and LCP array on the basis of a simple but non-trivial strategy. First, we construct the ψ array by using the space of the suffix array and LCP array and store it in the space of the LCP array. Then, we construct the suffix array in-place by using our linear time in-place algorithm, and we rewrite the ψ array to the LCP array as in Manzini's algorithm. Thus, we finally obtain both the suffix array and LCP array in linear time and in-place.

Organization. This paper is organized as follows. In Section 2, we introduce notations and definitions. In Section 3, we explain the induced sorting framework on which our algorithms are based. In Section 4 and Section 5, we propose optimal time and space algorithms for constructing suffix arrays and both suffix arrays and LCP arrays, respectively.

2 Preliminaries

Let Σ be an integer alphabet, the elements of which are $[1, \ldots, \sigma]$ for an integer $\sigma \geq 1$. An element of Σ^* is called a *string*. The length of a string $\mathbf{T}$ is denoted by $|\mathbf{T}|$. The empty string ϵ is a string of length 0. For a string $\mathbf{T} = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of $\mathbf{T}$, respectively. For a string $\mathbf{T}$ of length N, the i-th character of $\mathbf{T}$ is denoted by $\mathbf{T}[i]$ for $1 \leq i \leq N$, and the substring of $\mathbf{T}$ that begins at position i and that ends at position j is denoted by $\mathbf{T}[i \ldots j]$ for $1 \leq i \leq j \leq N$. For convenience, we assume that $\mathbf{T}[N] = \$$, where $\$$ is a special

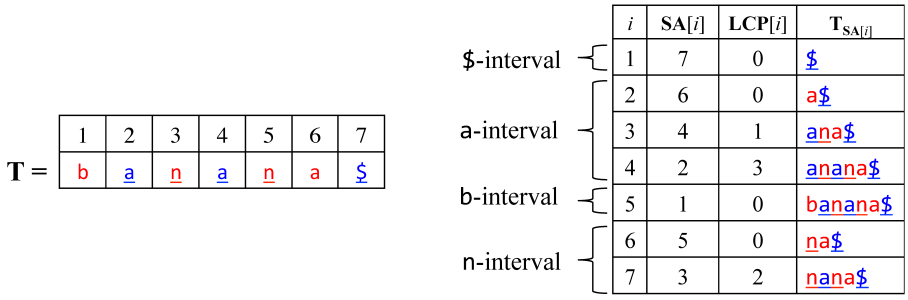

Figure 1. Suffix array and LCP array of $\mathbf{T}$ = banana\$. $\mathbf{T}[i]$ is colored red if $\mathbf{T}_i$ is an L-suffix, and blue otherwise. Moreover, $\mathbf{T}[i]$ is underlined if $\mathbf{T}_i$ is an LML- or LMS-suffix.

character lexicographically smaller than any characters in the string $\mathbf{T}[1 \dots N-1]$. In this paper, we also assume that $\sigma \leq N$ and that $\mathbf{T}$ contains σ distinct characters.

A suffix starting at a position i and its first character are denoted by $\mathbf{T}_i$ and t_i, respectively, and the position i is also called a *pointer* to $\mathbf{T}_i$. If no confusion occurs, we sometimes use $\mathbf{T}_i$ as a pointer i. For $1 \leq i \leq N$, $\mathbf{T}_i$ is called a *small type suffix* (S-suffix) if $i = N$ or $\mathbf{T}_i$ is lexicographically smaller than $\mathbf{T}_{i+1}$, and it is called a *large type suffix* (L-suffix) otherwise. An S-suffix/L-suffix $\mathbf{T}_i$ is also called a *leftmost-S-suffix*/*leftmost-L-suffix* (LMS-suffix/LML-suffix) if $i > 1$ and $\mathbf{T}_{i-1}$ is an L-suffix/S-suffix. For $i < N$, $\mathbf{T}'_i$ denotes the substring $\mathbf{T}[i \dots j]$, where $j > i$ is the leftmost position such that $\mathbf{T}_j$ is an LMS-suffix. Each type of $\mathbf{T}'_i$ is equal to that of $\mathbf{T}_i$, and $\mathbf{T}'_i$ is referred to as an L-, S-, LML-, or LMS-substring according to its type. The important property is that, for $1 \leq i < N$, $\mathbf{T}_i$ is an L-suffix if $t_i > t_{i+1}$, or $t_i = t_{i+1}$ and $\mathbf{T}_{i+1}$ is an L-suffix, and $\mathbf{T}_i$ is an S-suffix otherwise. From this property, each type of suffix can be obtained in $O(N)$ time with a right-to-left scan on $\mathbf{T}$ by comparing the first characters of adjacent suffixes. $suf(all)$ denotes the set of all suffixes of $\mathbf{T}$, and also $suf(L)$, $suf(S)$, $suf(LML)$, and $suf(LMS)$ denote the set of all L-, S-, LML-, and LMS-suffixes of $\mathbf{T}$, respectively. The size of a set M is denoted by N_M. Note that either $N_{suf(L)}$ or $N_{suf(S)}$ must be less than or equal to $N/2$ because all of the suffixes belong to either one. Moreover, $N_{suf(LMS)}$ must be less than or equal to the smallest of $N_{suf(L)}$ and $N_{suf(S)}$. For a subset M of $suf(all)$ and a suffix $\mathbf{T}_j$ of M, the rank of $\mathbf{T}_j$ is denoted by $rank_M(j)$, namely, $\mathbf{T}_j$ is the $rank_M(j)$-th smallest suffix of M. When the context is clear, we denote $rank_{suf(all)}$ as $rank$.

For a subset M of $suf(all)$, the suffix array $\mathbf{SA}_M$ of length N_M is an integer array that stores all pointers of M such that corresponding suffixes are lexicographically sorted. More precisely, for all suffixes $\mathbf{T}_i$ of M, $\mathbf{SA}_M[rank_M(i)] = i$. When the context is clear, we denote $\mathbf{SA}_{suf(all)}$ as $\mathbf{SA}$. For each character t, the maximum interval in which the first characters of suffixes are equal to t in $\mathbf{SA}$ is called the *t-interval*. Because L- and S-suffixes are respectively larger and smaller than their succeeding suffix, for any character t, L-suffixes that start with t are always located before S-suffixes starting with t in $\mathbf{SA}$.

The LCP array is an auxiliary array of $\mathbf{SA}$ such that $\mathbf{LCP}[i]$ contains the length of the longest common prefix of $\mathbf{T}_{\mathbf{SA}[i]}$ and $\mathbf{T}_{\mathbf{SA}[i-1]}$ for $1 < i \leq N$, and $\mathbf{LCP}[1] = 0$. See Figure 1.

3 Induced Sorting Framework

Our algorithm is based on the induced sorting framework [10,18,24,25], so, in this section, we explain the algorithm in [25] as an example of the framework. This algorithm runs in $O(N)$ time using $\sigma + N/\log N + O(1)$ extra words [4].

The key point of the framework is to sort a subset of suffixes once and then sort another subset of suffixes from the sorted subset. From this perspective, we say that the sorting of latter suffixes is induced from the former suffixes. Let $\mathbf{T}^0$ be $\mathbf{T}$ and let $\mathbf{T}^{i+1}$ be a string such that $|\mathbf{T}^{i+1}|$ is the number of LMS-substrings of $\mathbf{T}^i$ and $\mathbf{T}^{i+1}[j] = k$, where the j-th LMS-substring from the left of $\mathbf{T}^i$ is the k-th lexicographically smallest LMS-substring of $\mathbf{T}^i$. There are two important properties; the first is that $|\mathbf{T}^{i+1}| \leq \lfloor |\mathbf{T}^i|/2 \rfloor$ since the number of LMS-substrings in $\mathbf{T}^i$ is at most $\lfloor |\mathbf{T}^i|/2 \rfloor$, and the second is that the rank of the j-th LMS-suffix from the left of $\mathbf{T}^i$ within all LMS-suffixes in $\mathbf{T}^i$ corresponds to the rank of the j-th suffix from the left of $\mathbf{T}^{i+1}$ within all suffixes in $\mathbf{T}^{i+1}$. The algorithm recursively computes the suffix array $\mathbf{SA}^i$ of the string $\mathbf{T}^i$ at each recursive step i, namely, the algorithm sorts suffixes the number of which is smaller in more inner recursive steps. Note that $\mathbf{T}^i$ has the same property of $\mathbf{T}$ such that $\mathbf{T}^i$ consists of an integer alphabet of $[1, \ldots \sigma']$ for $1 \leq \sigma' \leq |\mathbf{T}^i|$ and $\mathbf{T}^i$ contains σ' distinct characters.

Below is an overview of the algorithm for computing the $\mathbf{SA}^i$ of $\mathbf{T}^i$ at a recursive step i. Note that all suffixes and substrings that appear in the overview indicate those of $\mathbf{T}^i$.

1. Sort all LMS-substrings.
2. Sort all LMS-suffixes from sorted LMS-substrings.
3. Sort all suffixes from sorted LMS-suffixes.
   (a) Perform preprocessing for Step 3b.
   (b) Sort all L-suffixes from sorted LMS-suffixes.
   (c) Sort all S-suffixes from sorted L-suffixes.

The essence of the algorithm is the part in which all suffixes are sorted from the sorted LMS-suffixes in Step 3. Step 1 runs in almost the same way as Step 3. Step 2 creates $\mathbf{T}^{i+1}$ and computes its suffix array recursively. Therefore, we herein explain only Step 3.

We consider only the case of computing the $\mathbf{SA}^0 = \mathbf{SA}$ of $\mathbf{T}^0 = \mathbf{T}$ at the recursive step 0 since we can also compute the $\mathbf{SA}^i$ of $\mathbf{T}^i$ similarly at each recursive step i. The algorithm requires three arrays, $\mathbf{A}$, $\mathbf{LE}/\mathbf{RE}$ [5], and $\mathbf{type}$. $\mathbf{A}$ is an integer array of length N to be $\mathbf{SA}$ at the end of the algorithm. At the beginning of the algorithm, we assume that each $\mathbf{A}[i]$ is initialized to *empty* in linear time, where empty is a special symbol that is used so that any element storing this symbol stores no meaningful value[6]. $\mathbf{type}$ is a binary array of length N, which indicates the type of $\mathbf{T}_j$ such that $\mathbf{type}[j] = L$ if $\mathbf{T}_j$ is an L-suffix, and $\mathbf{type}[j] = S$ otherwise. The $\mathbf{type}$ can

[4] The space for storing beginning and ending positions of sub-arrays in recursive steps is not accounted for.

[5] The notation was borrowed from $\mathbf{LF}/\mathbf{RF}$ used in [20], which is the abbreviation of leftmost/rightmost free. Although the definition is the same as the *bkt* array commonly used in previous research [6, 24, 25], the name $\mathbf{LF}/\mathbf{RF}$ is more specific. In our paper, we frequently use empty as the special symbol, so we prefer to use the notation $\mathbf{LE}/\mathbf{RE}$, which is the abbreviation for leftmost/rightmost empty.

[6] Practically, the special symbol is represented as an integer $N + 1$ indicating a position out of $\mathbf{A}$ so that we can distinguish the special symbol from pointers of $\mathbf{A}$.

be computed in $O(N)$ time with a right-to-left scan on $\mathbf{T}$ by comparing the first characters of the current suffix and its succeeding suffix. $\mathbf{LE}/\mathbf{RE}$ is an integer array of length σ such that $\mathbf{LE}[t]/\mathbf{RE}[t]$ indicates the next insertion position of a suffix starting with a character t in $\mathbf{A}$. $\mathbf{LE}[t]/\mathbf{RE}[t]$ is initially set to the head/tail of the t-interval of $\mathbf{SA}$, and it is managed in order to indicate the leftmost/rightmost empty position of the t-interval at any step of the algorithm. $\mathbf{LE}$ can be initialized in $O(N)$ time as follows. Let C_t be the number of suffixes starting with t. First, $\mathbf{LE}[t] = C_t$ is computed for all characters t by counting t_i with a single scan on $\mathbf{T}$ by using $\mathbf{LE}[t_i]$ as a counter, and last, $\mathbf{LE}[t] = 1 + \sum_{t'<t} C_{t'}$ is computed by accumulating $\mathbf{LE}[t] = C_t$ lexicographically. Similarly, $\mathbf{RE}$ can also be computed in $O(N)$ time.

We assume that $\mathbf{LE}$ is initialized at the beginning of Step 3b and that $\mathbf{RE}$ is also at the beginning of Step 3a and Step 3c. During the steps, types of suffixes are obtained by **type**.

Step 3a: As the result of Step 2, we have $\mathbf{SA}_{suf(LMS)} = \mathbf{A}[1 \dots N_{suf(LMS)}]$, and $\mathbf{A}[N_{suf(LMS)}+1 \dots N]$ is filled with empty. With a right-to-left scan on $\mathbf{SA}_{suf(LMS)}$, we move each $\mathbf{SA}_{suf(LMS)}[i] = \mathbf{T}_j$ into $\mathbf{A}[\mathbf{RE}[t_j]]$, which is the rightmost empty position of the t_j-interval, and decrease $\mathbf{RE}[t_j]$ by one to indicate the new rightmost empty position of the t_j-interval.

Step 3b: With a left-to-right scan on $\mathbf{A}$, we read all L- and LMS-suffixes $\mathbf{A}[i] = \mathbf{T}_j$ in lexicographic order, if $\mathbf{T}_{j-1}$ is an L-suffix, store $\mathbf{T}_{j-1}$ in $\mathbf{A}[\mathbf{LE}[t_{j-1}]]$, and increase $\mathbf{LE}[t_{j-1}]$ by one.

Step 3c: This step runs almost the same way as Step 3b. With a right-to-left scan on $\mathbf{A}$, we read all L- and S-suffixes $\mathbf{A}[i] = \mathbf{T}_j$ in reverse lexicographic order, if $\mathbf{T}_{j-1}$ is an S-suffix, store $\mathbf{T}_{j-1}$ in $\mathbf{A}[\mathbf{RE}[t_{j-1}]]$ and decrease $\mathbf{RE}[t_{j-1}]$ by one.

Steps 3a, 3b, and 3c run in $O(N)$ time because each step scans $\mathbf{A}$ only one time, and any of the operations take constant time per access. From Lemma 1, all induced L- and S-suffixes $\mathbf{T}_{j-1}$ are stored in $\mathbf{A}[rank(j-1)]$, so $\mathbf{A} = \mathbf{SA}$ is obtained at the end of Step 3. Roughly speaking, the correctness of Lemma 1 comes from the invariant that all suffixes stored in $\mathbf{A}$ are always sorted during the steps. When reading $\mathbf{A}[i] = \mathbf{T}_j$, the L-suffix $\mathbf{T}_{j-1}$ must be larger than any suffix $\mathbf{T}_{k-1}$ already stored in t_{j-1}-interval since $\mathbf{T}_k$ must appear at $\mathbf{A}[i']$ for $i' < i$, and it holds that $\mathbf{T}_k < \mathbf{T}_j$ from the invariant. Moreover, we do not miss any L-suffixes since we always store an induced L-suffix from a suffix stored in $\mathbf{A}[i]$ in a more rightward position $\mathbf{A}[i']$ for $i' > i$.

Lemma 1 ([25]). *When an L-suffix/S-suffix $\mathbf{T}_{j-1}$ is being induced in Step 3b/Step 3c, $\mathbf{LE}[t_{j-1}]/\mathbf{RE}[t_{j-1}]$ indicates $rank(j-1)$.*

Since $|\mathbf{T}^{i+1}| \leq \lfloor |\mathbf{T}^i|/2 \rfloor$ and the algorithm runs in $O(|\mathbf{T}^i|)$ time for all $\lceil \log N \rceil$ recursive steps i, the algorithm runs in $O(N)$ time in total. Moreover, the algorithm requires $\sigma + N/\log N + O(1)$ extra words, the first and second factors are for $\mathbf{LE}/\mathbf{RE}$ and **type**, respectively.

4 Optimal Time and Space Construction of Suffix Arrays

We propose a novel algorithm for constructing suffix arrays on the basis of the induced sorting framework. The space bottleneck of the previous algorithm [25] is the space of $\mathbf{LE}/\mathbf{RE}$ and **type**. Our algorithm embeds both arrays in the space of $\mathbf{A}$, and runs in $O(N)$ time and in-place, namely, in optimal time and space.

As seen in Section 3, the essence of the induced sorting framework is the part in which L-suffixes are sorted. Therefore, we focus on how to sort the L-suffixes from

sorted LMS-suffixes in $O(N)$ time and in-place. We can also sort S-suffixes in the same way (see Appendix A.2) and also LMS-substrings. Thus, we have the following theorem.

Theorem 2. *Given a read-only string* $\mathbf{T}$ *of length* N*, which consists of integers* $[1, \ldots, \sigma]$ *for* $1 \leq \sigma \leq N$ *and contains* σ *distinct characters, there is an algorithm for computing the* $\mathbf{SA}$ *of* $\mathbf{T}$ *in* $O(N)$ *time and in-place.*

Our main idea for reducing the space is to store sorted L- and LMS-suffixes in three internal sub-arrays in $\mathbf{A}$. We refer to these arrays as $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. The length of $\mathbf{Y}$ is σ, and for each character t, $\mathbf{Y}[t]$ stores either $\mathbf{LE}[t]$, the largest L-suffix starting with t, or the smallest LMS-suffix starting with t. $\mathbf{X}$ and $\mathbf{Z}$ store all L- and LMS-suffixes other than the ones stored in $\mathbf{Y}$, respectively.

We embed $\mathbf{LE}$ in $\mathbf{Y}$. Intuitively, this idea does not work because the total size of $\mathbf{X}$ and $\mathbf{LE}$ may exceed N, and if so, $\mathbf{X}$ overlaps with $\mathbf{LE}$ in $\mathbf{A}$, and elements of $\mathbf{LE}$ required in the future may be overwritten by induced L-suffixes. Moreover, we may not be able to even store $\mathbf{SA}_{suf(LMS)}$ and $\mathbf{LE}$ in $\mathbf{A}$ at the same time before sorting L-suffixes because their total size can also be greater than N. We avoid this problem by overwriting $\mathbf{LE}[t]$ only when it is no longer used in the future, namely, when all L-suffixes starting with t have been induced or there is no L-suffix starting with t. We detect such timing by causing a conflict between induced L-suffixes. Let CL_t be the number of L-suffixes starting with t. We try to store all L-suffixes in $\mathbf{X}$, whose space is limited that can store only $CL_t - 1$ L-suffixes starting with t for each character t. More precisely, for a character t, the beginning and ending position of t-interval overlaps with the ending position of the preceding t'-interval for $t' < t$ and the beginning position of the succeeding t''-interval for $t'' > t$, respectively. Therefore, conflict must occur between the largest (the last induced) L-suffix starting with a character t and the smallest (the first induced) L-suffix starting with $t'' > t$. We can detect the timing on the basis of conflicts, and we find that all L-suffixes starting with a smaller character t have been induced and that $\mathbf{LE}[t]$ is no longer needed in the future.

We do not need **type** anymore for detecting the type of suffixes being induced. We read L- and LMS-suffixes $\mathbf{T}_i$ stored either in $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ in lexicographic order. If $\mathbf{T}_i$ is read from $\mathbf{X}$ or $\mathbf{Z}$, we know the type of $\mathbf{T}_i$, so the type of $\mathbf{T}_{i-1}$ is easily obtained. Otherwise, we do not know the type of $\mathbf{T}_i$ in $\mathbf{Y}$, so the type of $\mathbf{T}_{i-1}$ is non-trivial. An important observation is that, for a suffix $\mathbf{T}_i$ in $\mathbf{Y}$, the preceding character t_{i-1} must be different from t_i since $\mathbf{T}_i$ is the largest L-suffix starting with t_i or the smallest LMS-suffix starting with t_i. Therefore, the type of $\mathbf{T}_{i-1}$ can be determined without **type** by comparing the characters t_{i-1} and t_i.

We use **type** of σ bits rather than N bits for distinguishing the L- and LMS-suffixes and the elements of $\mathbf{LE}$, which are stored in $\mathbf{Y}$. Although **type** require σ bits if it is stored naively, we can embed it in the space of $\mathbf{Y}$. Details on the in-place implementation of **type** will be given in Section 4.2.

The sub-arrays $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ and their more internal sub-arrays are located in $\mathbf{A}$ as shown in Figure 2. The figure also shows all of the steps of our algorithm. We partition the suffixes in certain subsets, which allows us to run each transition above in $O(N)$ time and in-place. Let $suf(LMSx)$ be the set of the LMS-suffixes that are the smallest among all LMS-suffixes starting with the same character, and let $suf(\overline{LMSx})$ be the set of all other LMS-suffixes. Let $suf(LMSy)$ be a subset of $suf(LMSx)$ that contains all $\mathbf{T}_i \in suf(LMSx)$ such that there is no L-suffix starting with t_i, and let $suf(\overline{LMSy})$ be the set of all other LMS-suffixes. We have $suf(LMSy) \subseteq$

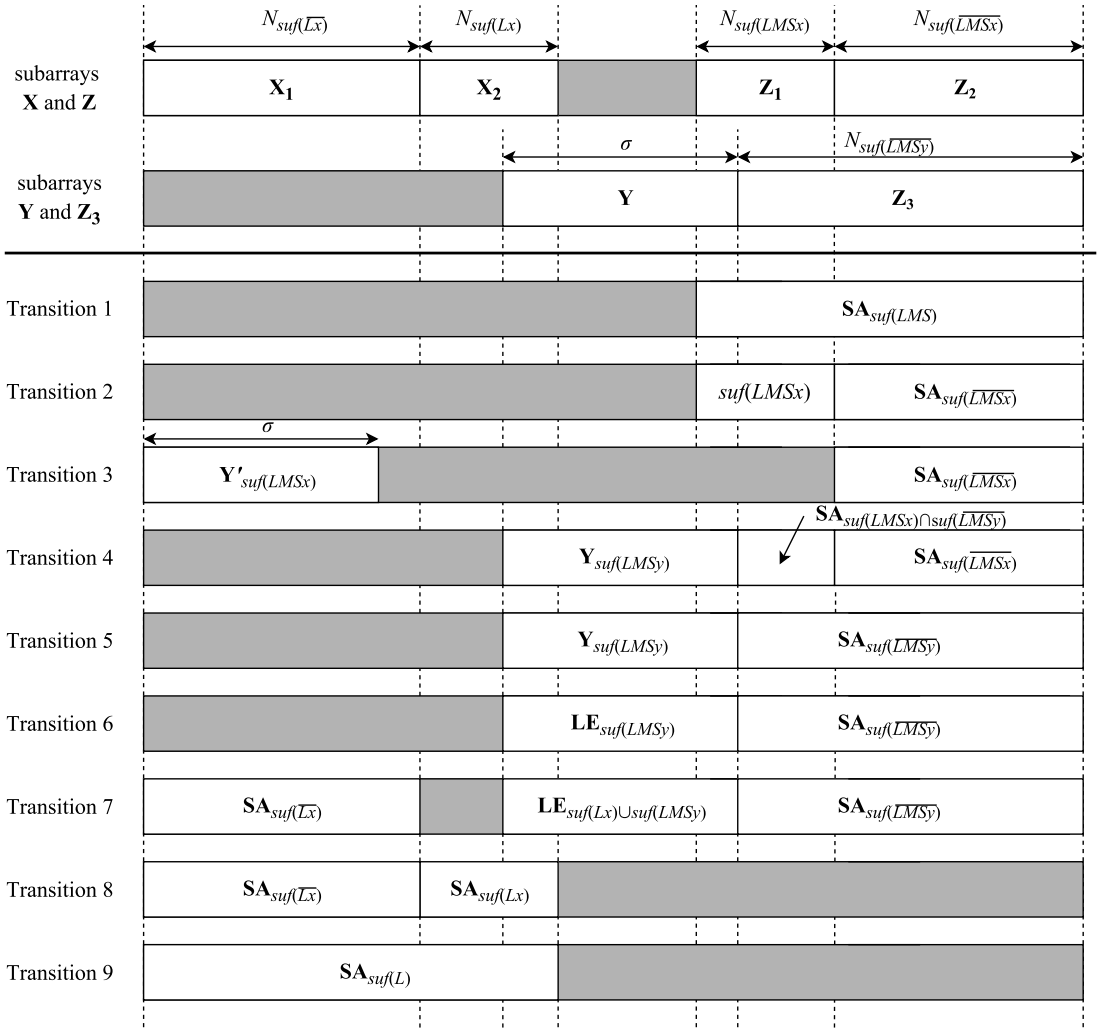

Figure 2. Inside transition of $\mathbf{A}$ while computing $\mathbf{SA}_{suf(L)}$ from $\mathbf{SA}_{suf(LMS)}$. Space colored with gray indicates empty space.

$suf(LMSx)$, $|suf(LMSx)| \leq \sigma$, and $suf(\overline{LMSx}) \subseteq suf(\overline{LMSy})$. Let $suf(Lx)$ be the set of all L-suffixes that are the largest among all L-suffixes starting with the same character, and let $suf(\overline{Lx})$ be the set of all other L-suffixes. Intuitively, $suf(Lx)$ and $suf(LMSx)$ consist of L- and LMS-suffixes that are closest to the border of L- and S-suffixes in each t-interval in $\mathbf{SA}_{suf(all)}$, respectively. Moreover, $suf(Lx) \cup suf(LMSy)$ is made by selecting one suffix for each interval from the set $suf(Lx) \cup suf(LMSx)$, where we give an L-suffix priority over an LMS-suffix if both exists. Thus, we have $|suf(Lx) \cup suf(LMSy)| \leq \sigma$.

We store various types of elements in $\mathbf{Y}$. To reduce ambiguity, $\mathbf{Y}_M$ denotes $\mathbf{Y}$ *overwritten* by a set of suffixes M whose first characters are distinct (we consider that the initial $\mathbf{Y}$ is filled with empty). More precisely, for a character t, $\mathbf{Y}_M[t] = \mathbf{T}_i$ if $\mathbf{T}_i$ starting with t exists in M, and $\mathbf{Y}_M[t] = \mathbf{Y}[t]$ otherwise. For example, $\mathbf{Y}_{suf(LMSy)}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{Y}_{suf(LMSy)}[t]$ is empty otherwise. Moreover, $\mathbf{LE}_{suf(LMSy)}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{LE}_{suf(LMSy)}[t] = \mathbf{LE}[t]$ otherwise.

4.1 Sort all L-suffixes

We compute $\mathbf{SA}_{suf(L)}$ in the head of $\mathbf{A}$ from $\mathbf{SA}_{suf(LMS)}$ stored in the head of $\mathbf{A}$, which is given by the result of sorting the LMS-suffixes. The internal transitions of $\mathbf{A}$ in the algorithm are shown in Figure 2, and each transition runs in $O(N)$ time and in-place. In Transitions 1-6, we compute $\mathbf{LE}$ and move LMS-suffixes in $\mathbf{Y}$ and $\mathbf{Z}$. Transition 7 induces all L-suffixes from LMS-suffixes stored in $\mathbf{Y}$ and $\mathbf{Z}$ and stores them in $\mathbf{X}$ and $\mathbf{Y}$. The concept of this transition is almost the same as Step 3b in Section 3. In Transitions 8-9, we merge the L-suffixes of $\mathbf{X}$ and $\mathbf{Y}$ and obtain $\mathbf{SA}_{suf(L)}$. The former part Transitions 1-5 is not so hard, so we omitted (we left the details in Appendix A.1), and we only describe the latter part Transitions 6-9 which is the most technical part of our algorithm. We assume that we have a bit array $\mathbf{type}$ of σ bits without extra space, and details on its in-place implementation are given in Section 4.2.

As the result of Transitions 1-5, we have $\mathbf{Y}_{suf(LMSy)}$ for which $\mathbf{Y}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{Y}[t]$ is empty otherwise, and we also have $\mathbf{type}$ for which $\mathbf{type}[t] = 1$ if an L-suffix starting with t exists, and $\mathbf{type}[t] = 0$ otherwise.

Transition 6: We transform $\mathbf{Y}_{suf(LMSy)}$ into $\mathbf{LE}_{suf(LMSy)}$. With a right-to-left scan on $\mathbf{T}$, we compute CL_t for each character t for which $\mathbf{type}[t] = 1$, namely for which $CL_t > 0$, and store it in $\mathbf{Y}[t]$ by using $\mathbf{Y}[t]$ as a counter. Note that we never overwrite a suffix of $suf(LMSy)$ stored in $\mathbf{Y}[t]$ since $\mathbf{type}[t] = 0$ and there is no L-suffix starting with t. With a left-to-right scan on $\mathbf{Y}$, we compute the prefix sum $\mathbf{Y}[t] = \mathbf{LE}[t] = 1 + \sum_{t'<t} \max(0, CL_{t'} - 1)$ for each t for which $\mathbf{type}[t] = 1$. Finally, we have $\mathbf{LE}_{suf(LMSy)}$ in $\mathbf{Y}$ that $\mathbf{Y}[t] = \mathbf{LE}[t]$ if $\mathbf{type}[t] = 1$, and $\mathbf{Y}[t]$ is $\mathbf{T}_i \in suf(LMSy)$ or empty otherwise.

Transition 7: We compute $\mathbf{SA}_{suf(\overline{Lx})}$ in $\mathbf{X}_1$ and $\mathbf{LE}_{suf(Lx) \cup suf(LMSy)}$ in $\mathbf{Y}$. This transition consists of the following three parts whose concept is almost same as Step 3b in Section 3. Part 1 reads all L- and LMS-suffixes $\mathbf{T}_i$ lexicographically from $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, Part 2 judges whether $\mathbf{T}_{i-1}$ is an L-suffix or not, and Part 3 stores $\mathbf{T}_{i-1}$ if it is an L-suffix. During the transition, we use $\mathbf{type}$ to determine whether $\mathbf{Y}[t]$ is a suffix (including both L- and LMS-suffixes) or an element of $\mathbf{LE}$. The invariant is that $\mathbf{type}[t] = 1$ if $\mathbf{Y}[t]$ is an element of $\mathbf{LE}$, and $\mathbf{type}[t] = 0$ otherwise, that is, $\mathbf{Y}[t]$ is empty or a suffix of $suf(Lx) \cup suf(LMSy)$. As the result of the previous transition, $\mathbf{type}$ has already satisfied the invariant.

We explain Part 1 last because it depends on Part 3.

Part 2: Judge whether $\mathbf{T}_{i-1}$ is an L-suffix or not. As already explained, the type of $\mathbf{T}_{i-1}$ can be obtained by comparing the first character t_{i-1} and its succeeding characters t_i.

Part 3: Store $\mathbf{T}_{i-1}$ if it is an L-suffix. We try to store an L-suffix $\mathbf{T}_{i-1}$ in $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$, which is the next insertion position of a suffix starting with t_{i-1}. If $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$ is empty, we simply store $\mathbf{T}_{i-1}$ there and increase $\mathbf{LE}[t_{i-1}]$ by one to update the next insertion position. Otherwise, $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$ has already stored a suffix $\mathbf{T}_j$. As already explained, a conflict must occur between the largest (the last induced) L-suffix starting with a character t and the smallest (the first induced) L-suffix starting with $t' > t$, and all L-suffixes starting with the smaller character t have already induced. Therefore, we compare the first characters t_{i-1} and t_j, store the smaller one in $\mathbf{LE}[\min(t_{i-1}, t_j)]$, store the larger one in $\mathbf{X}[\mathbf{LE}[t_{i-1}]]$, and update $\mathbf{type}[\min(t_{i-1}, t_j)] = 0$. Moreover, we increase $\mathbf{LE}[t_{i-1}]$ by one if $t_{i-1} > t_j$.

Part 1: Read all L- and LMS-suffixes $\mathbf{T}_i$ lexicographically. Recall that the arrays $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$ store sorted suffixes. With a left-to-right scan on $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, we scan $\mathbf{X}_1[i_X]$, $\mathbf{Y}[i_Y]$, and $\mathbf{Z}_3[i_Z]$ simultaneously in lexicographic order, where i_X, i_Y, and i_Z are the scanning positions of $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, respectively. We recall the types of suffixes stored in $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$:

- $\mathbf{X}_1[i_X]$ is either empty or an L-suffix.
- $\mathbf{Y}[i_Y]$ is either empty, a suffix of $suf(Lx) \cup suf(LMSy)$, or $\mathbf{LE}[i_Y]$.
- $\mathbf{Z}_3[i_Z]$ is a suffix of $\in suf(\overline{LMSy})$.

Let t_X, t_Y, and t_Z be the first characters of suffixes stored in $\mathbf{X}_1[i_X]$, $\mathbf{Y}[i_Y]$, and $\mathbf{Z}_3[i_Z]$, respectively, where t_Y equals i_Y. Here, we assume that t_X, t_Y, and t_Z are $\sigma + 1 \notin \Sigma$ if each index i_X, i_Y, or i_Z indicates a position out of the corresponding array, that is, such t_X, t_Y, and t_Z must not be chosen. We also assume that t_X is $\sigma + 1$ if $\mathbf{X}_1[i_X]$ is empty.

We choose the smallest character t_i of t_X, t_Y, and t_Z. In case we need to break a tie, we give t_X priority over t_Y, and t_Y priority over t_Z. Note that $\mathbf{T}_i$ is the smallest suffix of the three candidates because, for suffixes $\mathbf{T}_{j_1}$, $\mathbf{T}_{j_2}$, and $\mathbf{T}_{j_3}$ of $suf(\overline{Lx})$, $suf(Lx) \cup suf(LMSy)$, and $suf(\overline{LMSy})$, respectively, we have $\mathbf{T}_{j_1} < \mathbf{T}_{j_2} < \mathbf{T}_{j_3}$ if they all start with the same character, and $\mathbf{T}_i$ is chosen in this order of priority. Next, we increase the scanning position by one. Thus, we can read all L- and LMS-suffixes lexicographically.

One concern is that we may choose $t_Y = i_Y$ for which either $\mathbf{Y}[i_Y]$ is empty or $\mathbf{Y}[i_Y] = \mathbf{LE}[i_Y]$. The former case implies that none of the L- or LMS-suffixes start with t_Y, so we increase i_Y by one and choose the smallest character from the three candidates again. In the latter case, let $\mathbf{T}_i$ be the largest L-suffix starting with t, it implies that $\mathbf{type}[i_Y]$ must be 1, a conflict with $\mathbf{T}_i$ has not occured yet, and $\mathbf{T}_i$ has already been read and is still stored in $\mathbf{X}_1$. So, in this case also, we increase i_Y by one and choose the smallest character from the three candidates again. $\mathbf{T}_i$ stored in $\mathbf{X}_1$ will conflict with another L-suffix and be stored in $\mathbf{LE}[t_Y]$ in the future.

Transition 8: We compute $\mathbf{SA}_{suf(Lx)}$ in $\mathbf{X}_2$ and initialize the space except for $\mathbf{X}$ in $\mathbf{A}$ as empty. All L-suffixes of $suf(Lx)$ are stored in $\mathbf{Y}$, and we have $\mathbf{type}$ for which $\mathbf{type}[t] = 1$ if $\mathbf{Y}[t]$ stores an L-suffix of $suf(Lx)$, and $\mathbf{type}[t] = 0$ otherwise. With a left-to-right scan on $\mathbf{Y}$, we move all L-suffixes $\mathbf{T}_i$ for which $\mathbf{type}[t_i] = 1$ in back of $\mathbf{X}_1$ while preserving the order, and we obtain $\mathbf{SA}_{suf(Lx)}$ in $\mathbf{X}_2$. Finally, we fill $\mathbf{A}[N_{suf(L)} \dots N]$ with empty.

Transition 9: We compute $\mathbf{SA}_{suf(L)}$. By applying the in-place stable merge algorithm in Theorem 3 to $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{SA}_{suf(\overline{Lx})}$ considering the first characters as keys, we compute $\mathbf{SA}_{suf(L)}$ in $O(N)$ time and in-place.

Theorem 3 ([4]). *For two sorted integer arrays $\mathbf{A}_1 = \mathbf{A}[1 \dots N_1]$ and $\mathbf{A}_2 = \mathbf{A}[N_1 + 1 \dots N_1 + N_2]$ that are stored in an array $\mathbf{A}[1 \dots N_1 + N_2]$, there is an in-place linear time ($O(N_1 + N_2)$ time) algorithm that can stably merge $\mathbf{A}_1$ and $\mathbf{A}_2$ in $\mathbf{A}$.*

Remark: For ease of explanation, we use the complex stable merge algorithm in Transition 5 and 9 for sorting L-suffixes. We can optimize the algorithm so that the algorithm does not use the merge algorithm for sorting L-suffixes and use only two times for sorting S-suffixes. Since $\mathbf{SA}_{suf(\overline{LMSy})}$ is read only sequentially, we can simulate the sequential scan of $\mathbf{SA}_{suf(\overline{LMSy})}$ by scanning $\mathbf{SA}_{suf(LMSx) \cap suf(\overline{LMSy})}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$ sequentially. Moreover, Transition 9 is equal to Transition 2 in sorting S-suffixes (see Appendix A.2), so we can skip Transition 9 and avoid to use the stable merge algorithm.

4.2 In-place implementation of type

We store suffixes and elements of **LE** in **Y** in a compact representation so that whose most significant bits (MSBs) are vacant, and embed **type** in the MSBs of **Y**. Since each original value can be obtained from the simple compact representation in $O(1)$ time, it does not cause any problems for all transitions shown in Figure 2.

LE is a non-decreasing sequence, so we remember the leftmost m-interval that includes the position $2^{\lceil \log N \rceil - 1}$ in $\mathbf{X}_1$ whose MSB is 1, and also remember the MSB of $\mathbf{LE}[m]$ as *msb*. In Transition 7, *msb* is initially 0 but finally becomes 1. All elements of $\mathbf{LE}[t]$ are stored in **Y** in the compact representation by clearing the MSBs to 0. The original value of each $\mathbf{LE}[t]$ can be obtained in $O(1)$ time as follows;

- Set the MSB to 0 for $t < m$.
- Set the MSB to *msb* for $t = m$.
- Set the MSB to 1 for $t > m$.

A suffix $\mathbf{T}_i$ is stored as $\lfloor i/2 \rfloor$ so that the MSB is vacant. We use two important properties to obtain original values that, for a suffix $\mathbf{T}_i$ stored in $\mathbf{Y}[t]$, (1) the first character of $\mathbf{T}_i$ must be t, and (2) the preceding character t_{i-1} does not equal t (since $\mathbf{T}_i$ is the largest L-suffix starting with t or the smallest LMS-suffix starting with t). We can obtain an original suffix $\mathbf{T}_i$ from its compact representation $\mathbf{Y}[t] = j$. The candidate of i is $2j$ or $2j + 1$. If $t_{2j} \neq t_{2j+1}$, we choose one that equals t with Property 1. Otherwise, we choose $2j$ with Property 2.

Thus, we can store all elements of **LE** and suffixes in **Y** in a compact representation whose MSBs are vacant and store **type** in-place in the MSBs of **Y**.

5 Optimal Time and Space Construction of Suffix Arrays and LCP Arrays

We propose an algorithm for computing the suffix array and LCP array of a given read-only string **T** in $O(N)$ time and in-place. We revisit Manzini's algorithm [22], which constructs an LCP array **LCP** from a given string **T** and a suffix array **SA** in $O(N)$ time by using $\sigma + O(1)$ extra words. The algorithm uses a ψ array $\mathbf{\Psi}$ which is also called the *rank next array*, where $\mathbf{\Psi}[rank(i)] = rank(i + 1)$ for $1 \leq i < N$. The algorithm consists of two parts. The first part computes $\mathbf{\Psi}$ in $O(N)$ time by using $\sigma + O(1)$ extra words. The second part converts $\mathbf{\Psi}$ into **LCP** in $O(N)$ time and in-place. Therefore, **LCP** can be computed in $O(N)$ time and in-place if $\mathbf{\Psi}$ can be computed in $O(N)$ time and in-place.

Let **A** and **B** be integer arrays of length N to be **SA** and **LCP** at the end of the algorithm, respectively. Our algorithm computes $\mathbf{B} = \mathbf{\Psi}$ with both arrays **A** and **B** and $O(1)$ extra words. After that, it computes $\mathbf{A} = \mathbf{SA}$ in-place as described in Section 4 and converts $\mathbf{B} = \mathbf{\Psi}$ into $\mathbf{B} = \mathbf{LCP}$ in-place as in Manzini's way. For computing $\mathbf{\Psi}$, we use the inverse suffix array **ISA** such that $\mathbf{ISA}[\mathbf{SA}[i]] = i$, which is also called the *rank array* since $\mathbf{ISA}[i] = rank(i)$. The algorithm runs in the following steps.

1. Compute $\mathbf{B} = \mathbf{SA}$.
2. Compute $\mathbf{A} = \mathbf{ISA}$ from **SA**.
3. Compute $\mathbf{B} = \mathbf{\Psi}$, that is, drop **SA**. With a left-to-right scan on **ISA**, set $\mathbf{B}[\mathbf{ISA}[i]] = \mathbf{ISA}[i + 1]$ if $\mathbf{ISA}[i] < N$.

4. Compute $\mathbf{A} = \mathbf{SA}$ as described in Section 4.
5. Convert $\mathbf{B} = \mathbf{\Psi}$ into $\mathbf{B} = \mathbf{LCP}$ as in Manzini's way.

All of the steps run in $O(N)$ time and in-place. Thus, we have the following theorem.

Theorem 4. *Given a read-only string* $\mathbf{T}$ *of length* N*, which consists of integers* $[1, \ldots, \sigma]$ *for* $1 \leq \sigma \leq N$ *and contains* σ *distinct characters, there is an algorithm for computing both* $\mathbf{SA}$ *and* $\mathbf{LCP}$ *of* $\mathbf{T}$ *in* $O(N)$ *time and in-place.*

Acknowledgement

We wish to thank Takashi Kato, Shunsuke Inenaga, Hideo Bannai, Dominik Köppl, and anonymous reviewers for their many valuable suggestions on improving the quality of this paper, and especially, we also wish to thank an anonymous reviewer for giving us a simpler algorithm for computing LCP arrays as described in Section 5.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *The enhanced suffix array and its applications to genome analysis*, in Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings, 2002, pp. 449–463.
2. J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich: *Efficient fully-compressed sequence representations.* Algorithmica, 69(1) 2014, pp. 232–268.
3. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., 1994.
4. J. Chen: *Optimizing stable in-place merging.* Theor. Comput. Sci., 302(1-3) 2003, pp. 191–210.
5. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications.* Inf. Process. Lett., 106(2) 2008, pp. 75–80.
6. F. A. da Louza, S. Gog, and G. P. Telles: *Optimal suffix sorting and LCP array construction for constant alphabets.* Inf. Process. Lett., 118 2017, pp. 30–34.
7. O. Delpratt, N. Rahman, and R. Raman: *Engineering the LOUDS succinct tree representation*, in Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings, 2006, pp. 134–145.
8. J. Fischer: *Inducing the lcp-array*, in Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings, 2011, pp. 374–385.
9. J. Fischer, V. Heun, and S. Kramer: *Fast frequent string mining using suffix arrays*, in Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA, 2005, pp. 609–612.
10. G. Franceschini and S. Muthukrishnan: *In-place suffix sorting*, in Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings, 2007, pp. 533–545.
11. K. Goto and H. Bannai: *Space efficient linear time Lempel-Ziv factorization for small alphabets*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 163–172.
12. K. Goto, H. Bannai, S. Inenaga, and M. Takeda: *Fast q-gram mining on SLP compressed strings.* J. Discrete Algorithms, 18 2013, pp. 89–99.
13. G. Jacobson: *Space-efficient static trees and graphs*, in 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, 1989, pp. 549–554.
14. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* J. ACM, 53(6) 2006, pp. 918–936.

15. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings, 2001, pp. 181–192.
16. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
17. D. E. Knuth: *The art of computer programming, Volume III, 2nd Edition, Sorting and Searching*, Addison-Wesley, 1998.
18. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
19. F. Li and G. D. Stormo: *Selection of optimal DNA oligos for gene expression arrays.* Bioinformatics, 17(11) 2001, pp. 1067–1076.
20. Z. Li, J. Li, and H. Huo: *Optimal in-place suffix sorting*, in String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings, 2018, pp. 268–284.
21. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
22. G. Manzini: *Two space saving tricks for linear time LCP array computation*, in Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings, 2004, pp. 372–383.
23. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.
24. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, p. 15.
25. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction.* IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
26. N. Prezza: *In-place sparse suffix sorting*, in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018, 2018, pp. 1496–1508.

A Appendix

A.1 Sort all L-suffixes: Former Transitions

We describe Transitions 1-5, which are omitted in Section 4.1.

Transition 1: We shift $\mathbf{SA}_{suf(LMS)}$ stored in the head of $\mathbf{A}$ into $\mathbf{Z}$.

Transition 2: We store $suf(LMSx)$ in $\mathbf{Z}_1$ and compute $\mathbf{SA}_{suf(\overline{LMSx})}$ in $\mathbf{Z}_2$. Note that the suffixes in $\mathbf{Z}_2$ are sorted but may not be in $\mathbf{Z}_1$. Let j be the insertion position in $\mathbf{Z}_2$ for $\mathbf{SA}_{suf(\overline{LMSx})}$, which is initially set to $N_{suf(LMS)}$, namely, the end of $\mathbf{Z}$. With a right-to-left scan on $\mathbf{Z} = \mathbf{SA}_{suf(LMS)}$, we swap $\mathbf{SA}_{suf(LMS)}[i] = \mathbf{T}_k$ with $\mathbf{Z}[j]$ and decrease j by one if $\mathbf{T}_k \in suf(\overline{LMSx})$ and do nothing otherwise. Whether or not $\mathbf{T}_k$ belongs to $suf(\overline{LMSx})$ can be judged in $O(1)$ time by comparing the first characters because the first characters $t_{\mathbf{SA}_{suf(LMS)}[i]}$ and $t_{\mathbf{SA}_{suf(LMS)}[i-1]}$ are the same if and only if $\mathbf{T}_k \in suf(\overline{LMSx})$. Since we shift the suffixes of $suf(\overline{LMSx})$ to the end of $\mathbf{Z}$ while preserving the order of the shifted suffixes, we obtain $suf(LMSx)$ in $\mathbf{Z}_1$ (which may not be sorted) and $\mathbf{SA}_{suf(\overline{LMSx})}$ in $\mathbf{Z}_2$.

Unfortunately, we cannot compute $\mathbf{Y}_{suf(LMSy)}$ at this point directly because we currently do not know the size of $N_{suf(\overline{LMSy})}$ determining the starting position of $\mathbf{Y}$ within $\mathbf{A}$. We obtain this information in Transition 4. To start with, we consider a temporary array $\mathbf{Y}' = \mathbf{A}[1 \dots \sigma]$ and compute $\mathbf{Y}'_{suf(LMSx)}$.

Transition 3: We compute $\mathbf{Y}'_{suf(LMSx)}$. With a right-to-left scan on $\mathbf{Z}_1$, we try to move $\mathbf{Z}_1[i] = \mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$. However, $\mathbf{Y}'[t_{j_1}]$ may contain an LMS-suffix $\mathbf{T}_{j_2}$ because $\mathbf{Y}'$ may overlap with $\mathbf{Z}_1$. We simply move $\mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$ if $\mathbf{Y}'[t_{j_1}]$ is empty

and do nothing if $\mathbf{Y}'[t_{j_1}]$ is $\mathbf{T}_{j_1}$ because then $\mathbf{Z}_1[i]$ and $\mathbf{Y}'[t_{j_1}]$ are the same entry in $\mathbf{A}$. Otherwise, $\mathbf{Y}'[t_{j_1}]$ contains a suffix $\mathbf{T}_{j_2}$ such that $\mathbf{T}_{j_2} \neq \mathbf{T}_{j_1}$. In this case, we move $\mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$ and then try to move $\mathbf{T}_{j_2}$ into $\mathbf{Y}'[t_{j_2}]$. We repeat this procedure until we move $\mathbf{T}_{j_k}$ to $\mathbf{Y}'[t_{j_k}]$, which is empty, or encounter $\mathbf{Y}'[t_{j_k}] = \mathbf{T}_{j_k}$. Because $N_{suf(LMSx)} \leq \sigma$ and the first characters of $suf(LMSx)$ are all different, the number of insertions is $O(\sigma)$, and this transition can be done in $O(\sigma)$ time. Finally, we have $\mathbf{Y}'_{suf(LMSx)}$ such that $\mathbf{Y}'[t_i] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSx)$ or $\mathbf{Y}'[t_i]$ is empty otherwise.

Transition 4: We compute $\mathbf{Y}_{suf(LMSy)}$ and $\mathbf{SA}_{suf(LMSx)\cap suf(\overline{LMSy})}$. The set $suf(LMSx) \cap suf(\overline{LMSy})$ consists of each $suf(LMSx)$ suffix for which there is an L-suffix starting with the same character, and $suf(LMSy)$ is other $suf(LMSx)$. We compute $\mathbf{type}[t] = 1$ if there is an L-suffix starting with t, and $\mathbf{type}[t] = 0$ otherwise. We initialize $\mathbf{type}$ with 0. With a right-to-left scan on $\mathbf{T}$, we set $\mathbf{type}[t] = 1$ for an L-suffix starting with t. Now we know that a suffix stored in $\mathbf{Y}'_{suf(LMSx)}[t]$ with $\mathbf{type}[t] = 1$ belongs to $suf(\overline{LMSy}) \cap suf(LMSx)$. With a right-to-left scan on $\mathbf{Y}'_{suf(LMSx)}$, we move such suffixes in front of $\mathbf{Z}_2 = \mathbf{SA}_{suf(\overline{LMSx})}$ while preserving the order; then, we have $\mathbf{Z}_1 = \mathbf{SA}_{suf(LMSx)\cap suf(\overline{LMSy})}$. We just move $\mathbf{Y}'$ in front of $\mathbf{Z}_1$, and we have $\mathbf{Y}_{suf(LMSy)}$.

Transition 5: We compute $\mathbf{SA}_{suf(\overline{LMSy})}$. Because a suffix $\mathbf{T}_i$ of $suf(\overline{LMSy}) \cap suf(LMSx)$ is smaller than all suffixes of $suf(\overline{LMSx})$ starting with the same character t_i, $\mathbf{SA}_{suf(\overline{LMSy})}$ can be obtained by stably merging the last two arrays $\mathbf{SA}_{suf(\overline{LMSy})\cap suf(LMSx)}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$ with respect to the first characters as keys. The merged array contains *all* $suf(\overline{LMSy})$ suffixes since $(suf(\overline{LMSy}) \cap suf(LMSx)) \cup suf(\overline{LMSx}) = suf(\overline{LMSy})$. By applying Theorem 3 to $\mathbf{SA}_{suf(\overline{LMSy})\cap suf(LMSx)}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$, we compute $\mathbf{SA}_{suf(\overline{LMSy})}$ in $O(N)$ time and in-place.

A.2 Sort all S-suffixes

We can sort all S-suffixes in almost the same way as sorting L-suffixes but compute $\mathbf{SA}$ instead of $\mathbf{SA}_{suf(S)}$. The same can be said by switching the roles of $\mathbf{LE}$, L- and LMS-suffixes with $\mathbf{RE}$, S-, and L-suffixes, respectively. Let $suf(Sx)$ be the smallest suffixes starting with each character t, let $suf(\overline{Sx})$ be the set of the other S-suffixes, let $suf(Ly)$ be the set of the largest L-suffixes starting with each character t such that no S-suffix starts with t, and let $suf(\overline{Ly})$ be the set of the other L-suffixes. We compute $\mathbf{SA}_{suf(\overline{Ly})}$, $\mathbf{RE}_{suf(Ly)\cup suf(Sx)}$, and $\mathbf{SA}_{suf(\overline{Sx})}$ from $\mathbf{SA}_{suf(L)}$ in a similar way as Transitions 1-7 in Section 4.1. Note that $\mathbf{RE}_{suf(Ly)\cup suf(Sx)}$ equals $\mathbf{SA}_{suf(Ly)\cup suf(Sx)}$ from the definition. We compute $\mathbf{SA}$ in $O(N)$ time and in-place by considering the first characters as keys, by applying Theorem 3 to $\mathbf{SA}_{suf(\overline{Ly})}$ and $\mathbf{SA}_{suf(Ly)\cup suf(Sx)}$, and then by applying the result and $\mathbf{SA}_{suf(\overline{Sx})}$.

Thus, all S-suffixes can be sorted in $O(N)$ time and in-place as in Section 4.1. See Figure 3.

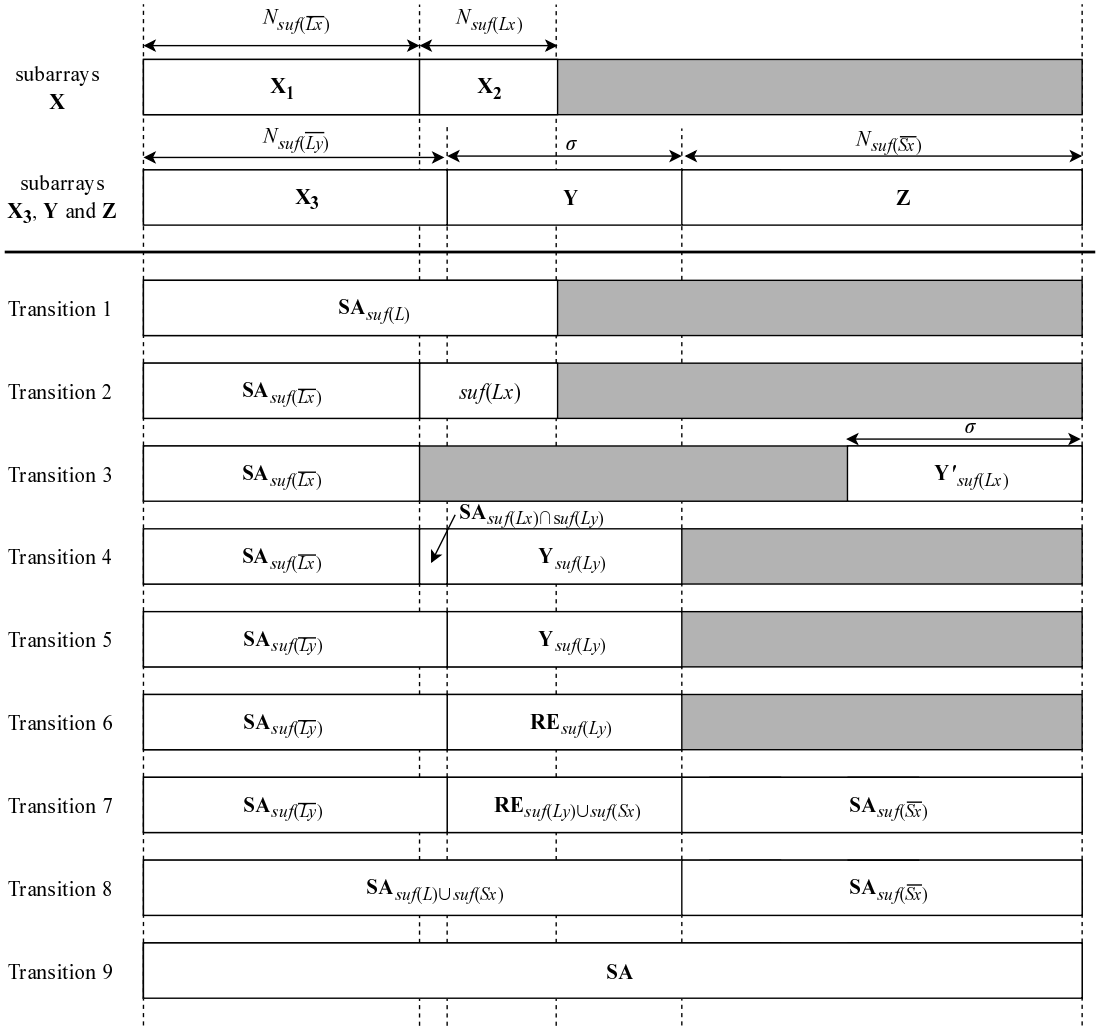

Figure 3. Inside transition of **A** while computing **SA** from $\mathbf{SA}_{suf(L)}$. Space colored with gray indicates empty space.