arXiv:1610.08305v5 [cs.DS] 28 May 2018

Optimal In-Place Suffix Sorting

Zhize Li
IIIS, Tsinghua University
zz-li14@mails.tsinghua.edu.cn

Jian Li
IIIS, Tsinghua University
lijian83@mail.tsinghua.edu.cn

Hongwei Huo
SCST, Xidian University
hwhuo@mail.xidian.edu.cn

Abstract

The suffix array is a fundamental data structure for many applications that involve string searching and data compression. Designing time/space-efficient suffix array construction algorithms has attracted significant attentions and considerable advances have been made for the past 20 years. We obtain the *first* in-place suffix array construction algorithms that are optimal both in time and space for (read-only) integer alphabets. Concretely, we make the following contributions:

1. For integer alphabets, we obtain the first suffix sorting algorithm which takes linear time and uses only $O(1)$ workspace (the workspace is the space needed beyond the input string and the output suffix array). The input string may be modified during the execution of the algorithm, but should be restored upon termination of the algorithm. Our C implementation of the algorithm requires only 8 Bytes of the workspace.
2. We strengthen the first result by providing the first linear time in-place algorithm for read-only integer alphabets with $|\Sigma| = O(n)$ (i.e., the input string cannot be modified). This algorithm settles the open problem posed by Franceschini and Muthukrishnan in ICALP 2007. The open problem asked to design in-place algorithms in $o(n \log n)$ time and ultimately, in $O(n)$ time for (read-only) integer alphabets with $|\Sigma| \leq n$. Our result is in fact slightly stronger since we allow $|\Sigma| = O(n)$.
3. Besides, for the read-only general alphabets (i.e., only comparisons are allowed), we present an optimal $O(n \log n)$ time in-place suffix sorting algorithm, recovering the result obtained by Franceschini and Muthukrishnan which was an open problem posed by Manzini and Ferragina in ESA 2002.

1 Introduction

In SODA 1990, suffix arrays were introduced by Manber and Myers [MM90] as a space-saving alternative to suffix trees [McC76, Far97]. Since then, it has been used as a fundamental data structure for many applications in string processing, data compression, text indexing, information retrieval and computational biology [FM00, AKO02, GV05, HCVN14, HSL+16]. Particularly, the suffix arrays are often used to compute the Burrows-Wheeler transform [BW94] and Lempel-Ziv factorization [ZL78]. Comparing with suffix trees, suffix arrays use much less space in practice. Abouelhoda et al. [AKO04] showed that any problem which can be computed using suffix trees can also be solved using suffix arrays with the same asymptotic time complexity, which makes suffix arrays very attractive both in theory and in practice. Hence, suffix arrays have been studied extensively over the last 20 years (see e.g., [KS03, KA03, KSB06, FM07, NZC09a, NZC11, Nong13]). We refer the readers to the surveys [PST07, DPT12] for many suffix sorting algorithms.

In 1990, Manber and Myers [MM90] obtained the first $O(n \log n)$ time suffix sorting algorithm over general alphabets. In 2003, Ko and Aluru [KA03], Kärkkäinen and Sanders [KS03] and Kim et al. [KSPP03]

independently obtained the first linear time algorithm for suffix sorting over integer alphabets. Clearly, these algorithms are optimal in terms of asymptotic time complexity. However, in many applications, the computational bottleneck is the *space* as we need the space-saving suffix arrays instead of suffix trees, and significant efforts have been made in developing *lightweight* (in terms of space usage) suffix sorting algorithms for the last decade (see e.g., [MF02, BK03, KA03, HSS03, MP06, FM07, NZ07, NZC09a, NZC11, Nong13]). In particular, the ultimate goal in this line of work is to obtain *in-place algorithms* (i.e., $O(1)$ workspace), which are also asymptotically optimal in time.

1.1 Problem Setting

Problem: Given a string $T = T[0 \dots n-1]$ with n characters, we need to construct the *suffix array* (SA) which contains the *indices* of all sorted suffixes of T (see Definition 1 for the formal definition of SA).

We consider the following three popular settings. Note that the constant alphabets (e.g., ASCII code) is a special case of integer alphabets, and the (read-only) integer alphabets is commonly used in practice. We measure the space usage of an algorithm in the unit of *words* same as [FM07, Nong13]. A word contains $\lceil \log n \rceil$ bits. One standard arithmetic or bitwise boolean operation on word-sized operands costs $O(1)$ time.

1. Integer alphabets: Each $T[i] \in [1, |\Sigma|]$ where the cardinality of the alphabets is $|\Sigma| \leq n$ and each $T[i]$ is stored in a word. The input string T may be modified by the algorithm, but should be restored upon termination of the algorithm.

2. Read-only integer alphabets: Each $T[i] \in [1, |\Sigma|]$ where $|\Sigma| = O(n)$. Moreover, the input string T is *read-only*. Each $T[i]$ can be read in $O(1)$ time. Note that we allow $|\Sigma| = O(n)$ rather than $|\Sigma| \leq n$ in Case 1.

3. Read-only general alphabets: The only operations allowed on the characters of T are comparisons. The input string T is read-only and we assume that each comparison takes $O(1)$ time. We *cannot* write the input space, make bit operations, even copy an input character $T[i]$ to the work space. Clearly, $\Omega(n \log n)$ time is a lower bound for suffix sorting in this case, as it generalizes comparison-based sorting.

The *workspace* used by an algorithm is the total space needed by the algorithm, excluding the space required by the input string T and the output suffix array SA. An algorithm which uses $O(1)$ words workspace to construct SA is called an *in-place* algorithm. See Tables 1 [1] and 2 for existing and new results.

1.2 Related Work and Our Contributions

1.2.1 Integer Alphabets

In this case, we allow the algorithm to modify the string T during the execution of the algorithm. We also describe how to restore T (if needed) at the end of the algorithm in Appendix C. Chan et al. [CMR14] denote this model as the *restore model* in their paper. We list several previous results and our new result in Table 1. Earlier algorithms that require more than $O(n)$ words workspace (see Table 1) do not need to modify the string T as they can afford to create a new array with n words to store the input.

Nong et al. [NZC09b, NZC11] obtained the first nearly linear time algorithm that used sublinear workspace. Recently, Nong [Nong13] obtained a linear time algorithm which used $|\Sigma|$ words workspace without modifying the string T. We improve their results as in the following theorem.

Theorem 1 *There is an in-place linear time algorithm for suffix sorting over integer alphabets with* $|\Sigma| \leq n$.

[1] Some previous algorithms state the space usages in terms of bits. We convert them into words.

Table 1: Time and workspace of suffix sorting algorithms for (read-only) integer alphabets Σ

Time	Workspace (words)	Algorithms
$O(n^2 \log n)$	$cn + O(1)$ $c < 1$	[MF02, MP06, MP08]
$O(n^2 \log n)$	$\|\Sigma\| + O(1)$	[IT99]
$O(n^2)$	$O(n)$	[SS07]
$O(n \log^2 n)$	$O(n)$	[Sad98]
$O(n \log n)$	$O(n)$	[MM90, LS07]
$O(vn)$	$O(n/\sqrt{v})$ $v \in [1, \sqrt{n}]$	[KSB06]
$O(n\sqrt{\|\Sigma\| \log(n/\|\Sigma\|)})$	$O(n)$	[BB05]
$O(n \log \log n)$	$O(n)$	[KJP04]
$O(n \log \log \|\Sigma\|)$	$O(n \log \|\Sigma\| / \log n)$	[HSS03]
$O(n \log \|\Sigma\|)$	$\|\Sigma\| + O(1)$	[NZ07]
$O(n)$	$O(n)$	[KSPP03, KS03, KA03, KSB06]
$O(n)$	$n + n/\log n + O(1)$	[NZC09a, NZC11]
$O(\frac{1}{\epsilon}n)$ *	$n^\epsilon + n/\log n + O(1)$ [2]	[NZC09b, NZC11]
$O(n)$	$\|\Sigma\| + O(1)$	[Nong13]
$O(n)$	$O(1)$	This paper

T is read-only in all algorithms except in the third to last row (marked with *).

Our algorithm is based on the induced sorting framework developed in [KA03] (which is also used in several previous algorithms [FM07, PST07, NZC09a, NZC09b, NZC11, Nong13]). We develop a few elementary, yet effective tricks to further reduce the space usage to constant. The proposed algorithm and the new tricks are also useful for the read-only integer and general alphabets.

Our algorithm is practical and easy to implement. Our C implementation of the algorithm requires only 8 Bytes workspace for any integer string and the running time is also competitive. We report our experimental results in Section 6.

1.2.2 Read-only Integer Alphabets

Now, we consider the more difficult case where the input string T is read-only. This is the main contribution of this paper. There are many existing algorithms for this case. See Table 1 for an overview. In ICALP 2007, Franceschini and Muthukrishnan [FM07] posed an open problem for designing an in-place algorithm that takes $o(n \log n)$ time or ultimately $O(n)$ time for (read-only) integer alphabets with $|\Sigma| \leq n$ (in fact, they did not specify whether the input string T is read-only or not). The current best result along this line is provided by Nong [Nong13], which used $|\Sigma|$ words workspace (Nong's algorithm is in-place if $|\Sigma| = O(1)$, i.e., constant alphabets). Note that in the worst case $|\Sigma|$ can be as large as $O(n)$.

In this paper, we settle down this open problem by providing the first optimal linear time in-place algorithm, as in the following theorem. Note that our result is in fact slightly stronger since we allow $|\Sigma| = O(n)$ instead of $|\Sigma| \leq n$ mentioned in the open problem [FM07].

Theorem 2 (Main Theorem) *There is an in-place linear time algorithm for suffix sorting over integer alphabets, even if the input string T is read-only and the size of the alphabets $|\Sigma| = O(n)$.*

[2] Nong et al. [NZC09b, NZC11] assumed that the word size is 32 bits and any integer can fit into one word. The result listed here is under the standard assumption that a word contains $\lceil \log n \rceil$ bits. It is not hard to verify that the bucket array B in their algorithm requires n^ϵ words. They also need an n bits array (or equivalently $n/\log n$ words).

Table 2: Time and workspace of suffix sorting algorithms for read-only general alphabets

Time	Workspace(words)	Algorithms
$O(n \log n)$	$O(n)$	[MM90, LS07]
$O(vn + n \log n)$	$O(v + n/\sqrt{v})$ $v \in [2, n]$	[BK03]
$O(vn + n \log n)$	$O(n/\sqrt{v})$ $v \in [1, \sqrt{n}]$	[KSB06]
$O(n \log n)$	$O(1)$	[FM07]
$O(n \log n)$	$O(1)$	This paper

1.2.3 Read-only General Alphabets

Now, we consider the case where the only operations allowed on the characters of string T (read-only) are comparisons. See Table 2 for an overview of the results. In 2002, Manzini and Ferragina [MF02] posed an open problem, which asked whether there exists an $O(n \log n)$ time algorithm using $o(n)$ workspace. In 2007, Franceschini and Muthukrishnan [FM07] obtained the first in-place algorithm that runs in optimal $O(n \log n)$ time. Their conference paper is somewhat complicated and densely-argued.

We also give an optimal in-place algorithm which achieves the same result, as in the following theorem. In addition, our algorithm does not make any bit operations while theirs uses bit operations heavily. Our algorithm is also arguably simpler.

Theorem 3 *There is an in-place $O(n \log n)$ time algorithm for suffix sorting over general alphabets, even if the input string T is read-only and only comparisons between characters are allowed.*

1.3 Difficulties and Our Approach

1.3.1 Difficulties

Typically, the suffix sorting algorithms are recursive algorithms. The size of the recursive (reduced) subproblem is usually less than half of the current problem. See e.g., [KA03, KSB06, PST07, FM07, NZC09a, NZC11, Nong13]. However, all previous algorithms require extra arrays, e.g., *bucket array* (which needs $|\Sigma|$ words at the top recursive level and $n/2$ words at the deep recursive levels), *type array* (which needs $n/\log n$ words) and/or other auxiliary arrays (which need up to $O(n)$ words), to construct the reduced problems and use the results of the reduced problems to sort the original suffixes. [3]

In particular, Nong et al. [NZC09a] made a breakthrough by providing the SA-IS algorithm which only required one bucket array (which needs $\max\{|\Sigma|, n/2\}$ words) and one type array ($n/\log n$ words). Note that the bucket array and type array are reused for each recursive level.

Currently, the best result is provided by Nong [Nong13]. However, Nong's algorithm still required the bucket array at the top recursive level, but not required at the deep levels. Hence, it needs $|\Sigma|$ words instead of $\max\{|\Sigma|, n/2\}$ words. Note that $|\Sigma|$ can be $O(n)$ in the worst case for integer alphabets. For the type array, Nong used this bucket array to implicitly indicate the type information.

Thus, *the main technical difficulty is to remove the workspace for the bucket array at the top recursive level since there is no extra space to use*. Note that it is non-trivial since T is read-only and SA needs to store the final order of all suffixes. Besides, the previous sorting steps or tricks may not work if one removes the bucket array. For example, Nong [Nong13] used the bucket array to indicate the type information. Then one may need the type array now since the bucket array has been removed.

[3] The definitions of bucket array and type array can be found in Section 2.

1.3.2 Our Approach

We briefly describe our optimal in-place linear time suffix sorting algorithms that overcome these difficulties. We provide an *interior counter trick* which can implicitly represent the dynamic *LF/RF-entry* information (see Section 2 for the definition) in SA. Besides, we provide a *pointer data structure* which can represent the bucket heads/tails in SA. Combining these two techniques, we can remove the workspace needed by the bucket array entirely. Note that it is non-trivial for the top recursive level which is the most difficult part, since the pointer data structure needs *nonconstant* workspace and we only have $O(1)$ extra workspace. In our algorithm, we divide the sorting step into two stages to address this issue. In order to remove the type array, we provide some useful properties and observations which allows us to retrieve the type information efficiently. For the general alphabets case, we provide simple sorting steps and extend the interior counter trick to obtain an optimal in-place $O(n \log n)$ time suffix sorting algorithm.

Organization: The remaining of the paper is organized as follows. Section 2 covers the preliminary knowledge. In Section 3, 4 and 5, we describe the framework and the details of our optimal in-place suffix sorting algorithms for the integer alphabets, read-only integer alphabets and read-only general alphabets, respectively. Next, we report the experimental results of our in-place algorithm for integer alphabets in Section 6. Finally, we conclude in Section 7.

2 Preliminaries

Given a string $T = T[0 \dots n-1]$ with n characters, the suffixes of T are $T[i \dots n-1]$ for all $i \in [0, n-1]$, where $T[i \dots j]$ denotes the substring $T[i]T[i+1] \dots T[j]$ in T. To simplify the argument, we assume that the final character $T[n-1]$ is a sentinel which is lexicographically smaller than any other characters in Σ. Without loss of generality, we assume that $T[n-1] = 0$. [4] Any two suffixes in T must be different since their lengths are different, and their lexicographical order can be determined by comparing their characters one by one until we see a difference due to the existence of the sentinel.

Definition 1 *The suffix array* SA *contains the indices of all suffixes of* T *which are sorted in lexicographical order, i.e.,* $\mathsf{suf}(\mathsf{SA}[i]) < \mathsf{suf}(\mathsf{SA}[j])$ *for all* $i < j$*, where* $\mathsf{suf}(i)$ *denotes the suffix* $T[i \dots n-1]$.

For example, if $T =$ "1220", then all suffixes are $\{1220, 220, 20, 0\}$ and $\mathsf{SA} = [3, 0, 2, 1]$. Note that SA always uses n words no matter what the alphabets Σ are, since it contains the permutation of $\{0, \dots, n-1\}$, where n is the length of T.

A suffix $\mathsf{suf}(i)$ is said to be *S-suffix* (S-type suffix) if $\mathsf{suf}(i) < \mathsf{suf}(i+1)$. Otherwise, it is *L-suffix* (L-type suffix) [KA03]. The last suffix $\mathsf{suf}(n-1)$ containing only the single character 0 (the sentinel) is defined to be an S-suffix. Equivalently, the $\mathsf{suf}(i)$ is S-suffix if and only if (1) $i = n-1$; or (2) $T[i] < T[i+1]$; or (3) $T[i] = T[i+1]$ and $\mathsf{suf}(i+1)$ is S-suffix. Obviously, the types can be computed by a linear scan of T (from $T[n-1]$ to $T[0]$). We further define the type of a character $T[i]$ to *S-type* (or *L-type* resp.) if $\mathsf{suf}(i)$ is S-suffix (or L-suffix resp.). A substring $T[i \dots j]$ is called an *S-substring* if both $T[i]$ and $T[j]$ are S-type, and there is no other S-type characters between them, or $i = j = n-1$ (the single sentinel). We can define *L-substring* similarly.

Example: We use the following running example for the integer alphabets case throughout the paper. Consider a

[4] Some previous papers use \$ to denote the sentinel. We use 0 here since we consider the integer alphabets.

string $T[0 \dots 12]$ = "2113311331210".

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S

E.g., $T[2]$ is S-type since $T[2] = 1 < T[3] = 3$. The S-substrings are $\{11, 1331, 11, 1331, 1210, 0\}$. □

Obviously, the indices of all suffixes, which begin with the same character, must appear consecutively in SA. We denote a subarray in SA for these suffixes with the same beginning character as a *bucket*, where the *head* and the *tail* of a bucket refer to the first and the last index of the bucket in SA respectively. Moreover, we define the first common character as its *bucket character*. We often use the bucket character to index the bucket. For example, if the bucket character is $T[i]$, we refer to this bucket as bucket $T[i]$. Sometimes we say that we place suffix $\mathsf{suf}(i)$ of T into SA, it always means that we place its corresponding index i into SA since $\mathsf{suf}(i)$ is a substring.

The *induced sorting* technique, developed by Ko and Aluru [KA03], is responsible for many recent advances of suffix sorting algorithms [PST07,FM07,NZC09a,NZC09b,NZC11,Nong13], and is also crucial to us. It can be used to induce the lexicographical order of L-suffixes from the sorted S-suffixes. Before introducing the induce sorting technique, we need the following useful property with respect to L-suffixes and S-suffixes (the proof simply follows from the definition of L- and S-suffix).

Property 1 *[KA03] In any bucket, S-suffixes always appear after the L-suffixes in* SA*, i.e., if an S-suffix and an L-suffix begin with the same character, the L-suffix is always smaller than the S-suffix.*

Now, we briefly introduce the standard induced sorting technique which needs the *bucket array* and *type array* explicitly. The bucket array contains $|\Sigma|$ integers and each denotes the position of a bucket head/tail in SA. The type array contains n bits and each entry denotes an L/S-type information for T (i.e., $0/1$ for L/S-type).

Inducing the order of L-suffixes from the sorted S-suffixes: Assume that all indices of the sorted S-suffixes are already in their correct positions in SA (i.e., in the tail of their corresponding buckets in SA). Now, we define some new notations (e.g., *LF/RF-entry*) to simplify the representation. We scan SA from left to right (i.e., from $\mathsf{SA}[0]$ to $\mathsf{SA}[n-1]$). We maintain an *LF-pointer* (leftmost free pointer) for each bucket which points to the leftmost free entry (called the *LF-entry*) of the bucket. The LF-pointers initially point to the head of their corresponding buckets. When we scan $\mathsf{SA}[i]$, let $j = \mathsf{SA}[i] - 1$. If $\mathsf{suf}(j)$ is an L-suffix (indicated by the type array), we place the index of $\mathsf{suf}(j)$ (i.e., j) into the LF-entry of bucket $T[j]$, and then let the LF-pointer of this bucket $T[j]$ point to the next free entry. The LF-pointers are maintained in the bucket array. If $\mathsf{suf}(j)$ is an S-suffix, we do nothing (since all S-suffixes are already sorted in the correct positions). We give a running example in Appendix A.1.

Sorting all S-suffixes from the sorted L-suffixes is completely symmetrical: we scan SA from right to left, maintaining an *RF-pointer* (rightmost free pointer) for each bucket which points to the *RF-entry* (rightmost free entry) of the bucket.

Lemma 1 *[KA03] Suppose all S-suffixes (or L-suffixes resp.) of T are already sorted. Then using induced sorting, all L-suffixes (or S-suffixes resp.) can be sorted correctly.*

The idea of induced sorting is that the lexicographical order between $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ is decided by the order of $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ if $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ are in the same bucket (i.e., $T[i] = T[j]$). We only need to specify the correct order of these L-suffixes in the same buckets since we always place the

L-suffixes in their corresponding buckets. Consider two L-suffixes $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ in the same bucket. We have $\mathsf{suf}(i+1) < \mathsf{suf}(i)$ and $\mathsf{suf}(j+1) < \mathsf{suf}(j)$ by the definition of L-suffix. Since we scan SA from left to right, $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ must appear earlier than $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$. Hence the correctness of induced sorting is not hard to prove by induction.

Inducing the order of L-suffixes from the sorted LMS-suffixes: A suffix $\mathsf{suf}(i)$ is called an *LMS-suffix* (leftmost S-type) if $T[i]$ is S-type and $T[i-1]$ is L-type, for $i \geq 1$. Nong et al. [NZC09a] observed that we can sort all L-suffixes from the sorted LMS-suffixes (instead of S-suffixes) if they are stored in the tail of their corresponding buckets in SA. Roughly speaking, the idea is that in the induced sorting, only LMS-suffixes are useful for sorting L-suffixes. One difference from the standard induced sorting is that we may scan some empty entries in SA. However, the empty entries can be ignored and all L-suffixes can still be sorted correctly. We provide a running example in Appendix A.2.

Lemma 2 *[NZC09a] Suppose all LMS-suffixes of T are already sorted and stored in the tail of their buckets. Then using induced sorting, all L-suffixes can be sorted correctly.*

After we sort all L-suffixes from the sorted LMS-suffixes, we can induce the order of all S-suffixes from the sorted L-suffixes by Lemma 1, and sort all suffixes. Now, we introduce how to sort the LMS-suffixes. First, we define some notations. A character $T[i]$ of T is called *LMS-character* if $\mathsf{suf}(i)$ is LMS-suffix. A substring $T[i \dots j]$ is called an *LMS-substring* if both $T[i]$ and $T[j]$ are LMS-characters, and there is no other LMS-characters between them, or $i = j = n-1$ (the single sentinel). Similarly, we can define *LML-suffix* (leftmost L-type) and *LML-substring*.

Sort the LMS-suffixes: If we know the lexicographical order of all LMS-substrings, then we can use their ranks to construct the reduced problem T_1. Sorting the suffixes of T_1 is equivalent to sorting the LMS-suffixes of T (see the following example). Nong et al. [NZC09a] showed that we can use the same induced sorting step to sort all LMS-substrings from the sorted LMS-characters of T. We briefly sketch their idea. We refer the readers to [NZC09a] for the details. We define the *LMS-prefix* of a suffix $\mathsf{suf}(i)$ to be $T[i \dots j]$, where $j > i$ is the smallest position in $\mathsf{suf}(i)$ such that $T[j]$ is an LMS character (e.g., the LMS-prefix of $\mathsf{suf}(4)$ is "31"). Suppose all LMS-characters are stored in the tail of their corresponding buckets in SA. First, we sort all LMS-prefix of L-suffixes from the sorted LMS-characters, using one scan of induced sorting from left to right (the same as induce the order of L-suffixes from LMS-suffixes). Then we sort all LMS-prefix of S-suffixes from the sorted LMS-prefix of L-suffixes (in the same way as inducing the order of S-suffixes from L-suffixes). After this, we have sorted all LMS-substrings since all LMS-substrings are LMS-prefix of S-suffixes by the definition of LMS-prefix. The correctness proof follows the same argument as in the standard setting.

Example: Continue the running example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
LMS		$*$				$*$				$*$			$*$

Note that the LMS-substrings are $\{11331, 11331, 1210, 0\}$. Their ranks in lexicographical order are $\{1, 1, 2, 0\}$. Thus, the reduced problem is $T_1 = 1120$. The order of the suffixes of T_1 is the same as the order of corresponding LMS-suffixes of T. The suffix array of the reduced problem T_1 is solved recursively. □

Note that in this preliminary section, the induced sorting steps are not *in-place* since they require explicit storage for the bucket and type arrays.

3 Suffix Sorting for Integer Alphabets

3.1 Framework

To avoid confusion, we recall that an LMS-character is a single character, an LMS-substring is a substring which begins with an LMS-character and ends with an LMS-character, and an LMS-suffix is a suffix of T which begins with an LMS-character. Our optimal in-place suffix sorting algorithm for integer alphabets consists of the following steps.

1. (Section 3.2) Rename T.
2. (Section 3.3) Sort all LMS-characters of T.
3. (Section 3.4) Induced sort all LMS-substrings from the sorted LMS-characters.
4. (Section 3.5) Construct the reduced problem T_1 (in which we need to sort all LMS-suffixes) from the sorted LMS-substrings.
5. (Section 3.6) Sort the LMS-suffixes by solving T_1 recursively.
6. (Section 3.7) Induced sort all suffixes of T from the sorted LMS-suffixes.

In a high level, the framework is similar to several other previous algorithms based on induced sorting [KA03, FM07, PST07, NZC09a, NZC09b, NZC11, Nong13], and in particular to [NZC09a]. Our algorithm differs in the detailed implementation of the above steps to obtain the first in-place algorithm (i.e., remove the bucket array and type array). We describe the details of the above steps in the following sections. We also describe how to restore T (if needed) at the end of the algorithm in Appendix C.

3.2 Rename T

In this section, we rename each L-type character of T to be the index of its bucket head and each S-type character of T to be the index of its bucket tail (Nong et al. [NZC11] has a similar renaming step). The correctness of the step is shown in the following Lemma 3.

Lemma 3 *The renaming step does not change the lexicographical order of all suffixes of T.*

Proof: For any two suffixes, beginning with the same character, the L-suffix is smaller than the S-suffix (Property 1). Hence, the renaming step does not change the relative orders of all suffixes. □

Example: We illustrate the renaming process in our running example.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	(12)	(11	1	5	9	2	6)	(10	0)	(4	8	3	7)
Bucket	(0)	(1	1	1	1	1	1)	(2	2)	(3	3	3	3)

After renaming, we get T' as following:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T'	7	6	6	9	9	6	6	9	9	6	7	1	0

E.g., $T'[0] = 7$ since $T[0]$ is L-type and the head of bucket 2 (i.e., bucket $T[0]$) is 7, and $T'[1] = 6$ since $T[1]$ is S-type and the tail of bucket 1 (i.e., bucket $T[1]$) is 6. Note that the heads of bucket $0, 1, 2, 3$ are $0, 1, 7, 9$, respectively. □

Now, we describe how to implement this step using linear time and $O(1)$ workspace. This step is very simple and similar to the counting sort (see e.g., [CLRS01, Ch. 8]). We first rename all L-type characters to be the index of its bucket head, and then rename all S-type characters to be the index of its bucket tail.

1. First we scan T once to compute the number of times each character occurs in T and store them in SA. Then we perform a *prefix sum computation* to determine the starting position of each character (i.e., bucket head) in SA. Finally we scan T once again to rename each character as the index of its bucket head.

2. Now we need to let the S-type characters of T to be the index of its bucket tail. Same as before, we scan T to compute the number of times each character occurs in T and store them in SA. Then, we scan T once again from right to left. For each S-type $T[i]$, we let it be the index of its bucket tail (i.e., just add the number of characters belonging to this bucket to $T[i]$). Note that if we scan T from right to left, for each $T[i]$, we can know its type is L-type or S-type in $O(1)$ time. There are two cases: 1) if $T[i] \neq T[i+1]$, we can know its type immediately by definition; 2) if $T[i] = T[i+1]$ then its type is the same as the type of $T[i+1]$. We only need to maintain *one* boolean variable which represent the type of previous scanned character $T[i+1]$.

3.3 Sort all LMS-characters

Now, we sort all LMS-characters of T, i.e., place the indices of the LMS-characters in the tail of their corresponding buckets in SA. Note that we do not have extra space to store the LF/RF-pointers/counters for each bucket to indicate the position of the free entries in the process. For this purpose, we develop a simple trick, called *interior counter trick*, which allows us to carefully use the space in SA to store the information of both the indices and the pointers. The implementation details are described below. In the steps, we use three special symbols which are Unique, Empty and Multi. [5]

Step 1. Initializing SA: First we clear SA (i.e., $\mathsf{SA}[i] = \mathsf{Empty}$, for all $i \in [0, n-1]$). Then we scan T from right to left. For every $T[i]$ which is an LMS-character (this can be easily decided in constant time), do the following:

(1) If $\mathsf{SA}[T[i]] = \mathsf{Empty}$, let $\mathsf{SA}[T[i]] = \mathsf{Unique}$ (meaning it is the unique LMS-character in this bucket). Note that after the renaming step, $T[i]$ is the index of its bucket tail.

(2) If $\mathsf{SA}[T[i]] = \mathsf{Unique}$, let $\mathsf{SA}[T[i]] = \mathsf{Multi}$ (meaning the number of LMS-characters in this bucket is at least 2).

(3) Otherwise, do nothing.

Step 2. Placing all indices of LMS-characters into SA: We scan T from right to left. For every $T[i]$ which is an LMS-character, we distinguish the following cases:

(1) $\mathsf{SA}[T[i]] = \mathsf{Unique}$: In this case, we let $\mathsf{SA}[T[i]] = i$ (i.e., $T[i]$ is the unique LMS-character in its bucket, and we just put its index into its bucket).

[5] We use at most five special symbols in this paper. The special symbol is only used to simplify the argument and we do not have to impose any additional assumption to accommodate these symbols (including the read-only cases in Section 4 and 5). These special symbols can be handled using an extra $O(1)$ workspace. We defer the details to Appendix B.

(2) $\mathsf{SA}[T[i]] = \mathsf{Multi}$ and $\mathsf{SA}[T[i]-1] = \mathsf{Empty}$: In this case, $T[i]$ is the first (i.e. largest index, since we scan T from right to left) LMS-character in its bucket. So if $\mathsf{SA}[T[i]-2] = \mathsf{Empty}$, we let $\mathsf{SA}[T[i]-2] = i$ and $\mathsf{SA}[T[i]-1] = 1$ (i.e., we use $\mathsf{SA}[T[i]-1]$ as the counter for the number of LMS-characters which has been added to this bucket so far). Otherwise, $\mathsf{SA}[T[i]-2] \neq \mathsf{Empty}$ (i.e., $\mathsf{SA}[T[i]-2]$ is in a different bucket, which implies that this bucket has only two LMS-characters). Then we let $\mathsf{SA}[T[i]] = i$ and $\mathsf{SA}[T[i]-1]$ be Empty (We do not need a counter in this case and the last LMS-character belonging to this bucket will be dealt with in the later process).

(3) $\mathsf{SA}[T[i]] = \mathsf{Multi}$ and $\mathsf{SA}[T[i]-1] \neq \mathsf{Empty}$: In this case, $\mathsf{SA}[T[i]-1]$ is maintained as the counter. Let $c = \mathsf{SA}[T[i]-1]$. We check whether the position ($\mathsf{SA}[T[i]-c-2]$), i.e. $c+2$ positions before its tail, is Empty or not. If $\mathsf{SA}[T[i]-c-2] = \mathsf{Empty}$, let $\mathsf{SA}[T[i]-c-2] = i$ and increase $\mathsf{SA}[T[i]-1]$ by one (i.e., update the counter number). Otherwise $\mathsf{SA}[T[i]-c-2] \neq \mathsf{Empty}$ (i.e., reaching another bucket), we need to shift these c indices to the right by two positions (i.e., move $\mathsf{SA}[T[i]-c-1 \ldots T[i]-2]$ to $\mathsf{SA}[T[i]-c+1 \ldots T[i]]$), and let $\mathsf{SA}[T[i]-c] = i$ and $\mathsf{SA}[T[i]-c-1] = \mathsf{Empty}$. After this, only one LMS-character needs to be added into this bucket in the later process.

(4) $\mathsf{SA}[T[i]]$ is an index: From case (2) and (3), we know the current $T[i]$ must be the last LMS-character in its bucket. So we scan SA from right to left, starting with $\mathsf{SA}[T[i]]$, to find the first position j such that $\mathsf{SA}[j] = \mathsf{Empty}$. Then we let $\mathsf{SA}[j] = i$. Now, we have filled the entire bucket. However, we note that not every bucket is fully filled as we have only processed LMS-characters so far.

After the above Step 1 and 2, there may be still some special symbols Multi and the counters (because the bucket is not fully filled, so we have not shifted these indices to the right in the bucket). We need to free these position. We scan SA once more from right to left. If $\mathsf{SA}[i] = \mathsf{Multi}$, we shift the indices of LMS-characters in this bucket to the right by two positions (i.e., $\mathsf{SA}[i-c-1 \ldots i-2]$ to $\mathsf{SA}[i-c+1 \ldots i]$) and let $\mathsf{SA}[i-c-1] = \mathsf{SA}[i-c] = \mathsf{Empty}$, where $c = \mathsf{SA}[i-1]$ denotes the counter.

Lemma 4 *The indices of the LMS-characters can be placed in the tail of their corresponding buckets in* SA *using linear time and* $O(1)$ *workspace.*

Proof: We only need to show that the Step 2, i.e., placing all indices of LMS-characters into SA, takes $O(n)$ time. For each scanned $T[i]$, it takes $O(1)$ time except when the $T[i]$ is the last two LMS-characters of its bucket. In this case, we need to shift the indices in this bucket when dealing with the penultimate LMS-character in its bucket, and scan the bucket when dealing with the last one. It takes $O(n)$ time since every bucket only needs to be shifted and scanned once. The space usage of this step is obvious. □

Example: Continue our example (U, E and M denote Unique, Empty and Multi, respectively):
Step 1. Initializing SA:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	7	6	6	9	9	6	6	9	9	6	7	1	0
LMS		*				*				*			*
SA	E	E	E	E	E	E	E	E	E	E	E	E	E

After initialization:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(**U**)	(E)	(E	E	E	E	**M**)	(E	E)	(E	E	E	E)

Step 2. Placing all indices of LMS-characters into SA:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(**12**)	(E)	(E	E	E	E	M)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(E	E	**9**	**1**	**M**)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(E	**5**	9	**2**	**M**)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(**1**	5	9	**3**	**M**)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(E	E	**1**	**5**	**9**)	(E	E)	(E	E	E	E)

In the last row, we remove all Multi symbols and counters. □

3.4 Induced sort all LMS-substrings from the sorted LMS-characters

In this section, we sort all LMS-substrings from the sorted LMS-characters using induced sorting. Since all LMS-substrings are LMS-prefix of S-suffixes (Recall that the LMS-prefix of $\mathsf{suf}(i)$ is $T[i \dots j]$, where $j > i$ is the smallest position in $\mathsf{suf}(i)$ such that $T[j]$ is an LMS character) and sorting the LMS-prefix of all suffixes from the sorted LMS-characters is the same as sorting all suffixes from the sorted LMS-suffixes (see the preliminary Section 2).

Now, we describe the details. We divide this step into two parts.

(1) First, we sort the LMS-prefix of all suffixes from the sorted LMS-characters. Since this part is the same as sorting all suffixes from the sorted LMS-suffixes, we will describe the details in Section 3.7.

(2) Then, we place the indices of all sorted LMS-substrings in $\mathsf{SA}[n - n_1 \dots n - 1]$, where n_1 denotes the number of LMS-characters. Note that the number of LMS-characters, LMS-suffixes, and LMS-substrings are the same. Moreover, $n_1 \leq \frac{n}{2}$ since any two LMS-characters are not adjacent.

The first part is described and proved in Section 3.7. Here, we only need to explain the second part how to place the indices of all sorted LMS-substrings in $\mathsf{SA}[n - n_1 \dots n - 1]$. First, we need the following Observation 1 and Lemma 5. Then we give a lemma to show that this step can be done in linear time using $O(1)$ workspace.

Observation 1 *For any bucket in* SA*, let* t *be its bucket tail. Then* $T[\mathsf{SA}[t]]$ *is S-type if and only if* $T[\mathsf{SA}[t]] < T[\mathsf{SA}[t]+1]$*. Similarly,* $T[\mathsf{SA}[h]]$ *is L-type if and only if* $T[\mathsf{SA}[h]] > T[\mathsf{SA}[h]+1]$*, where* h *is the bucket head.*

Lemma 5 *If a bucket contains S-type characters, then one can scan this bucket once to compute the number of S-type characters in this bucket using* $O(1)$ *workspace.*

Proof: We scan this bucket from its tail to its head. For the current scanning entry $\mathsf{SA}[i]$, there are two possibilities: 1). If $T[\mathsf{SA}[i]] \geq T[\mathsf{SA}[i]+1]$, do nothing; 2). Otherwise, let j be the smallest index such that $T[k] = T[\mathsf{SA}[i]]$ for any $k \in [j, \mathsf{SA}[i]]$. Then we increase num by $\mathsf{SA}[i] - j + 1$. Here variable num counts the number of S-type characters in this bucket and initially is 0. □

Lemma 6 *The indices of all sorted LMS-substrings can be placed in* $\mathsf{SA}[n - n_1 \dots n - 1]$ *using linear time and* $O(1)$ *workspace.*

Proof: In Step (2), we scan SA from right to left to place the indices of all LMS-substrings at the end of SA. We only need to explain how to distinguish whether $T[\mathsf{SA}[i]]$ is LMS-character or not when we scanning $\mathsf{SA}[i]$. Note that if we can tell if $T[\mathsf{SA}[i]]$ is S-type or not, we can also tell if $T[\mathsf{SA}[i]]$ is an LMS-character or not since $T[\mathsf{SA}[i]]$ is an LMS-character if and only if $T[\mathsf{SA}[i]]$ is S-type and $T[\mathsf{SA}[i]-1] > T[\mathsf{SA}[i]]$. In the scanning process, when we reach a new bucket, we can see whether this bucket contains S-type characters or not from Observation 1. Furthermore, if we know the number of S-type characters in this bucket (Lemma 5), we are done with this step since all S-type suffixes appear after the L-suffixes in any bucket (Property 1). This step costs $O(n)$ time overall since each character is scanned at most twice. □

Example: Continue our example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
LMS		*				*				*			*
SA	(12)	(*E*)	(*E*	*E*	1	5	9)	(*E*	*E*)	(*E*	*E*	*E*	*E*)

(1) Sorting the LMS-prefixes of all suffixes (see Section 3.7):

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	(1	5	9	2	6)	(10	0)	(4	8	3	7)

(2) Placing the indices of all sorted LMS-substrings in $\mathsf{SA}[n-n_1 \dots n-1]$:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	**12**	**1**	**5**	**9**

□

3.5 Construct the reduced problem T_1

In this section, we construct the smaller recursive problem T_1. We rename the sorted LMS-substrings (obtained from the previous step) using their ranks to obtain T_1. Note that this step is not difficult and similar to the previous algorithms (e.g., [NZC09a, Nong13]).

Now, we spell out the details for this step. Initially, all LMS-substrings are sorted in $\mathsf{SA}[n-n_1 \dots n-1]$. First, we let the rank of the smallest LMS-substring corresponding to $\mathsf{SA}[n-n_1]$ (i.e., the LMS-substring which begins from index $\mathsf{SA}[n-n_1]$) be 0 (it must be the sentinel). Then, we scan $\mathsf{SA}[n-n_1+1 \dots n-1]$ from left to right to compute the rank for each LMS-substring. When scanning $\mathsf{SA}[i]$, we compare the LMS-substring corresponding to $\mathsf{SA}[i]$ and that corresponding to $\mathsf{SA}[i-1]$. If they are the same, $\mathsf{SA}[i]$ gets the same rank as $\mathsf{SA}[i-1]$. Otherwise, the rank of $\mathsf{SA}[i]$ is the rank of $\mathsf{SA}[i-1]$ plus 1. Since we have no extra space, we need to store the ranks in SA as well. In particular, the rank of $\mathsf{SA}[i]$ is stored in $\mathsf{SA}[\lfloor \frac{\mathsf{SA}[i]}{2} \rfloor]$. There is no conflict since any two LMS-characters are not adjacent. Finally, we shift nonempty entries in $\mathsf{SA}[0 \dots n-n_1-1]$ to the head of SA, so that the ranks occupy a consecutive segment of the space. Now, we have obtained the reduced problem T_1 which is stored in $\mathsf{SA}[0 \dots n_1-1]$. In other words, $\mathsf{SA}[i]$ ($i \in [0, n_1-1]$) stores the new name of the i-th LMS-substring with respect to its appearance in the input string T.

Example: Continue our example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	7	6	6	9	9	6	6	9	9	6	7	1	0
LMS		*				*				*			*
SA	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	12	1	5	9

After scanning $\mathsf{SA}[n-n_1 \ldots n-1]$ (which stores the sorted LMS-substrings):

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	**1**	E	**1**	E	**2**	E	**0**	E	E	12	1	5	9

Finally, we get T_1 stored in $\mathsf{SA}[0 \ldots n_1-1]$ by shifting nonempty items in $\mathsf{SA}[0 \ldots n-n_1-1]$ to the head of SA.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	**1**	**1**	**2**	**0**	E	E	E	E	E	12	1	5	9

Note that $T_1 =$ "1120" which corresponds to the LMS-substrings {"66996", "66996", "6710", "0"}. □

First, we give an observation which helps us to identify the S-type and L-type characters of T. Then we obtain the following lemma which implies that T_1 can be obtained in linear time.

Observation 2 *For any index i of T, let $j \in [i+1, n-1]$ be the smallest index such that $T[j] < T[j+1]$ (So $T[j]$ is S-type). Furthermore let $k \in [i+1, j]$ be the smallest index such that $T[l] = T[j]$ for any $k \le l \le j$. Then $T[k]$ is the first S-type character after index i. Moreover, all characters between $T[i]$ and $T[k]$ are L-type, and characters between $T[k]$ and $T[j]$ are S-type.*

Lemma 7 *T_1 can be obtained using $O(n)$ time and $O(1)$ workspace.*

Proof: For the workspace part, it is obvious since we do not use any extra space beyond SA in the above step. For the time part, we only need to explain the running time of the comparison process. When we compare $\mathsf{SA}[i]$ and $\mathsf{SA}[i-1]$, we can know the length of these two LMS-substrings (indicated by $\mathsf{SA}[i]$ and $\mathsf{SA}[i-1]$) from the Observation 2. Note that each character of T is scanned at most twice since it is only scanned when identifying it length and its adjacent predecessor LMS-substring. Thus the comparison process takes $O(n)$ time because the total length of all LMS-substrings is less than $2n$. □

3.6 Sort the LMS-suffixes by solving T_1 recursively

In this section, we sort all LMS-suffixes and place their indices in the tail of their corresponding buckets in SA. This can be done as follows:

1. We first solve T_1 recursively. From Section 3.5, T_1 is stored in $\mathsf{SA}[0 \ldots n_1-1]$. We define SA_1 to be $\mathsf{SA}[n-n_1 \ldots n-1]$ and use SA_1 to store the output of the subproblem T_1.

2. Now, we put all indices of LMS-suffixes in SA. First we move SA_1 to $\mathsf{SA}[0 \ldots n_1-1]$ (i.e., move $\mathsf{SA}[n-n_1 \ldots n-1]$ to $\mathsf{SA}[0 \ldots n_1-1]$). Then we scan T from right to left. For every LMS-character $T[i]$, place i (i.e., index of $\mathsf{suf}(i)$) in the tail of SA.

3. For notational convenience, we define $\mathsf{LMS}[0 \ldots n_1] \triangleq \mathsf{SA}[n-n_1 \ldots n-1]$. Now, we obtain the sorted order of all LMS-suffixes of the original string T by letting $\mathsf{SA}[i] = \mathsf{LMS}[\mathsf{SA}[i]]$ for all $i \in [0, n_1-1]$.

4. Finally, we scan $\mathsf{SA}[0 \ldots n_1-1]$ once more from right to left, and move the indices of LMS-suffixes in the same bucket to the tail of its bucket and clear other entries. This is easy to do since each S-type $T[i]$ (after the renaming step in Section 3.2) has pointed to the tail of its bucket.

Example: Continue our example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	1	1	2	0	E	E	E	E	E	E	E	E	E

Step 1. Solve T_1 recursively:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	1	1	2	0	E	E	E	E	E	**3**	**0**	**1**	**2**

Step 2. After move SA_1 to $\mathsf{SA}[0 \dots n_1 - 1]$ and put all LMS-suffixes in SA:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	**3**	**0**	**1**	**2**	E	E	E	E	E	**1**	**5**	**9**	**12**

Step 3. Get all sorted LMS-suffixes:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	**12**	**1**	**5**	**9**	E	E	E	E	E	1	5	9	12

Step 4. Move the indices of LMS-suffixes in the same bucket to the tail of its bucket and clear other entries:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(**12**)	(E)	(E	E	**1**	**5**	**9**)	(E	E)	(E	E	E	E)

□

Lemma 8 *All LMS-suffixes can be sorted by solving the reduced problem T_1 recursively and placed in the tail of their corresponding buckets in* SA *using $O(n)$ time and $O(1)$ workspace.*

Proof: Each LMS-substring corresponds to a character of T_1 and this character is the rank of the LMS-substring from Section 3.5. Hence, the lexicographical order of LMS-suffixes of T is the same as the order of suffixes in T_1. □

3.7 Induced sort all suffixes of T

Now, we sort all suffixes of T from the sorted LMS-suffixes using induced sorting with our interior counter trick (Note that this step is the same as what we did in Section 3.4. Now, we describe the details here). First, we induced-sort the order of all L-suffixes from LMS-suffixes. Then we induce the order of S-suffixes from the L-suffixes. Now, we show how to carry out these steps with the desired optimal time and space.

Step 1. Induced sort all L-suffixes from the sorted LMS-suffixes: Some details of this step is similar to our previous step in Section 3.3 where we introduced our interior counter trick. We divide this step into two parts as follows:

(1) First initialize SA: We scan T from right to left. For every $T[i]$ which is L-type, do the following:

  (i) If $\mathsf{SA}[T[i]] = \mathsf{Empty}$, let $\mathsf{SA}[T[i]] = \mathsf{Unique}$ (unique L-type character in this bucket).

  (ii) If $\mathsf{SA}[T[i]] = \mathsf{Unique}$, let $\mathsf{SA}[T[i]] = \mathsf{Multi}$ (the number of L-type characters in this bucket is at least 2).

  (iii) Otherwise do nothing.

(2) Then we scan SA from left to right to sort all the L-suffixes.

  (i) If $\mathsf{SA}[i] = \mathsf{Empty}$, do nothing.

(ii) If $\mathsf{SA}[i]$ is an index, we let $j = \mathsf{SA}[i] - 1$. Then, if $\mathsf{suf}(j)$ is L-suffix (this can be identified in constant time from the following Lemma 9), we place $\mathsf{suf}(j)$ into the LF-entry (recall that LF-entry denotes the leftmost free entry in its bucket) of its bucket and increase the counter by one.

(iii) If $\mathsf{SA}[i] = \mathsf{Multi}$, which means $\mathsf{SA}[i]$ is the head of its bucket, and this bucket has at least two L-suffixes which are not sorted, we use $\mathsf{SA}[i]$ and $\mathsf{SA}[i+1]$ as the bucket head (the symbol Multi) and the counter of this bucket, respectively. Then we skip these two entries and continue to scan $\mathsf{SA}[i+2]$.

Now, all L-suffixes have been sorted. Note that we still need to scan SA once more to free these positions occupied by Multi and counters. After this, the indices of all L-suffixes are in their final positions in SA.

Step 2. Remove LMS-Suffixes from SA: We can use a trick similar to the previous Step 2 in Section 3.3, i.e., placing the indices of LMS-characters into SA. The difference is that instead of placing the actual LMS-characters, we place the Empty symbol instead. Also note that we do not delete the sentinel since it must be in the final position. Now, SA contains only all L-suffixes and the sentinel, and all of them are in their final positions in SA.

Step 3. Induced sort all S-suffixes from the sorted L-suffixes: Now, this step is completely symmetrical to the above Step 1 (Sort all L-suffixes using induced sorting). We use S-type and RF-entry instead of L-type and LF-entry, and we do not repeat the details here.

Example: Continue our example:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	7	6	6	9	9	6	6	9	9	6	7	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	(12)	(E)	(E	E	1	5	9)	(E	E)	(E	E	E	E)

Step 1. Induced-sort all L-suffixes from the sorted LMS-suffixes:
(1) After initialization:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(**U**)	(E	E	1	5	9)	(**M**	E)	(**M**	E	E	E)

(2) Scan SA from left to right to sort all L-suffixes:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(**12**)	(**11**)	(E	E	1	5	9)	(M	E)	(M	E	E	E)
SA	(12)	(**11**)	(E	E	1	5	9)	(**10**	E)	(M	E	E	E)
SA	(12)	(11)	(E	E	**1**	5	9)	(10	**0**)	(M	E	E	E)
SA	(12)	(11)	(E	E	1	**5**	9)	(10	0)	(**M**	**1**	**4**	E)
SA	(12)	(11)	(E	E	1	5	**9**)	(10	0)	(**M**	**2**	4	**8**)
SA	(12)	(11)	(E	E	1	5	9)	(10	0)	(M	2	**4**	8)
SA	(12)	(11)	(E	E	1	5	9)	(10	0)	(**4**	**8**	**3**	**E**)
SA	(12)	(11)	(E	E	1	5	9)	(10	0)	(4	**8**	3	**7**)

Note that the third to last line is the case (iii). So we skip these two entries (i.e., 'M' and '2').
Step 2. Remove LMS-Suffixes from SA:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	(E	E	**E**	**E**	**E**)	(10	0)	(4	8	3	7)

Step 3. Induced-sort all S-suffixes from the sorted L-suffixes:
(1) After initialization:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	(E	E	E	E	**M**)	(10	0)	(4	8	3	7)

(2) Scan SA from right to left to sort all S-suffixes:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	(E	E	**6**	**1**	**M**)	(10	0)	(4	8	3	**7**)
SA	(12)	(11)	(E	**2**	6	**2**	**M**)	(10	0)	(4	8	**3**	7)
SA	(12)	(11)	(**9**	2	6	**3**	**M**)	(**10**	0)	(4	8	3	7)
SA	(12)	(11)	(9	2	**6**	3	M)	(10	0)	(4	8	3	7)
SA	(12)	(11)	(**E**	**5**	**9**	**2**	**6**)	(10	0)	(4	8	3	7)
SA	(12)	(11)	(**1**	5	9	**2**	6)	(10	0)	(4	8	3	7)
SA	(12)	(11)	(1	5	9	2	6)	(10	0)	(4	8	3	7)

□

In order to show the time used in this step, we need the following useful lemma to identify the L/S-type information in the induced-sorting step.

Lemma 9 *When we scan* SA *from left to right to induced-sort L-suffixes, for each scanned* $\mathsf{SA}[i]$*, we want to identify the type of* $\mathsf{suf}(\mathsf{SA}[i]-1)$*. If* $T[\mathsf{SA}[i]-1] \neq T[\mathsf{SA}[i]]$*, the type of* $\mathsf{suf}(\mathsf{SA}[i]-1)$ *can be obtained immediately. Otherwise* $T[\mathsf{SA}[i]-1] = T[\mathsf{SA}[i]]$ *(this case* $\mathsf{suf}(\mathsf{SA}[i]-1)$ *belongs to the current scanned bucket* $T[\mathsf{SA}[i]]$*). If all L-suffixes of* T *belonging to bucket* $T[\mathsf{SA}[i]]$ *are not already sorted, then* $\mathsf{suf}(\mathsf{SA}[i]-1)$ *is an L-suffix.*

Proof: According to Property 1, in any bucket, the S-suffixes always appear after the L-suffixes in SA. Besides, it is obvious that every suffix of T is considered exactly once. Furthermore, we can distinguish whether all L-suffixes of T belonging to the current bucket $T[\mathsf{SA}[i]]$ are already sorted or not by scanning the current bucket once, when we are reaching a new bucket. Combining these observations, the lemma is proved. □

Lemma 10 *Given all sorted LMS-suffixes of* T*, all suffixes can be sorted correctly using* $O(n)$ *time and* $O(1)$ *workspace according to the induced sorting steps.*

Proof: For the correctness: we can sort all L-suffixes correctly from the sorted LMS-suffixes using induced sorting step from Lemma 2 and we can sort all S-suffixes correctly from the sorted L-suffixes using induced sorting step from Lemma 1. □

Now, we obtain the following theorem for our optimal in-place algorithm.

Theorem 4 *Our Algorithm takes* $O(n)$ *time and* $O(1)$ *workspace to compute the suffix array of string* T *over integer alphabets with* $|\Sigma| \leq n$*.*

Proof: The time complexity simply follows from the recursion $T(n) = T(n/2) + O(n) = O(n)$. For the workspace, every step in our algorithm uses $O(1)$ workspace and different steps can reuse the $O(1)$ workspace too. In the recursive subproblem, we can also reuse the $O(1)$ workspace. Note that we do not need to store n_i (i.e. the size of reduced problem T_i at recursive level i) for each recursive level i. We can use a global variable to denote the size of T_i (i.e. n_i) at the current level i. Before we return to the upper level $i-1$, we can recover the value of n_{i-1} using the same global variable and scan SA not SA_i) from the end to T_i (note that it is stored in SA). For the other case (i.e., go into the deeper level $i+1$), n_{i+1} is very easy to obtain. □

4 Suffix Sorting for Read-only Integer Alphabets

4.1 Framework

First, we define some notations. Let n_L and n_S denote the number of L-suffixes and S-suffixes, respectively. Let n_1 denote the length of the reduced problem T_1, i.e., n_1 equals to the number of LMS-suffixes (Case 1) or LML-suffixes (Case 2). Now, we describe the framework of our algorithm as follows:

1. If $n_L \leq n_S$ (i.e., the number of L-suffixes is no larger than that of S-suffixes), then:

   (1) (Section 4.2) Sort all LMS-characters of T.
   We use counting sort to sort all LMS-characters of T in $\mathsf{SA}[n-n_1 \dots n-1]$. In the counting sort step, we use $\mathsf{SA}[0 \dots n/2]$ as the temporary space (counting array). After this step, all indices of the sorted LMS-characters are stored in $\mathsf{SA}[n-n_1 \dots n-1]$. Note that this step is different from our previous algorithm in Section 3 since we sort and store the LMS-characters in the end of SA instead of their corresponding buckets, because T is read-only now.

   (2) (Section 4.4) Induced sort all LMS-substrings from the sorted LMS-characters.
   This induced-sorting step is the same as Step (4) below where we induced-sort all suffixes from the sorted LMS-suffixes. Thus, we only describe the details of this step in Section 4.4. After this step, all indices of the sorted LMS-substrings are stored in $\mathsf{SA}[n-n_1 \dots n-1]$.

   (3) (Section 4.3) Construct and solve the reduced problem T_1 from the sorted LMS-substrings.
   We construct the reduced problem T_1 using the ranks of all sorted LMS-substrings which are stored in $\mathsf{SA}[n-n_1 \dots n-1]$, where the ranks of LMS-substrings correspond to the lexicographical order of the sorted LMS-substrings (this construction step is the same as that in Section 3.5). Then we get the reduced problem T_1 in $\mathsf{SA}[0 \dots n_1-1]$ and solve T_1 recursively to obtain the sorted LMS-suffixes. In the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[n-n_1 \dots n-1]$ as the output space for T_1. After this step, all indices of the sorted LMS-suffixes are stored in $\mathsf{SA}[n-n_1 \dots n-1]$.

   (4) (Section 4.4) Induced sort all suffixes of T from the sorted LMS-suffixes (T_1).
   We induced-sort all suffixes of T from the sorted LMS-suffixes which are stored in $\mathsf{SA}[n-n_1 \dots n-1]$. In the sorting step, we use the *interior counter trick*, as in Section 3, which can implicitly represent the dynamic LF/RF-entry information in SA. Besides, we provide a *pointer data structure* which can represent the bucket heads/tails in SA. Combining these two techniques, we can remove the workspace needed by the bucket array. Due to the space required by our pointer data structure is nonconstant, we divide this induced sorting step into two stages to address this issue. Note that this step is the main technical part of our optimal in-place algorithm. After this step, all indices of the suffixes of T are sorted and stored in $\mathsf{SA}[0 \dots n-1]$.

2. Otherwise, execute the above steps switching the role of LMS with LML.

Without loss of generality, we assume that $n_L \leq n_S$. Note that we compare the number of L-suffixes and S-suffix at the beginning since we need half of the space of SA to construct our pointer data structure for induced-sorting the L-suffixes (from the sorted LMS-suffixes) and S-suffixes (from the sorted L-suffixes) in Step (4). The in-place implementation of this induced sorting step is the main technical part of our optimal in-place algorithm. Note that the empty space is enough since the number of LMS-suffixes (i.e., n_1) and L-suffixes (i.e., n_L) both are less than or equal to $n/2$, where $n_1 \leq n/2$ since any two LMS-characters are not adjacent by the definition of LMS-characters, and $n_L \leq n/2$ since $n_L \leq n_S$. Note that for previous algorithms (e.g., [NZC09a, Nong13]), they do not need the comparison at the beginning since they use the

bucket array (which needs $|\Sigma|$ words workspace) in the induced sorting step. Here, we construct the pointer data structure and combine our interior counter trick to remove the workspace.

Now, we describe the details of our in-place algorithm in the following sections.

4.2 Sort all LMS-characters of T

In this section, we sort all LMS-characters of T and place their indices in $\mathsf{SA}[n-n_1 \dots n-1]$. Recall that n_1 denotes the number of LMS-characters.

Now, we describe the details. Since $|\Sigma| = O(n)$, we can assume that $|\Sigma| \leq dn$ for some constant d. We divide the LMS-characters of T into $2d$ partitions and sort each partition one by one. The partition i contains the LMS-characters which belong to $\left[\frac{i|\Sigma|}{2d}+1, \frac{(i+1)|\Sigma|}{2d}\right]$, for $0 \leq i < 2d$. We use m_i to denote the number of LMS-characters in partition i. Then for each partition i, we use the standard *counting sort* (see e.g., [CLRS01, Chap. 8]) to sort these m_i LMS-characters (the LMS-characters can be identified by scanning T once from right to left). Concretely, we use $\mathsf{SA}[0 \dots n/2]$ as the temporary counting array, and use $\mathsf{SA}[n/2+\sum_{j=0}^{i-1} m_j+1 \dots n/2+\sum_{j=0}^{i} m_j]$ as the output array. After this counting sort step, the indices of these m_i sorted LMS-characters have been placed in $\mathsf{SA}[n/2+\sum_{j=0}^{i-1} m_j+1 \dots n/2+\sum_{j=0}^{i} m_j]$.

Note that we can use the counting sort step for each partition. Because the gap of each partition is $\frac{|\Sigma|}{2d} \leq \frac{dn}{2d} = \frac{n}{2}$, the space of $\mathsf{SA}[0 \dots n/2]$ is enough for the temporary counting array (its size equals to the gap) of counting sort step. It is not hard to see that the sorting step takes $O(n)$ time and uses $O(1)$ workspace since we only make $2d$ times of counting sort steps (each step takes linear time).

After sorting all $2d$ partitions, all indices of the sorted LMS-characters are placed in $\mathsf{SA}[n/2+1, n/2+\sum_{j=0}^{2d-1} m_j]$ (i.e., $\mathsf{SA}[n/2+1, n/2+n_1]$). Then we move them to $\mathsf{SA}[n-n_1 \dots n-1]$, which can be easily done in linear time and $O(1)$ workspace.

4.3 Construct and solve the reduced problem T_1 from the sorted LMS-substrings

Construct the reduced problem T_1: We construct the reduced problem T_1 using the ranks of all sorted LMS-substrings which are stored in $\mathsf{SA}[n-n_1 \dots n-1]$ from the Step (2) (see Section 4.1 above), where the ranks of LMS-substrings are corresponding to the lexicographical order of the sorted LMS-substrings.

Note that this construction step is exactly the same as that in Section 3.5. Thus we omit the details and recall the same lemma as follows.

Lemma 11 *T_1 can be constructed using $O(n)$ time and $O(1)$ workspace.*

Solve T_1 recursively: Now, we sort all LMS-suffixes by solving T_1 recursively and place their indices in the tail of SA (i.e. $\mathsf{SA}[n-n_1 \dots n-1]$). This step is carried out as follows (similar to the Section 3.6 except Step 4 below):

1. We first solve T_1 recursively. Recall that T_1 is stored in $\mathsf{SA}[0 \dots n_1-1]$. We define SA_1 to be $\mathsf{SA}[n-n_1 \dots n-1]$ and use SA_1 to store the output of the subproblem T_1.

2. Now, we put all indices of LMS-suffixes in SA. First we move SA_1 to $\mathsf{SA}[0 \dots n_1-1]$ (i.e., move $\mathsf{SA}[n-n_1 \dots n-1]$ to $\mathsf{SA}[0 \dots n_1-1]$). Then we scan T from right to left. For every LMS-character $T[i]$, place i (i.e., index of $\mathsf{suf}(i)$) in the tail of SA.

3. For notational convenience, we define $\mathsf{LMS}[0 \dots n_1] \triangleq \mathsf{SA}[n-n_1 \dots n-1]$. Now, we obtain the sorted order of all LMS-suffixes of the original string T by letting $\mathsf{SA}[i] = \mathsf{LMS}[\mathsf{SA}[i]]$ for all $i \in [0, n_1-1]$.

4. Finally, we finish this step by moving $\mathsf{SA}[0 \dots n_1 - 1]$ to $\mathsf{SA}[n - n_1 \dots n - 1]$. Now, all indices of the sorted LMS-suffixes are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$.

Lemma 12 *All LMS-suffixes can be sorted by solving the reduced problem T_1 recursively and placed in the tail of SA using $O(n)$ time and $O(1)$ workspace.*

Proof: The time and space used in this step are easy to verify. We only show the correctness of this step. Each character of T_1 corresponds to an LMS-substring of T and this character is the rank of the corresponding sorted LMS-substring. Hence, the lexicographical order of LMS-suffixes of T is the same as the order of suffixes in T_1. □

4.4 Induced sort all suffixes of T from the sorted LMS-suffixes

In this section, we show that how to induced-sort all suffixes from the sorted LMS-suffixes. All indices of the sorted LMS-suffixes have been placed in $\mathsf{SA}[n - n_1 \dots n - 1]$ from the previous step (see Lemma 12). Note that this step is the main technical part of our in-place algorithm. We develop two techniques (*interior counter trick* and *pointer data structure*) for implementing this induced sorting in-place.

Let $\mathsf{SA}_L = \mathsf{SA}[0 \dots n_L - 1]$ and $\mathsf{SA}_S = \mathsf{SA}[n_L \dots n - 1]$. Recall that n_S and n_L denote the number of S-suffixes and L-suffixes, respectively. Also note that $n_L + n_S = n$. First, we sort all n_L L-suffixes from the sorted LMS-suffixes which are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$ and store the sorted L-suffixes in SA_L. Then, we sort all n_S S-suffixes from the sorted L-suffixes and store the sorted S-suffixes in SA_S. Finally, we merge the sorted L-suffixes (stored in SA_L) and S-suffixes (stored in SA_S) to sort all suffixes in $\mathsf{SA}[0 \dots n - 1]$.

Note that sorting the n_L L-suffixes from the sorted LMS-suffixes is totally symmetrical as sorting the n_S S-suffixes from the sorted L-suffixes, as stated in Section 2. Thus, we only need to show the details of how to sort all n_S S-suffixes from the sorted L-suffixes which has already been stored in SA_L, and store the sorted S-suffixes in SA_S. Before we show the details, we briefly recall the original induced sorting step here (we have introduced it in Section 2 and provided a running example in Appendix A.1.). Now, we recall the symmetrical case here, i.e., inducing the order of S-suffixes from the sorted L-suffixes.

Inducing the order of S-suffixes from the sorted L-suffixes: We scan SA from right to left (i.e., from $\mathsf{SA}[n - 1]$ to $\mathsf{SA}[0]$). When we scan $\mathsf{SA}[i]$, let $j = \mathsf{SA}[i] - 1$. If $T[j]$ is S-type, i.e., $\mathsf{suf}(j)$ is an S-suffix (indicated by the type array), we place the index of $\mathsf{suf}(j)$ (i.e. j) into the RF-entry of bucket $T[j]$, and then let the RF-pointer of this bucket $T[j]$ point to the next entry. If $T[j]$ is L-type, we do nothing (since all L-suffixes are sorted in the correct positions). The RF-pointers are maintained by the bucket array.

In order to obtain the in-place algorithm, we need to show how to remove the workspace needed by the bucket array and type array in the induced sorting step. Briefly speaking, the purpose of the pointer data structure is to indicate the bucket tails of S-suffixes, and the purpose of the interior counter trick is to maintain the RF-pointers of the buckets dynamically. Thus, for a query of RF-entry for $\mathsf{suf}(j)$ in bucket $T[j]$, we know the tail of the bucket $T[j]$ from the pointer data structure in constant time (Lemma 16), then we use the interior counter trick to indicate the RF-entry in this bucket (Lemma 13). For removing the type array, we use the Lemma 14 to identify the L/S-suffixes in the induced sorting step.

Now, we describe the details. First, we introduce our interior counter trick, extending the one from Section 3. Here, we assume that the tail of the bucket of any S-suffix is known (which is indicated by the Lemma 16).

Interior counter trick: Note that the buckets of the S-suffixes we discussed in this section are in $\mathsf{SA}_S = \mathsf{SA}[n_L \dots n-1]$, since we already have placed the sorted L-suffixes in $\mathsf{SA}_L = \mathsf{SA}[0 \dots n_L - 1]$. Thus, we only need to sort all S-suffixes to their corresponding buckets in SA_S and the buckets only contains S-suffixes now.

Here we only describe the details of interior counter trick for one bucket since other buckets are the same. Note that we assume that the tail of the bucket of any S-suffix is known (Lemma 16). To simplify the representation, we assume the bucket from index 0 to index $m-1$ of SA_S, where m is the size of this bucket (i.e. the number of S-suffixes in this bucket is m). We only describe the case where $m > 3$ since other cases with $m \leq 3$ are similar and simpler. We define five special symbols $\mathsf{B_H}$ (head of the bucket), $\mathsf{B_T}$ (tail of the bucket), E (Empty), $\mathsf{R_1}$ (one remaining S-suffix) and $\mathsf{R_2}$ (two remaining S-suffixes) [6].

First, we use three special symbols to initialize this bucket, i.e., let $\mathsf{SA}_S[0] = \mathsf{B_H}$, $\mathsf{SA}_S[m-2] = \mathsf{E}$ and $\mathsf{SA}_S[m-1] = \mathsf{B_T}$. Let S_i denote the index of the i-th S-suffix which needs to be placed into the RF-entry of this bucket. Now, we describe how to place the indices of these m S-suffixes into the RF-entry of this bucket one by one. We distinguish the following four cases:

(1) If $\mathsf{SA}_S[m-1] = \mathsf{B_T}$, and $\mathsf{SA}_S[m-2] = \mathsf{E}$ or $\mathsf{SA}_S[m-\mathsf{SA}_S[m-2]-3] \neq \mathsf{B_H}$: In this case, we place the index of the current S-suffix (i.e., S_i) into the RF-entry of this bucket, where $1 \leq i \leq m-3$. Concretely, we know the position of the tail of this bucket in SA_S, i.e., $m-1$ according to the assumption. Then, we use $\mathsf{SA}_S[m-2]$ as the counter to denote the number of the indices of S-suffixes has been placed so far. Note that the RF-entry of this bucket is pointed by this counter (i.e. RF-pointer). Thus, we can place the index of the current S-suffix (S_i) into the RF-entry of this bucket in constant time, and then update the counter $\mathsf{SA}_S[m-2]$.

(2) If $\mathsf{SA}_S[m-1] = \mathsf{B_T}$ and $\mathsf{SA}_S[m-\mathsf{SA}_S[m-2]-3] = \mathsf{B_H}$: In this case, we place the index of the third to last S-suffix (i.e. S_{m-2}) into the RF-entry of this bucket. Concretely, we shift the previous $m-3$ S-suffixes which stored in $\mathsf{SA}_S[1, \dots, m-3]$ to $\mathsf{SA}_S[2, \dots, m-2]$. Then, we place S_{m-2} into $\mathsf{SA}_S[1]$ and let $\mathsf{SA}_S[m-1] = \mathsf{R_2}$. This step takes $O(m)$ time since we shift $m-3$ S-suffixes.

(3) If $\mathsf{SA}_S[m-1] = \mathsf{R_2}$: In this case, we place the index of the second to last S-suffix (i.e. S_{m-1}) into the RF-entry of this bucket. We shift the previous $m-2$ S-suffixes which stored in $\mathsf{SA}_S[1, \dots, m-2]$ to $\mathsf{SA}_S[2, \dots, m-1]$. Then, we place S_{m-1} into $\mathsf{SA}_S[1]$ and let $\mathsf{SA}_S[0] = \mathsf{R_1}$. This step takes $O(m)$ time since we shift $m-2$ S-suffixes.

(4) Otherwise: In this case, we place the index of the last S-suffix (i.e. S_m) into the RF-entry of this bucket. First, we know the tail of the bucket indicated by our pointer data structure in constant time. Then, we search the entries before the tail one by one until that we find the special symbol $\mathsf{R_1}$. We let this entry to be S_m. This step takes $O(m)$ time since we search $m-1$ S-suffixes.

In order to demonstrate these four cases more clearly, we also provide a demonstration as follows:

[6] Recall that the special symbol is only used to simplify the argument. See Appendix B for the details.

Index	0	1	...	$m-3$	$m-2$	$m-1$
SA_S	$\mathsf{SA}_S[0]$	$\mathsf{SA}_S[1]$	...	$\mathsf{SA}_S[m-3]$	$\mathsf{SA}_S[m-2]$	$\mathsf{SA}_S[m-1]$
After initialization :						
SA_S	$\mathsf{B_H}$	$\mathsf{SA}_S[1]$	...	$\mathsf{SA}_S[m-3]$	E	$\mathsf{B_T}$
Case (1) :						
SA_S	$\mathsf{B_H}$	$\mathsf{SA}_S[1]$	...	$\mathbf{S_1}$	$\mathbf{1}$	$\mathsf{B_T}$
$\vdots$					$\vdots$	
SA_S	$\mathsf{B_H}$	$\mathbf{S_{m-3}}$	...	S_1	$\mathbf{m-3}$	$\mathsf{B_T}$
Case (2) :						
SA_S	$\mathsf{B_H}$	$\mathbf{S_{m-2}}$	...	$\mathbf{S_2}$	$\mathbf{S_1}$	R_2
Case (3) :						
SA_S	R_1	$\mathbf{S_{m-1}}$	...	$\mathbf{S_3}$	$\mathbf{S_2}$	$\mathbf{S_1}$
Case (4) :						
SA_S	$\mathbf{S_m}$	S_{m-1}	...	S_3	S_2	S_1

Note that this step uses $O(1)$ workspace since there is no bucket array and type array, and the space needed by our interior counter trick and pointer data structure is in SA_S. The purpose of the interior counter trick is to dynamic maintain the RF-pointers of the buckets. E.g., for a query of RF-entry for $\mathsf{suf}(j)$ in bucket $T[j]$, first we know the tail of the bucket $T[j]$ by the assumption, then we use the interior counter trick to indicate the RF-entry in this bucket. We have the following lemma.

Lemma 13 *If the tail of the bucket of any S-suffix is known, one can sort the S-suffixes from the sorted L-suffixes using the induced sorting step with the interior counter trick in linear time and $O(1)$ workspace.*

Note that in the induced sorting step, one uses the type array to identify whether the $\mathsf{suf}(j)$ is S-suffix or not. For removing the type array, we use the following Lemma 14 to identify the type of the L- or S-suffix in the induced-sorting step.

Lemma 14 *If $T[j] \neq T[\mathsf{SA}[i]]$, the type of $\mathsf{suf}(j)$ can be obtained immediately, where $j = \mathsf{SA}[i] - 1$. Otherwise $T[j] = T[\mathsf{SA}[i]]$ (this case $\mathsf{suf}(j)$ belongs to the current scanning bucket $T[\mathsf{SA}[i]]$), if all S-suffixes of T that belong to bucket $T[\mathsf{SA}[i]]$ are not already sorted, then the $\mathsf{suf}(j)$ is S-suffix.*

It is not hard to decide whether all S-suffixes in the current scanning bucket $T[\mathsf{SA}[i]]$ are already sorted or not. One can use an extra variable to denote how many S-suffixes remain in the current scanning bucket $T[\mathsf{SA}[i]]$. When we begin to scan a new bucket, we scan this bucket once to initialize this variable. Note that we can do this initialization, since there are special symbols in SA_S for each bucket after we initialize the buckets in our interior counter trick.

Now, there is only one thing left: how to know the tails of the bucket of S-suffixes in the induced sorting step. The purpose of the pointer data structure is to indicate the tails of the bucket of S-suffixes. However, the pointer data structure requires c_p words, where the value of c_p will be specified later. Thus, we need to divide this induced-sorting step into two stages. The first stage (Section 4.4.1) is to sort the first $n_S - c_p$ S-suffixes (i.e. the largest $n_S - c_p$ S-suffixes), where our pointer data structure still exists. The second stage (Section 4.4.2) is to sort the last c_p S-suffixes, where there is no space for the pointer data structure.

4.4.1 The first stage

Pointer data structure: Now, we construct our pointer data structure which supports to find the tails of the buckets in constant time. We store the pointer data structure in the tail of SA_S, recall that $\mathsf{SA}_S = \mathsf{SA}[n_L, \ldots, n-1]$. Now, we describe the details. We divide the S-suffixes of T into $4d$ parts according to their first characters, and construct the pointer data structure for each part respectively. The $4d$ parts are divided by $T[j] \in \left[\frac{i|\Sigma|}{4d}+1, \frac{(i+1)|\Sigma|}{4d}\right]$, for $0 \leq i < 4d$. Let D_i denote the pointer data structure of the i-th part. We only show the details how we construct the pointer data structure D_0 as follows, since constructing D_i is similar for $0 < i < 4d$ (one only need to shift $T[j]$ with $\frac{i|\Sigma|}{4d}$).

(1) First, we let $\mathsf{SA}_S[i] = 1$ for all $i \in [1, \frac{|\Sigma|}{4d}]$. Then we scan T from right to left. For every S-type $T[i] \in [1, \frac{|\Sigma|}{4d}]$, we increase $\mathsf{SA}_S[T[i]]$ by one.

(2) Then we scan $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ from left to right. We use a variable sum to count the sum, first initialize $sum = -1$. For each $\mathsf{SA}_S[i]$ which is being scanned, first let $sum = sum + \mathsf{SA}_S[i]$, then let $\mathsf{SA}_S[i] = sum$. Now, for any S-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4d}]$, $\mathsf{SA}_S[T[i]] - T[i]$ must indicate the tail of bucket $T[i]$ in SA_S. Since we want every entry in $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ to be distinct, we initialize $\mathsf{SA}_S[i] = 1$ for all $i \in [1, \frac{|\Sigma|}{4d}]$ in Step (1). Hence the tail of bucket $T[i]$ is $\mathsf{SA}_S[T[i]] - T[i]$.

(3) Finally, we construct D_0 for $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ according to Lemma 15. D_0 uses at most $c(n + \frac{|\Sigma|}{4d})/\log n$ words space. We store D_0 in the tail of SA_S (i.e., $\mathsf{SA}_S[n_S - c(n + \frac{|\Sigma|}{4d})/\log n \ldots n_S - 1]$). D_0 supports to find the tail of the bucket of any S-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4d}]$ in constant time.

Lemma 15 *For any m distinct integers $0 \leq a_0 < a_1 \ldots < a_{m-1} \leq n$, where $m \leq n$ and $n > 1024$, one can construct a data structure using linear time (i.e., $O(n)$ time) and at most $cn/\log n$ words, where $1 < c < 2$, such that each query to the i-th smallest integer a_i ($\mathsf{select}(i)$) can be answered in constant time.*

Proof: We first construct a bitmap $B[0 \ldots n]$. we initialize B by $B[a_i] = 1$ for all $i \in [0, m-1]$. We need a data structure to support query $\mathsf{select}(i)$, which asks for the index of i-th 1 in B. There is an auxiliary data structure using $O(n/\log\log n)$ bits (more precisely $3n/\log\log n + n^{\frac{1}{4}}(\frac{1}{4}\log n \log\log n + \log\log n)$) which can be constructed in $O(n)$ time to support constant time select query in B [Jac89, Cla96, NP12]. Converting bits to words, we can see that the data structure uses at most $cn/\log n$ words (for $1 < c < 2$ if $n > 1024$). □

After this step, the pointer data structure (i.e. D_i for all $0 \leq i < 4d$) is stored in $\mathsf{SA}_S[n_S - c_p \ldots n_S - 1]$, where $c_p = \lceil 4d \cdot (c(n + \frac{|\Sigma|}{4d})/\log n)\rceil \leq \lceil 5dcn/\log n\rceil$. Recall that the sorted n_L L-suffixes are stored in SA_L and $n_S \geq n_L$. Thus the empty space (i.e., $\mathsf{SA}_S = \mathsf{SA}[n_L, \ldots, n-1]$) is at least half of the space of SA. It is enough to construct and store the pointer data structure, i.e., $\frac{|\Sigma|}{4d} \leq \frac{dn}{4d} = \frac{n}{4}$ for constructing it and $\frac{n}{4}$ for storing it (which uses c_p words). Note that we assume that $n > 4c_p$, otherwise it is easy to solve since n is constant. Hence the pointer data structure is constructed in linear time and supports to find the tail of the bucket of any S-suffix in constant time.

Lemma 16 *We can construct the pointer data structure in linear time, and this pointer data structure uses at most c_p words and can support to find the bucket tail of any S-suffix in constant time.*

Proof: We only need to specify the query time constant. We use $4d$ values, i.e., $m_1, \ldots, m_{4d}$, which denote the number of S-suffixes in each interval, respectively (They can be obtained from the variable sum which

is computed in the final stage of the Step (2)). Now, if we want to find the bucket tail of an S-suffix $\mathsf{suf}(i)$, we first compare $T[i]$ with $\frac{i|\Sigma|}{4d}$ (for $0 \leq i < 4d$) to see which pointer data structure $T[i]$ belongs to. Assume that it belongs to D_j. Then we do a $\mathsf{select}(T[i] - (j-1)\frac{|\Sigma|}{4d})$ query on D_j, and combine the select result with the corresponding m_k $(k < j)$ to identify the tail of bucket $T[i]$. All the above operations can be done in constant time. □

Now according to Lemma 13, 14 and 16, we can sort the first largest $n_S - c_p$ S-suffixes from the sorted L-suffixes which stored in SA_L using the induced sorting step with the interior counter trick and the pointer data structure which stored in $\mathsf{SA}[n - c_p \ldots n - 1]$, and we store the indices of the sorted $n_S - c_p$ S-suffixes in $\mathsf{SA}_S[0, \ldots, n_S - c_p - 1]$.

4.4.2 The second stage

Now, we only need to show that how to sort the last c_p S-suffixes which is occupied by our pointer data structure. First, we specify that how to identify whether an S-suffix belongs to the first $n_S - c_p$ S-suffixes or not. We can scan T once to find the character of $n_S - c_p$ largest S-suffix using the pointer data structure and let ch denote this character. If the tail of the bucket ch is exactly $c_p - 1$ in SA_S, then we compare the beginning character of the S-suffix with ch to identify which part it belongs to. Otherwise, the $n_S - c_p$ largest S-suffix belongs to bucket ch, and it will be stored in $\mathsf{SA}_S[c_p - 1]$. We only need two variables to indicate whether the S-suffix belongs to the first $n_S - c_p$ S-suffixes or not. One is the number to denote the tail of bucket ch, and the other is also an integer number which denotes the gap between the tail of bucket ch and $c_p - 1$.

Now, we describe the details to sort the last c_p S-suffixes. First, we move these largest $n_S - c_p$ S-suffixes to the tail of SA_S, i.e., $\mathsf{SA}_S[c_p, n_S - 1]$. Then we scan the T from right to left to place the smallest c_p S-suffixes into $\mathsf{SA}_S[0, c_p - 1]$. Now, we use merge sort with the in-place linear time merging algorithm [SS87] to sort these c_p S-suffixes, the sorting key for each S-suffix is its beginning character. After this sorting step, these c_p S-suffixes have been placed in their corresponding buckets in $\mathsf{SA}_S[0, c_p - 1]$. Note that we can use the same sorting step (which we used for sorting the first $n_S - c_p$ S-suffixes) to sort the last c_p S-suffixes without the pointer data structure. Now, the key point is that we can use the binary search (instead of the pointer data structure) to find the tails of the bucket for these c_p S-suffixes, since $c_p \leq \lceil 5dcn/\log n \rceil$ is small enough (i.e. $c_p \log n = O(n)$) to maintain that the time complexity of our algorithm is $O(n)$.

Using the binary search to extend interior counter trick is not very difficult, one can see the details in Section 5.5 which we induced sort all L-suffixes from the sorted S-suffixes for the general alphabets (Note that the optimal time is $O(n \log n)$ for the general alphabets case. Thus, we directly use the binary search to extend interior counter trick and do not use the pointer data structure in that case.).

After this step, all n_S S-suffixes are sorted in SA_S. Now we have all sorted L-suffixes in SA_L (i.e., $\mathsf{SA}[0 \ldots n_L - 1]$) and all sorted S-suffixes in SA_S (i.e., $\mathsf{SA}[n_L \ldots n - 1]$). We use the stable, in-place, linear time merging algorithm [SS87] to merge the ordered SA_L and SA_S (the merging key for $\mathsf{SA}[i]$ is $T[\mathsf{SA}[i]]$, i.e., the first character of $\mathsf{suf}(\mathsf{SA}[i])$). After this merging step, all suffixes of T have be sorted in $\mathsf{SA}[0 \ldots n - 1]$.

Finally, we obtain the following theorem for our optimal in-place algorithm.

Theorem 5 (Main Theorem) *Our Algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of string T over integer alphabets Σ, where T is read-only and $|\Sigma| = O(n)$.*

5 Suffix Sorting for Read-only General Alphabets

5.1 Framework

The framework of our algorithm for read-only general alphabets (i.e., the only operations allowed on the characters of T (read-only) are comparisons) is described as follows:

1. If $n_S \leq n_L$ (i.e., the number of S-suffixes is no larger than that of L-suffixes), then

   (1) (Section 5.2) Sort all S-substrings of T using mergesort directly.
   We use mergesort to sort all S-substring of T in $\mathsf{SA}[n - n_S \ldots n - 1]$. In the merging step of mergesort, we use $\mathsf{SA}[0 \ldots n_S - 1]$ as the temporary space. After this step, all S-substrings should be in the lexicographical order stored in $\mathsf{SA}[n - n_S \ldots n - 1]$.

   (2) (Section 5.3) Construct the reduced problem T_1 from the sorted S-substrings.
   We construct the reduced problem T_1 using the ranks of all sorted S-substrings which are stored in $\mathsf{SA}[n - n_S \ldots n - 1]$. The ranks of S-substrings are corresponding to the lexicographical order of the sorted S-substrings. After this step, we get the reduced problem T_1 in $\mathsf{SA}[0 \ldots n_S - 1]$.

   (3) (Section 5.4) Sort the S-suffixes by solving T_1 recursively.
   We sort $T_1 = \mathsf{SA}[0 \ldots n_S - 1]$ recursively. In the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[n - n_S \ldots n - 1]$ as the output space for T_1. Then we use the suffix array of T_1 (i.e. SA_1) to place all indices of the sorted S-suffixes of T into $\mathsf{SA}[n - n_S \ldots n - 1]$.

   (4) (Section 5.5) Induced sort all suffixes from the sorted S-suffixes.
   First, we place all indices of S-suffixes in their final positions in SA by using mergesort together with a stable, in-place, linear time merging algorithm [SS87]. Then we extend our *interior counter trick* to sort all L-suffixes from the sorted S-suffixes. Finally, all indices of the sorted suffixes of T are stored in $\mathsf{SA}[0 \ldots n - 1]$.

2. Otherwise, execute the above steps switching the roles of L and S.

Note that our algorithm is very simple since the optimal time complexity in this case is $O(n \log n)$ instead of $O(n)$. Moreover, our algorithm does not make any bit operations rather than previous algorithms, e.g., [FM07].

Without loss of generality, we assume that $n_S \leq n_L$. Now, we describe the details of our in-place algorithm in the following sections.

5.2 Sort all S-substrings of T

In this section, we sort all S-substrings of T as follows:

1. First, we scan T from right to left and place all indices of S-type characters into $\mathsf{SA}[n - n_S \ldots n - 1]$. Note that $n_S \leq n/2$ since we assume that $n_S \leq n_L$.

2. Then, we sort $\mathsf{SA}[n - n_S \ldots n - 1]$ using mergesort (the sorting key for $\mathsf{SA}[i]$ is the S-substring of T which begins at $T[\mathsf{SA}[i]]$). We use $\mathsf{SA}[0 \ldots n_S - 1]$ as the temporary space for mergesort. To compare two keys (i.e., two S-substrings) in mergesort, we simply do the straightforward character-wise comparisons.

After the above two steps, all the S-substring have been sorted in $\mathsf{SA}[n-n_S \ldots n-1]$. We have the following lemma.

Lemma 17 *We can sort all S-substrings using $O(n \log n)$ time and $O(1)$ workspace.*

Proof: Step 1 does not need any extra space and costs linear time, because we can compute the type of each character in $O(1)$ time during the right-to-left scan of T. Moreover, we know mergesort needs linear workspace. Hence, it is sufficient to use $\mathsf{SA}[0 \ldots n_S - 1]$ as the workspace for mergesort. For Step 2, it suffices to show that the time spent for comparison process in any recursive level of mergesort (there are $O(\log n)$ recursive levels) can be bounded by $O(n)$. In any level, each S-substring is compared to exactly one other S-substring. The length of the S-substrings can be obtained according to Observation 2. Recall that each character of T is scanned at most twice since it only be scanned when identifying the length of its adjacent predecessor S-substring and itself. Thus the comparison process takes $O(n)$ time in any level because the total length of all S-substrings is less than $2n$. □

5.3 Construct the reduced problem T_1 from the sorted S-substrings

In this section, we construct the reduced problem T_1 by renaming the sorted S-substrings. After Section 5.2, all S-substrings have been sorted in $\mathsf{SA}[n - n_S \ldots n - 1]$. The construction of T_1 consists of the following two steps:

1. We rename the S-substrings by their ranks. First let the rank of $\mathsf{SA}[n - n_S]$ be 0. We scan $\mathsf{SA}[n - n_S + 1 \ldots n - 1]$ from left to right. When scanning $\mathsf{SA}[i]$, we compare S-substring beginning with $T[\mathsf{SA}[i]]$ and S-substring beginning with $T[\mathsf{SA}[i-1]]$. If they are different, let the rank of $\mathsf{SA}[i]$ be the rank of $\mathsf{SA}[i-1]$ plus one. Otherwise, the rank of $\mathsf{SA}[i]$ is the same as that of $\mathsf{SA}[i-1]$. We store the rank of $\mathsf{SA}[i]$ in $\mathsf{SA}[i - n + n_S]$.

2. Next, we use the heapsort to sort $\mathsf{SA}[n-n_S \ldots n-1]$ (the sorting key for $\mathsf{SA}[i]$ is $\mathsf{SA}[i]$ itself). When we exchange two entries (say, $\mathsf{SA}[i]$ and $\mathsf{SA}[j]$, $i, j \in [n-n_S \ldots n-1]$) in $\mathsf{SA}[n-n_S \ldots n-1]$ during heapsort, we also exchange the corresponding two entries (i.e., $\mathsf{SA}[i - n + n_S]$ and $\mathsf{SA}[j - n + n_S]$) in $\mathsf{SA}[0 \ldots n_S - 1]$. Note that we use heapsort here since it is in-place, so we do not need any extra space.

After the above two steps, we get the reduced problem T_1 in $\mathsf{SA}[0 \ldots n_S - 1]$.

Lemma 18 *T_1 can be constructed in $O(n \log n)$ time and $O(1)$ workspace.*

Proof: In Step 1, each S-substring beginning with $T[\mathsf{SA}[i]]$ is compared with S-substring beginning with $T[\mathsf{SA}[i+1]]$. So each S-substring only participates in two comparisons. Now the argument is similar to a comparison process in a recursive level of mergesort in Lemma 17, thus it costs linear time. Obviously, Step 2 takes $O(n \log n)$ time and $O(1)$ workspace. □

5.4 Sort the S-suffixes by solving T_1 recursively

In this section, we solve $T_1 = \mathsf{SA}[0 \ldots n_S - 1]$ recursively to obtain the order of all S-suffixes in $\mathsf{SA}[n - n_S \ldots n - 1]$. For the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[n - n_S \ldots n - 1]$ as the output space for T_1. After the recursive call, SA_1 stores the suffix array of T_1. We need to restore their names back to the indices of S-suffixes in T they represented. This step can be done as follows.

1. First, we scan T from right to left. We maintain a counter sum for the number of S-type characters we have scanned so far. Initially sum is 0. If $T[i]$ is S-type, we increase sum by 1 and place $\mathsf{suf}(i)$ into $\mathsf{SA}[n_S - sum]$ (i.e., let $\mathsf{SA}[n_S - sum] \leftarrow i$). Now $\mathsf{SA}[0 \dots n_S - 1]$ stores the indices of all S-suffixes of T.

2. Then for $i \in [n - n_S, n - 1]$, let $\mathsf{SA}[i] \leftarrow \mathsf{SA}[\mathsf{SA}[i]]$.

Now, we have obtained all S-suffixes in the lexicographical order in $\mathsf{SA}[n - n_S \dots n - 1]$.

5.5 Induced sort all suffixes of T

From Section 5.4, we have obtained the sorted S-suffixes in $\mathsf{SA}[n - n_S \dots n - 1]$. Now, we sort all suffixes from these sorted S-suffixes.

Preprocessing: First, we scan T from right to left to place all indices of L-suffixes into $\mathsf{SA}[0 \dots n - n_S - 1]$. Then, we sort $\mathsf{SA}[0 \dots n - 1]$ (the sorting key of $\mathsf{SA}[i]$ is $T[\mathsf{SA}[i]]$ i.e., the first character of $\mathsf{suf}(\mathsf{SA}[i])$) using the mergesort, with the merging step implemented by the stable, in-place, linear time merging algorithm developed by Salowe and Steiger [SS87]. After this sorting step, we show some useful observations.

Observation 3 *All suffixes of T have been sorted by their first characters in SA, i.e., in their corresponding buckets.*

Observation 4 *All indices of L-suffixes beginning with the same character in SA are in increasing order, due to the stableness of the above sorting algorithm.*

Lemma 19 *All S-suffixes are already in their final position in SA.*

Proof: Before the sorting step, all sorted S-suffixes are in $\mathsf{SA}[n - n_S \dots n - 1]$ and all L-suffixes are in $\mathsf{SA}[0 \dots n - n_S - 1]$. Because the merging step is stable, the S-suffixes are behind the L-suffixes in the same bucket and hence are already in their final positions in SA from Observation 3 and Property 1. □

Induced Sorting: Now, we induce the order of all L-suffixes from the sorted S-suffixes (which are already in their final position in SA by Lemma 19) using induced sorting. Now, we extend the interior counter trick in Section 4.4 to handle the read-only general alphabets. We use five special symbols $\mathsf{B_H}$ (Head of L-suffixes), $\mathsf{B_T}$ (Tail of L-suffixes), E (Empty), $\mathsf{R_1}$ (one remaining L-suffix) and $\mathsf{R_2}$ (two remaining L-suffixes) [7]. We do the following two steps to sort all L-suffixes:

Step 1. Initializing SA: Firstly, we initialize all buckets in SA by placing some special symbols in each bucket in order to inform us the number of L-suffixes in the bucket. Concretely, we scan T from right to left. For each scanning character $T[i]$ which is L-type, if bucket $T[i]$ has not been initialized, we need to initialize bucket $T[i]$ (we will show that how to identify the bucket is initialized or not in the end of this step). Before to initialize bucket $T[i]$, we first need to obtain the value N_L, which is the number of L-suffixes in this bucket. Let l denote the head of bucket $T[i]$ in SA (i.e. l is the smallest index in SA such that $T[\mathsf{SA}[l]] = T[i]$) and r denote the tail of bucket $T[i]$ in SA (i.e. r is the largest index in SA such that $T[\mathsf{SA}[r]] = T[i]$). Furthermore, we let r_L denote the tail of L-suffixes in this bucket (i.e., r is the largest

[7] Recall that the special symbol is only used to simplify the argument. See Appendix B for the details. The only difference is that we need to use the in-place linear time merging algorithm [SS87] to place all indices of the suffixes of T in their corresponding buckets in SA (the merging key is the beginning character of the suffix), then we scan the SA once to know the bucket heads/tails for the five integers (instead of scanning the string T for the (read-only) integer alphabets).

index in SA such that $T[\mathsf{SA}[r_L]] = T[i]$ and $T[\mathsf{SA}[r]]$ is L-type). Note that $N_L = r_L - l + 1$. Hence, it suffices to compute l and r_L. The following steps compute l and r_L, respectively.

(i) We can find l by searching $T[i]$ in SA (the search key for $\mathsf{SA}[i]$ is $T[\mathsf{SA}[i]]$) using *binary search*. This uses $O(\log n)$ time from Observation 3.

(ii) For r_L, since the bucket $T[i]$ has not been initialized, $\mathsf{suf}(i)$ is the first L-suffix in its bucket being scanned. From Observation 4, $\mathsf{suf}(i)$ must be stored in $\mathsf{SA}[r_L]$ (i.e., $\mathsf{SA}[r_L] = i$) since we scan T from right to left. Hence, we can scan this bucket from l to r to find r_L which satisfies $\mathsf{SA}[r_L] = i$.

After this, we have obtained the value of N_L. Now, we initialize the bucket $T[i]$ as follows:

(1) If $N_L = 1$, we do nothing (there is only one L-suffix in this bucket, and obviously it is in the final position).

(2) If $N_L = 2$, let $\mathsf{SA}[l+1] = \mathsf{B_T}$ (recall that l is the head of bucket $T[i]$ and r is the bucket tail, i.e., $\mathsf{SA}[l \dots r]$ is the bucket $T[i]$. Moreover, r_L is the tail of L-suffixes in this bucket)

Index	l	$l+1(r_L)$	$l+2$	$\dots$	r
Type	L	L	S	$\dots$	S
SA	$\mathsf{SA}[l]$	$\underline{\mathsf{B_T}}$	$\mathsf{SA}[l+2]$	$\dots$	$\mathsf{SA}[r]$

(3) If $N_L = 3$, let $\mathsf{SA}[l+1] = \mathsf{B_H}$ and $\mathsf{SA}[l+2] = \mathsf{B_T}$.

Index	l	$l+1$	$l+2(r_L)$	$l+3$	$\dots$	r
Type	L	L	L	S	$\dots$	S
SA	$\mathsf{SA}[l]$	$\underline{\mathsf{B_H}}$	$\underline{\mathsf{B_T}}$	$\mathsf{SA}[l+3]$	$\dots$	$\mathsf{SA}[r]$

(4) If $N_L > 3$, let $\mathsf{SA}[l+1] = \mathsf{B_H}$, $\mathsf{SA}[l+2] = \mathsf{E}$ and $\mathsf{SA}[l+N_L-1] = \mathsf{B_T}$.

Index	l	$l+1$	$l+2$	$l+3$	$\dots$	$l+N_L-1(r_L)$	$l+N_L$	$\dots$	r
Type	L	L	L	L	$\dots$	L	S	$\dots$	S
SA	$\mathsf{SA}[l]$	$\underline{\mathsf{B_H}}$	$\underline{\mathsf{E}}$	$\mathsf{SA}[l+3]$	$\dots$	$\underline{\mathsf{B_T}}$	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$

Note that we can find out whether the bucket $T[i]$ is already initialized or not in $O(\log n)$ time. We do a binary search find l, then check $\mathsf{SA}[l+1]$ is $\mathsf{B_H}$, $\mathsf{B_T}$ or others. If the bucket $T[i]$ has been initialized, $\mathsf{SA}[l+1]$ is $\mathsf{B_H}$ or $\mathsf{B_T}$. Otherwise, it has not been initialized yet [8]. It is not hard to see that this initialization step uses $O(n \log n)$ time and $O(1)$ workspace.

Step 2. Sort all L-suffixes using induced sorting: We scan SA from left to right to sort all L-suffixes. The step is similar to sorting all suffixes in Section 4.4. The main difference is that we use binary search to find the bucket head. Specifically, we scan SA from left to right. For every $\mathsf{SA}[i]$, let $j = \mathsf{SA}[i] - 1$. If $T[j]$ is L-type, then place $\mathsf{suf}(j)$ into the LF-entry of its bucket, and increase the head counter by one. To specify how to place the L-suffix into the LF-entry of its bucket in SA, We only specify the case where $N_L > 3$ for the bucket. The other cases with $N_L \leq 3$ are similar and simpler. Let L_i denote the index of i-th L-suffix which needs to be placed into the LF-entry of this bucket. We distinguish the following four cases:

[8] Note that we are able to do the binary search in SA, though there are special symbols (i.e. $\mathsf{B_H}$, $\mathsf{B_T}$, E) in SA. Since the longest continuous special symbol entries in SA is 2, i.e., any three of continuous entries in SA must have at least one suffix entry (this entry represents the index of a suffix of T).

(1) If $\mathsf{SA}[l+1] = \mathsf{B_H}$ and $\mathsf{SA}[l+2] = \mathsf{E}$: The first L-suffix (i.e. L_1) need to be placed into this bucket. We let $\mathsf{SA}[l] = j$ and $\mathsf{SA}[l+2] = 1$ (use $\mathsf{SA}[l+2]$ as the counter to denote the number of L-suffixes have been placed so far). Recall that $\mathsf{suf}(j)$ is the current L-suffix we want to place.

(2) If $\mathsf{SA}[l+1] = \mathsf{B_H}$ and $\mathsf{SA}[l+2] \neq \mathsf{E}$: These L-suffixes except the first L-suffix (L_1) and the last two L-suffixes (L_{N_L-1} and L_{N_L}) need to be placed.
Let $c = \mathsf{SA}[l+2]$ (counter). If $\mathsf{SA}[l+c+2] \neq \mathsf{B_T}$, we let $\mathsf{SA}[l+c+2] = j$ and $\mathsf{SA}[l+2] = c+1$. Otherwise (this is the third to last L-suffix, i.e. L_{N_L-2}), we shift these $c-1$ L-suffixes to the left by one position (i.e., move $\mathsf{SA}[l+3 \ldots r_L - 1]$ to $\mathsf{SA}[l+2 \ldots r_L - 2]$) and let $\mathsf{SA}[r_L - 1] = j$ and $\mathsf{SA}[l+1] = \mathsf{R_2}$.

(3) If $\mathsf{SA}[l+1] = \mathsf{R_2}$: The penultimate L-suffix (i.e. L_{N_L-1}) need to be placed.
We scan this bucket to find r_L such that $\mathsf{SA}[r_L] = \mathsf{B_T}$. Then, we move $\mathsf{SA}[l+2 \ldots r_L - 1]$ to $\mathsf{SA}[l+1 \ldots r_L - 2]$. After, we let $\mathsf{SA}[r_L - 1] = j$ and $\mathsf{SA}[r_L] = \mathsf{R_1}$.

(4) Otherwise, the last L-suffix (i.e. L_{N_L}) need to be placed.
We scan this bucket to find r_L such that $\mathsf{SA}[r_L] = \mathsf{R_1}$. Then let $\mathsf{SA}[r_L] = j$.

Index	l	$l+1$	$l+2$	$l+3$	$\ldots$	$l+N_L-1(r_L)$	$l+N_L$	$\ldots$	r
Type	L	L	L	L	$\ldots$	L	S	$\ldots$	S
Case (1) :									
SA	$\mathsf{SA}[l]$	$\mathsf{B_H}$	E	$\mathsf{SA}[l+3]$	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
SA	$\mathbf{L_1}$	$\mathsf{B_H}$	**1**	$\mathsf{SA}[l+3]$	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
Case (2) :									
SA	L_1	$\mathsf{B_H}$	1	$\mathsf{SA}[l+3]$	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
SA	L_1	$\mathsf{B_H}$	**2**	$\mathbf{L_2}$	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
$\vdots$			$\vdots$						
SA	L_1	$\mathsf{B_H}$	$\mathbf{N_L - 3}$	L_2	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
SA	L_1	$\mathsf{R_2}$	$\mathbf{L_2}$	$\mathbf{L_3}$	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
Case (3) :									
SA	L_1	$\mathsf{R_2}$	L_2	L_3	$\ldots$	$\mathsf{B_T}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
SA	L_1	$\mathbf{L_2}$	$\mathbf{L_3}$	$\mathbf{L_4}$	$\ldots$	$\mathsf{R_1}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
Case (4) :									
SA	L_1	L_2	L_3	L_4	$\ldots$	$\mathsf{R_1}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$
SA	L_1	L_2	L_3	L_4	$\ldots$	$\mathbf{L_{N_L}}$	$\mathsf{SA}[l+N_L]$	$\ldots$	$\mathsf{SA}[r]$

We have the following lemmas and theorem.

Lemma 20 *When we scan* $\mathsf{SA}[i]$ *in the induced sorting step, whether* $T[\mathsf{SA}[i]-1]$ *is L-type or S-type can be identified in* $O(1)$ *time. The only exception is when* $\mathsf{suf}(\mathsf{SA}[i]-1)$ *is the last L-suffix which needs to be inserted into the bucket* $T[\mathsf{SA}[i]-1]$. *This special case needs* $O(N_L)$ *time, recall that* N_L *denote the number of L-suffixes in its bucket (i.e.* $N_L = r_L - l + 1$*).*

Proof: Let $j = \mathsf{SA}[i] - 1$. First if $T[j] \neq T[j+1]$, by definition it is trivial. Otherwise, $T[j] = T[j+1]$. From Lemma 14, we only need to know whether all L-suffixes in the bucket $T[j]$ (i.e., bucket $T[\mathsf{SA}[i]]$) have already been sorted or not. In our algorithm, we use the extended interior counter trick which maintains the counters of the buckets. So we can identify whether all L-suffixes in the bucket $T[j]$ have already been sorted or not immediately except when $\mathsf{suf}(j)$ is the last L-suffixes which needs to be placed into the bucket

$T[j]$ (corresponding to case (4)). However, we can scan this bucket from left to right to identify whether the special symbol R_1 exists or not. If exists, which means there is one L-suffix remained, this must be the $\mathsf{suf}(j)$. Otherwise, all L-suffixes in the bucket $T[j]$ have already been sorted. This scanning operation takes $O(N_L)$ time. □

Lemma 21 *All the suffixes can be sorted correctly from the sorted S-suffixes in $O(n \log n)$ time.*

Proof: All S-suffixes have been sorted correctly according to Lemma 19. All L-suffixes can be sorted correctly from the sorted S-suffixes using induced sorting steps according to Lemma 1. For time, in the scanning process, we use $O(\log n)$ time binary search to find the head of its bucket for each L-suffix (identified by Lemma 20), and use the extended interior counter trick to find the LF-entry in constant time except for the last two L-suffixes of the bucket. We need to scan all L-suffixes in this bucket in order to find the final position (corresponding to case (4) in Step 2) and shift these L-suffixes by one position (corresponding to case (3) in Step 2). This only takes $O(n)$ time overall since each bucket is scanned at most twice. To sum up, this sorting step takes $O(n \log n)$ time. □

Theorem 6 *Our Algorithm takes $O(n \log n)$ time and $O(1)$ workspace to compute the suffix array for string T over general alphabets, where T is read-only and only comparisons between characters are allowed.*

Proof: All steps in our algorithm takes $O(n \log n)$ time. Besides, the size of recursive problem T_1 is no larger than half of $|T|$. We have $T(n) = T(n/2) + O(n \log n)$, thus $T(n) = O(n \log n + \frac{n}{2} \log \frac{n}{2} + \cdots) = O(n \log n)$. For workspace, every step uses $O(1)$ workspace, and in the recursive subproblem we can also reuse the $O(1)$ workspace. Moreover, at the same recursive level, the different steps can reuse the $O(1)$ workspace too. Also note that we do not need to store n_i at each recursive level i during the recursion. □

6 Experiments

In this section, we report the experimental results for our optimal linear time in-place algorithm for suffix sorting over integer alphabets (the algorithm was described in Section 3). The experiments were conducted on an Intel(R) Core(TM) i5-3470 Processor (3.2GHz, 4 cores) and 4GB RAM. The operating system was Ubuntu 14.04.3LTS x86_64. The compiler was gcc (version 4.8.4) executed with the "-W -Wall -fomit-frame-pointer -DNDEBUG -O3" options. See Table 3 for the experimental results.

The datasets were generated by choosing a random number in Σ independently for each position of T. We test our algorithm for $|\Sigma| = 100$, $|\Sigma| = 1000$ and $|\Sigma| = n$ (n is the length of the integer string). Each integer occupies 4 Bytes. The maximum input size we can handle is 1600MB as the main memory is only 4GB and we also need 1600MB for the output.

For the running time, we used the mean over three runs (measured by the clock() function). Note that we only record the time interval for sorting SA, excluding the time for reading the input string T into the main memory and writing the output SA to disk. The total space is the heap peak measured by the *memusage* command same as Nong [Nong13]. The workspace is the total space subtracting the space of T and SA. The workspace of our algorithm is invariably 8 Bytes. We can also see that the running time grows approximately linearly with the size of the input. The overall running time is quite competitive: the algorithm can sort 20MB input data in about 1.5 second and 1.6GB data in less than 4 minute. The size of the alphabets does not significantly affect the running time.

Table 3: Experimental results of our optimal linear time in-place suffix sorting algorithm

Input	Time (Seconds)	Speed (MB/s)	Workspace (Bytes)	Total space (Bytes)	Space of T and SA (Bytes)
20MB-100	1.120	17.857	8	41,943,048	41,943,040
20MB-1k	1.137	17.590	8	41,943,048	41,943,040
20MB	1.557	12.845	8	41,943,048	41,943,040
100MB-100	8.456	11.826	8	209,715,208	209,715,200
100MB-1k	8.745	11.435	8	209,715,208	209,715,200
100MB	8.476	11.798	8	209,715,208	209,715,200
1000MB-100	116.156	8.609	8	2,097,152,008	2,097,152,000
1000MB-1k	127.473	7.845	8	2,097,152,008	2,097,152,000
1000MB	142.648	7.010	8	2,097,152,008	2,097,152,000
1600MB-100	192.387	8.317	8	3,355,443,208	3,355,443,200
1600MB-1k	210.308	7.607	8	3,355,443,208	3,355,443,200
1600MB	234.348	6.827	8	3,355,443,208	3,355,443,200

Note that $n \times 4$ is the space usage (in Bytes) of the input string. Input name 20MB-100 indicates that the input size is 20MB and $|\Sigma| = 100$, and name 20MB indicates that the input size is 20MB and $|\Sigma| = n = 5,242,880$.

7 Conclusion

In this paper, we present three in-place algorithms for suffix sorting over (read-only) integer alphabets and read-only general alphabets. All of them are optimal both in time and space.

Concretely, we provide the first optimal linear time in-place suffix sorting algorithm for (read-only) integer alphabets. Our algorithms solve the open problem posed by Franceschini and Muthukrishnan in ICALP 2007 [FM07]. Besides, our algorithm is easy to implement and competitive in practice. For the read-only general alphabets, we provide simple sorting steps to obtain an optimal in-place $O(n \log n)$ time suffix sorting algorithm, which recovers the result obtained by Franceschini and Muthukrishnan [FM07] which was an open problem posed by Manzini and Ferragina [MF02]. Our techniques may be used in other scenarios, e.g., one can obtain an in-place sorting algorithm by using the interior counter trick and pointer data structure to remove the workspace needed by the counting array in the classical counting sort algorithm.

There is a surge of interests in developing external memory algorithms for suffix sorting in recent years [FGM12, NCHW15]. Many such algorithms are extensions of existing lightweight internal memory algorithms. It would be interesting to investigate the external memory setting and see whether our tricks and data structures are applicable in this setting. We also plan to consider other string processing problems that are tightly connected with suffix array, such as compressed suffix arrays, longest common prefixes, Burrows-Wheeler transform and Lempel-Ziv factorization.

Acknowledgments

We would like to thank Ge Nong for his help in our experiments, and Gonzalo Navarro for helpful suggestions.

References

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Algorithms in Bioinformatics*, pages 449–463. Springer, 2002.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[BB05] Dror Baron and Yoram Bresler. Antisequential suffix sorting for bwt-based data compression. *Computers, IEEE Transactions on*, 54(4):385–397, 2005.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Combinatorial Pattern Matching (CPM)*, pages 55–69. Springer, 2003.

[BW94] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical Report 124*, 1994.

[Cla96] David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[CLRS01] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press Cambridge, 2001.

[CMR14] Timothy M Chan, J Ian Munro, and Venkatesh Raman. Selection and sorting in the restore model. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 995–1004. Society for Industrial and Applied Mathematics, 2014.

[DPT12] Jasbir Dhaliwal, Simon J Puglisi, and Andrew Turpin. Trends in suffix sorting: a survey of low memory algorithms. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, pages 91–98. Australian Computer Society, Inc., 2012.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE, 1997.

[FGM12] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000.

[FM07] G Franceschini and S Muthukrishnan. In-place suffix sorting. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 533–545. Springer-Verlag, 2007.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[HCVN14] Hongwei Huo, Longgang Chen, Jeffrey Scott Vitter, and Yakov Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *Data Compression Conference (DCC)*, pages 292–301. IEEE, 2014.

[HSL+16] Hongwei Huo, Zhigang Sun, Shuangjiang Li, Jeffrey Scott Vitter, Xinkun Wang, Qiang Yu, and Jun Huan. Cs2a: a compressed suffix array-based method for short read alignment. In *Data Compression Conference (DCC)*, pages 271–278. IEEE, 2016.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 251–260. IEEE, 2003.

[IT99] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*, pages 81–88. IEEE, 1999.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE, 1989.

[KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching (CPM)*, pages 200–210. Springer, 2003.

[KJP04] Dong K Kim, Junha Jo, and Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Experimental and Efficient Algorithms*, pages 301–314. Springer, 2004.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 943–955. Springer, 2003.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching (CPM)*, pages 186–199. Springer, 2003.

[LS07] N Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[McC76] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[MF02] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. In *European Symposium on Algorithms (ESA)*, pages 698–710. Springer, 2002.

[MM90] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

[MP06] Michael A Maniscalco and Simon J Puglisi. Faster lightweight suffix array construction. In *Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, pages 16–29. Citeseer, 2006.

[MP08] Michael A Maniscalco and Simon J Puglisi. An efficient, versatile approach to suffix sorting. *Journal of Experimental Algorithmics (JEA)*, 12:1–2, 2008.

[NCHW15] Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. Induced sorting suffixes in external memory. *ACM Transactions on Information Systems (TOIS)*, 33(3):12, 2015.

[Nong13] Ge Nong. Practical linear-time O (1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems (TOIS)*, 31(3):15, 2013.

[NP12] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, pages 295–306, 2012.

[NZ07] Ge Nong and Sen Zhang. Optimal lightweight construction of suffix arrays for constant alphabets. In *Algorithms and Data Structures*, pages 613–624. Springer, 2007.

[NZC09a] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference (DCC)*, pages 193–202. IEEE, 2009.

[NZC09b] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In *Combinatorial Pattern Matching (CPM)*, pages 54–67. Springer, 2009.

[NZC11] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *Computers, IEEE Transactions on*, 60(10):1471–1484, 2011.

[PST07] Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4, 2007.

[Sad98] Kunihiko Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *Data Compression Conference (DCC)*, pages 129–138. IEEE, 1998.

[SS87] Jeffrey Salowe and William Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, 1987.

[SS07] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

A Running Examples

A.1 Induced sorting all L-suffixes from the sorted S-suffixes

In this appendix, we give a running example for the standard induced sorting step which needs the bucket array and type array explicitly. Assume that all indices of the sorted S-suffixes are already in their correct positions in SA (i.e., in the tail of their corresponding buckets in SA). We scan SA from left to right (i.e., from SA[0] to SA[$n-1$]). When we scan SA[i], let $j = \mathsf{SA}[i]-1$. If $T[j]$ is L-type, i.e., $\mathsf{suf}(j)$ is an L-suffix (indicated by the type array), we place the index of $\mathsf{suf}(j)$ (i.e. j) into the LF-entry of bucket $T[j]$, and then let the LF-pointer of this bucket $T[j]$ point to the next entry. The LF-pointers are maintained by the bucket array. If $\mathsf{suf}(j)$ is an S-suffix, we do nothing (since all S-suffixes are already sorted in the correct positions).

The idea of induced sorting is that the lexicographical order between $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ are decided by the order of $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ if $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ are in the same bucket (i.e., $T[i] = T[j]$). Considering two L-suffixes $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ in the same bucket, we have $\mathsf{suf}(i+1) < \mathsf{suf}(i)$ and $\mathsf{suf}(j+1) < \mathsf{suf}(j)$ by the definition of L-suffix. Since we scan SA from left to right, $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ must appear earlier than $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$. Hence the correctness of induced sorting is not hard to prove by induction. Note that the order of $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ with $T[i] \neq T[j]$ is already correct, since we always place the L-suffixes in their corresponding buckets.

Example: We use the following running example to demonstrate the induced sorting step. Consider a string $T[0\ldots12]$ = "2113311331210" (the integer alphabets).

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S

$T[2]$ is S-type since $T[2] = 1 < T[3] = 3$. The S-substrings are $\{11, 1331, 11, 1331, 1210, 0\}$. The S-suffixes are $\{\mathsf{suf}(1), \mathsf{suf}(2), \mathsf{suf}(5), \mathsf{suf}(6), \mathsf{suf}(9), \mathsf{suf}(12)\}$.

Now, we show the induced sorting step in our running example. Suppose all indices of the sorted S-suffixes (i.e., $1, 2, 5, 6, 9, 12$) are already stored in the tail of their corresponding buckets in SA: (E denotes an Empty entry.)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	(**12**)	(E	**1**	**5**	**9**	**2**	**6**)	(E	E)	(E	E	E	E)
Bucket	(0)	(1	1	1	1	1	1)	(2	2)	(3	3	3	3)

(The entries between a pair of parentheses denote a bucket in SA which are these suffixes that start with the same character. The heads of bucket $0, 1, 2, 3$ are $0, 1, 7, 9$, respectively.)

The scanning process is as follows. An arrow on top of a number indicates that it is the current entry we

are scanning.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	($\overrightarrow{\mathbf{12}}$)	(**11**)	1	5	9	2	6)	(E	E)	(E	E	E	E)
SA	(12)	($\overrightarrow{\mathbf{11}}$)	1	5	9	2	6)	(**10**)	E)	(E	E	E	E)
SA	(12)	(11	$\overrightarrow{\mathbf{1}}$	5	9	2	6)	(10	**0**)	(E	E	E	E)
SA	(12)	(11	1	$\overrightarrow{\mathbf{5}}$	9	2	6)	(10	0)	(**4**	E	E	E)
SA	(12)	(11	1	5	$\overrightarrow{\mathbf{9}}$	2	6)	(10	0)	(4	**8**	E	E)
SA	(12)	(11	1	5	9	2	6)	(10	0)	($\overrightarrow{\mathbf{4}}$	8	**3**	E)
SA	(12)	(11	1	5	9	2	6)	(10	0)	(4	$\overrightarrow{\mathbf{8}}$	3	**7**)

We first scan $\mathsf{SA}[0] = 12$. We place 11 (since $T[11]$ is L-type) to the LF-entry of bucket 1 (i.e., $\mathsf{SA}[1]$), note that the LF-pointer of bucket 1 initially points to $\mathsf{SA}[1]$ (head of bucket 1). Next, we scan $\mathsf{SA}[1] = 11$, and we place 10 ($T[10]$ is also L-type) to the LF-entry of its bucket (i.e., bucket 2), and so on. □

A.2 Induced sorting all L-suffixes from the sorted LMS-suffixes

In this appendix, we show the induce sorting step which sorting the L-suffixes from LMS-suffixes in our example. Note that the empty entries can be ignored, and all L-suffixes can still be sorted correctly.

Example: Suppose all LMS-suffixes (i.e., $1, 5, 9, 12$) are already sorted in the tail of their corresponding bucket in SA: (E denotes an Empty entry.)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
T	2	1	1	3	3	1	1	3	3	1	2	1	0
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	(**12**)	(E	E	E	**1**	**5**	**9**)	(E	E)	(E	E	E	E)
Bucket	(0)	(1	1	1	1	1	1)	(2	2)	(3	3	3	3)

The scanning process is as follows. An arrow on top of a number indicates that it is the current entry we are scanning. When we are scanning an empty entry in SA, we ignore this entry (i.e., do nothing).

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Type	L	S	S	L	L	S	S	L	L	S	L	L	S
SA	($\overrightarrow{\mathbf{12}}$)	(**11**)	E	E	1	5	9)	(E	E)	(E	E	E	E)
SA	(12)	($\overrightarrow{\mathbf{11}}$)	E	E	1	5	9)	(**10**)	E)	(E	E	E	E)
SA	(12)	(11	E	E	$\overrightarrow{\mathbf{1}}$	5	9)	(10	**0**)	(E	E	E	E)
SA	(12)	(11	E	E	1	$\overrightarrow{\mathbf{5}}$	9)	(10	0)	(**4**	E	E	E)
SA	(12)	(11	E	E	1	5	$\overrightarrow{\mathbf{9}}$)	(10	0)	(4	**8**	E	E)
SA	(12)	(11	E	E	1	5	9)	(10	0)	($\overrightarrow{\mathbf{4}}$	8	**3**	E)
SA	(12)	(11	E	E	1	5	9)	(10	0)	(4	$\overrightarrow{\mathbf{8}}$	3	**7**)

□

B Handling Special Symbols

We use at most five special symbols (e.g. U (denoted as Unique), E (denoted as Empty)) in this paper. The special symbols are only used to simplify the argument, i.e., we do not need any additional assumption. Now we describe how to handle these special symbols. Note that we introduce these special symbols and use them in our interior counter trick (see Section 3.3, 3.7 and 4.4). Recall that, if one uses the bucket array (which needs extra $\max\{|\Sigma|, n/2\}$ words workspace) without using the interior counter trick, then the suffix array SA would only contain the indices of T (i.e., $\{0, 1, \ldots, n-1\}$) (see the preliminary section). Now the key point is to distinguish whether an entry in SA is one of these five integers (chosen to replace the special symbols) or just an index as before.

First, consider the simpler case where $n < 2^{\lceil \log n \rceil} - 5$. We can simply use integers $\{n, n+1, \ldots, n+4\}$ as these five special symbols since they are different from all indices (identifiable) and each of them can be stored in one entry of SA same as an index of T.

Otherwise, we use any five integers (belong to $[0, n-1]$) as these five special symbols. Without loss of generality, we assume that these five integers are $\{n-5, n-4, \ldots, n-1\}$. Then we use ten extra variables (as the previous bucket array) to indicate the head/tail of the five buckets (which the five suffixes $\{\mathsf{suf}(n-5), \ldots, \mathsf{suf}(n-1)\}$ belong to) in SA and their LF/RF-entries. Thus we do not need the interior counter trick for these five buckets. Note that we can obtain the head/tail of these five buckets by scanning T once to count how many characters are smaller/larger than these five characters $\{T[n-5], \ldots, T[n-1]\}$, respectively. To identify an entry of SA, if the entry belongs to one of these five buckets, it is just an index. Otherwise, we check its value: if it equals to one of $\{n-5, n-4, \ldots, n-1\}$, then it is a special symbol since none of these five integers belongs to this bucket.

There is one more subtlety. In our interior counter trick, there is a counter for each bucket to count how many suffixes have been placed into this bucket. Thus there are three types of entries in SA: 1) indices of T; 2) these five integers (as special symbols); 3) counters for the buckets. We want to point out that the position of each counters is alway fixed to be adjacent to the special symbol. Besides, the value of any counter is less than its bucket size minus 2 since the counter is deleted when we place the last two indices into its bucket (see e.g. the figure in Section 4.4). Moreover, it is not hard to verify that the counter can only conflict with the special symbol E (which happens when we first insert an index to a bucket). Thus we choose to use $n-1$ to denote the special symbol E. There is no conflict since the counter is always less than $n-2$.

C Restoring the original string T

In this Appendix, we show that we can restore the string T in our first algorithm which is designed for the string T over the integer alphabets $\{1, 2, \ldots, \Sigma\}$. First, we can see that in the termination of our algorithm. Suffix array SA contains the indices of all suffixes of T which are in lexicographical order. Note that if we do not modify T, we will have the following observation.

Observation 5 *For each suffix* $\mathsf{suf}(\mathsf{SA}[i])$ *in* SA*, let* b_i *denote its bucket character (i.e., the first common character), then* $T[\mathsf{SA}[i]] = b_i$.

The key point to recover T is that we need to maintain the equal relationship of the characters of T. So if we modify T to T' under this condition such that $T'[i] = T'[j]$ (or $T'[i] \neq T'[j]$, resp.) if and only if $T[i] = T[j]$ (or $T[i] \neq T[j]$, resp.). Then, we can recover T from SA and T' using linear time (scan SA once) and $O(1)$ workspace from above Observation 5. Now, we need to modify the first renaming step in our algorithm to rename each character $T[i]$ to be its bucket tail (note that this modification maintains the equal

relationship). This change only lead the details in the later induced sorting step changed. In the induced sorting step, since we let all $T[i]$ points to its bucket tail, so the induced sort LMS-suffixes or S-suffixes will be the same as before. The only thing we need to explain in the induced sorting step is that we induced sort L-suffixes from the sorted LMS-suffixes since there are not exist pointers which point to the bucket head (see Step 1 of Section 3.7 which sort all L-suffixes from the sorted LMS-suffixes using induced sorting). However, we can fix this step using our interior counter trick which we widely used in this paper. Now, we describe the details. We consider the buckets in SA into two types, one type does not contain LMS-suffixes, the other contains LMS-suffixes. These two types are easy to identify since the LMS-suffixes have already been sorted in the tail of their corresponding buckets in SA.

Type 1. The buckets do not contain LMS-suffixes: In this type, we initialize the bucket in the following steps. Scanning T from right to left. For every $T[i]$ which is L-type and its bucket is this type, do the following:

(1) If $\mathsf{SA}[T[i]] = \mathsf{Empty}$, let $\mathsf{SA}[T[i]] = \mathsf{Unique1}$ (unique L-type character in this bucket).

(2) If $\mathsf{SA}[T[i]] = \mathsf{Unique1}$, let $\mathsf{SA}[T[i]] = \mathsf{Multi1}$ and $\mathsf{SA}[T[i]-1] = 2$ (number of L-type characters in this bucket is 2).

(3) If $\mathsf{SA}[T[i]] = \mathsf{Multi1}$, increase $\mathsf{SA}[T[i]-1]$ by one. ($\mathsf{SA}[T[i]-1]$ denote the number of L-type characters in this bucket)

After this initialization, the head of this type bucket can be indicated by $\mathsf{SA}[t]$ and $\mathsf{SA}[t-1]$, where t is its bucket tail.

Type 2. The buckets contain LMS-suffixes: In this type, we initialize the bucket in the following steps. Scanning T from right to left. For every $T[i]$ which is L-type and its bucket is this type, do the following:

(1) If $\mathsf{SA}[T[i]]$ is an index, shift these LMS-suffixes (which are sorted in this bucket tail) to left by one position and let $\mathsf{SA}[T[i]] = \mathsf{Unique2}$ (unique L-type character in this bucket).

(2) If $\mathsf{SA}[T[i]] = \mathsf{Unique2}$, shift these LMS-suffixes (which have been shifted by one position) to left by one position again and let $\mathsf{SA}[T[i]-1] = 2$ (number of L-type characters in this bucket is 2).

(3) If $\mathsf{SA}[T[i]] = \mathsf{Multi2}$, increase $\mathsf{SA}[T[i]-1]$ by one. ($\mathsf{SA}[T[i]-1]$ denote the number of L-type characters in this bucket)

After this initialization, the head of this type bucket can be indicated by $\mathsf{SA}[t]$ and $\mathsf{SA}[t-1]$ too, where t is its bucket tail.

Now, all L-suffixes can be sorted using induced sort like before, but their indices are not in their final positions in SA. We need scan T once more from right to left to compute the number of suffixes in each bucket, then shift these sorted L-suffixes to their bucket head (it is not hard to see that this shift step can be done in linear time). Now, all L-suffixes are placed in their final positions in SA, then using induced sort as before we can sort all S-suffixes, so all suffixes have been sorted. In conclusion, we have the following lemma.

Lemma 22 *The original string T can be restored using linear time and $O(1)$ workspace.*