Optimal In-Place Suffix Sorting

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

Jian Li
Tsinghua University
lapordge@gmail.com

Hongwei Huo
Xidian Univeristy
hwhuo@mail.xidian.edu.cn

Abstract

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 in the last 20 years. We obtain the suffix array construction algorithms that are optimal both in time and space for both integer alphabets and general alphabets. Concretely, we make the following contributions:

1. For integer alphabets, we obtain the first 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 algorithm is easy to implement. Our C implementation of the algorithm requires only 8 Bytes of workspace.
2. We strengthen the first result by providing the first linear time in-place algorithm for read-only integer alphabets (i.e., we cannot modify the input string T). This settles the open problem posed by Franceschini and Muthukrishnan in ICALP 2007 [FM07].
3. For read-only general alphabets (i.e., only comparisons are allowed), we present an in-place $O(n \log n)$ time algorithm, recovering the result obtained by Franceschini and Muthukrishnan [FM07].

1 Introduction

Suffix arrays were introduced by Manber and Myers [MM90, MM93] 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]. 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.

Given string $T = T[0 \dots n-1]$, 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. The suffix array SA for string T is the sorted array of all the suffixes of the string T, according to their lexicographical order, i.e., SA stores a permutation of $0, \dots, n-1$, such that $T[\mathsf{SA}[i] \dots n-1] < T[\mathsf{SA}[j] \dots n-1]$ for $i < j$. Suffix arrays have been studied extensively over the last 20 years (see e.g., [MM93, KS03, KSB06, KA05, NZC09a, NZC11, Non13]). We refer the readers to the surveys [PST07, DPT12] for many suffix sorting algorithms.

In 1990, Manber and Myers [MM90, MM93] 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, and significant efforts have been made in developing *lightweight* (in terms of space usage) suffix sorting algorithms for the last decade [MF02, BK03, KA03, HSS03, HSS09, MP06, NZ07, NZC11, Non13]. 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 Related Work and Our Contribution

Before we discuss in details of the previous and our algorithms, we need some terminologies. We measure the space usage in the unit of *words*. Each word can store $O(\log W)$ bits, where $n < W$. One standard arithmetic or bitwise boolean operation on word-sized operands costs $O(1)$ time. 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. As usual, we can use the space of SA when construct SA.

We consider the following three popular settings.

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.

2. Read-only integer alphabets: Each $T[i] \in [1, \Sigma]$ where $|\Sigma| \leq n$ and $T[i]$ is stored in a word or $\log |\Sigma|$ bits. In addition, the input string T is read-only.

3. Read-only general alphabets: We can only read the input string T and compare characters. 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, as it is a generalization of ordinary sorting (if all $T[i]$s are distinct).

There are many suffix sorting algorithms over these alphabets. See Tables 1 and 2 for an overview.

1.1.1 Integer Alphabets

We first consider integer alphabets. We allow the algorithm to temporarily modify T. However, the original string T must be restored upon the termination of our algorithm. Chan et al. [CMR14] denote this situation as *restore model* in their paper.

We list many previous results and our new result in Table 1 [1]. Earlier algorithms that require more than $O(n)$ workspace (See Table 1 for many of them) do not need to modify the input as they can create a new array with n words.

Nong et al. [NZC09b, NZC11] obtained the first nearly linear time algorithm that uses sublinear workspace. They modified the input T in their algorithm. Recently, Nong [Non13] obtained a linear time in-place algorithm if $|\Sigma| = O(1)$. In fact, the algorithm needs $|\Sigma|$ words workspace and it does not need to modify the input T. Note that in the worst case $|\Sigma|$ can be as large as $O(n)$. We improve their results as follows.

Theorem 1 *There is an in-place linear time algorithm for suffix sorting over integer alphabets.*

Our algorithm is based on the induced sorting framework developed in [KA03] (which are also used in several previous algorithms [KA05, FM07, PST07, NZC09a, NZC09b, NZC11, Non13]). We develop a few

[1] Some previous works state their space usages in terms of bits. We convert into words.

Table 1: Time and workspace of suffix array construction algorithms for integer alphabets

Time	Workspace (words)	Algorithms
$O(n^2 \log n)$	$cn + O(1)$ $c < 1$	[MF04, 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, MM93, LS99, 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, HSS09]
$O(n \log \|\Sigma\|)$	$\|\Sigma\| + O(1)$	[NZ07]
$O(n)$	$O(n)$	[KSPP03, KS03, KA03, KA05, 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]
$O(n)$	$\|\Sigma\| + O(1)$	[Non13]
$O(n)$	$O(1)$	this paper

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

elementary, yet effective tricks to further reduce the space usage to constant. This algorithm and the new tricks are also useful for 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.

1.1.2 Read-only Integer Alphabets

In this section, we consider the harder case where the input string T is read-only. There are many algorithms for this case. See Table 1 for an overview. Again, the alphabets is $[1, \Sigma]$ where $|\Sigma| \leq n$. In 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 integer alphabets (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 [Non13], which uses $|\Sigma|$ words workspace, as we just mentioned. In this paper, we settle down this open problem.

Theorem 2 *There is an in-place linear time algorithm for suffix sorting over integer alphabets, even if the input string T is read-only.*

1.1.3 Read-only General Alphabets

Now, we consider the general case where the string T is over an arbitrary alphabet. The only operations allowed on the characters of string T (read-only) are comparisons. See table 2 for an overview of results. In 2002, Manzini and Ferragina [MF02] posed an open problem, which asked whether there exists an

[2] Nong et al. [NZC09b, NZC11] assume 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 $O(\log n)$ bits. It is easy to verify 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: Read-only general alphabets

Time	Workspace(words)	Algorithms
$O(n \log n)$	$O(n)$	[MM90, MM93, LS99, 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

$O(n \log n)$ time algorithm using $o(n)$ workspace. In ICALP 2007, Franceschini and Muthukrishnan [FM07] obtained the first in-place algorithm that runs in optimal $O(n \log n)$ time. Their conference paper is complicated and densely-argued. We also give an algorithm which achieves the same result. In addition, we do not make any bit operations while they use bit operations in many places in their algorithm.

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.*

1.2 Our techniques

Almost all of the previous algorithms [KA03,KA05,KSB06,PST07,NZC09a,NZC11] require extra arrays to sort the suffixes of T, e.g., *bucket array* (which needs $|\Sigma|$ words), *type array* (needs $n/\log n$ words) and/or other arrays which need $O(n)$ words. Currently the best result is provided by Nong [Non13]. However, he still requires *bucket array* in the first level, which needs $|\Sigma|$ words. Note that the workspace needed in the first level is the most difficult part to be removed because it seems no extra space we can use since SA needs to store the final order of all suffixes now. The deeper recursive level is, the more extra space we can reuse in SA, because the size of the recursive sub-problems in the deeper level are less than half of the problems in current level for many algorithms. Therefore the highest bits in SA can be reused as extra space in the recursive level. So the main difficulty is to remove the workspace which are needed by bucket array and type array in the first level. In addition, we note that our in-place algorithms do not need the highest bits in SA to store the type information.

Now, we describe our algorithms that overcome these difficulties as follows:

1. (Section 3) integer alphabets:
   In a high level, the framework of our algorithm is based on induced sorting [KA03] which will be introduced in Section 2. However, the actual implementation is different, as we must carefully avoid to store the information explicitly (e.g. types and pointers) to obtain an in-place algorithm.

   We give some properties and observations between string T and suffix array SA which are useful to obtain the type information. Furthermore, we provide an *interior counter trick* which can represent the dynamic pointer information in SA, and we temporarily occupy the space of T (note that in this case T is not read-only) to represent the bucket information. Combining these methods, we can overcome these difficulties (i.e., remove the space needed by *bucket array* and *type array*) to obtain an in-place linear time algorithm for integer alphabets.

2. (Section 4) read-only general alphabets:
   We provide simple sorting steps and extend our interior counter trick to obtain the optimal in-place algorithm for this case.

3. (Section 5) read-only integer alphabets:
   Our in-place algorithm for this case is a bit complex than the above algorithms since this is the hardest case. In general, we provide a *pointer data structure* which can implicitly represent the bucket information and combine these methods that we used in the general alphabets case to obtain the optimal in-place algorithm.

Organization : The remaining of the paper is organized as follows. Section 2 is a preliminary section where we introduce some useful notions and tools. In Section 3, 4 and 5, we describe the framework of our in-place suffix sorting algorithms and detail the steps of the algorithms for integer alphabets, read-only general alphabets and read-only integer alphabets respectively. Next, we report the experimental results of our algorithm for integer alphabets in Section 6. Finally, we give a conclusion in Section 7.

2 Preliminaries

We consider a string $T = T[0 \ldots n-1]$ with n characters over the alphabets Σ, where $|\Sigma| \le n$. We use $T[i \ldots j]$ denote the substring $T[i]T[i+1]\ldots T[j]$ 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 $T[n-1] = 0$ [3]. Let $\mathsf{suf}(i)$ denote the suffix $T[i \ldots n-1]$ of T. Any two suffixes in T must be different since their length 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.

The suffix array of a string T, which we write as SA, is the array which contains the indices of the suffixes of T, sorted in lexicographical order. Formally, SA is an array $\mathsf{SA}[0 \ldots n-1]$ that contains the permutation of the integer of $[0 \ldots n-1]$, such that $\mathsf{suf}(\mathsf{SA}[i]) < \mathsf{suf}(\mathsf{SA}[j])$ for all $i < j$.

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 S-suffix. Equivalently, we can see that $\mathsf{suf}(i)$ 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 $T[0]$). Given the type of a suffix, we further define the type of a character $T[i]$ is *S-type* (or *L-type* resp.) if $\mathsf{suf}(i)$ is S-suffix (or L-suffix resp.). A substring $T[i \ldots j]$ is called an *S-substring* if (1) $i = j = n-1$; or (2) $i < j$, both $T[i]$ and $T[j]$ are S-type, and there is no other S-type characters between them. We can define *L-substring* symmetrically.

Example: We use the following running example throughout the paper. Consider string $T[0 \ldots 12] =$ "2113311331210" (the integer alphabet).

$index$	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 $\{11, 1331, 11, 1331, 1210, 0\}$. □

Obviously, the indices of all suffixes, which start with the same character, must appear consecutively in SA. We denote a subarray in SA for these suffixes with the same first 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

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

bucket. For example, if the bucket character is $T[i]$, we refer to it 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.

The *induced sorting* technique, developed by Ko and Aluru [KA03], is responsible for many recent advances of suffix sorting algorithms [KA05, PST07, FM07, NZC09a, NZC09b, NZC11, Non13], 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.

Inducing the order of L-suffixes from S-suffixes : Assume that all S-suffixes are already sorted and in their correct positions in SA. We scan SA from left to right (i.e., from SA$[0]$ to SA$[n-1]$). We maintain an *LF-pointer* (leftmost free pointer) for each bucket which points to the *LF-entry* (leftmost free entry) of the bucket. The LF-pointers initially point to the head of their corresponding buckets. When we scan SA$[i]$, let $j = \mathsf{SA}[i] - 1$. If $T[j]$ is L-type (i.e., $\mathsf{suf}(j)$ is L-suffix), we place $\mathsf{suf}(j)$ into the LF-entry in bucket $T[j]$, and then let the LF-pointer of this bucket (i.e., bucket $T[j]$) point to the next entry. If $T[j]$ is S-type, we do nothing. Sort all S-suffixes from sorted L-suffixes is completely symmetrical: we scan SA from right to left, maintain 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.*

Example: Now, we show the induce sorting step in our running example.

Suppose all S-suffixes (i.e., $1, 2, 5, 6, 9, 12$) are already sorted 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	($\underline{\mathbf{12}}$)	(E	$\underline{\mathbf{1}}$	$\underline{\mathbf{5}}$	$\underline{\mathbf{9}}$	$\underline{\mathbf{2}}$	$\underline{\mathbf{6}}$)	(E	E)	(E	E	E	E)
bucket	(0)	(1	1	1	1	1	1)	(2	2)	(3	3	3	3)

(Note SA$[0] = 12$ since $\mathsf{suf}(12) =$ "0" is the smallest suffix. 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	($\overrightarrow{\mathbf{12}}$)	($\underline{\mathbf{11}}$	1	5	9	2	6)	(E	E)	(E	E	E	E)
SA	(12)	($\overrightarrow{\mathbf{11}}$	1	5	9	2	6)	($\underline{\mathbf{10}}$	E)	(E	E	E	E)
SA	(12)	(11	$\overrightarrow{\mathbf{1}}$	5	9	2	6)	(10	$\underline{\mathbf{0}}$)	(E	E	E	E)
SA	(12)	(11	1	$\overrightarrow{\mathbf{5}}$	9	2	6)	(10	0)	($\underline{\mathbf{4}}$	E	E	E)
SA	(12)	(11	1	5	$\overrightarrow{\mathbf{9}}$	2	6)	(10	0)	(4	$\underline{\mathbf{8}}$	E	E)
SA	(12)	(11	1	5	9	2	6)	(10	0)	($\overrightarrow{\mathbf{4}}$	8	$\underline{\mathbf{3}}$	E)
SA	(12)	(11	1	5	9	2	6)	(10	0)	(4	$\overrightarrow{\mathbf{8}}$	3	$\underline{\mathbf{7}}$)

We first scan $\mathsf{SA}[0] = 12$. Now, $j = 11$ and $T[11]$ is L-type. We place 11 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. □

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., $T[i] = T[j]$). 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. 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.

Inducing the order of L-suffixes from 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 all 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.

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.

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 (1) $i = j = n-1$; or (2) $i < j$, both $T[i]$ and $T[j]$ are LMS-characters, and there is no other LMS-characters between them (similar to above S-substring). Similarly, we can define *LML-suffix* (Leftmost L-type) and *LML-substring*. 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.

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.

Nong et al. [NZC09a] showed that we can use the same induced sorting step to sort all LMS-substrings from 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 an 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 bucket 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 (the same as induce 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.

3 Suffix Sorting for Integer Alphabets

3.1 Framework

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

1. (Section 3.2) Rename T.
2. (Section 3.3) Sort all LMS-characters of T.
3. (Section 3.4) 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 LMS-suffixes by solving T_1 recursively.
6. (Section 3.7) Sort all suffixes from the sorted LMS-suffixes (T_1).

In a high level, the framework is similar to several other previous algorithms based on induced sorting [KA03, KA05, FM07, PST07, NZC09a, NZC09b, NZC11, Non13], and in particular to [NZC09a]. Our algorithm differs in the detailed implementation of the above steps to obtain an in-place algorithm. We describe the details of the above steps in the following Sections. Finally, see Appendix B for restoring T.

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 Lemma 3 below. Now, we describe how to implement this step using linear time and $O(1)$ workspace. We divide this into two part, one part is for renaming all L-type characters to be the index of its bucket head and the other part is for renaming all S-type characters to be the index of its bucket tail. This step is similar to counting sort (see e.g., [CLRS01, Ch. 8]).

1. First we scan T once to compute the number of times each character occurs in T and store them in SA (i.e., first initialize $\mathsf{SA}[i] = 0$ for all $i \in [0, n-1]$, then for each $T[i]$ we increase $\mathsf{SA}[T[i]]$ by one). Then we perform a *prefix sum computation* to determine the starting position of each character (i.e. bucket head) in SA (i.e., scan SA once, for each $\mathsf{SA}[i]$, let $\mathsf{SA}[i] = sum$, and $sum = sum + \mathsf{SA}[i]$, where sum denote how many characters so far). Finally we scan T once again, for each $T[i]$ we let $T[i] = \mathsf{SA}[T[i]]$ (the index of its bucket head). Now, all characters of T has renamed as the index of its bucket head.

2. Then we need to let the S-type characters of T to be the index of its bucket tail. First we scan T once 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 $T[i] = T[i] + \mathsf{SA}[T[i]] - 1$ (the index of its bucket tail). (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 case : 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 a Boolean variable which represent the type of previous character $T[i+1]$)

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

Proof: For two suffixes, beginning with the same character, the L-suffix is smaller than the S-suffix. 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

$T'[0] = 7$ since $T[0]$ is L-type and the head of bucket 2 (i.e., bucket $T[0]$) is 7. $T'[1] = 6$ since $T[1]$ is S-type and the tail of bucket 1 (i.e., bucket $T[1]$) is 6.

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 pointers/counters for each bucket to indicate how many entries we have used 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. [4]

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 once 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, $T[i]$ 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.

[4] We assume that a word contains enough bits to represent any character in Σ and all such special symbols (there are at most five special symbols).

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

(1) SA$[T[i]]$ = Unique: In this case, we let 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).

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

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

(4) 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 SA$[T[i]]$, to find the first position j such that SA$[j]$ = Empty. Then we let 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 scan step, all indices of LMS-characters have been placed in SA. Note that there may be still some special symbols Multi and the counters (due to the bucket is not fully filled, so we have not shifted these indices to right in this bucket). We need to free these position. We scan SA once more from right to left. If SA$[i]$ = Multi, we shift the indices of LMS-characters in this bucket to right by two positions (i.e., SA$[i-c-1 \ldots i-2]$ to SA$[i-c+1 \ldots i]$) and let SA$[i-c-1]$ = SA$[i-c]$ = Empty, where c = SA$[i-1]$ denote the counter.

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	($\underline{\mathbf{U}}$)	(E)	(E	E	E	E	$\underline{\mathbf{M}}$)	(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	($\underline{\mathbf{12}}$)	(E)	(E	E	E	E	M)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(E	E	$\underline{\mathbf{9}}$	**1**	**M**)	(E	E)	(E	E	E	E)
SA	(12)	(E)	(E	$\underline{\mathbf{5}}$	9	**2**	**M**)	(E	E)	(E	E	E	E)
SA	(12)	(E)	($\underline{\mathbf{1}}$	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 step, we remove all Multi symbols and counters.

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 explain the Step 2 which 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 (the last but one) and scan the bucket once (the last one). It takes $O(n)$ time since every bucket only need to be shifted and scanned once. □

3.4 Sort all LMS-substrings

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 LMS-prefix of an suffix $\mathsf{suf}(i)$ 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 sort the LMS-prefix of all suffixes of T from the sorted LMS-characters is the same as sort all suffixes of T from the sorted LMS-suffixes (see the Preliminary Section 2). Now, 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 this 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 denote 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.

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) Sort the LMS-prefix 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) Place 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**

We only need to explain the second part which place the indices of all sorted LMS-substrings in $\mathsf{SA}[n - n_1 \dots n - 1]$. First, we need the following observation. Then we give a lemma to show that this step can be done in linear time using $O(1)$ workspace.

Observation 1 *For every bucket in* SA, *let t to 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 *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: After Step (1), we can scan SA once from right to left to place the indices of all LMS-substrings into the end of SA. We only need to explain that when we scanning $\mathsf{SA}[i]$ how to identify $T[\mathsf{SA}[i]]$ is LMS-character or not. Note that if we can identify $T[\mathsf{SA}[i]]$ is S-type or not, we can also identify $T[\mathsf{SA}[i]]$ is LMS-character or not since $T[\mathsf{SA}[i]]$ is 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 identify this bucket contains S-type characters or not from Observation 1. Furthermore, if we can compute the number of S-type characters in this bucket, we will have done this step. To compute the number of S-type characters in this bucket, we continue to scan this bucket from its tail. For the current scanning entry $\mathsf{SA}[i]$, 1).If $T[\mathsf{SA}[i]] \geq T[\mathsf{SA}[i]+1]$, do nothing; 2).Otherwise, let j to 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 $j - \mathsf{SA}[i] + 1$, where we maintain a variable num to count the number of S-type characters in this bucket and initially to be 0. This step cost $O(n)$ time overall since each character is scanned at most twice. □

3.5 Get reduced problem T_1

In this section, we construct the smaller problem T_1 which we need to solve recursively. We rename the sorted LMS-substrings (obtained from the previous step) using their ranks.

Now, we spell out the details. Initially, all LMS-substrings are sorted in $\mathsf{SA}[n-n_1\dots n-1]$. First let the rank of the smallest LMS-substring $\mathsf{SA}[n-n_1]$ to be 0 (it must be the sentinel). Then we scan $\mathsf{SA}[n-n_1+1\dots n-1]$ once from left to right to compute the rank for each LMS-substring. When we 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 shifting nonempty entries in $\mathsf{SA}[0\dots n-n_1-1]$ to the left, so that the ranks occupy a continuous segment of space. Now, we have obtained the reduced problem T_1 which are 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 (w.r.t. their 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 scan $\mathsf{SA}[n-n_1\dots n-1]$ (which stored 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 let T_1 in the $\mathsf{SA}[0\dots n_1-1]$ by shifting nonempty items in $\mathsf{SA}[0\dots n-n_1-1]$ 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 $T_1 =$ “1120” corresponding to the LMS-substrings {“66996”, “66996”, “6710”, “0”}.

First, we give an observation which help us to identify the S-type and L-type character of T. Then we can obtain the following lemma showed that T_1 can be obtained in linear time using this observation.

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 between $T[k]$ and $T[j]$ are S-type.*

Lemma 6 *T_1 can be obtained using $O(n)$ time and $O(1)$ workspace.*

Proof: For the workspace term is obvious since we do not use extra space beyond SA in above step. For the time, we only need to explain the time of the comparison process used. 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 only be scanned when identify the length of its adjacent predecessor LMS-substring and itself. Thus the comparison process takes $O(n)$ time because the total length of all LMS-substrings is less than $2n$. □

3.6 Sort all LMS-suffixes

In this section, we sort all LMS-suffixes and place their indices in the tail of their corresponding buckets in SA, which is carried out as follows:

1. We first solve T_1 recursively. From Section 3.5, we have T_1 stored in $\mathsf{SA}[0 \ldots n_1 - 1]$. Define SA_1 to be $\mathsf{SA}[n - n_1 \ldots n - 1]$. We 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 once 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 $\mathsf{LMS}[0 \ldots n_1] = \mathsf{SA}[n - n_1 \ldots n - 1]$. 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 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 3.3) 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 $\mathsf{SA}[0 \ldots n_1 - 1]$ 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 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 7 *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 correspond 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 are the same as the order of suffixes in T_1. $\square$

3.7 Sort all suffixes

Now, we sort all suffixes of T from the sorted LMS-suffixes using induced sorting (Note that this step is the same as we do in Section 3.4, we give the details here). First, we induce 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 time and space.

Step 1. Sort all L-suffixes using induced sorting : We divide this step into two parts, and describe the details as follows:

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

- (i) If SA$[T[i]]$ = Empty, let SA$[T[i]]$ = Unique (unique L-type character in this bucket).
- (ii) If SA$[T[i]]$ = Unique, let SA$[T[i]]$ = Multi (number of L-type characters in this bucket is at least 2).
- (iii) Otherwise do nothing.

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

- (i) If SA$[i]$ = Empty, do nothing.
- (ii) If SA$[i]$ is an index: let j = SA$[i] - 1$. If suf(j) is L-suffix (This can be identified in constant time from the following lemma 8), we place suf(j) into the LF-entry of its bucket and increase the counter by one. This is similar to previous Step 2 which placing the indices of LMS-characters into SA in Section 3.3.
- (iii) If SA$[i]$ = Multi, which means SA$[i]$ is the head of its bucket, and this bucket has at least two L-suffixes which are not sorted. In this case, SA$[i]$ and SA$[i+1]$ are used as bucket head (the symbol Multi) and counter of this bucket, respectively. Then we skip these two entries and continue to scan SA$[i+2]$.

Now, all L-suffixes have be 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 position in SA.

Step 2. Remove LMS-Suffixes from SA : We can use a trick similar to previous Step 2 which placing the indices of LMS-characters into SA in Section 3.3. The difference is that instead of placing actual LMS-characters, we place 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 position in SA.

Step 3. Sort all S-suffixes using induced sorting : Now, this step is completely symmetrical to 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 it here.

In order to show the time used in this step, we need the the following useful lemma in the induced sort step which scan SA from left to right to sort L-suffixes.

Lemma 8 *When we are scanning* $\mathsf{SA}[i]$, *we want to identity 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 scanning bucket* $T[\mathsf{SA}[i]]$*), if all L-suffixes of* T *that belong to bucket* $T[\mathsf{SA}[i]]$ *are not already sorted, then the* $\mathsf{suf}(\mathsf{SA}[i]-1)$ *is L-suffix.*

Proof: From the Property 1, in any bucket, S-suffixes always appear after the L-suffixes in SA. Moreover, it is obvious that every suffix of T is considered exactly once. Combine these observations imply this lemma. Furthermore, whether all L-suffixes of T that belong to current bucket $T[\mathsf{SA}[i]]$ are already sorted or not is not hard to identify by scanning current bucket once when we are reaching a new bucket. □

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. Sort all L-suffixes using induced sorting:
(1) After initialization:

$index$	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	$(\underline{\mathbf{U}})$	$(E$	E	1	5	9)	$(\underline{\mathbf{M}}$	$E)$	$(\underline{\mathbf{M}}$	E	E	$E)$

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

$index$	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	$(\overrightarrow{\mathbf{12}})$	$(\underline{\mathbf{11}})$	$(E$	E	1	5	9)	$(M$	$E)$	$(M$	E	E	$E)$
SA	(12)	$(\overrightarrow{\mathbf{11}})$	$(E$	E	1	5	9)	$(\underline{\mathbf{10}}$	$E)$	$(M$	E	E	$E)$
SA	(12)	(11)	$(E$	E	$\overrightarrow{\mathbf{1}}$	5	9)	(10	$\underline{\mathbf{0}})$	$(M$	E	E	$E)$
SA	(12)	(11)	$(E$	E	1	$\overrightarrow{\mathbf{5}}$	9)	(10	0)	$(\underline{\mathbf{M}}$	$\underline{\mathbf{1}}$	$\underline{\mathbf{4}}$	$E)$
SA	(12)	(11)	$(E$	E	1	5	$\overrightarrow{\mathbf{9}})$	(10	0)	$(\underline{\mathbf{M}}$	$\underline{\mathbf{2}}$	4	$\underline{\mathbf{8}})$
SA	(12)	(11)	$(E$	E	1	5	9)	(10	0)	$(M$	2	$\overrightarrow{\mathbf{4}}$	8)
SA	(12)	(11)	$(E$	E	1	5	9)	(10	0)	$(\overrightarrow{\mathbf{4}}$	$\underline{\mathbf{8}}$	$\underline{\mathbf{3}}$	$\underline{\mathbf{E}})$
SA	(12)	(11)	$(E$	E	1	5	9)	(10	0)	(4	$\overrightarrow{\mathbf{8}}$	3	$\underline{\mathbf{7}})$

The third 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	$\underline{\mathbf{E}}$	$\underline{\mathbf{E}}$	$\underline{\mathbf{E}})$	(10	0)	(4	8	3	7)

Step 3. Sort all S-suffixes using induced sorting:
(1) After initialization:

$index$	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	$(E$	E	E	E	$\underline{\mathbf{M}})$	(10	0)	(4	8	3	7)

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

$index$	0	1	2	3	4	5	6	7	8	9	10	11	12
SA	(12)	(11)	$(E$	E	$\underline{\mathbf{6}}$	$\underline{\mathbf{1}}$	$\underline{\mathbf{M}})$	(10	0)	(4	8	3	$\overleftarrow{\mathbf{7}})$
SA	(12)	(11)	$(E$	$\underline{\mathbf{2}}$	6	$\underline{\mathbf{2}}$	$\underline{\mathbf{M}})$	(10	0)	(4	8	$\overleftarrow{\mathbf{3}}$	7)
SA	(12)	(11)	$(\underline{\mathbf{9}}$	2	6	$\underline{\mathbf{3}}$	$\underline{\mathbf{M}})$	$(\overleftarrow{\mathbf{10}}$	0)	(4	8	3	7)
SA	(12)	(11)	(9	2	$\overleftarrow{\mathbf{6}}$	3	$M)$	(10	0)	(4	8	3	7)
SA	(12)	(11)	$(\underline{\mathbf{E}}$	$\underline{\mathbf{5}}$	$\underline{\mathbf{9}}$	$\underline{\mathbf{2}}$	$\overleftarrow{\mathbf{6}})$	(10	0)	(4	8	3	7)
SA	(12)	(11)	$(\underline{\mathbf{1}}$	5	9	$\overleftarrow{\mathbf{2}}$	6)	(10	0)	(4	8	3	7)
SA	(12)	(11)	(1	5	9	2	6)	(10	0)	(4	8	3	7)

Lemma 9 *Given all sorted LMS-suffixes of T, all the suffixes use these induced sorting steps can be sorted correctly using $O(n)$ time and $O(1)$ workspace.*

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 have the following theorem.

Theorem 4 *Our algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of string T over an integer alphabet.*

4 Suffix sorting for read-only general alphabets

4.1 Framework

In this section, we give the framework of our suffix sorting algorithm for read-only general alphabets. Let n_L and n_S denote the number of L-suffixes and S-suffixes, respectively. Now, the framework is described as follows:

1. If $n_S \leq n_L$ (i.e., the number of S-suffixes is less), then

   (1) (Section 4.2) Sort all S-substrings of T using mergesort directly.
   We use mergesort to sort all S-substring of T in SA$[0 \ldots n_S - 1]$. In the merging step of mergesort, we use SA$[n_S \ldots 2n_S - 1]$ as the temporary space. After this step, all S-substrings should be in the lexicographical order stored in SA$[0 \ldots n_S - 1]$.

   (2) (Section 4.3) 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 SA$[0 \ldots n_S - 1]$. 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 SA$[n_S \ldots 2n_S - 1]$.

(3) (Section 4.4) Sort S-suffixes by solving T_1 recursively.
We sort $T_1 = \mathsf{SA}[n_S \dots 2n_S - 1]$ recursively, in the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[0 \dots n_S - 1]$ as the output space for T_1. Then we use the suffix array of T_1 which stored in SA_1 to place all indices of S-suffixes of T in lexicographical order into $\mathsf{SA}[0 \dots n_S - 1]$.

(4) (Section 4.5) Sort all suffixes from the sorted S-suffixes.
First we place all indices of S-suffixes in the their final positions in SA by using mergesort together with a stable, in-place, linear time merging algorithm [SS87]. Then we use induced sorting step to sort all L-suffixes from the sorted S-suffixes. After this, all suffixes of T have been sorted in $\mathsf{SA}[0 \dots n - 1]$.

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

The purpose of comparing n_L and n_S is to guarantee the size of the reduced problem T_1 is no more than half of $|T|$ (i.e., $|T_1| \leq |T|/2$). Without lost of generality, we assume that $n_S \leq n_L$.

4.2 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 $\mathsf{SA}[0 \dots n_S - 1]$. Note that $n_S \leq n/2$ since we assume that $n_S \leq n_L$.

2. Then, we sort $\mathsf{SA}[0 \dots n_S - 1]$ using mergesort (the sorting key for $\mathsf{SA}[i]$ is the S-substring of T which begins at $T[\mathsf{SA}[i]]$). We use $\mathsf{SA}[n_S \dots 2n_S - 1]$ 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 $\mathsf{SA}[0 \dots n_S - 1]$. We have the following lemma.

Lemma 10 *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 $\mathsf{SA}[n_S \dots 2n_S - 1]$ as the workspace for mergesort. We need to show that Step 2 takes $O(n \log n)$ time. It suffices to show that the time spent for comparison process in one recursive level of mergesort (there are $O(\log n)$ recursive levels) can be bounded by $O(n)$. Since each S-substring is compared to exactly one other S-substring. The length of the S-substrings can be obtained according to Observation 2. Note that each character of T is scanned at most twice since it only be scanned when identify the length of its adjacent predecessor S-substring and itself. Thus the comparison process takes $O(n)$ time because the total length of all S-substrings is less than $2n$. □

4.3 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 4.2, all S-substrings have been sorted in $\mathsf{SA}[0 \dots n_S - 1]$. The construction of T_1 consists of the following two steps:

1. We rename the S-substrings by their ranks (this step is similar to the step of renaming the LMS-substrings by their ranks in Section 3.5). First let the rank of $\mathsf{SA}[0]$ be 0. We scan $\mathsf{SA}[1 \dots n_S - 1]$ 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 $\mathsf{SA}[n_S + i]$.

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

After the above two steps, we get the reduced problem T_1 in $\mathsf{SA}[n_S \dots 2n_S - 1]$.

Lemma 11 *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 can only participate in two comparisons. Now the argument is similar to a comparison process in one recursive level of mergesort in Lemma 10, thus it costs linear time. Obviously, Step 2 takes $O(n \log n)$ time and $O(1)$ workspace. $\square$

4.4 Sort all S-suffixes by solving T_1 recursively

In this section, we solve $T_1 = \mathsf{SA}[n_S \dots 2n_S - 1]$ recursively to obtain the order of all S-suffixes in $\mathsf{SA}[0 \dots n_S - 1]$. For the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[0 \dots n_S - 1]$ 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 represent. This 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 $\mathsf{SA}[2n_S - sum]$ (i.e., let $\mathsf{SA}[2n_S - sum] \leftarrow i$). Now $\mathsf{SA}[n_S \dots 2n_S - 1]$ stores the indices of all S-suffixes of T.

2. Then for $i \in [0, n_S - 1]$, let $\mathsf{SA}[i] \leftarrow \mathsf{SA}[n_S + \mathsf{SA}[i]]$.

Now, we have obtained all S-suffixes in the lexicographical order in $\mathsf{SA}[0 \dots n_S - 1]$.

4.5 Sort all suffixes of T

From Section 4.4, we have obtained the sorted S-suffixes in $\mathsf{SA}[0 \dots n_S - 1]$. Now, we sort all suffixes from these sorted S-suffixes.

Preprocessing : First, we move these S-suffixes from $\mathsf{SA}[0 \dots n_S - 1]$ to $\mathsf{SA}[n - n_S \dots n - 1]$. Then we scan T from right to left to place all indices of L-suffixes into $\mathsf{SA}[0 \dots n - n_S - 1]$. Now, 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 make some useful observations.

Observation 3 *All suffixes of T have been sorted by their first characters in* SA.

Observation 4 *All the 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 12 *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 12) using induced sorting. Now, we extend the interior counter trick in Section 3.3 to handle the read-only general alphabets. We use five special symbols H (Head), T_L (Tail of L-suffixes), E (Empty), R_1 (one remaining L-suffix) and R_2 (two remaining L-suffixes). 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 once 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 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 binary search $T[i]$ in SA (the search key for $\mathsf{SA}[i]$ is $T[\mathsf{SA}[i]]$) using $O(\log n)$ 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 once 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 $\mathsf{SA}[l+1] = T_L$ (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	$\mathsf{SA}[l]$	$\underline{\mathbf{T_L}}$	$\mathsf{SA}[l+2]$	$\dots$	$\mathsf{SA}[r]$

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

index	l	$l+1$	$l+2(r_L)$	$l+3$	$\dots$	r
type	L	L	L	S	$\dots$	S
SA	$\mathsf{SA}[l]$	$\underline{\mathbf{H}}$	$\underline{\mathbf{T_L}}$	$\mathsf{SA}[l+3]$	$\dots$	$\mathsf{SA}[r]$

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

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	$\mathsf{SA}[l]$	$\underline{\mathbf{H}}$	$\underline{\mathbf{E}}$	$\mathsf{SA}[l+3]$	$\dots$	$\underline{\mathbf{T_L}}$	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$

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 H, T_L or others. If the bucket $T[i]$ has been initialized, $\mathsf{SA}[l+1]$ is H or T_L. Otherwise, it has not been initialized yet. [5] 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 our first algorithm in Section 3.7. The main difference is that we use binary search to find the head of bucket (while in the first algorithm, the renamed $T[i]$ is used to point to the head of bucket). Specifically, we scan SA once 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 i-th L-suffix which needs to be placed into the LF-entry of this bucket. We distinguish the following four cases:

(1) $\mathsf{SA}[l+1] = H$ and $\mathsf{SA}[l+2] = E$: 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) $\mathsf{SA}[l+1] = H$ and $\mathsf{SA}[l+2] \neq E$: 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 $\mathsf{SA}[l+c+2] \neq T_L$, we let $\mathsf{SA}[l+c+2] = j$ and $\mathsf{SA}[l+2] = c+1$. Otherwise (this is the last but two L-suffix, i.e.,L_{N_L-2}), we shift these $c-1$ L-suffixes to the left by one position (i.e., move $\mathsf{SA}[l+3\dots r_L-1]$ to $\mathsf{SA}[l+2\dots r_L-2]$) and let $\mathsf{SA}[r_L-1] = j$ and $\mathsf{SA}[l+1] = R_2$.

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

(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 $\mathsf{SA}[r_L] = R_1$. Then let $\mathsf{SA}[r_L] = j$.

[5] Note that we can also do binary search in SA, though there are special symbols (i.e, H, T_L, 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 (i.e., this entry represent an index of suffix of 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
case (1) :									
SA	$\mathsf{SA}[l]$	H	E	$\mathsf{SA}[l+3]$	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
SA	$\underline{\mathbf{L_1}}$	H	$\underline{\mathbf{1}}$	$\mathsf{SA}[l+3]$	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
case (2) :									
SA	L_1	H	1	$\mathsf{SA}[l+3]$	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
SA	L_1	H	$\underline{\mathbf{2}}$	$\underline{\mathbf{L_2}}$	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
			$\vdots$						
SA	L_1	H	$\underline{\mathbf{N_L-3}}$	L_2	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
SA	L_1	$\underline{\mathbf{R_2}}$	$\underline{\mathbf{L_2}}$	$\underline{\mathbf{L_3}}$	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
case (3) :									
SA	L_1	R_2	L_2	L_3	$\dots$	T_L	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
SA	L_1	$\underline{\mathbf{L_2}}$	$\underline{\mathbf{L_3}}$	$\underline{\mathbf{L_4}}$	$\dots$	$\underline{\mathbf{R_1}}$	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
case (4) :									
SA	L_1	L_2	L_3	L_4	$\dots$	R_1	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$
SA	L_1	L_2	L_3	L_4	$\dots$	$\underline{\mathbf{L_{N_L}}}$	$\mathsf{SA}[l+N_L]$	$\dots$	$\mathsf{SA}[r]$

We have the following lemmas and theorem.

Lemma 13 *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-suffixes which needs to be inserted into the bucket $T[\mathsf{SA}[i]-1]$. This special case needs $O(N_{LS})$ time, where N_{LS} denote the size of the bucket $T[\mathsf{SA}[i]-1]$ (i.e., $N_{LS}=r-l+1$).*

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 8, 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 interior counter trick which maintain 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 $O(N_{LS})$ time. □

Lemma 14 *All the suffixes can be sorted correctly from the sorted S-suffixes in $O(n\log n)$ time and $O(1)$ workspace.*

Proof: We can sort all the S-suffixes correctly using induced sorting by Lemma 1. For workspace, it is obvious. For time, we use $O(\log n)$ time binary search to find the head of its bucket for each L-suffix . Then we use the above step to locate the position of the L-suffix in $O(1)$ time except for the last two L-suffixes of the bucket. For the last two L-suffixes, 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). But this only takes $O(n)$ time overall since each bucket is scanned at most twice. Thus the time is mainly spent in the binary search step. To sum up, this sorting step takes $O(n\log n)$ time. □

Theorem 5 *Our algorithm takes $O(n \log n)$ time and $O(1)$ workspace to compute the suffix array for string T over a general alphabet.*

Proof: All steps in our algorithm takes $O(n \log n)$ time. Besides, the size of recursive problem T_1 is not 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, in the same recursive level, the different steps can reuse this $O(1)$ workspace too. $\square$

5 Suffix sorting for read-only integer alphabets

5.1 Framework

Our suffix sorting algorithm for read-only integer alphabets requires to use some steps of the above two algorithms. First, we provide a framework of our algorithm as follows:

1. If the number of S-suffixes of T is more than that of the L-suffixes (i.e., $n_S > n_L$), then:
   1) (Section 5.2) Sort all LMS-characters of T.
   2) (Section 5.3) Sort all LMS-substrings from the sorted LMS-characters.
   3) Construct and solve the reduced problem T_1 from the sorted LMS-substrings.
   4) (Section 5.3) Sort all suffix from sorted LMS-suffixes (T_1).
2. Otherwise, execute above steps switching the role of LMS with LML.

Note that the number of LMS-characters are less than $|T|/2$, because any two LMS-characters are not adjacent by the definition of LMS-characters. The same holds for LML-characters. Hence the size of the reduced problem T_1 constructed by our algorithm is less than $|T|/2$. Without loss of generality, we assume that $n_S > n_L$.

Recall that the purpose for renaming T is to access the heads or tails of the buckets in constant time during the scanning process in the induced sorting step. Fortunately, we can accomplish these steps without modifying T. In Section 5.2, we show how to sort all LMS-characters (i.e. Step 1)). Since Step 2) and Step 4) are almost the same (see our first algorithm for integer alphabets in Section 3.4 and Section 3.7), we only need to show how to sort all suffixes from the sorted LMS-suffixes (i.e. Step 4)) in Section 5.3. Step 3) is exactly the same as our first algorithm which was in Section 3.5 and Section 3.6. Because the operation (which did with T) in this step is only to compare characters (i.e., T is renamed or not does not influence this step), we omit the Step 3) in this algorithm.

5.2 Sort all LMS-characters

In this section, we sort all LMS-characters of T. Since we can not modify the input string T, we do not place the indices of LMS-characters in the tail of their corresponding bucket in SA as we did in our first algorithm. Note that this placing step in our first algorithm is equivalent to sorting all LMS-characters since bucket characters are in the lexicographical order in SA. Moreover, the next immediate step is to use these sorted LMS-characters to sort all LMS-substrings using induced sorting. So the observation here is that we can place the sorted LMS-characters in arbitrary positions in SA, as long as we can sort all LMS-substrings. In Section 5.3, we will show how to sort all suffixes from sorted LMS-suffixes (same as sort all LMS-substrings from sorted LMS-characters), when the sorted LMS-suffixes are placed in $\mathsf{SA}[n - n_1 \dots n-1]$, where n_1 denote the number of LMS-suffixes and $n = |T|$. In this section, we first show how to place the sorted LMS-characters in $\mathsf{SA}[n - n_1 \dots n-1]$. Recall that when we say that placing characters or suffixes of T into SA, it always means that placing its corresponding indices of T into SA.

Now, we show how to place the sorted LMS-characters into $\mathsf{SA}[n - n_1 \dots n-1]$. We use m to denote the number of LMS-characters which belong to $[1, |\Sigma|/2]$ (i.e., these LMS-characters $T[i]$ satisfy $T[i] \in [1, |\Sigma|/2]$), where the size of alphabets is $|\Sigma| \leq n$. We do the following:

1. Sort these m LMS-characters in $[1, |\Sigma|/2]$.
   We use counting sort (see e.g., [CLRS01, Ch. 8]) to sort these m LMS-characters. Concretely, we use $\mathsf{SA}[1 \dots |\Sigma|/2]$ as the *count array*, and use $\mathsf{SA}[|\Sigma|/2 + 1 \dots |\Sigma|/2 + m]$ as the output array. After this counting sort step, the indices of these m sorted LMS-characters have been placed in $\mathsf{SA}[|\Sigma|/2 + 1 \dots |\Sigma|/2 + m]$.

2. Sort the remaining $n_1 - m$ LMS-characters in $[|\Sigma|/2 + 1, |\Sigma|]$.
   Similar to the above step, we also use counting sort to sort the remaining LMS-characters. In the counting sort step, we use $\mathsf{SA}[1 \dots |\Sigma|/2]$ as the *count array* (note that we use $\mathsf{SA}[T[i] - |\Sigma|/2]$ to count the number of times LMS-charcter $T[i] \in [|\Sigma|/2+1, |\Sigma|]$), and use $\mathsf{SA}[|\Sigma|/2+m+1 \dots \Sigma|/2+n_1]$ as the output array. After this counting sort step, the indices of these $n_1 - m$ remaining LMS-characters have been placed in $\mathsf{SA}[|\Sigma|/2 + m + 1 \dots |\Sigma|/2 + n_1]$.

After the above two steps, we have sorted all LMS-characters in $\mathsf{SA}[|\Sigma|/2 + 1, |\Sigma|/2 + n_1]$ (i.e., placed their corresponding indices in $\mathsf{SA}[|\Sigma|/2 + 1, |\Sigma|/2 + n_1]$). Then we move them to $\mathsf{SA}[n - n_1 \dots n-1]$, which can be easily done in $O(n)$ time and $O(1)$ workspace.

5.3 Sort all suffixes from sorted LMS-suffixes

In this section, we need to sort all suffixes from the sorted LMS-suffixes. The sorted LMS-suffixes have been placed in $\mathsf{SA}[n - n_1 \dots n-1]$) from the previous steps.

Let $\mathsf{SA}_L = \mathsf{SA}[0 \dots n_L - 1]$ and $\mathsf{SA}_S = \mathsf{SA}[n_L \dots n-1]$ (note that $n_L + n_S = n$). Now, we do the following three steps to sort all suffixes:

1. (Section 5.3.1) 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.

2. (Section 5.3.2) Sort all n_S S-suffixes from the sorted L-suffixes which are stored in SA_L and store the sorted S-suffixes in SA_S.

3. (Section 5.3.3) Merge the L-suffixes (stored in SA_L) and S-suffixes (stored in SA_S) to sort all suffixes in $\mathsf{SA}[0 \dots n-1]$

5.3.1 Sort all L-suffixes from the sorted LMS-suffixes

In this section, 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. Before we sort L-suffixes, we need the following lemma.

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

Let $c_p = \lceil 5cn/\log n \rceil$. Without loss of generality, we assume $n > 2c_p$ (otherwise it is easy to solve since n is constant). Now, we describe how to sort all n_L L-suffixes from the sorted LMS-suffixes which are stored in $\mathsf{SA}[n - n_1 \ldots n-1]$. We divide this sorting step into three steps as follows:

Step 1. Construct the *pointer data structures* for all L-suffixes : Since we cannot modify T, we need to find another method to get the bucket heads for the L-suffixes efficiently. Especially, it should be space-efficient. For this purpose, we construct a space-efficient *pointer data structures* to represent the bucket heads of all L-suffixes and support to find the bucket head of any L-suffix in constant time. We store the pointer data structures in $\mathsf{SA}[n/2 - c_p \ldots n/2 - 1]$. Note that the buckets of the L-suffixes we talked in this section is in SA_L (recall that $\mathsf{SA}_L = \mathsf{SA}[0 \ldots n_L - 1]$). Moreover, the space storing the pointer data structures (i.e., $\mathsf{SA}[n/2 - c_p \ldots n/2 - 1]$) has no conflict to the space storing the sorted LMS-suffixes (i.e., $\mathsf{SA}[n - n_1 \ldots n-1]$) since $n_1 \leq \frac{n}{2}$. Now, we show the details how to construct the pointer data structures. This contains four parts as follows:

(1) Construct the *pointer data structure* for these L-suffixes $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4}]$. We use D_1 to denote this pointer data structure. Because these four steps (1)-(4) are almost the same, we only show the details how we construct the pointer data structure D_1. We do the following:

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

   (ii) Then we scan $\mathsf{SA}[1 \ldots \frac{|\Sigma|}{4}]$ once from left to right. We use a variable sum to count the sum, first initialize $sum = 1$. Then for each $\mathsf{SA}[i]$ which is being scanned, let $\mathsf{SA}[i] = sum$, and $sum = sum + \mathsf{SA}[i]$(similar to the prefix sum computation in the counting sort). Now, for any L-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4}]$, $\mathsf{SA}[T[i]] - T[i]$ must indicate the head of bucket $T[i]$ in SA_L. Since we want every entry in $\mathsf{SA}[1 \ldots \frac{|\Sigma|}{4}]$ to be distinct, we initialize $\mathsf{SA}[i] = 1$ for all $i \in [1, \frac{|\Sigma|}{4}]$ in above Step (i). Hence the head of bucket $T[i]$ is $\mathsf{SA}[T[i]] - T[i]$.

   (iii) Finally, we construct the pointer data structure D_1 for $\mathsf{SA}[1 \ldots \frac{|\Sigma|}{4}]$. D_1 uses $c(n + \frac{|\Sigma|}{4})/\log n$ words space, and it can support to find the bucket head of any L-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4}]$ in $O(1)$ time according to Lemma 15. We store D_1 in the tail of $\mathsf{SA}[0 \ldots n/2 - 1]$ (i.e., $\mathsf{SA}[n/2 - c(n + \frac{|\Sigma|}{4})/\log n \ldots n/2 - 1]$).

(2) Construct the pointer data structure for these L-suffixes $\mathsf{suf}(i)$ satisfying $T[i] \in [\frac{|\Sigma|}{4} + 1, \frac{|\Sigma|}{2}]$. We use D_2 to denote this pointer data structure.

(3) Construct the pointer data structure for these L-suffixes $\mathsf{suf}(i)$ satisfying $T[i] \in [\frac{|\Sigma|}{2} + 1, \frac{3|\Sigma|}{4}]$. We use D_3 to denote this pointer data structure.

(4) Construct the pointer data structure for these L-suffixes $\mathsf{suf}(i)$ satisfying $T[i] \in [\frac{3|\Sigma|}{4} + 1, |\Sigma|]$. We use D_4 to denote this pointer data structure.

Step (2),(3) and (4) do the same as Step (1). After this four steps, the pointer data structures (D_1, D_2, D_3 and D_4) are stored in $\mathsf{SA}[n/2 - c_p \dots n/2 - 1]$.

Now, we can have the following lemma:

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

Proof: D_1, D_2, D_3 and D_4 takes at most $4c(n + \frac{|\Sigma|}{4})/\log n \leq c_p$ words. We need four values m_1, m_2, m_3 and m_4, which denote the number of L-suffixes in above step (1), (2), (3) and (4), respectively (They can be obtained from the variable sum which is computed in the final stage of the step (ii)). Now, if we want to find the bucket head of an L-suffix $\mathsf{suf}(i)$, we first compare $T[i]$ with $\frac{|\Sigma|}{4}$, $\frac{|\Sigma|}{2}$ and $\frac{3|\Sigma|}{4}$ to see which pointer data structure $T[i]$ belongs to. Assume it belongs to D_j. Then we do a $\mathsf{select}(T[i] - (j-1)\frac{|\Sigma|}{4})$ query on D_j and combine the select result with the corresponding m_k ($k < j$) to identify the head of bucket $T[i]$. All the above operations can be done in $O(1)$ time. $\square$

Step 2. Sort the first $n_L - c_p$ smallest L-suffixes : Now, we show how to sort the first $n_L - c_p$ smallest L-suffixes in $\mathsf{SA}_L[0 \dots n_L - c_p - 1]$. From Step 1, we have obtained the pointer data structures stored in $\mathsf{SA}[n/2 - c_p \dots n/2 - 1]$ which can support finding the bucket head of any L-suffix in constant time by Lemma 16. The sorted LMS-suffixes are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$. Thus, the first $n_L - c_p$ smallest L-suffixes can be sorted into $\mathsf{SA}_L[0 \dots n_L - c_p - 1]$ using the same induced sorting step in our first algorithm (which sorts all L-suffixes in Section 3.7) except in the scanning process, we use the select query in the pointer data structures to find the bucket head. Note that we do nothing when the L-suffix is in the remaining c_p largest L-suffixes. It is not hard to see that the first $n_L - c_p$ smallest L-suffixes can be sorted correctly since the remaining c_p largest L-suffixes can not influence the order of the first $n_L - c_p$ smallest L-suffixes in the induced sorting step.

Now, we specify that how to identify an L-suffix belongs to the first $n_L - c_p$ smallest L-suffixes or not. First, we can scan T once to find the character of $n_L - c_p$ smallest L-suffix using this pointer data structures (let ch to denote this character). If the head of the bucket ch is exactly $n_L - c_p$ in SA_L, then we identify the L-suffix by comparing it with ch. Otherwise, the $n_L - c_p$ smallest L-suffix belongs to bucket ch, and it will be stored in $\mathsf{SA}[n_L - c_p]$. We only need two variables to indicate whether the L-suffix belongs to the first $n_L - c_p$ smallest L-suffixes or not. One is the number to denote the head of bucket ch, and the other is a number to denote the gap between the head of bucket ch and $\mathsf{SA}[n_L - c_p]$.

We have the following lemma:

Lemma 17 *The first $n_L - c_p$ smallest L-suffixes can be sorted into $\mathsf{SA}_L[0 \dots n_L - c_p - 1]$ in linear time and $O(1)$ workspace.*

Step 3. Sort the remaining c_p L-suffixes : Now we sort the remaining c_p L-suffixes into $\mathsf{SA}_L[n_L - c_p \dots n_L - 1]$. Since the pointer data structures has occupied c_p words in $\mathsf{SA}[n/2 - c_p \dots n/2 - 1]$, we need to free these entries in order to store the remaining c_p L-suffixes. We cannot do the same as in our first algorithm since we do not have the pointer data structures. Fortunately, we can sort the remaining c_p L-suffixes using the similar step in our second algorithm (sort L-suffixes from the sorted S-suffixes in Section 4.5). Since $c_p = \lceil 5cn/\log n \rceil$, this step takes $O(n)$ time. Now we describe how we do this. First we scan T to place the remaining c_p L-suffixes into $\mathsf{SA}_L[n_L - c_p \dots n_L - 1]$ (not sorted yet). Since $c_p = \lceil 5cn/\log n \rceil$, we can sort these remaining c_p L-suffixes using their first characters in $O(n)$ time. After this, the remaining c_p L-suffixes are in their buckets (recall that the buckets are defined in SA_L). Now we can do the same induced sorting step as we do in our second algorithm (sort L-suffixes from the sorted S-suffixes in Section 4.5).

Lemma 18 *The remaining c_p L-suffixes can be sorted into $\mathsf{SA}_L[n_L - c_p \dots n_L - 1]$ in linear time and O(1) workspace.*

From the above steps and lemmas, we have the following lemma:

Lemma 19 *All L-suffixes can be sorted correctly from the sorted LMS-suffixes in linear time and $O(1)$ workspace.*

5.3.2 Sort all S-suffixes from the sorted L-suffixes

In this section, we sort all n_S S-suffixes from the sorted L-suffixes which are stored in SA_L (i.e., $\mathsf{SA}[0 \dots n_L - 1]$) and store the sorted S-suffixes in SA_S (i.e., $\mathsf{SA}[n_L \dots n-1]$). Note that this step is almost the same as the step in Section 5.3.1 where we sort all L-suffixes from the sorted LMS-suffixes. More concretely, we do the following three similar steps:

Step 1. Construct the *pointer data structures* for all S-suffixes : Since $n_L \leq n/2$ (note that we have assumed $n_S \geq n_L$ at the beginning of our algorithm), we can use $\mathsf{SA}[n/2 \dots n/2 + c_p - 1]$ to store the pointer data structures which represent the bucket tails of all S-suffixes in SA_S. The step is the same as Step 1 in Section 5.3.1 where we construct the pointer data structures for all L-suffixes.

Step 2. Sort the last $n_S - c_p$ largest S-suffixes : We sort the last $n_S - c_p$ largest S-suffixes into $\mathsf{SA}_S[c_p \dots n_S - 1]$. This step is the same as above Step 2 in Section 5.3.1 where we sort the first $n_L - c_p$ smallest L-suffixes.

Step 3. Sort the remaining c_p S-suffixes : We sort the remaining c_p S-suffixes into $\mathsf{SA}_S[0 \dots c_p - 1]$. This step is the same as above Step 3 in Section 5.3.1 where we sort the remaining c_p L-suffixes.

Therefore, we have the similar lemma as follows :

Lemma 20 *All S-suffixes can be sorted correctly from the sorted L-suffixes in linear time and $O(1)$ workspace.*

5.3.3 Sort all suffixes

Now we have all sorted L-suffixes in SA_L (i.e., $\mathsf{SA}[0 \dots n_L - 1]$) and all sorted S-suffixes in SA_S (i.e., $\mathsf{SA}[n_L \dots n-1]$). 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 $\mathsf{SA}[0 \dots n-1]$.

Lemma 21 *All suffixes can be sorted correctly from all sorted L-suffixes and S-suffixes in linear time and $O(1)$ workspace.*

Proof: Since the merging step is stable, after merging, all L-suffixes and S-suffixes are still sorted. We only need to show the order between L-suffixes and S-suffixes are right. Note that before the merging step all L-suffixes are in front of all S-suffixes. Since the merging step is stable, it guarantees that an L-suffix is still in front of the S-suffix if they are in the same bucket in SA. So the order between L-suffixes and S-suffixes are correct according to Property 1. □

From above lemmas, we can obtain the following theorem.

Theorem 6 *Our algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of string T over a read-only integer alphabet.*

6 Experiments

In this section, we report our experimental results for our first algorithm (the linear time in-place algorithm for integer alphabets). The experiments were conducted on a 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.

The datasets were generated by choosing a random number in Σ independently for each position. We test our algorithm for $|\Sigma| = 100$, $|\Sigma| = 1000$ and $|\Sigma| = n$ (n is the length of the input). 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.

See Table 3 for the results. For the running times, we took the mean over three runs (measured using 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, writing the output SA to disk and restoring the input string T. The total space is the heap peak measured by memusage command. 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.

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

Table 3: Experimental Results: 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 Conclusions

In this paper we present three in-place algorithms for suffix sorting over (read-only) integer alphabets and read-only general alphabets. For (read-only) integer alphabets, our in-place algorithm takes linear time. The algorithm is also easy to implement and competitive in practice. For read-only general alphabets, our in-place algorithm takes $O(n \log n)$ time. All of them are optimal both in time and space. Our algorithms for integer alphabets solve the open problems posed by Franceschini and Muthukrishnan [FM07], and our algorithm for general alphabets recovers the result obtained by Franceschini and Muthukrishnan [FM07] which was an open problem posed by Manzini and Ferragina [MF02].

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 SA, such as compressed suffix arrays, longest common prefixes, Burrows-Wheeler transform and Lempel-Ziv factorization.

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*, pages 55–69. Springer, 2003.

[BW94] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 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*, 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 *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, 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 *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[FM07] G Franceschini and S Muthukrishnan. In-place suffix sorting. In *Proceedings of the 34th international conference on Automata, Languages and Programming*, 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.

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

[HSS09] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[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 *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

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

[KA05] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.

[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 *Automata, Languages and Programming*, 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*, pages 186–199. Springer, 2003.

[LS99] N Jesper Larsson and Kunihiko Sadakane. *Faster suffix sorting*. Citeseer, 1999.

[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 *AlgorithmsSA 2002*, pages 698–710. Springer, 2002.

[MF04] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[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*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[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.

[Non13] Ge Nong. Practical linear-time O (1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems (TOIS)*, 31(3):15, 2013.

[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, 2009. DCC'09.*, 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*, 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, 1998. DCC'98. Proceedings*, 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 Example

In this section, we show the induce sorting step which sorting the L-suffixes from LMS-suffixes in our example.

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	($\underline{\mathbf{12}}$)	(E	E	E	$\underline{\mathbf{1}}$	$\underline{\mathbf{5}}$	$\underline{\mathbf{9}}$)	(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	($\overrightarrow{\mathbf{12}}$)	($\underline{\mathbf{11}}$	E	E	1	5	9)	(E	E)	(E	E	E	E)
SA	(12)	($\overrightarrow{\mathbf{11}}$	E	E	1	5	9)	($\underline{\mathbf{10}}$	E)	(E	E	E	E)
SA	(12)	(11	E	E	$\overrightarrow{\mathbf{1}}$	5	9)	(10	$\underline{\mathbf{0}}$)	(E	E	E	E)
SA	(12)	(11	E	E	1	$\overrightarrow{\mathbf{5}}$	9)	(10	0)	($\underline{\mathbf{4}}$	E	E	E)
SA	(12)	(11	E	E	1	5	$\overrightarrow{\mathbf{9}}$)	(10	0)	(4	$\underline{\mathbf{8}}$	E	E)
SA	(12)	(11	E	E	1	5	9)	(10	0)	($\overrightarrow{\mathbf{4}}$	8	$\underline{\mathbf{3}}$	E)
SA	(12)	(11	E	E	1	5	9)	(10	0)	(4	$\overrightarrow{\mathbf{8}}$	3	$\underline{\mathbf{7}}$)

B Restore 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 $\{1, 2, \dots, \Sigma\}$. 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 maintain 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 bucker 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 dose 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 once 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 once 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 position 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 position 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 string T can be restored using linear time and $O(1)$ workspace.*