# Practical Linear-Time $O(1)$-Workspace Suffix Sorting for Constant Alphabets

GE NONG, Sun Yat-sen University

This article presents an $O(n)$ time algorithm called SACA-K for sorting the suffixes of an input string $T[0, n-1]$ over an alphabet $A[0, K-1]$. The problem of sorting the suffixes of $T$ is also known as constructing the suffix array (SA) for $T$. The theoretical memory usage of SACA-K is $n \log K + n \log n + K \log n$ bits. Moreover, we also have a practical implementation for SACA-K that uses $n$ bytes + $(n + 256)$ words and is suitable for strings over any alphabet up to full ASCII, where a word is $\log n$ bits. In our experiment, SACA-K outperforms SA-IS that was previously the most time and space efficient linear-time SA construction algorithm (SACA). SACA-K is around 33% faster and uses a smaller deterministic workspace of $K$ words, where the workspace is the space needed beyond the input string and the output SA. Given $K = O(1)$, SACA-K runs in linear time and $O(1)$ workspace. To the best of our knowledge, such a result is the first reported in the literature with a practical source code publicly available.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*Sorting and searching*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Suffix array, sorting algorithm, linear time, $O(1)$-workspace

**ACM Reference Format:**
Ge Nong, 2013. Practical Linear-Time $O(1)$-Workspace Suffix Sorting for Constant Alphabets. *ACM Trans. Inf. Syst.* V, N, Article A (January YYYY), 15 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

Given a string $T[0, n-1]$ of $n$ characters from an ordered alphabet $A[0, K-1]$, the suffix array (SA) of $T$ is an array $SA[0, n-1]$ of integers storing the pointers for all the suffixes in increasing lexicographical order [Manber and Myers 1993]. To simplify presentation, we assume that there is always $T[n-1] = 0$ which is the unique smallest character in $T$ and called the *sentinel*. Because of the sentinel, any two suffixes in $T$ must be different, and their lexicographical order is determined by comparing their characters one by one, from left to right, until we see a difference. Let $\mathsf{suf}(T, i)$ denote the suffix $T[i, n-1]$ in $T$. Given that all the suffixes of $T$ have been sorted in $SA$, there must be $\mathsf{suf}(T, SA[i]) < \mathsf{suf}(T, SA[j])$ for all $i < j$.

In this article, we propose an $O(n)$ time suffix array construction algorithm (SACA) called SACA-K (SACA with $K$-word workspace). The theoretical memory usage of

The author was supported by Program for New Century Excellent Talents in University (NCET-10-0854), the Project of DEGP (2012KJCX0001), the NSFC (60873056) and the Fundamental Research Funds for the Central Universities of China (11lgzd04 and 11lgpy93).
Author's address: G. Nong, Computer Science Department, Sun Yat-sen University, Guangzhou 510275, China, e-mail: issng@mail.sysu.edu.cn.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1046-8188/YYYY/01-ARTA $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

SACA-K is $n \log K + n \log n + K \log n$ bits. Moreover, we also have a practical implementation for SACA-K that uses $n$ bytes + $(n + 256)$ words and is suitable for strings over any alphabet up to full ASCII, where a word is $\log n$ bits.

SA and its variants are fundamental data structures for building information systems. During the past two decades, a plethora of SACAs of different time and space complexities have been proposed. Among them, a few notable ones are [Manber and Myers 1993; Sadakane 1998; Itoh and Tanaka 1999; Larsson and Sadakane 1999; Burkhardt and Kärkkäinen 2003; Hon et al. 2003; Manzini and Ferragina 2004; Schürmann and Stoye 2005; Maniscalco and Puglisi 2006]. Readers may want to read [Puglisi et al. 2007] for a thorough survey up to 2007. The SA can be computed in linear time [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen et al. 2006; Nong et al. 2011][1] on a RAM model. In practice the best time and space performance for linear-time SACAs is currently achieved by algorithms SA-IS and SA-DS [Nong et al. 2011]. Both algorithms use a common divide-and-conquer method to recursively compute the SA in linear time. In general, SA-IS runs faster but SA-DS can use less space in the worst case. Of particular interest to us in this article is to further improve the induced sorting technique in SA-IS to make it run faster and use less space. The key for SA-IS to achieve linear time is the combined use of the linear-time methods for problem reduction and solution induction. The time complexity of SA-IS is given by $\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$, where $\mathcal{T}(\lfloor n/2 \rfloor)$ counts for reducing $T$ into a new shortened string $T_1$ of size not greater than half of $T$ (see Lemma 3.5 in [Nong et al. 2011]), and $O(n)$ is due to inducing the SA of $T$ from that of $T_1$. The core of the whole SA-IS algorithm is the induced sorting technique for sorting the suffixes as well as the sampled substrings, which is developed on top of the following classification of L-type and S-type suffixes [Itoh and Tanaka 1999; Ko and Aluru 2005; Nong et al. 2011].

The suffix composed of only the sentinel itself, i.e. $\mathsf{suf}(T, n-1)$, is S-type. For $i \in [0, n-2]$, a suffix $\mathsf{suf}(T, i)$ is defined as L-type or S-type if $\mathsf{suf}(T, i) > \mathsf{suf}(T, i+1)$ or $\mathsf{suf}(T, i) < \mathsf{suf}(T, i+1)$, respectively. Equivalently, $\mathsf{suf}(T, i)$ is S-type if and only if (1) $i = n-1$; or (2) $T[i] < T[i+1]$; or (3) $T[i] = T[i+1]$ and $\mathsf{suf}(T, i+1)$ is S-type. Moreover, $\mathsf{suf}(T, i)$ is L-type if it is not S-type. From the suffix type definitions, an L-type or S-type suffix is larger or smaller than its succeeding suffix, respectively. Further, an S-type suffix $\mathsf{suf}(T, i)$, $i > 0$, is called an LMS-suffix (leftmost S-type) if $\mathsf{suf}(T, i-1)$ is L-type. Given the type of a suffix, we further define the type of a character: $T[i]$ is L-type or S-type if $\mathsf{suf}(T, i)$ is L-type or S-type, respectively. Furthermore, $T[i]$ is called an LMS-character if $\mathsf{suf}(T, i)$ is an LMS-suffix. A substring $T[i, j]$ 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-character in between them. $\mathsf{lms}(T, i)$ denotes the LMS-substring starting at LMS-character $T[i]$, $i \in [1, n-1]$.

The following diagram illustrates the concepts of suffix/character type and LMS-substring. Given a string $T = "\mathsf{ococonut0}"$ , by scanning the string from right to left, we find the type of each suffix and character and store it in an $n$-bit type array $t[0, n-1]$, where $t[i]$ gives the type of $\mathsf{suf}(T, i)$: 0 for L-type and 1 for S-type, respectively. Also, all the LMS-substrings, in their positional order from left to right in $T$, are found to be $\{"\mathsf{coc}", "\mathsf{con}", "\mathsf{nut0}", "\mathsf{0}"\}$ (notice that the sentinel itself is an LMS-substring), where each pair of neighboring LMS-substrings overlap on a common LMS-character.

```
              T: o c o c o n u t 0
 character type: L S L S L S L L S
              t: 0 1 0 1 0 1 0 0 1
LMS-substrings: coc , con , nut0 , 0
```

[1] Only the journal versions of articles reporting these algorithms are cited here.

The induced sorting method in SA-IS is a kind of bucket sorting developed in the context of SA construction. Given that a set of elements are sorted by their keys into an array, each subset of elements of equivalent keys must locate consecutively in a sub-array called a bucket. If we sort all the characters of $T$ into $SA$, we will see a set of *buckets in* $SA$, where each bucket comprises a set of equivalent characters. Hence, if we lexicographically sort all the suffixes of $T$ into $SA$, then all the suffixes with a common first character must fall into the bucket for their first characters. Let $\mathsf{bucket}(SA, T, i)$ denote the bucket in $SA$ for character $T[i]$ as well as suffix $\mathsf{suf}(T, i)$. Furthermore, the first and the last items of a bucket are called the *start* and the *end* of the bucket, respectively.

An important property utilized to develop the linear-time algorithms in [Ko and Aluru 2005; Nong et al. 2011] is that in each bucket in $SA$, an L-type suffix of $T$ must be lexicographically less than and hence locate before an S-type suffix. This property was exploited by SA-IS for induced sorting of both the sampled substrings and the suffixes at each recursion level. The induced sorting algorithms in SA-IS are bucket sorting in principle, using a bucket counter array $bkt$ for keeping track of the status of each bucket on-the-fly. The term **"induced sorting"** is coined to reflect that the order of suffixes of a string is induced from that of another string that is at least half shorter. The new linear-time algorithm SACA-K is developed based on a novel naming method different from that in SA-IS. Such a naming method enables us to design SACA-K with the following distinct advantages over SA-IS: (1) type array $t$ is not needed at all; and (2) bucket counter array $bkt$ is needed only at the top recursion level. As a result, the workspace is deterministic $K$ words for computing the suffix array of input string $T$, where the **workspace** is the space needed beyond the input $T$ and the output $SA$. Therefore, for any $n$-character string $T$ over a constant alphabet of size $K = O(1)$, SACA-K solves the problem in $O(n)$ time and $O(1)$ workspace.[2]

In the rest of this article, we present the SACA-K algorithm framework in Section 2, and explain the underlying ideas in Sections 3-5. The practical time and space performance of SACA-K are evaluated by experiments on a set of typical corpora in Section 6, and the main results are summarized in Section 7.

## 2. SACA-K

### 2.1. Framework

Fig. 1 shows the framework of SACA-K. Similar to SA-IS in [Nong et al. 2011], SACA-K first samples all the LMS-substrings of $T$, and sorts them, then replaces each LMS-substring by an integer name to produce a shortened string $T_1$ (which is at least 1/2 shorter than $T$, i.e. $n_1 \leq \lfloor n/2 \rfloor$, see Lemma 3.5 in [Nong et al. 2011]) for computing the SA of $T$ recursively. Both SA-IS and SACA-K sample the same set of LMS-substrings to compute the new shortened string $T_1$.[3] As a result, SACA-K will output the same SA as that from SA-IS, in the same time complexity of $\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$ as that of SA-IS too. The major improvement of SACA-K over SA-IS lies in the reduced workspace. The design of SA-IS uses a workspace reserved for bucket counter array $bkt$ and type array $t$ linear to $n$. However, SACA-K uses only a deterministic workspace which is solely decided by $K$ instead.

[2]If not specified explicitly, a space is measured in $\log n$ bits, as commonly adopted in the literature for SACAs, e.g. [Franceschini and Muthukrishnan 2007; Kärkkäinen et al. 2006]. In [Franceschini and Muthukrishnan 2007], a SACA is said to be "in-place" if it uses $O(1)$ workspace.

[3]SACA-K names the sorted LMS-substrings by a new method (presented in Section 5) different from that in SA-IS. Hence, the string $T_1$ produced in SACA-K may be different from that in SA-IS, although both are of the same length $n_1$.

```
SACA-K(T, SA, K, n, level)
      ▷ T: input string;
      ▷ SA: suffix array of T;
      ▷ K: alphabet size of T;
      ▷ n: size of T;
      ▷ level: recursion level;

      ▷ Stage 1: induced sort the LMS-substrings of T.
 1    if level = 0
        then
 2          Allocate an array of K integers for bkt;
 3          Induced sort all the LMS-substrings of T, using bkt for bucket counters;
        else
 4          Induced sort all the LMS-substrings of T, reusing the start or
            the end of each bucket as the bucket's counter;

      ▷ SA is reused for storing T_1 and SA_1.

      ▷ Stage 2: name the sorted LMS-substrings of T.
 5    Compute the lexicographic names for the sorted LMS-substrings to produce T_1;

      ▷ Stage 3: sort recursively.
 6    if K_1 = n_1 ▷ each character in T_1 is unique.
        then
 7          Directly compute SA(T_1) from T_1;
        else
 8          SACA-K(T_1, SA_1, K_1, n_1, level + 1);

      ▷ Stage 4: induced sort SA(T) from SA(T_1).
 9    if level = 0
        then
10          Induced sort SA(T) from SA(T_1), using bkt for bucket counters;
11          Free the space allocated for bkt;
        else
12          Induced sort SA(T) from SA(T_1), reusing the start or the end of
            each bucket as the bucket's counter;
13    return ;
```

Fig. 1. The algorithm framework of SACA-K.

Some more notations are introduced here for further presentation of SACA-K. To denote a symbol in SACA-K at level $l \geq 0$, we add " $(l)$" to the symbol's subscript, e.g. $T_{(l)}$ and $T_{1(l)}$ for $T$ and $T_1$ at level $l$, respectively. Further, let $\mathsf{SA}(T_{(l)})$ denote the suffix array of $T_{(l)}$, and $SA_{(l)}$ be the space for storing $\mathsf{SA}(T_{(l)})$. That is, the notation $\mathsf{SA}(T_{(l)})$ means that all the suffixes of $T_{(l)}$ are already sorted and stored in $SA_{(l)}$; however, the notation $SA_{(l)}$ means *only* the space for storing $\mathsf{SA}(T_{(l)})$, regardless of what and how the data are stored. Notice that due to the recursion, $T_{1(l)}$ and $SA_{1(l)}$ are actually $T_{(l+1)}$ and $SA_{(l+1)}$, respectively.

### 2.2. Reusing $SA_{(0)}$

The space of $SA$ at level 0, i.e. $SA_{(0)}$, is reused throughout all the recursion levels of SACA-K. In Fig. 2, the upper and the lower 3 rows show the statuses of $SA_{(0)}$ immediately before and after the recursive call at line 8, in problem reduction (i.e. Stage 1-2) and solution induction (i.e. Stage 4) at levels 0-2, respectively.

At level 0 shown in the first row, $T_{(1)}$ (i.e. $T_{1(0)}$) is stored in the rightmost $n_{(1)}$ items in $SA_{(0)}$ (i.e. $SA_{(0)}[n_{(0)}-n_{(1)}, n_{(0)}-1]$), where $n_{(l)}$ is the size of $T_{(l)}$, and the first $n_{(0)}-n_{(1)} \geq n_{(1)}$ items in $SA_{(0)}$ are unoccupied and can be reused for $SA_{(1)}$ (recalling $n_{(l)} \geq 2n_{1(l)}$ at each level $l$). At level 1 shown in the 2nd row, $T_{(2)}$ is stored immediately on the left hand side of $T_{(1)}$, and the leftmost $n_{(0)} - n_{(1)} - n_{(2)} \geq n_{(2)}$ items are free and can be reused for $SA_{(2)}$. We keep on recursively reducing the string from level to level. At level $l$, the sub-array $SA_{(0)}[0, n_{(l+1)} - 1]$ is always free, and is enough for the space required for $SA_{(l+1)}$.

Suppose that we are at line 9 (in Fig. 1) for level 2. At this point, $\mathsf{SA}(T_{(3)})$ has been computed and stored in $SA_{(3)}$ which is reusing $SA_{(0)}[0, n_{(3)} - 1]$ as shown by row 4 (in Fig. 2). Then in line 12, $\mathsf{SA}(T_{(2)})$ is induced from $\mathsf{SA}(T_{(3)})$ and stored in $SA_{(2)}$ which is reusing $SA_{(0)}[0, n_{(2)} - 1]$. Further in line 13, we return to the upper recursion level and reach line 9 for level 1, and now the status of $SA_{(0)}$ is shown by row 5. Then, we continue to compute $\mathsf{SA}(T_{(1)})$ from $\mathsf{SA}(T_{(2)})$ by line 12, and get $\mathsf{SA}(T_{(1)})$ stored in $SA_{(1)}$ shown in the last row when we reach line 9 at level 0. Finally, $\mathsf{SA}(T_{(0)})$ is induced from $\mathsf{SA}(T_{(1)})$ by line 10 to produce the output suffix array.

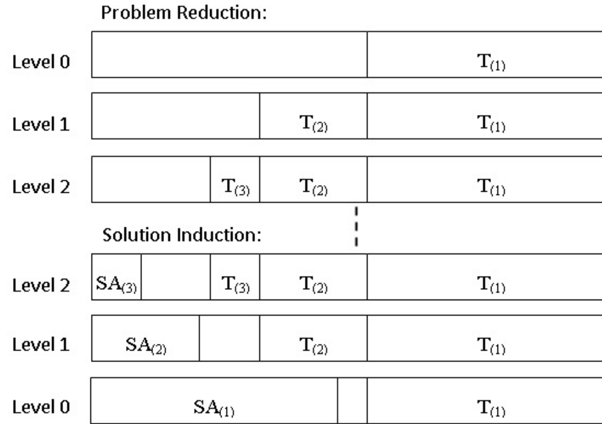


Fig. 2. Reusing $SA_{(0)}$ in SACA-K.

### 2.3. Induced Sorting

After level 0, SACA-K follows a common execution path for levels 1, 2 and etc. Hence, it is enough for us to explain SACA-K for levels 0 and 1 only. The details of the algorithm for induced sorting the suffixes at levels 0 and 1 are different, however, they can be fit into the following common algorithm framework. At each level, provided that all the LMS-suffixes of $T$ have been sorted and stored in $SA_1$ which is reusing $SA[0, n_1 - 1]$, we can perform the induced sorting of all the suffixes of $T$ by this 4-step procedure:

(1) Initialize each item of $SA[n_1, n - 1]$ as empty;
(2) Scan $SA[0, n_1 - 1]$ once from right to left to put all the sorted LMS-suffixes of $T$ into their buckets in $SA$, from the end to the start in each bucket.
(3) Scan $SA$ once from left to right. For each non-empty $SA[i]$, $j = SA[i] - 1$, if $T[j]$ is L-type, then put $\mathsf{suf}(T, j)$ into the current leftmost empty position in $\mathsf{bucket}(SA, T, j)$.

(4) Scan $SA$ once from right to left. For each non-empty $SA[i]$, $j = SA[i] - 1$, if $T[j]$ is S-type, then put $\mathsf{suf}(T, j)$ into the current rightmost empty position in $\mathsf{bucket}(SA, T, j)$.

The above algorithm can also be reused to induce the sorting of all the LMS-substrings of $T$, by keeping the last two steps unchanged and modifying the first two steps as follows:

(1) Initialize each item of $SA[0, n-1]$ as empty;
(2) Scan $T$ once from right to left to put all the LMS-substrings of $T$ into the buckets for their first characters, i.e., $\mathsf{lms}(T, i)$ is put into $\mathsf{bucket}(SA, T, i)$, from the end to the start in each bucket.

Each step of the aforementioned algorithms for induced sorting suffixes and LMS-substrings clearly runs in $O(n)$ time, resulting in a total time complexity of $O(n)$ for both algorithms. Because there is no bucket counter array $bkt$ for level 1, the last 3 steps for the induced sorting algorithms on levels 0 and 1 are different. Specifically, the major difference is in the last 2 steps for: (1) how to determine the type of $T[j]$ when we are scanning a non-empty item $SA[i]$; and (2) how to keep track of the current leftmost or rightmost positions of each bucket. Since the last 2 steps for induced sorting suffixes and LMS-substrings at each level are identical, and the first 2 steps are straightforward, we will concentrate on presenting the algorithms for induced sorting suffixes at levels 0 and 1, respectively.

## 3. SORTING SUFFIXES AT LEVEL 0

Different from SA-IS where an $n$-bit type array $t$ is available at each level for induced sorting suffixes, there is no $t$ in SACA-K, neither implicitly nor explicitly. Under this constraint, the general algorithm for induced sorting suffixes in Section 2.3 is further developed as follows:

(1) Initialize each item of $SA[n_1, n-1]$ as empty.
(2) Compute into $bkt[0, K-1]$ the end position of each bucket in $SA$. Scan $SA[0, n_1 - 1]$ once from right to left to put all the sorted LMS-suffixes of $T$ into their buckets in $SA$, from the end to the start in each bucket, in this way: for each scanned item $SA[i]$, $j = SA[i]$ and $c = T[j]$, set $SA[i]$ as empty, $SA[bkt[c]] = j$ and decrease $bkt[c]$ by 1.
(3) Compute into $bkt[0, K-1]$ the start position of each bucket in $SA$. Scan $SA$ once from left to right to induced sort the L-type suffixes of $T$ into their buckets in $SA$, from the start to the end in each bucket, in this way: for each scanned non-empty item $SA[i]$, $j = SA[i] - 1$ and $c = T[j]$, if $T[j]$ is L-type, then set $SA[bkt[c]] = j$ and increase $bkt[c]$ by 1.
(4) Compute into $bkt[0, K-1]$ the end position of each bucket in $SA$. Scan $SA$ once from right to left to induced sort the S-type suffixes of $T$ into their buckets in $SA$, from the end to the start in each bucket, in this way: for each scanned non-empty item $SA[i]$, $j = SA[i] - 1$ and $c = T[j]$, if $T[j]$ is S-type, then set $SA[bkt[c]] = j$ and decrease $bkt[c]$ by 1.

In the last two steps of the above algorithm, how to determine the type of $T[j]$ without type array $t$? For step 3, because each non-empty item $SA[i]$ stores either an LMS-suffix or an L-type suffix, $T[j]$ must be L-type for $T[j] \geq T[SA[i]]$. However, for step 4, we need to utilize the following property to help determine if $T[j]$ is S-type or not when we see $T[j] = T[SA[i]]$. In this property, $bkt[T[j]] < i$ means that the newly induced S-type suffix must be stored into an item in front of (i.e. on the left hand side of) the non-empty item $SA[i]$ that we are currently scanning.

PROPERTY 3.1. *At level $l = 0$, when induced sorting the S-type suffixes of $T$ from the sorted L-type suffixes, for each non-empty $SA[i]$ and $j = SA[i] - 1$, $\mathsf{suf}(T, j)$ is S-type if and only if: (i) $T[j] < T[SA[i]]$ or (ii) $T[j] = T[SA[i]]$ and $bkt[T[j]] < i$.*

For the SA-IS algorithm, there is an optimized implementation by Yuta Mori at https://sites.google.com/site/yuta256/sais/. In Mori's program, a technique similar to Property 3.1 is employed to remove the array $t$. As seen in file `saic.c` of package `sais-lite-2.4.1`, when an S-type suffix is induced into $SA[j]$, the highest bit of $SA[j]$ is reset as 0 if $\mathsf{suf}(T, SA[j] - 1)$ is detected as S-type for $T[SA[j] - 1] \leq T[SA[j]]$, or else set as 1. Later on, when we further scan to $SA[j]$, we can determine whether $\mathsf{suf}(T, SA[j] - 1)$ is S-type or not by simply checking the highest bit of $SA[j]$.[4] Such a technique occupies the highest bit of each $SA[j]$ to mark whether $\mathsf{suf}(T, SA[j] - 1)$ is S-type or not. However, in Property 3.1, we do not use, by any means, any space in $SA$ for removing the array $t$. This is the difference between Mori's technique and ours, and is critical for SACA-K to achieve $K$-word workspace. Regardless of this difference, SACA-K is distinct from all the known SACAs, by the new naming rule for LMS-substrings proposed in the sequel.

## 4. SORTING SUFFIXES AT DEEPER LEVELS

At each recursion level of SACA-K, bucket sorting is employed for induced sorting of both LMS-substrings and suffixes. At level 0, we use $K$ words to store a bucket counter array $bkt[0, K - 1]$ for induced sorting when reducing $T$ into $T_1$, as well as augmenting $\mathsf{SA}(T_1)$ to $\mathsf{SA}(T)$. However, at level 1, if we still use a specific bucket counter array for bucket sorting, the bucket counter array will require $O(n)$ words. In order to achieve a workspace of $K$ words for the whole algorithm, no specific bucket counter array is allowed for bucket sorting at levels 1, 2 and thereafter. Fortunately, we have found a novel way for induced sorting using no specific bucket counter array, in case of the following property is held.

PROPERTY 4.1. *At level $l > 0$, each L-type or S-type character in $T$ itself also points to the start or the end of its bucket in $SA$, respectively.*

In Section 5, we will show how to produce $T$ with this property. Now, given this property for $T$ at level 1, we show how to compute $\mathsf{SA}(T)$ without using a specific bucket counter array.

### 4.1. Induced Sorting L-Type Suffixes

Without the bucket counter array $bkt$ that we had for induced sorting the L-type suffixes at level 0 in Section 3, the algorithm for induced sorting the L-type suffixes at level 1 relies on Property 4.1. The key idea is to reuse the start item of each bucket in $SA$ to maintain a counter for tracking the location where an L-type suffix being sorted into this bucket should be stored. At any level $l > 0$, each item of $SA$ is reusing an item of $SA_{(0)}$ and the highest bit in each item is not needed to store the index for a suffix in $T$ (due to $n_1 \leq \lfloor n/2 \rfloor$ at each level). Hence, at level $l > 0$, the highest bit of $SA[i]$ is always available to be used for indicating what data is currently stored in the rest bits of $SA[i]$: 0 for a suffix index, 1 for a bucket counter or empty value.

At the beginning of line 12 in Fig. 1, an item in $SA$ may be empty (marked by the least negative integer denoted by EMPTY) or store the index of an LMS-suffix in $T$, and all the LMS-suffixes stored in $SA$ have been sorted in their correct order. To induced

[4]This can also speed up the running process, because it avoids one random access to array $t$ for the type of $\mathsf{suf}(T, SA[j] - 1)$). The array $t$ previously in SA-IS is now replaced by another cache-friendly sparse $n$-bit array consisting of the highest bits of $SA$.

sort all the L-type suffixes, we scan $SA$ once from left to right to do as follows. For each $SA[i] > 0$ being scanned, $j = SA[i] - 1$, if $T[j]$ is L-type (in this case, $T[j]$ is L-type if $T[j] \geq T[j+1]$), we will put $\mathsf{suf}(T, j)$ into its bucket in $SA$. Recalling that $T$ in this case holds Property 4.1, so $T[j]$ points to the start of its bucket in $SA$. That is, let $c = T[j]$, the start of $\mathsf{bucket}(SA, T, j)$ is $SA[c]$. To indicate an item in $SA$ is being reused as a bucket counter, the value stored in this item is set as a non-empty negative value. Now, we check the value of $SA[c]$ for these cases:

(1) If $SA[c]$ is empty, then $\mathsf{suf}(T, j)$ is the first suffix being put into its bucket. In this case, we further check $SA[c+1]$ to see if it is empty or not. If it is, we sort $\mathsf{suf}(T, j)$ into $SA[c+1]$ by setting $SA[c+1] = j$ and start to reuse $SA[c]$ as a counter by setting $SA[c] = -1$. Otherwise, $SA[c+1]$ may be non-negative for a suffix index or negative for a counter, and $\mathsf{suf}(T, j)$ must be the only element of its bucket, we hence simply put $\mathsf{suf}(T, j)$ into its bucket by setting $SA[c] = j$.
(2) If $SA[c]$ is non-negative, then $SA[c]$ is "borrowed" by the left-neighboring bucket (of $\mathsf{bucket}(SA, T, j)$). In this case, $SA[c]$ is storing the largest item in the left-neighboring bucket, and we need to shift-left one step of all the items in the left-neighboring bucket to their correct locations in $SA$. The start item of the left-neighboring bucket can be found by scanning from $SA[c]$ to the left, until we see the first item $SA[x]$ that is negative for being reused as a counter. That is, $x$ is the largest for $SA[x] < 0$, $SA[x] \neq \mathsf{EMPTY}$ and $x < c$. Having found $SA[x]$, we shift-left one step all the items in $SA[x+1, c]$ to $SA[x, c-1]$, and set $SA[c]$ as empty. Now, we see the same condition as that in case 1, hence the operations in case 1 are performed to further sort $\mathsf{suf}(T, j)$ into its bucket.
(3) If $SA[c]$ is negative and non-empty, then $SA[c]$ is being reused as a counter for $\mathsf{bucket}(SA, T, j)$. In this case, let $d = SA[c]$ and $pos = c - d + 1$, then $SA[pos]$ is the item that $\mathsf{suf}(T, j)$ should be stored into. However, $\mathsf{suf}(T, j)$ may be the largest suffix in its bucket. Therefore, we further check the value of $SA[pos]$ to proceed as follows. If $SA[pos]$ is empty, we simply put $\mathsf{suf}(T, j)$ into its bucket by setting $SA[pos] = j$, and increase the counter of its bucket by 1, i.e. $SA[c] = SA[c] - 1$ (notice that $SA[c]$ is negative for a counter). Otherwise, it indicates that $SA[pos]$ is the start item of the right-neighboring bucket, which must be currently non-negative for a suffix index or negative for a counter. Hence, we need to shift-left one step the items in $SA[c+1, pos-1]$ to $SA[c, pos-2]$, then sort $\mathsf{suf}(T, j)$ into its bucket by setting $SA[pos-1] = j$.

In the algorithm described above, because we reuse the start item of a bucket as a counter for recording how many L-type suffixes are already stored in the bucket, it is possible that the largest suffix of a bucket is temporarily put into the start item of its right-neighboring bucket. In other words, the rightmost item of a bucket runs into the start item of the right-neighboring bucket. Hence, in case 2, if we see $SA[c]$ non-negative for a suffix index, it means that $SA[c]$ is borrowed by the largest suffix in the left-neighboring bucket (of $\mathsf{bucket}(SA, T, j)$). Hence, we need to adjust all the items of the left-neighboring bucket to their correct locations. This is done by shifting left one step all the items in the left-neighboring bucket, where the start of the left-neighboring bucket is currently the first non-empty negative item in front of $SA[c]$. Notice that in cases 2 and 3, the suffixes in a bucket are shifted left only when the bucket is fully filled. In other words, no other suffix will be sorted into the bucket thereafter. Hence, the counter for this bucket is not needed any more. Shifting left a bucket in case 3 is simpler than that in case 2, for we have already known the exact positions for the first and the last items of the bucket.

The time complexity of this algorithm is determined by the loop for scanning $SA$ once to perform the induced sorting operations. Each iteration of this loop will sort at

most an L-type suffix into $SA$, and each L-type suffix already sorted into $SA$ can be shifted at most once. Hence, this loop has a time complexity dominated by the loop's size, i.e. $O(n)$.

### 4.2. Induced Sorting S-Type Suffixes

Given all the L-type suffixes of $T$ are already sorted into their correct positions in $SA$, we can scan $SA$ once from right to left to induced sort all the S-type suffixes. When induced sorting the L-type suffixes, the start item of each bucket is reused as a counter for the bucket. However, to induced sort the S-type suffixes, because we are now scanning $SA$ in a reverse direction, i.e. from right to left, and each S-type character of $T$ points the end of its bucket in $SA$, it is now the end item instead of the start item of a bucket is reused as the counter for the bucket. Hence, with some minor and symmetric changes to that for induced sorting the L-type suffixes, here comes the algorithm for inducing the order of S-type suffixes from the sorted L-type suffixes.

We scan $SA$ once from right to left to do as follows. For each $SA[i] > 0$ being scanned, $j = SA[i] - 1$, we first check if $T[j]$ is S-type or not, using Property 4.2. In this property, case (ii) means that the newly induced S-type suffix must be stored into an item in front of (i.e. on the left hand side of) the item $SA[i]$ that we are currently scanning. Now in $T$, a characters itself also points to either the start or the end of its bucket in $SA$. Hence, when we see $T[j] = T[SA[i]]$ and $T[j] > i$, then $T[j]$ must point to the end of $\mathsf{bucket}(SA, T, j)$. This implies that $T[j]$ must be S-type, because Property 4.1 is now held by $T$.

PROPERTY 4.2. *At level $l > 0$, when induced sorting the S-type suffixes of $T$ from the sorted L-type suffixes, for each $SA[i] > 0$ and $j = SA[i] - 1$, $\mathsf{suf}(T, j)$ is S-type if and only if: (i) $T[j] < T[SA[i]]$ or (ii) $T[j] = T[SA[i]]$ and $T[j] > i$.*

If $T[j]$ is S-type, we will put $\mathsf{suf}(T, j)$ into its bucket in $SA$. Recalling that $T$ in this case holds Property 4.1, so $T[j]$ points to the end of its bucket in $SA$. That is, let $c = T[j]$, the end of $\mathsf{bucket}(SA, T, j)$ is $SA[c]$. Now, we check the value of $SA[c]$ for these cases:

(1) If $SA[c]$ is empty, then $\mathsf{suf}(T, j)$ is the first suffix being put into its bucket. In this case, we further check $SA[c - 1]$ to see if it is empty or not. If it is, we sort $\mathsf{suf}(T, j)$ into $SA[c - 1]$ by setting $SA[c - 1] = j$ and start to reuse $SA[c]$ as a counter by setting $SA[c] = -1$. Otherwise, $SA[c - 1]$ may be non-negative for a suffix index or negative for a counter, and $\mathsf{suf}(T, j)$ must be the only element of its bucket, we hence simply put $\mathsf{suf}(T, j)$ into its bucket by setting $SA[c] = j$.
(2) If $SA[c]$ is non-negative, then $SA[c]$ is "borrowed" by the right-neighboring bucket (of $\mathsf{bucket}(SA, T, j)$). In this case, $SA[c]$ is storing the smallest item in the right-neighboring bucket, and we need to shift-right one step all the items in the right-neighboring bucket to their correct locations in $SA$. The end item of the right-neighboring bucket can be found by scanning from $SA[c]$ to the right, until we see the first item $SA[x]$ that is negative for being reused as a counter. That is, $x$ is the smallest for $SA[x] < 0$, $SA[x] \neq \mathsf{EMPTY}$ and $x > c$. Having found $SA[x]$, we shift-right one step all the items in $SA[c, x - 1]$ to $SA[c + 1, x]$, and set $SA[c]$ as empty. Now, we see the same condition as that in case 1, hence the operations in case 1 are performed to further sort $\mathsf{suf}(T, j)$ into its bucket.
(3) If $SA[c]$ is negative and non-empty, then $SA[c]$ is reused as a counter for $\mathsf{bucket}(SA, T, j)$. In this case, let $d = SA[c]$ and $pos = c + d - 1$, then $SA[pos]$ is the item that $\mathsf{suf}(T, j)$ should be stored into. However, $\mathsf{suf}(T, j)$ may be the smallest S-type suffix in its bucket. Therefore, we further check the value of $SA[pos]$ to proceed as follows. If $SA[pos]$ is empty, we simply put $\mathsf{suf}(T, j)$ into its bucket by setting $SA[pos] = j$, and increase the counter of its bucket by 1, i.e. $SA[c] = SA[c] - 1$

(notice that $SA[c]$ is negative for a counter). Otherwise, $SA[pos]$ must be currently non-negative for a suffix index or negative for a counter. Hence, we need to shift-right one step the items in $SA[pos+1, c-1]$ to $SA[pos+2, c]$, then sort $\mathsf{suf}(T, j)$ into its bucket by setting $SA[pos+1] = j$.

### 5. NAMING SORTED LMS-SUBSTRINGS

We now describe how to calculate the names for the sorted LMS-substrings of $T$ to get a new reduced string $T_1$ (which is also the input string $T$ at the next recursion level) with Property 4.1.

Define s-rank and se-rank of a character in $T$ as follows. The s-rank of $T[i]$ is the number of characters in $T$ smaller than $T[i]$, and the se-rank of $T[i]$ is the number of characters in $T$ smaller than or equal to $T[i]$ (excluding $T[i]$ itself), respectively. Given that all the LMS-substrings of $T$ have been sorted into $SA_1$, we use the following novel naming method to produce $T_1$ in time $O(n)$. Notice that in this section, each set of identical LMS-substrings in $T$ constitutes a **substring bucket** in $SA_1$, such a bucket definition for LMS-substrings is different from that for suffixes and characters in Section 1.

(1) Scan $SA_1$ once from left to right to name each LMS-substring of $T$ by the start position of the substring's bucket in $SA_1$, resulting in an interim reduced string denoted by $Z_1$ (where each character points to the start of its bucket in $SA_1$);
(2) Scan $Z_1$ once from right to left to replace each S-type character in $Z_1$ by the end position of its bucket in $SA_1$, resulting in the new string $T_1$. (Notice that in this step, the type of each character in $Z_1$ can be determined on-the-fly when $Z_1$ is being scanned from right to left.)

This naming method is different from that used in SA-IS. Naming the LMS-substrings of $T$ in this way, in the new string $T_1$, each L-type or S-type character itself is also its s-rank or se-rank in $T_1$, respectively. As a result, we now get the reduced string $T_1$, in which each L-type or S-type character points to the start or the end of the character's bucket in $SA_1$, respectively. However, there is still a problem to be solved in this naming algorithm. To detect the start of each bucket in the first step, we need to compare any two neighboring LMS-substrings of $T$ stored in $SA_1$. Without type array $t$, how to determine the ends of two LMS-substrings when they are compared? Because the type of $\mathsf{suf}(T, i-1)$ relies on the type of $\mathsf{suf}(T, i)$ when $T[i-1] = T[i]$ (see Section 1), there is a difficulty in determining the end of an LMS-substring when traversing from the start of the LMS-substring. However, fortunately, we can still traverse an LMS-substring from its start to detect its end by utilizing the following observation.

An LMS-substring has a type pattern governed by this regular expression $S^+L^+S$, where $S^+$ and $L^+$ mean a string of one or multiple S-type and L-type characters, respectively. In other words, an LMS-substring consists of three segments in sequence: one or multiple S-type characters, one or multiple L-type characters, and a single S-type character. Suppose that we are going to retrieve $\mathsf{lms}(T, x)$ from its start character $T[x]$, $\mathsf{lms}(T, x)$ together with its succeeding LMS-substring will follow such a pattern $S^+L^+S^+L^+S$ (notice that any two neighboring LMS-substrings must overlap on a common LMS-character). This fact is utilized to design the following 2-step algorithm for retrieving $\mathsf{lms}(T, x)$ from $T[x]$:

(1) Traverse the LMS-substring from its first character $T[x]$ until we see a character $T[x+i]$ less than its preceding $T[x+i-1]$. Now, $T[x+i-1]$ must be L-type.
(2) Continue to traverse the remaining characters of the LMS-substring and terminate when we see a character $T[x+i]$ greater than its preceding $T[x+i-1]$ or $T[x+i]$ is the sentinel. At this point, we know that the start of the succeeding LMS-substring

has been traversed and its position was previously recorded when we saw $T[x+i] < T[x+i-1]$ the last time.

Consider the following example for the above algorithm. Suppose that we have two neighboring LMS-substrings ″suffix0″, where the first and second LMS-substrings are ″suf″ and ″ffix0″, respectively. Starting from the character ″s″, the first step traverses the character ″u″, then breaks when the first character ″f″ is seen, for ″f″ < ″u″. Further in the 2nd step, the next two characters ″f″, ″i″ are traversed. When the first ″f″ is visited, its position is saved, for ″f″ < ″u″ and it is probably the start of the 2nd LMS-substring. However, when the 2nd ″f″ is approached, we do not save its position, for it must not be the start of the 2nd LMS-substring (suppose that it is, then the first ″f″ must be S-type and hence the start of the 2nd LMS-substring instead, resulting in a contradiction). When we reach the character ″i″, because ″i″ > ″f″, the traversal is terminated and the first ″f″ is confirmed to be the end of the first LMS-substring.

### 5.1. Correctness

In the SA-IS algorithm [Nong et al. 2011], having sorted and stored in $SA_1$ all the LMS-substrings of $T$, we name each LMS-substring by the *index of its bucket* in $SA_1$ to produce the reduced string called $Y_1$ here, where the buckets in $SA_1$ are indexed from 0. If we name each LMS-substring by the *start position of its bucket* instead to produce another string $Y_2$ (i.e. $Z_1$ in our new naming algorithm), then for any $Y_1[i] < Y_1[j]$ or $Y_1[i] = Y_1[j]$, we must have $Y_2[i] < Y_2[j]$ or $Y_2[i] = Y_2[j]$, respectively. Therefore $\mathsf{SA}(Y_1)$ and $\mathsf{SA}(Y_2)$ must be identical. Further, we rename each S-type character in $Y_2$ by the end position of its bucket instead to produce yet another string called $Y_3$. Now for any $Y_2[i] < Y_2[j]$, there must be $Y_3[i] < Y_3[j]$. In case of $Y_2[i] = Y_2[j]$, we further consider two more cases in respect to whether the types of $Y_2[i]$ and $Y_2[j]$ are the same. If so, we must have $Y_3[i] = Y_3[j]$; or else without loss of generality, suppose $Y_2[i]$ and $Y_2[j]$ are L-type and S-type, respectively, we must have $Y_3[i] < Y_3[j]$, $\mathsf{suf}(Y_2, i) < \mathsf{suf}(Y_2, j)$ and $\mathsf{suf}(Y_3, i) < \mathsf{suf}(Y_3, j)$. Hence $\mathsf{SA}(Y_2)$ and $\mathsf{SA}(Y_3)$ must be identical too. Given $\mathsf{SA}(Y_1)$ and $\mathsf{SA}(Y_3)$ are identical, because $Y_1$ and $Y_3$ are in effect $T_1$, as produced by SA-IS and SACA-K, respectively, we get that $\mathsf{SA}(T_1)$ and therefore $\mathsf{SA}(T)$ computed by both algorithms must be identical.

## 6. PERFORMANCE

Four programs are used in this performance evaluation experiment: `saca-k`, `sa-is`, `sais-lite` and `divsort`. The first two were made by us for the algorithms SACA-K and SA-IS, respectively; the last two were made by Yuta Mori: `sais-lite-2.4.1` at https://sites.google.com/site/yuta256/sais/ and `libdivsufsort-2.0.1` at http://code.google.com/p/libdivsufsort/, respectively. The first three are linear-time (`sais-lite` is an optimized implementation of `sa-is`, so it is linear-time too), only `divsort` has a super-linear worst-case time of $O(n \log n)$ (stated in README of `libdivsufsort-2.0.1`). For each input string in this experiment, all the outputs from these four programs were compared to be identical to ensure that all these programs worked correctly.

The experiment was performed on a notebook with configuration: 1 Intel(R) Core i3-370M Processor (2.4GHz, Dual Core, 3MB L3), 4GB 1333MHz DDR3 SDRAM, CentOS 6.3 (Final) 64-bit. Specifically, `divsort` and `sais-lite` were compiled using the default `makefile` provided in their source packages, and our two programs `saca-k` and `sa-is` were compiled by `g++` with options "`-fomit-frame-pointer -W -Wall -Winline -DNDBUG -O3`". Our source packages for `saca-k` and `sa-is` are publicly available at http://code.google.com/p/ge-nong/.

Table I lists the datasets used in this experiment, they are a subset of the Pizza Chili corpus at http://pizzachili.dcc.uchile.cl/. The first 4 are from the main text corpus,

and the last 4 are from the highly repetitive corpus. The investigated performance measures are the time and space consumptions for each algorithm running on the datasets. With these settings, in the design of the four programs, each integer takes 4 bytes and each character of an input string takes 1 byte. The theoretical maximum workspace in bytes for each program is given as follows: $4K$ for `saca-k`, $2.125n$ for `sa-is`, $\max\{4096, 2n\}$ for `sais-lite`, $O(1)$ for `divsort` (the total space is given as $5n + O(1)$ bytes in README of `libdivsufsort-2.0.1`).

Table I. Corpora. One byte per character.

| Corpus | $n$ | $K$ | Description |
|---|---|---|---|
| dna | 403,927,746 | 16 | A sequence of newline-separated gene DNA sequences from the Gutenberg Project. |
| english.600MB | 629,145,600 | 239 | The 600MB-prefix of the original corpus "english" which is the concatenation of English text files from the Gutenberg Project. |
| proteins.600MB | 629,145,600 | 27 | The 600MB-prefix of the original corpus "proteins" which is a sequence of newline-separated protein sequences from the Swissprot database. |
| sources | 210,866,607 | 230 | A file formed by C/Java source code by concatenating all the files of the linux-2.6.11.6 and gcc-4.0.0 distributions. |
| cere | 461,286,644 | 5 | A file containing 37 sequences of Saccharomyces Cerevisiae. |
| einstein.en.txt | 467,626,544 | 139 | The English article of Albert Einstein downloaded up to November 10, 2006. |
| fib41 | 267,914,296 | 2 | Fibonacci sequence. |
| kernel | 257,961,616 | 160 | A collection of all 1.0.x and 1.1.x versions of the Linux Kernel6. |

Table II. Workspace in bytes. The smallest workspace results are always achieved by `saca-k`; while the workspace results for `sa-is` are linear to $n$ and the largest.

| Corpus | divsort | sais-lite | sa-is | saca-k |
|---|---|---|---|---|
| dna | 263,168 | 4,096 | 148,438,208 | **1,029** |
| english.600MB | 263,168 | 4,096 | 212,770,873 | **1,029** |
| proteins.600MB | 263,168 | 4,096 | 235,596,366 | **1,029** |
| sources | 263,168 | 4,096 | 68,884,168 | **1,029** |
| cere | 263,168 | 4,096 | 82,561,594 | **1,029** |
| einstein.en.txt | 263,168 | 4,096 | 85,152,774 | **1,029** |
| fib41 | 263,168 | 4,096 | 54,186,838 | **1,029** |
| kernel | 263,168 | 4,096 | 46,767,439 | **1,029** |

Table III. Time in $\mu s/ch$. The mean speeds of `divsort` and `sais-lite` are very close and the fastest. The average speedup of `saca-k` over `sa-is` is $0.533/0.402 = 1.33$.

| Corpus | divsort | sais-lite | sa-is | saca-k |
|---|---|---|---|---|
| dna | **0.201** | 0.276 | 0.601 | 0.426 |
| english.600MB | **0.221** | 0.300 | 0.766 | 0.512 |
| proteins.600MB | **0.227** | 0.327 | 0.804 | 0.504 |
| sources | **0.121** | 0.177 | 0.334 | 0.287 |
| cere | 0.167 | **0.152** | 0.516 | 0.402 |
| einstein.en.txt | **0.149** | 0.154 | 0.348 | 0.310 |
| fib41 | 0.308 | **0.103** | 0.456 | 0.423 |
| kernel | **0.139** | 0.146 | 0.435 | 0.354 |
| mean | **0.192** | 0.204 | 0.533 | 0.402 |
| stdev | **0.061** | 0.084 | 0.178 | 0.082 |

### 6.1. Space

The workspace is obtained by subtracting $5n$ bytes (the necessary space for input and output) from the total space usage measured by command `memusage`. The workspace results measured in bytes for our experiments are shown in Table II. While the workspace of `sa-is` is linearly corpus size dependent, the workspace for each of the rest is a constant. The smallest workspace results are always achieved by `saca-k`: $256 \times 4 = 1024$ bytes, plus an extra integer to account for the sentinel, for a total of 1029 bytes.

### 6.2. Time

In Table III, each runtime in microseconds per character ($\mu s/ch$) is the mean of three runs measured using the `clock()` function of C to record only the time interval for computing the SA, which doesn't include the latency for reading the input corpus from disk and writing the output SA to disk. In the last two rows, the statistics of mean and standard deviation are given for the samples of each program. From the mean results, we have these observations: (1) `divsort` is the fastest; (2) the speed of `sais-lite` is very close to that of `divsort`; (3) `saca-k` takes twice the time of `divsort`; (4) `sa-is` is the slowest.

On two repetitive corpora “cere” and “fib41”, `sais-lite` is observed to be running faster than `divsort`. In particular, for “fib41”, the speedup of `sais-lite` over `divsort` is $0.308/0.103 = 2.99$. This is also an evidence for a well-known drawback of engineered super-linear time algorithms: their speeds are input dependent, and can become much slower than linear-time algorithms for some inputs.

From Table III, `saca-k` is observed to be running about 33% faster than `sa-is` on average, i.e. a mean speedup of $0.533/0.402 = 1.33$. The speed improvement is mainly due to that at each level $l > 0$, `saca-k` need not scan $T$ to find the start or the end of each bucket in $SA$, due to Property 4.1. However, `sa-is` needs to scan $T$ six times to compute the bucket counter array: three times for induced sorting the LMS-substrings, and three times for induced sorting the suffixes. As a summary, `saca-k` not only consumes less space than `sa-is`, but also runs faster.

In order to see the runtime for increasing file size, two files “english” and “proteins” were chosen to record the runtimes for each program on their increasing prefixes of sizes in MB: 10, 20, 40, 60, 100, 200, 400, 600. The time results in $\mu s/ch$ for these two files are shown in Fig. 3 and 4, respectively. A consistent trend for all the curves is that, when the size of input string increases, all programs slow down. The reason is when $n$ increases, more total space is needed by each program, which in turn causes the on-chip cache miss ratio to increase and results in a longer latency for random accesses of data from the main memory.

In Table III, Fig. 3 and 4, we have seen that the results for `divsort` and `sais-lite` are quite close. Because `sais-lite` is an optimized implementation of `sa-is` and `saca-k` is faster than `sa-is`, we believe that an optimized implementation of `saca-k` will have better time and space performance than `sais-lite` and hence runs in a speed even closer to that of `divsort`. We anticipate that such an optimized implementation can be engineered after the publication of this work.

## 7. CONCLUSION

Each step of SACA-K in Fig. 1 has a time complexity $O(n)$, so the total time remains linear as that of SA-IS, i.e. $\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$. For the space complexity of SACA-K, besides $T$ and $SA$, we have an additional array $bkt$ of $K$ words at recursion level 0 *only*. Hence we have the following result:

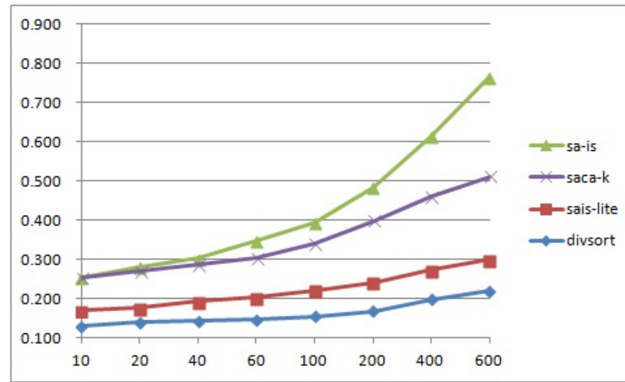


Fig. 3. Time in $\mu s/ch$ vs. prefix of “english” in MB.

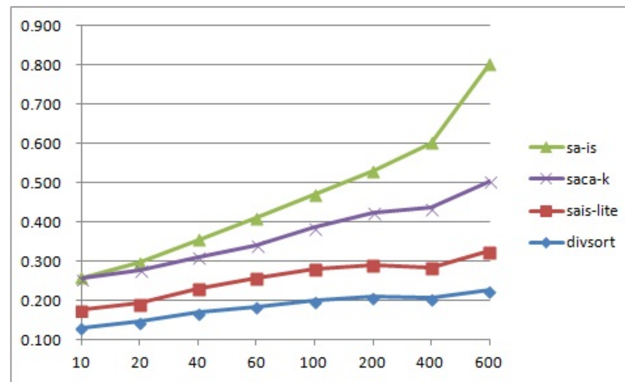


Fig. 4. Time in $\mu s/ch$ vs. prefix of “proteins” in MB.

LEMMA 7.1. *For a string $T[0, n-1]$ over an alphabet $A[0, K-1]$, SACA-K requires $O(n)$ time and $K$-word workspace for constructing the suffix array of $T$, where a word is $\log n$ bits.*

From Lemma 7.1, we have an immediate result: given $K = O(1)$, SACA-K runs in linear time and $O(1)$ workspace. To the best of our knowledge, such a result is the first reported in the literature with a practical source code publicly available.

Besides being used in SA construction, the idea of induced sorting suffixes has also been exploited to design algorithms for other problems, e.g. direct BWT computation using induced sorting by Okanohara and Sadakane [Okanohara and Sadakane 2009] and inducing the LCP-array by Fischer [Fischer 2011]. The methods proposed here might also be used to develop more time and space efficient algorithms for solving these problems.

Recently, some external memory (EM) SACAs have been proposed for constructing large SAs, where the space needed by an EM algorithm is mainly supplied by low-cost massive disks, e.g. bwt-disk [Ferragina et al. 2012] and DC3 [Dementiev et al. 2008]. In bwt-disk[5], the original input string is split into a number of blocks so that the BWT computation of each block can be completely executed in RAM. The whole BWT is built

[5]bwt-disk computes the Burrows-Wheeler Transform (BWT), however, it was also analyzed in [Ferragina et al. 2012] that bwt-disk can be adapted for SA construction.

incrementally, by first computing the solution for each block and then merging these solutions block-by-block. For a given input string, the speed of bwt-disk is inversely proportional to the number of blocks: a smaller number of blocks means a faster speed. To compute the BWT of each block, the SA of the block needs to be constructed using a SACA. Hence, efficient internal memory SACAs with good worst-case time and space performance, such as SACA-K, can also find an important role in the design of efficient EM algorithms for SA related problems.

## ACKNOWLEDGMENTS

The author wish to thank the reviewers and the editor for this article, Prof. Sen Zhang in the State University of New York College at Oneonta and Dr. Wai Hong Chan in the Hong Kong Institute of Education, for their constructive suggestions for improving the presentation of this paper.

## REFERENCES

S. Burkhardt and J. Kärkkäinen. 2003. Fast Lightweight Suffix Array Construction and Checking. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 2676. 55–69.

R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. 2008. Better External Memory Suffix Array Construction. *ACM Journal of Experimental Algorithmics* 12 (August 2008), 3.4:1–3.4:24.

P. Ferragina, T. Gagie, and G. Manzini. 2012. Lightweight Data Indexing and Compression in External Memory. *Algorithmica* 63, 3 (2012), 707–730.

J. Fischer. 2011. Inducing the LCP-Array. In *Algorithms and Data Structures*. Lecture Notes in Computer Science, Vol. 6844. 374–385.

G. Franceschini and S. Muthukrishnan. 2007. In-Place Suffix Sorting. In *Automata, Languages and Programming*. Lecture Notes in Computer Science, Vol. 4596. 533–545.

W. K. Hon, K. Sadakane, and W. K. Sung. 2003. Breaking a Time-and-Space Barrier for Constructing Full-Text Indices. In *Proceedings of FOCS'03*. 251–260.

H. Itoh and H. Tanaka. 1999. An efficient method for in memory construction of suffix arrays. In *Proceedings of SPIRE'99*. 81–88.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. 2006. Linear work suffix array construction. *JACM* 53, 6 (Nov. 2006), 918–936.

D. K. Kim, J. Jo, H. Park, and K. Park. 2005. Constructing Suffix Arrays in Linear Time. *Journal of Discrete Algorithms* 3, 2-4 (2005), 126–142.

P. Ko and S. Aluru. 2005. Space-efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* 3, 2-4 (2005), 143–156.

N. J. Larsson and K. Sadakane. 1999. *Faster Suffix Sorting*. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999). Department of Computer Science, Lund University, Sweden.

U. Manber and G. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948.

M. A. Maniscalco and S. J. Puglisi. 2006. Faster lightweight suffix array construction. In *Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*. 16–29.

G. Manzini and P. Ferragina. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica* 40, 1 (Sept. 2004), 33–50.

G. Nong, S. Zhang, and W. H. Chan. 2011. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. Comput.* 60, 10 (Oct. 2011), 1471–1484.

D. Okanohara and K. Sadakane. 2009. A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In *Proceedings of SPIRE'09*. Lecture Notes in Computer Science, Vol. 5721. 90–101.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. 2007. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.* 39, 2 (2007), 1–31.

K. Sadakane. 1998. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *Proceedings of DCC'98*. Snowbird, UT, USA, 129–38.

K. B. Schürmann and J. Stoye. 2005. An incomplex algorithm for fast suffix array construction. In *Proceedings of ALENEX/ANALCO 2005*. 77–85.