Algorithmica (2026) 88:44
https://doi.org/10.1007/s00453-026-01383-7

# Sparse Suffix and LCP Array: Simple, Direct, Small, and Fast

**Lorraine A. K. Ayad[1] · Grigorios Loukides[2] · Solon P. Pissis[3] · Hilde Verbeek[3]**

Received: 2 May 2024 / Accepted: 30 March 2026 / Published online: 14 May 2026


**Abstract**

Sparse suffix sorting is the problem of sorting $b = o(n)$ suffixes of a string of length $n$. Efficient sparse suffix sorting algorithms have existed for more than a decade. Despite the multitude of works and their justified claims for applications in text indexing, the existing algorithms have not been employed by practitioners. Arguably this is because there are no simple, direct, *and* efficient algorithms for sparse suffix array construction. We provide two new algorithms for constructing the sparse suffix and LCP arrays that are simultaneously simple, direct, small, and fast. In particular, our algorithms are: *simple* in the sense that they can be implemented using only basic data structures; *direct* in the sense that the output arrays are not a byproduct of constructing the sparse suffix tree or an LCE data structure; *fast* in the sense that they run in $\mathcal{O}(n \log b)$ time, in the worst case, or in $\mathcal{O}(n)$ time, when the total number of suffixes with an LCP value greater than $2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ is in $\mathcal{O}(b/\log b)$, matching the time of optimal yet much more complicated algorithms [Gawrychowski and Kociumaka, SODA 2017; Birenzwige et al., SODA 2020]; and *small* in the sense that they can be implemented using *only* $8b + o(b)$ machine words. Our algorithms are non-trivial space-efficient adaptations of the Monte Carlo algorithm by I et al. for constructing the sparse suffix tree in $\mathcal{O}(n \log b)$ time [STACS 2014]. We provide extensive experiments to justify our claims on simplicity and on efficiency. A preliminary version of this paper appeared in the proceedings of LATIN 2024.

**Keywords** sparse suffix sorting · suffix array · LCP array · text indexing

✉ Solon P. Pissis
solon.pissis@cwi.nl

Lorraine A. K. Ayad
lorraine.ayad@brunel.ac.uk

Grigorios Loukides
grigorios.loukides@kcl.ac.uk

Hilde Verbeek
hilde.verbeek@cwi.nl

[1] Brunel University of London, Kingston Lane, UB8 3PH Uxbridge, UK

[2] King's College London, 30 Aldwych, WC2B 4BG London, UK

[3] Networks & Optimization group, Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands

## 1 Introduction

Let $T = T[1\,..\,n]$ be a string of length $n$ over an ordered alphabet $\Sigma$. Further let $\mathcal{B} \subseteq [1, n]$ be a set of $b > 1$ positions in $T$. *Sparse suffix sorting* is the problem of sorting the set of suffixes $T_{\mathcal{B}} = \{T[i\,..\,n] : i \in \mathcal{B}\}$ lexicographically [1]. This is achieved by constructing the sparse suffix array. The *sparse suffix array* $\mathsf{SSA} = \mathsf{SSA}[1\,..\,b]$ is the array containing the positions in $\mathcal{B}$ in the lexicographical order of the suffixes in $T_{\mathcal{B}}$. The associated *sparse longest common prefix array* $\mathsf{SLCP} = \mathsf{SLCP}[1\,..\,b]$ stores the length $\mathsf{SLCP}[i]$ of the longest common prefix of $T[\mathsf{SSA}[i-1]\,..\,n]$ and $T[\mathsf{SSA}[i]\,..\,n]$ when $i \in [2, n]$ or 0 when $i = 1$. The SSA and SLCP array can be used to construct the sparse suffix tree in linear time using the algorithm by Kasai et al. [2]. The *sparse suffix tree* is the compacted trie of the set $T_{\mathcal{B}}$. Vice-versa, the SSA and SLCP array can be obtained in linear time using a pre-order traversal of the sparse suffix tree.

Sparse suffix sorting was introduced as a fundamental step in the construction of compressed or sparse text indexes [1]. Modern compressed text indexes [3, 4], practical indexes for long patterns [5–8], and sublinear-space string algorithms [9–11] rely on sparse suffix sorting: they first sample a sublinear number of "important" suffixes, which they next sort to construct their final solution. Efficient sparse suffix sorting algorithms have existed for more than a decade. The following algorithms construct SSA explicitly, or implicitly by first constructing the sparse suffix tree. Since the size of SSA (and the size of sparse suffix tree) is $\Theta(b)$, the goal of these algorithms is to use $\mathcal{O}(b)$ words of space assuming read-only random access to $T$:

- Kärkkäinen et al. presented a deterministic $\mathcal{O}(n^2/s)$-time and $\mathcal{O}(s)$-space algorithm, for any $s \in [b, n]$ [12, Section 8].
- Bille et al. presented a Monte Carlo $\mathcal{O}(n \log^2 b)$-time and $\mathcal{O}(b)$-space algorithm [13]. They also presented a Las Vegas $\mathcal{O}(n \log^2 n + b^2 \log b)$-time and $\mathcal{O}(b)$-space algorithm.
- I et al. presented a Monte Carlo $\mathcal{O}(n + (bn/s) \log s)$-time and $\mathcal{O}(b)$-space algorithm, for any $s \in [b, n]$ [14]. They also presented a Las Vegas $\mathcal{O}(n \log b)$-time and $\mathcal{O}(b)$-space algorithm.
- Gawrychowski and Kociumaka [15] presented a Monte Carlo $\mathcal{O}(n)$-time and $\mathcal{O}(b)$-space algorithm. They also presented a Las Vegas $\mathcal{O}(n\sqrt{\log b})$-time and $\mathcal{O}(b)$-space algorithm.
- Birenzwige et al. [16] presented a Las Vegas algorithm running in $\mathcal{O}(n)$ time using $\mathcal{O}(b)$ space. They also presented a deterministic $\mathcal{O}(n \log \frac{n}{b})$-time and $\mathcal{O}(b)$-space algorithm, for any $b = \Omega(\log n)$. See also [17] for a subsequent improvement.

The following algorithms also construct SSA, but they work in the *restore model* [18]: an algorithm is allowed to overwrite parts of the input, as long as it can restore it to its original form at termination.

- Fischer et al. [19] presented a deterministic $\mathcal{O}(c\sqrt{\log n} + b \log b \log n \log^* n)$-time and $\mathcal{O}(b)$-space algorithm, where $c$ is the number of letters that must be compared for distinguishing the suffixes in $T_{\mathcal{B}}$. In some cases, this runs in sublinear extra time; extra refers to the linear cost of loading $T$ in memory.
- Prezza [20] presented a Monte Carlo $\mathcal{O}(n + b \log^2 n)$-time algorithm using $\mathcal{O}(1)$ words of space.

***Motivation.***
Despite the multitude of works on sparse suffix sorting and their justified claims for applications in text indexing, the existing algorithms have not been employed by practitioners. Arguably this is because there are no simple, direct, *and* efficient algorithms for SSA construction. The $\mathcal{O}(n)$-time algorithms of Gawrychowski and Kociumaka [15] and of Birenzwige et al. [16] are far from simple and do not seem to be practically promising either. The former (Monte Carlo) algorithm relies heavily on the construction of compacted tries, which induce high constants in space usage, and on a recursive application of difference cover to construct a Longest Common Extension (LCE) data structure. The latter (Las Vegas) algorithm relies on an intricate partitioning scheme (sampling) to construct SSA and on an LCE data structure to compute the SLCP array. The Monte Carlo $\mathcal{O}(n \log b)$-time algorithm of I et al. [14] is simple but it also relies heavily on compacted tries, which makes it less likely to be employed by practitioners for SSA construction. The Monte Carlo $\mathcal{O}(n + b \log^2 n)$-time algorithm of Prezza [20] makes heavy usage of an LCE data structure as well: constructing the SSA and SLCP array is a byproduct of an in-place LCE data structure. This algorithm is, to the best of our knowledge, the only algorithm which has been implemented (at least in a simplified form). Due to the interest in sparse suffix sorting and the above characteristics of the existing algorithms, we were motivated to revisit this problem to develop efficient, yet simple and direct, algorithms for SSA construction. Such algorithms may serve as baselines for practitioners to engineer the SSA and SLCP array construction.

***Our Results.***
We assume the standard word RAM model with word size $\Theta(\log n)$; basic arithmetic and bit-wise operations on $\mathcal{O}(\log n)$-bit integers take $\mathcal{O}(1)$ time. We assume that we have a read-only random access string $T$ of length $n$ over an integer alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$, a read-only integer array $A$ of size $b$ storing the $b$ elements of $\mathcal{B}$, and two write-only integer arrays SSA and SLCP , each of size $b$. We count the amount of *extra space in machine words* used to construct the SSA and SLCP array. We present two algorithms:

1. Our first algorithm, Main- Algo, constructs SSA and SLCP *directly*; i.e., without first explicitly constructing the sparse suffix tree or an LCE data structure. Its time complexity is $\mathcal{O}(n+(bn/s)\log s)$ and its space complexity is $s+7b+o(b)$ machine words, for any chosen $s \in [b, n]$. It is a Monte Carlo algorithm that returns the correct output *with high probability*; i.e., with probability at least $1 - n^{-c}$, for any constant $c \geq 1$ chosen at construction time. Main- Algo is *simple* in the sense that it can be implemented using only *basic data structures* (e.g., dictionaries and arrays) readily available in widely-used programming languages (e.g., `C++`, `Java`, or `Python`). Main- Algo is a non-trivial space-efficient simulation of the algorithm by I et al. for sparse suffix tree construction [14]. A disadvantage of these two algorithms is that they attain the $\Theta(n \log b)$ time bound for $s = b$ *in any case*. To address this, we develop Parameterized- Algo, a parameterized algorithm which is sensitive to the repetitive structure of the input string.
2. Our second algorithm, Parameterized- Algo, also constructs SSA and SLCP directly. Its time complexity is $\mathcal{O}(n + (b'n/b)\log b)$ and its space complexity is

$8b + 4b' + o(b)$ machine words, where $b'$ is the total number of suffixes $\mathsf{SSA}[i] \in \mathcal{B}$ with $\mathsf{SLCP}[i] \geq \ell$ or $\mathsf{SLCP}[i+1] \geq \ell$, where $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$. When $b' = \mathcal{O}(b/\log b)$, Parameterized- Algo runs in $\mathcal{O}(n)$ time, thus matching the time of the optimal yet much more complicated algorithms in [15, 16], using *only* $8b + o(b)$ machine words. It is a Monte Carlo algorithm that returns the correct output with high probability. It is *remarkably simple* as it consists of two calls of Main- Algo and a linear-time step that merges the partial results (however, the proof of correctness requires some work). The running time of Parameterized- Algo is good in the following sense: if the instance is reasonably sparse, then $\ell$ is large and likely $b' = \mathcal{O}(b/\log b)$, thus it runs in $\mathcal{O}(n)$ time. In any case, it runs in $\mathcal{O}(n \log b)$ time. For instance, for the full human genome (v. GRCh38) as $T$, where $n \approx 3 \cdot 10^9$, and for $b = \lfloor \sqrt{n} \rfloor = 56137$ suffixes selected uniformly at random, $b' = 2525 < \lfloor b/\log b \rfloor = 3558$. We also analyze the time complexity of Parameterized- Algo on random strings and show that it works in $\mathcal{O}(n)$ time (after a small amendment), for any string chosen uniformly at random from $\Sigma^n$ and any set $T_{\mathcal{B}}$ of $b$ suffixes of $T$, with high probability.

To fully justify our claims, we also perform an experimental evaluation. For comparison purposes, we had to resort to Prezza's implementation of his algorithm from [20], which works in the restore model, and it is the only available implementation. Our experiments show that for realistically *sparse instances* (i.e., for $b \in [n/10^7, n/10^3]$) of real-world datasets, our implementation of Parameterized- Algo: (i) runs in linear time (i.e., it is not substantially affected when $b$ increases); and (ii) clearly outperforms Prezza's implementation both in terms of time (even if the latter uses the $n\lceil \log |\Sigma| \rceil = \Omega(n \log |\Sigma|)$ bits of $T$) *and* space. For *dense instances* (i.e., for $b \in [2 \cdot n/10^2, n/10]$), our implementation of Parameterized- Algo still runs faster than Prezza's implementation but, as expected, has a larger memory footprint.

The rest of the paper is organized as follows. Section 2 provides definitions and notation as well as a description of how Karp-Rabin fingerprints [21] are computed, stored, and used. In Section 3, we present Main- Algo, as well as a full working example; and, in Section 4, we present Parameterized- Algo. Finally, in Section 5, we present extensive experiments showing how Main- Algo and Parameterized- Algo perform in comparison to Prezza's implementation. A preliminary version of this paper, *without Section 5 that is new*, appeared in the proceedings of LATIN 2024 [22].

## 2 Preliminaries

We consider finite strings over an integer alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. The elements of $\Sigma$ are called *letters*. A *string* $T = T[1..n]$ is a sequence of letters from $\Sigma$; we denote by $|T| = n$ the *length* of $T$. The fragment $T[i..j]$ of $T$ is an *occurrence* of the underlying *substring* $P = T[i] \ldots T[j]$ occurring at *position* $i$ in $T$. A *prefix* of $T$ is a substring of $T$ of the form $T[1..j]$ and a *suffix* of $T$ is a substring of $T$ of the form $T[i..n]$.

***Karp-Rabin Fingerprints.***
Let $T$ be a string of length $n$ over an integer alphabet. Let $p$ be a prime and choose $r \in [0, p-1]$ uniformly at random. The Karp-Rabin (KR) fingerprint [21] of $T[i \,..\, j]$ is defined as:

$$\phi_T(i, j) = (\sum_{k=i}^{j} T[k] r^{j-k} \bmod p, r^{j-i+1} \bmod p).$$

Clearly, if $T[i \,..\, i+\ell] = T[j \,..\, j+\ell]$ then $\phi_T(i, i+\ell) = \phi_T(j, j+\ell)$. On the other hand, if $T[i \,..\, i+\ell] \neq T[j \,..\, j+\ell]$ then $\phi_T(i, i+\ell) \neq \phi_T(j, j+\ell)$ with probability at least $1 - \ell/p$ [23]. Since we are comparing only substrings of equal length, the number of different possible substring comparisons is less than $n^3$. Thus, for any constant $c \geq 1$, we can set $p$ to be a prime larger than $\max(|\Sigma|, n^{c+3})$ to make the KR fingerprint function perfect with probability at least $1 - n^{-c}$. Any KR fingerprint of $T$ or $p$ fits in one machine word of size $\Theta(\log n)$.

***Remark 1*** In our algorithms (MAIN- ALGO and PARAMETERIZED- ALGO), *the length* $j - i + 1$ of the substrings of $T$ considered at each of the $\mathcal{O}(\log n)$ iterations is fixed, hence the quantity $r^{j-i+1} \bmod p$ only needs to be computed once before every iteration; and thus we do not store it together with the actual fingerprint to save machine words.

**Lemma 1** *([14]) Any string $T \in \Sigma^n$ can be preprocessed in $\mathcal{O}(n)$ time using $s + \mathcal{O}(1)$ machine words, for any $s \in [1, n]$, so that the KR fingerprint of any length-$k$ fragment of $T$ is computed in $\mathcal{O}(\min\{k, n/s\})$ time.*[1]

I et al. [14] employ the *distribute-and-collect* technique [24] to group $b$ suffixes, according to a fixed-length common prefix by using their KR fingerprints (see also Remark 1), in $\mathcal{O}(b \log_s n)$ time. We instead use hashing to achieve the same result in $\mathcal{O}(b)$ time with high probability. This gives improved running times in some special regimes (see Theorem 2 and Theorem 3). It also simplifies our algorithms in terms of implementation since efficient hashing implementations are readily available in any widely-used programming language.

## 3 Main Algorithm

***Overview.***
A summary of our main algorithm (MAIN- ALGO) follows with references to the pseudocode given in Algorithms 1 and 2.[2] As input, it takes a string $T$ from $\Sigma^n$ and an array $A$ of $b$ elements, indicating the starting positions of the suffixes that need to be sorted. As additional input, it takes an integer $j_{\text{start}}$, which defines the number of

[1] I et al. [14] claim $\mathcal{O}(s)$ space but from their construction it is evident that in fact $s + \mathcal{O}(1)$ machine words are used.

[2] The pseudocode is complete in the sense that it only assumes the implementation of Lemma 1 (Line 11).

iterations. In this section, $j_{\text{start}}$ is set to $\lfloor \log n \rfloor$, but a different value is used for the parameterized algorithm in Section 4.

During the first phase (Algorithm 1, Lines 1-25), the suffixes are distributed into groups such that all suffixes belonging to a particular group share a common prefix. At the end of this process, we are left with a hierarchy of groups that describes the exact longest common prefixes between suffixes. The members of each group are then sorted lexicographically (Algorithm 1, Lines 26-27), which is made possible by knowing their longest common prefixes, such that a traversal of the hierarchy will yield the suffixes in lexicographic order. This traversal is the second phase (Algorithm 1, Line 28 and Algorithm 2): a simple depth-first search is used to construct the sparse suffix array and accompanying sparse LCP array from the hierarchy.

### 3.1 Computing and Sorting the LCP Groups

During the first phase, the suffixes of $A$ are organized into several LCP groups stored in set $B$. Each group in $B$ is represented by a triple $(i, k, \{v_1, \ldots, v_{n_i}\})$, where $i$ is the index (id) of the group, $k$ is its associated LCP value and $v_1$ through $v_{n_i}$ are its members, which are either suffixes or other groups. To distinguish between suffixes and groups, the indices 1 through $b$ are reserved for the suffixes in $A$ and the group numbering starts at $b + 1$. At every point of the algorithm, it holds that in a group $(i, k, \{v_1, \ldots, v_{n_i}\})$, all suffixes and groups (with their respective suffixes) in $\{v_1, \ldots, v_{n_i}\}$ share a prefix of length *at least* $k$. At the start (base case), there exists just one group $(b + 1, 0, \{1, \ldots, b\})$ (Line 3) containing all suffixes as members.

The LCP groups are then "refined" (the refinement process will be explained shortly) over the course of $\lfloor \log n \rfloor$ iterations, such that in the end each group describes the exact longest common prefix of its members rather than just a lower bound. Specifically, by the end of iteration $j$ (where $j$ descends from $\lfloor \log n \rfloor$ down to zero), in a group with LCP value $k$, two suffixes will have an actual longest common prefix of at least $k$ and at most $k + 2^j - 1$ letters. This gap is closed once $j$ has reached zero, at which point the refinement process is completed. The algorithm allows specifying a starting value for $j$ different from $\lfloor \log n \rfloor$, through the parameter $j_{\text{start}}$. This is used in the parameterized algorithm described in Section 4.

The refinement process works as follows in iteration $j$. We refine every existing group; let one such group be $(i, k, \{v_1, \ldots, v_{n_i}\})$. We create a hash table (Line 8) and for every group member, with index $v_i$, we take the KR fingerprint as per Lemma 1 of $T[A[v_i] + k \,..\, A[v_i] + k + 2^j - 1]$ (Line 11).[3] If $v_i$ denotes a group, we do the same thing using any given suffix belonging to that group; this is easily achieved by appending "witness" suffixes to $A$ for every created group as seen in Lines 4 and 22. (Any suffix can be a witness but we choose the one with the smallest index.) All the members are grouped in the hash table based on their KR fingerprints: if two suffixes have the same KR fingerprint, they will end up in the same entry of the hash table and with high probability have the same prefix of length $k + 2^j$. To save space, entries are removed from the group as they are added to the hash table (Line 13). All entries of the hash table are then inspected (Line 14). We distinguish three cases. In case 1

[3] This assumes that $A[v_i] + k + 2^j - 1 \leq n$; if this is not the case, the suffix ends at position $n$.

(Lines 16-17), if all suffixes in a group end up having the same KR fingerprint, we update the LCP value of the old group to $k + 2^j$ rather than creating a new group. In case 2 (Lines 18-22), if two or more suffixes have the same KR fingerprint, a new group is made with LCP value $k+2^j$, containing these suffixes, and added to $B$ (Lines 18-22). After removing the suffixes from their original group, we replace them with the index of the newly created group (Line 20). In case 3 (Lines 23-24), if a suffix is not grouped with any other suffix, we append it back to the group it originated from.

Once the iteration with $j = 0$ ends, all LCP groups describe the exact longest common prefix of their members.[4] We now sort the members of every group lexicographically (Lines 26-27). Sorting can be done using merge sort or radix sort, because these algorithms can be performed in place using $\mathcal{O}(1)$ additional memory. Moreover, since we now know the exact LCP value for each group, two members in the same group can easily be compared in constant time: if they have a longest common prefix of length $k$, then the first position in which they differ is $k + 1$, meaning they can be compared by only comparing their $k + 1$-th letters. After this, set $B$ contains the complete and sorted LCP groups, which are passed on to the second step of the algorithm.

### 3.2 Constructing the SSA and SLCP Array

The second phase (Algorithm 2) of the main algorithm involves traversing the groups created in the previous phase in order to construct the SSA and SLCP array. At this point, the members of each group are sorted lexicographically, which means that the SSA can be obtained by a simple pre-order walk along the hierarchy of the groups. For any two members, their exact longest common prefix is stored by their lowest common ancestor; that is, the group with the greatest LCP value that both suffixes fall under.

This part of the algorithm is thus a simple depth-first search of the underlying hierarchy that records all encountered suffixes in SSA in the order they appear. For every group that is visited, the LCP value of its direct "parent" is stored with it (Lines 4 and 17). Throughout, a value $\ell$ is tracked that takes the value of the lowest LCP value that has been seen since the last suffix was encountered (Lines 8-9); every time a suffix is appended to SSA , $\ell$ is appended to the SLCP array (Lines 11-12). This completes the construction.

### 3.3 A Full Working Example

A step-by-step example of the workings of this algorithm is described below. In this example, we construct the SSA and SLCP arrays on the string $T =$ `abracadabrarabia` for the positions $A = (1, 3, 8, 10, 11, 13)$. $A$ contains the starting indices of the suffixes we wish to process. The corresponding suffixes are, in order of their starting positions in $T$, `abracadabrarabia`, `racadabrarabia`, `abrarabia`, `rarabia`, `arabia` and `abia`.

[4] This is generally not true when $j_{\text{start}}$ was set to a value less than $\lfloor \log n \rfloor$; in this case, the LCP values are only correct if they are at most $2^{j_{\text{start}}+1} - 1$; this is expanded on in Section 4.

**Algorithm 1** MAIN- ALGO

**Input:** string $T \in \Sigma^n$, integer $b$, array $A$ of $b$ distinct integers from $[n]$, and integer $j_{\text{start}}$ (default $\lfloor \log n \rfloor$)
**Output:** SSA and SLCP

$m \leftarrow b + 1$ ▷ The number of currently used id's
$L_m \leftarrow (1, \ldots, b)$ ▷ The first group
$B \leftarrow \{(m, 0, L_m)\}$ ▷ Set containing the first group $L_m$ with id $m$ and LCP value 0
$A[m] \leftarrow A[1]$ ▷ Choose a witness for group with id $m$ and extend $A$
**for** $j = j_{\text{start}}, \ldots, 0$ **do** ▷ For all relevant powers of 2 down to 0
  $B' \leftarrow \emptyset$ ▷ Set to store the new groups
  **for** $(i, k, L_i) \in B$ **do** ▷ For every group
    $H_i \leftarrow$ empty hash table ▷ key: the fingerprint of the group member; value: id's of the members
    $s \leftarrow |L_i|$ ▷ To memorize the original size of $L_i$
    **for** $l \in L_i$ **do** ▷ Partition group $L_i$ according to LCP value $k + 2^j$
      $h \leftarrow \phi_T(A[l] + k, A[l] + k + 2^j - 1)$ ▷ Fingerprint of $T[A[l] \,..\, A[l] + k + 2^j - 1]$ using Lemma 1
      $H_i[h]$.append$(l)$ ▷ Append id $l$ as a value of fingerprint $h$ to a satellite vector
      $L_i$.erase$(l)$ ▷ Remove $l$ from $L_i$ (to save space) – $L_i$ is empty at the end of the loop
    **for** $h \in H_i$ **do** ▷ For every fingerprint in the hash table
      $f \leftarrow H_i[h]$ ▷ Let $f$ denote the vector of id's with fingerprint $h$
      **if** $|f| = s$ **then** ▷ All members belong to the same group
        $B \leftarrow B \setminus \{(i, k, L_i)\} \cup \{(i, k + 2^j, f)\}$ ▷ The old group is replaced due to a longer LCP
      **else if** $|f| \geq 2$ **then** ▷ New group $f$ must be created
        $m \leftarrow m + 1$ ▷ New group id
        $L_i$.append$(m)$ ▷ Append the new id to $L$ of tuple $(i, k, L) \in B$
        $B' \leftarrow B' \cup \{(m, k + 2^j, f)\}$ ▷ New group $f$ with id $m$ and LCP value $k + 2^j$ is created
        $A[m] \leftarrow A[f[1]]$ ▷ Choose witness $A[f[1]]$ for group with id $m$ and extend $A$
      **else if** $|f| = 1$ **then**
        $L_i$.append$(f)$ ▷ Append the old id (only element of $f$) back to $L_i$
  $B \leftarrow B \cup B'$ ▷ Append $B'$ to $B$
**for** $(i, k, L_i) \in B$ **do**
  $L_i$.sort$(l \mapsto T[A[l] + k])$ ▷ Sort group in ascending order w.r.t. $T[A[l] + k]$ using in-place sort
**return** OUTPUT- ARRAYS$(B, b, A)$ ▷ See Algorithm 2

**Algorithm 2** OUTPUT- ARRAYS

**Input:** Set $B$ of tuples $(i, k, L_i)$ expected in ascending order by $i$, integer $b$, and array $A$
**Output:** SSA and SLCP

SSA $\leftarrow$ empty array
SLCP $\leftarrow$ empty array
$S \leftarrow$ empty stack
$S$.push$((b + 1, 0))$ ▷ The initial entry of the stack relating to the first group which has LCP value 0
$\ell \leftarrow 0$ ▷ Variable tracking the least LCP value encountered since the last suffix
**while** $S$ is not empty **do**
  $(i, \ell') \leftarrow S$.pop()
  **if** $\ell' < \ell$ **then** ▷ Update $\ell$ if it is less than before
    $\ell \leftarrow \ell'$
  **if** $i \leq b$ **then** ▷ Member $i$ is a suffix: add it to SSA and SLCP
    SSA .append$(A[i])$
    SLCP .append$(\ell)$
    $\ell \leftarrow \infty$ ▷ Reset $\ell$
  **else** ▷ Member $i$ is a group: add its members to the stack
    $(i, k, L_i) \leftarrow B[i - b]$
    **for** $i' \in L_i$ in reverse order **do** ▷ Since the stack is FILO, the first entry needs to be added last
      $S$.push$((i', k))$
**return** SSA and SLCP

We start with a singular group containing all indices and having a shared prefix of length zero (see Table 1). For decreasing powers of 2 starting with $2^{\lfloor \log n \rfloor}$, groups are refined in order to have more and more precise LCP values. For every new group that we create, we take the index corresponding to one of its members and add it to the array $A$, so that we can easily find the shared prefix corresponding to that group.

In the actual algorithm, the prefixes are efficiently grouped by taking their KR fingerprints as per Lemma 1 and indexed in a hash table. For the sake of the example, each relevant prefix is listed instead of its KR fingerprint. The members of each group are sorted lexicographically using merge sort or radix sort. At this point, the groups are refined to the point where they define the exact longest common prefix between their members, so sorting them is done by looking at just the letter succeeding their longest common prefix.

The second phase of the algorithm is a fairly straightforward DFS on the groups in order to build the two output arrays (see Table 2). The variable $\ell$, at every point, takes as value the length of the longest common prefix stored in the lowest common ancestor between the last discovered suffix, in the (conceptual) underlying hierarchy that records all encountered suffixes in SSA in the order they appear, and the currently inspected member. To facilitate this, every member is stored with its *parent's* LCP value in the search stack.

At every iteration, the top of the stack (represented as the first member) is popped. If it is a group, its children are pushed onto the stack and $\ell$ is updated if necessary. If it is a suffix, the element $A[i]$, where $i$ is the index of the suffix, is added to SSA, and the current value of $\ell$ is added to SLCP.

Thus the resulting arrays are $\mathsf{SSA} = (13, 1, 8, 11, 3, 10)$ and $\mathsf{SLCP} = (0, 2, 4, 1, 0, 2)$. Indeed, the corresponding suffixes are `abia`, `abracadabrarabia`, `abrarabia`, `arabia`, `racadabrarabia` and `rarabia`, which are now lexicographically sorted. Moreover, between each consecutive pair of suffixes in SSA, the longest common prefixes correspond to the values in SLCP (the first value in SLCP is 0, by definition).

### 3.4 Analysis

We prove the following result (Theorem 1) by analyzing the time (Lemma 2) and space (Lemma 3) complexity of the MAIN- ALGO. The correctness of the algorithm follows directly from [14].

**Theorem 1** *For any string $T \in \Sigma^n$, any set $T_{\mathcal{B}}$ of $b$ suffixes of $T$, and any $s \in [b, n]$,* MAIN- ALGO *with $j_{start}$ set to $\lfloor \log n \rfloor$ computes the SSA and SLCP of $T_{\mathcal{B}}$ in $\mathcal{O}(n + (bn/s) \log s)$ time using $s + 7b + o(b)$ machine words. The output is correct with high probability.*

**Lemma 2** MAIN- ALGO *with $j_{start}$ set to $\lfloor \log n \rfloor$ runs in $\mathcal{O}(n + (bn/s) \log s)$ time.*

***Proof*** The first phase of the algorithm consists of $\mathcal{O}(\log n)$ iterations and a sorting step. During every iteration, each existing group is considered and every member within the group is hashed. After being hashed, it is either re-added to the same group or put into a new group. The total number of groups is at most $b - 1$, as the group

**Table 1** Computing and sorting the LCP groups.

| | |
|---|---|
| $T =$ `abracadabrarabia` $A = (1, 3, 8, 10, 11, 13)$ | |
| Initial group: $(7, 0, (1, 2, 3, 4, 5, 6))$ | |
| $2^j = 16, 8$ | None of the suffixes share a prefix of length 16 or 8, so no refinement is possible. |
| $2^j = 4$ | Consider the prefixes of length 4 of every suffix: |
| | 1: **`abra`** |
| | 2: `raca` |
| | 3: **`abra`** |
| | 4: `rara` |
| | 5: `arab` |
| | 6: `abia` |
| | Combine the members 1 and 3 into a new group with id 8 and LCP value 4: |
| | $\{(7, 0, (2, 4, 5, 6, \mathbf{8})), (8, 4, (\mathbf{1, 3}))\}$ |
| $2^j = 2$ | Extend the prefixes by 2 and re-group: |
| | 2: **`ra`** 1: `(abra)ca` |
| | 4: **`ra`** 3: `(abra)ra` |
| | 5: `ar` |
| | 6: **`ab`** |
| | 8: **`ab`** |
| | New groups: |
| | $\{(7, 0, (5, \mathbf{9}, \mathbf{10})), (8, 4, (1, 3)), (9, 2, (\mathbf{2, 4})), (10, 2, (\mathbf{6, 8}))\}$ |
| $2^j = 1$ | Extend the prefixes of every suffix by 1 and regroup: |
| | 5: **`a`** 1: `(abra)c` 2: `(ra)c` 6: `(ab)i` |
| | 9: `r` 3: `(abra)r` 4: `(ra)r` 8: `(ab)r` |
| | 10: **`a`** |
| | New groups: |
| | $\{(7, 0, (9, \mathbf{11})), (8, 4, (1, 3)), (9, 2, (2, 4)), (10, 2, (6, 8)), (11, 1, (\mathbf{5, 10}))\}$ |
| Sorting | Sort every group based on the letter appearing directly after the prefix of each member: |
| | 9: `r` 1: `(abra)c` 2: `(ra)c` 6: `(ab)i` 5: `(a)r` |
| | 11: `a` 3: `(abra)r` 4: `(ra)r` 8: `(ab)r` 10: `(a)b` |
| | Final groups: |
| | $\{(7, 0, (\mathbf{11, 9})), (8, 4, (1, 3)), (9, 2, (2, 4)), (10, 2, (6, 8)), (11, 1, (\mathbf{10, 5}))\}$ |

structure represents a conceptual tree (hierarchy) with $b$ leaves in which all internal nodes have at least two children. The number of members in each group is at most $b$. However, by amortization, it can be seen that every member (that is, every suffix and every group other than the "root") is processed precisely once during every iteration. Thus, we have $2b - 2 = \mathcal{O}(b)$ members in total.

For every group member, a KR fingerprint is computed. After the one-time preprocessing of $T$ in $\mathcal{O}(n)$ time, the KR fingerprint of a length-$k$ substring can be computed in $\mathcal{O}(\min\{k, n/s\})$ time (Lemma 1). In the first $\log s$ iterations, the cost is $\mathcal{O}(n/s)$, so the total cost of these iterations is $\mathcal{O}((bn/s)\log s)$. After $\log s$ iterations,

**Table 2** Constructing the SSA and SLCP array

| | $\ell$ | **Stack** | SSA | SLCP |
|---|---|---|---|---|
| 1 | 0 | (7, 0) | ∅ | ∅ |
| 2 | 0 | (11, 0), (9, 0) | ∅ | ∅ |
| 3 | 0 | (10, 1), (5, 1), (9, 0) | ∅ | ∅ |
| 4 | 0 | (6, 2), (8, 2), (5, 1), (9, 0) | ∅ | ∅ |
| 5 | 2 | (8, 2), (5, 1), (9, 0) | (13) | (0) |
| 6 | 2 | (1, 4), (3, 4), (5, 1), (9, 0) | (13) | (0) |
| 7 | 4 | (3, 4), (5, 1), (9, 0) | (13, 1) | (0, 2) |
| 8 | 1 | (5, 1), (9, 0) | (13, 1, 8) | (0, 2, 4) |
| 9 | 0 | (9, 0) | (13, 1, 8, 11) | (0, 2, 4, 1) |
| 10 | 0 | (2, 2), (4, 2) | (13, 1, 8, 11) | (0, 2, 4, 1) |
| 11 | 2 | (4, 2) | (13, 1, 8, 11, 3) | (0, 2, 4, 1, 0) |
| 12 | 2 | ∅ | (13, 1, 8, 11, 3, 10) | (0, 2, 4, 1, 0, 2) |

the length $k$ of the substring whose KR fingerprint is computed is $k < n/2^{\log s} = n/s$ and so the total cost of all remaining iterations is $bn/s + bn/(2s) + bn/(4s) + \cdots + b = \mathcal{O}(bn/s)$. Thus the total cost of computing all KR fingerprints is $\mathcal{O}(n + (bn/s)\log s)$.

Every group member has its KR fingerprint taken and added to a hash table supporting constant worst-case operations with high probability [25, 26]. Afterwards, all members from the hash table are re-added to the groups; for every KR fingerprint collision a new group is created with its respective members, and all other members are re-added to the original group. The number of newly created groups is at most half the number of members in the original group, as every new group has to contain at least two members. So other than the fingerprinting, all operations for a single member are performed in constant time with high probability,[5] meaning that the total time for every iteration is $\mathcal{O}(b \log n) = \mathcal{O}((bn/s)\log s)$.

In the sorting step, we have two cases: (a) $b < n/\log n$ and (b) $b \geq n/\log n$. The members in each group are sorted using in-place merge sort [27, 28] (Case (a)) or in-place radix sort [29] (Case (b)) in $\mathcal{O}(n)$ time.

*Case (a):* $b < n/\log n$. Sorting $k$ members with merge sort takes $\mathcal{O}(k \log k)$ time. Recall that there are at most $b - 1$ groups and that the total number of members over all groups is at most $2b - 2 = \mathcal{O}(b)$. If the number of members to be sorted in group $i$ is $k_i$, then $k_1 + \cdots + k_{b-1} = \mathcal{O}(b)$ so the time needed to sort all groups is $\mathcal{O}(k_1 \log k_1 + \cdots + k_{b-1} \log k_{b-1}) = \mathcal{O}((k_1 + \cdots + k_{b-1}) \log b) = \mathcal{O}(b \log b) = \mathcal{O}(n)$.

*Case (b):* $b \geq n/\log n$. We employ the algorithm by Franceschini et al. [29], which given an array $A$ of $k$ $\mathcal{O}(\log k)$-bit integers, sorts $A$ in place in $\mathcal{O}(k)$ time. Sorting the $2b - 2$ members takes $\mathcal{O}(b) = \mathcal{O}(n)$ time, because every member can be encoded by its group id, which is a $\mathcal{O}(\log b)$-bit integer, and a letter, which is also a $\log \sigma = \mathcal{O}(\log n) = \mathcal{O}(\log b)$-bit integer, where $\sigma = |\Sigma|$.

[5] If this is not the case, we output incorrect arrays deliberately to ensure that our algorithm is Monte Carlo.

The second phase of the algorithm is a simple stack-based DFS. Each of the $\mathcal{O}(b)$ members is pushed to and popped from the stack precisely once. The further operations applied to each member all take constant time, so this step takes $\mathcal{O}(b)$ time.

Adding all this together gives $\mathcal{O}(n) + \mathcal{O}(bn/s) \cdot \mathcal{O}(\log s) + \mathcal{O}(n) + \mathcal{O}(b) = \mathcal{O}(n + (bn/s)\log s)$ time. □

Like the algorithm by I et al. [14], Main- Algo can be amended to work in $\mathcal{O}(n)$ time, when $s = b \log b$.

**Lemma 3** Main- Algo *can be implemented using* $s + 7b + o(b)$ *machine words, excluding the read-only string* $T$*, the array* $A$ *representing the set of* $b$ *suffixes, and the write-only output arrays* SSA *and* SLCP*.*

***Proof*** We analyze the peak space used by the algorithm neglecting the use of $\mathcal{O}(1)$ machine words:

- **KR fingerprints**: Pre-processing $T$ to compute KR fingerprints takes $s$ machine words by Lemma 1.
- **Array** $A$: Array $A$ starts with $b$ integers as input, but at most $b-1$ more integers are appended to it during the algorithm to store witness suffixes for new groups. (Even if $A$ is read-only we can simulate the append operation by using an extra array.)
- **Set** $B$: We implement set $B$ using three integer arrays: $K$ of size $b-1$; $C$ of size $b-1$; and $L$ of size $2b-2$. $K[i]$ stores the LCP value for the group with id $i+b$; and $L[C[i-1]+1], \ldots, L[C[i]]$ are the group's member id's.[6] There are at most $b-1$ groups; for every group one integer is stored in $K$ and $C$ as well as the group member id's in $L$. The total number of group members is at most $2b-2$, since all groups except the "root" group are a member. Thus set $B$ can be implemented using $4b-4$ integers.
- **Hash table** $H_i$: While processing group $i+b$, every group member is (removed from the group and) added to a hash table $H_i$ as satellite value of the corresponding KR fingerprint key. We use a space-efficient hash table storing $c_i = C[i]-C[i-1]$ integers (KR fingerprints) as keys: By using [25, 26], we implement $H_i$ using $(1+\epsilon)c_i$ machine words, for any $\epsilon = \Omega(\log\log c_i / \log c_i)$. We need at most $b$ integers to maintain the size of the satellite values per KR fingerprint because every group can have at most $b$ members. By choosing $\epsilon = \log\log c_i / \log c_i$ we need at most $2b + o(b)$ machine words in total to maintain the hash table.

We can delete the fingerprint data structure and the hash table before moving to the sorting step. Sorting does not use any additional space because merge sort and radix sort can be implemented in-place [27–29], thus using only $\mathcal{O}(1)$ additional machine words. The first phase of the algorithm uses at most $s + 7b + o(b)$ machine words but at the end of it we have $5b + \mathcal{O}(1)$ machine words stored: array $A$ and set $B$.

We now analyze the space used in the second phase (Algorithm 2); in particular, the space taken by the search stack. The stack stores at most every group and every suffix. However, the stack never simultaneously stores a member and one of its ancestors,

[6] If $i = 1$ then the group member id's are $L[1], \ldots, L[C[i]]$.

meaning the maximum size of the stack at any point is at most the maximum width of the sparse suffix tree, which is $b$. Every element in the stack consists of two integers, so the stack takes up at most $2b$ machine words. No other machine words need to be stored as the maximum stack size $b$ is known in advance and so the stack is implemented using an array.

Adding this together gives at most $s + 7b + o(b)$ machine words in total needed for the algorithm. □

## 4 A Simple Parameterized Algorithm

In real-world datasets, the $b$ suffixes in $T_{\mathcal{B}}$ will generally not share very long prefixes. Even when they do, it is highly unlikely that all of them have this property. While MAIN- ALGO is theoretically efficient, it would waste a lot of time with such datasets by considering large overlaps between suffixes when in reality the longest common prefixes are much shorter or when only very few suffixes share very long prefixes. Below, we show a simple method to take advantage of this, by only considering short common prefixes in the beginning and then extending them *only* for the suffixes that happen to share longer prefixes. By considering an extra parameter $b'$ indicating the number of suffixes that share longest common prefixes longer than a certain threshold, we arrive at a time and space complexity that appears favorable for such real-world datasets.

***Main Idea.***
We design an algorithm for constructing SSA and SLCP which is parameterized by the total number $b' \leq b$ of suffixes which have an LCP value of at least $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ with some other suffix. We show that partitioning the $b$ suffixes into two classes (one with suffixes with LCP value strictly less than $\ell$; and another with suffixes with LCP value greater than or equal to $\ell$) can be done in $\mathcal{O}(n)$ time. In particular, we show that it suffices to invoke Theorem 1 twice: once (with a small change) for the $b$ suffixes; and once (as is) for the $b'$ suffixes; and then merge the partial results in $\mathcal{O}(b)$ time to obtain the final SSA and SLCP array.

***Description and Pseudocode.***
The pseudocode for the algorithm is given as PARAMETERIZED- ALGO (Algorithm 3).[7] A line-by-line explanation of the algorithm follows.

PARAMETERIZED- ALGO invokes the original algorithm MAIN- ALGO twice with different arguments. In Line 1, it calls MAIN- ALGO with the full array $A$ as argument (and $s = b$). We set the parameter $j_{\text{start}}$ that indicates the starting value of $j$ (Line 5 of Algorithm 1) to $\lfloor \log \frac{n}{b} \rfloor$, meaning that $j$ starts at a lower value than the value $\lfloor \log n \rfloor$ used in the MAIN- ALGO and so it will take less time to complete. The result of this is that SSA will only be sorted up to $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$ positions. This means that for every consecutive pair of suffixes in SSA, if their LCP value is less than $\ell$, they will already be sorted correctly, whereas the other suffixes, with associated LCP values of $\ell$, will need to be further sorted in the second phase (Lines 8 to 13) of PARAMETERIZED- ALGO.

[7] The pseudocode is complete in the sense that it only assumes the implementation of MAIN- ALGO.

What remains is to identify the suffixes that need to be further sorted, sort these suffixes separately from the others, and re-insert them into the output arrays along with the corrected LCP values. We use two arrays $A'$ and $P$ for this purpose: $A'$ contains the suffixes; and $P$ tracks the positions in SSA that these suffixes are taken from, to ensure that they will later be re-inserted at the correct positions. In Line 5, we ensure that the right suffixes are tracked in these arrays, namely those that have an LCP value of $\ell$ with their predecessor or successor suffix. If any such suffixes are found, we invoke MAIN- ALGO again (Line 9), but with just these suffixes (those in array $A'$) as input, and with the default value of $j_{\text{start}} = \lfloor \log n \rfloor$. This means that the suffixes of $A'$ will now be fully sorted rather than being sorted up to $\ell$ positions. Then, in Lines 10 and 11, we insert these re-sorted suffixes at the same positions that they were taken from before, but in the corrected order. In Lines 12 and 13, we also copy the associated LCP values, but only at the positions in-between two re-sorted suffixes, as all other LCP values were already correct.

We next state and prove Theorem 2.

**Theorem 2** *For any string $T \in \Sigma^n$ and any set $T_{\mathcal{B}}$ of $b$ suffixes of $T$,* PARAMETERIZED-ALGO *computes the SSA and SLCP of $T_{\mathcal{B}}$ in $\mathcal{O}(n + (b'n/b)\log b)$ time using $8b + 4b' + o(b)$ machine words, where $b'$ is the total number of $i$ such that $\mathsf{SSA}[i] \in \mathcal{B}$ and $\mathsf{SLCP}[i] \geq \ell$ or $\mathsf{SLCP}[i+1] \geq \ell$, with $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$. The output is correct with high probability. When $b' = \mathcal{O}(b/\log b)$,* PARAMETERIZED- ALGO *runs in $\mathcal{O}(n)$ time using $8b + o(b)$ machine words.*

**Algorithm 3** PARAMETERIZED- ALGO

**Input:** string $T \in \Sigma^n$, integer $b$, and array $A$ of $b$ distinct integers from $[n]$
**Output:** SSA and SLCP

1: SSA, SLCP $\leftarrow$ MAIN- ALGO($T$, $A$, $b$, $j_{\text{start}} = \lfloor \log \frac{n}{b} \rfloor$) ▷ Quasi-sort the $b$ suffixes
2: $\ell \leftarrow 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$
3: $P$, $A' \leftarrow$ empty arrays
4: **for** $i = 1, \ldots, b$ **do** ▷ Check which of the $b$ suffixes need further sorting
5: **if** $\mathsf{SLCP}[i] = \ell \vee (i < b \wedge \mathsf{SLCP}[i+1] = \ell)$ **then** ▷ Suffix $T[\mathsf{SSA}[i]\,..\,n]$ needs further sorting
6: $P$.append($i$)
7: $A'$.append(SSA[$i$])
8: **if** $|A'| > 0$ **then**
9: SSA$'$, SLCP$'$ $\leftarrow$ MAIN- ALGO($T$, $A'$, $|A'|$, $j_{\text{start}} = \lfloor \log n \rfloor$) ▷ Sort $|A'|$ out of $b$ suffixes
10: **for** $i = 1, \ldots, |A'|$ **do** ▷ Merge the partial arrays
11: SSA[$P[i]$] $\leftarrow$ SSA$'[i]$
12: **if** SLCP[$P[i]$] $= \ell$ **then** ▷ Only copy the SLCP value if the suffix at $P[i]-1$ was also re-sorted
13: SLCP[$P[i]$] $\leftarrow$ SLCP$'[i]$
14: **return** SSA and SLCP

***Time Complexity.***
The first phase of the algorithm (Line 1) runs in $\mathcal{O}(\log \frac{n}{b})$ iterations. The longest prefixes whose KR fingerprints are computed have length $\mathcal{O}(\frac{n}{b})$, and there are $\mathcal{O}(b)$ KR fingerprints computed in each iteration. This means that computing the KR fingerprints during the first phase takes $\mathcal{O}(b) \cdot (\mathcal{O}(\frac{n}{b}) + \mathcal{O}(\frac{n}{2b}) + \mathcal{O}(\frac{n}{4b}) + \ldots) = \mathcal{O}(n)$ time. Hashing the fingerprints takes $\mathcal{O}(b \log \frac{n}{b}) = \mathcal{O}(n)$ worst-case time in total with high

probability. (Grouping the fingerprints via distribute-and-collect, like the algorithm by I et al. [14], would incur a multiplicative factor of $\log_s n$.) Sorting takes $\mathcal{O}(n)$ time (see Lemma 2). Therefore the entire first phase runs in $\mathcal{O}(n)$ time. The second phase (Lines 8 to 13) computes KR fingerprints of longer prefixes as well and otherwise runs the same as MAIN- ALGO, with the exception that only $b'$ suffixes are now sorted. By Lemma 2, for $s = b$, this takes $\mathcal{O}(n + (b'n/b)\log b)$ time. All other operations run in single loops over arrays of size $b$ or $b'$ with constant-time operations, and thus take $\mathcal{O}(b)$ time. Adding everything together gives $\mathcal{O}(n + (b'n/b)\log b)$ time. When $b' = \mathcal{O}(b/\log b)$, the running time becomes $\mathcal{O}(n)$.

***Space Complexity.***
The first phase of the algorithm uses $s+7b+o(b)$ machine words as per Lemma 3. The additional arrays $P$, $A'$, $\mathsf{SSA}'$ and $\mathsf{SLCP}'$ use $4b'$ machine words in total. The second invocation of MAIN- ALGO uses $s + 7b' + o(b')$ machine words as per Lemma 3. By setting $s = b$, the algorithm uses $8b + 4b' + o(b)$ machine words in total. If $b' = \mathcal{O}(b/\log b)$, the algorithm uses $8b + o(b)$ machine words.

***Correctness.***
The correctness of $\mathsf{SSA}$ is proved by Lemma 6 and that of $\mathsf{SLCP}$ is proved by Lemma 7. To prove these lemmas, we first show the auxiliary Lemmas 4 and 5.

**Lemma 4** *Let $\mathsf{SSA}_1$ be the instance of $\mathsf{SSA}$ after the first invocation of* MAIN- ALGO *(Line 1). The strings $T[\mathsf{SSA}_1[i]\,..\,n]$, $i \in [1, b]$, are sorted up to their prefix of length $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$.*

***Proof*** In MAIN- ALGO, all LCP values can be increased by powers of two in each iteration. With the starting value $j_{\text{start}} = \lfloor \log \frac{n}{b} \rfloor$, this adds up to a maximum LCP value of $\ell$ in any group. At any point during MAIN- ALGO, two suffixes that are in the same group with LCP value $k$ share a longest common prefix of length at least $k$. Thus, this invocation of MAIN- ALGO will compute the LCP values between suffixes correctly if they are at most $\ell$, and all other LCP values will be $\ell$. The sorting step takes into account only the letter which appears after the computed (longest) common prefix, so if the LCP between any two suffixes is less than $\ell$ the suffixes will be sorted correctly. The statement follows. □

**Lemma 5** *Let $\mathsf{SSA}_1$ be the instance of $\mathsf{SSA}$ after the first invocation of* MAIN- ALGO *(Line 1), and let $\mathsf{SSA}_2$ be the instance of $\mathsf{SSA}$ returned at the end of* PARAMETERIZED-ALGO *(Line 14). For every $i \in [1, b]$, either $\mathsf{SSA}_1[i] = \mathsf{SSA}_2[i]$ and $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ have a longest common prefix of length $n - \mathsf{SSA}_1[i] + 1$, or $\mathsf{SSA}_1[i] \neq \mathsf{SSA}_2[i]$ and $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ have a longest common prefix of length at least $\ell$.*

***Proof*** If $\mathsf{SSA}_1[i] = \mathsf{SSA}_2[i]$, this is trivial, so we only concern ourselves with the case $\mathsf{SSA}_1[i] \neq \mathsf{SSA}_2[i]$. In this case, the value was overwritten in Line 11, meaning that the suffix $\mathsf{SSA}_1[i]$ was stored in $A'$ in Line 7 to be re-sorted in the second invocation of MAIN- ALGO. The same must hold for $\mathsf{SSA}_2[i]$.

Consider the array $A'$ as it is built in Lines 4-7. By Lemma 4, the suffixes of $\mathsf{SSA}_1$ are sorted up to their length-$\ell$ prefix; since the entries of $A'$ appear in the same order as they appear in $\mathsf{SSA}_1$, this must also be the case for $A'$. Because the suffixes of $A'$ are

already sorted correctly up to their length-$\ell$ prefix, it must be that for every position $j \in [1, b']$, $A'[j]$ and $\mathsf{SSA}'[j]$ have the same length-$\ell$ prefix. Now note that if $\mathsf{SSA}_1[i]$ appears in position $j$ in $A'$, then $\mathsf{SSA}_2[i]$ will take the value from $\mathsf{SSA}'[j]$. Since $A'[j]$ and $\mathsf{SSA}'[j]$ have a length-$\ell$ common prefix, $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i]$ must as well. □

**Lemma 6** *The instance $SSA_2$ of SSA returned at the end of* PARAMETERIZED- ALGO *(Line 14),*

*contains the suffixes of A sorted lexicographically.*

***Proof*** We prove this by showing that for any two consecutive positions $i$ and $i+1$, $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ appear in the right order. Let $\mathsf{SSA}_1$ be the instance of SSA after the first invocation of MAIN- ALGO.

We already know that $\mathsf{SSA}_1$ is sorted correctly up to $\ell$ positions. This means that for any $i$, if the longest common prefix of $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ is shorter than $\ell$, they already appear in the correct order in this array. If neither suffix is overwritten after the second phase, this is also trivially the case for them in $\mathsf{SSA}_2$.

Now suppose that exactly one of the two (wlog $\mathsf{SSA}_1[i+1]$) is replaced by some other suffix $s$ while the other remains the same. Let $k$ be the LCP of $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$. By Lemma 5, $\mathsf{SSA}_1[i+1]$ and $s$ have a longest common prefix of length at least $\ell$. This is longer than $k$, which is strictly less than $\ell$. This means that the $(k+1)$-th letter of $s$ is the same as that of $\mathsf{SSA}_1[i+1]$, which is the first position in which it differs from $\mathsf{SSA}_1[i]$. Thus $\mathsf{SSA}_2[i] = \mathsf{SSA}_1[i]$ and $\mathsf{SSA}_2[i+1] = s$ are sorted correctly relative to one another.

The remaining case is when $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ have a longest common prefix of length $\ell$ or longer. In this case, both suffixes are added to $A'$ to be re-sorted in the second invocation, and both $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ may take the value of another suffix. The second invocation of the main algorithm sorts all suffixes in $A'$ completely, returning $\mathsf{SSA}'$. The suffixes in $\mathsf{SSA}'$ are then re-inserted into $\mathsf{SSA}_2$, in which they will appear in the same order as they did in $\mathsf{SSA}'$. Therefore, no matter which suffixes end up at $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$, they also appeared consecutively in $\mathsf{SSA}'$ and therefore must be sorted correctly. □

**Lemma 7** *For any two consecutive positions $i$ and $i+1$, $SLCP[i+1]$, as returned by Algorithm 3, gives the length of the longest common prefix of $SSA[i]$ and $SSA[i+1]$.*

***Proof*** Let $\mathsf{SSA}_1$ and $\mathsf{SLCP}_1$ be the arrays returned by the first invocation of MAIN-ALGO, and $\mathsf{SSA}_2$ and $\mathsf{SLCP}_2$ the arrays produced at the end. By Lemma 4, if $\mathsf{SLCP}_1[i+1] < \ell$, this value is correct. Therefore, the only values that need to be overwritten for $\mathsf{SLCP}_2$ are when $\mathsf{SLCP}_1[i+1] = \ell$. The check at Line 12 ensures this. Of course, when $\mathsf{SLCP}_1[i+1] = \ell$, then both $\mathsf{SSA}_1[i]$ and $\mathsf{SSA}_1[i+1]$ are added to $A'$ in order to be re-sorted in the second invocation. The values at $\mathsf{SSA}_2[i]$ and $\mathsf{SSA}_2[i+1]$ are then replaced by two suffixes that appear consecutively in $\mathsf{SSA}'$, say $\mathsf{SSA}'[j]$ and $\mathsf{SSA}'[j+1]$. By the correctness of MAIN- ALGO, the LCP value of these two suffixes is given by $\mathsf{SLCP}'[j+1]$, which is the value that $\mathsf{SLCP}_2[i+1]$ takes. □

***Random Strings.***

Finally, we show that PARAMETERIZED- ALGO can be trivially amended to work in $\mathcal{O}(n)$ time for any string chosen uniformly at random from $\Sigma^n$.

**Theorem 3** *For any string $T$ chosen uniformly at random from $\Sigma^n$ and any set $T_{\mathcal{B}}$ of $b$ suffixes of $T$, SSA and SLCP of $T_{\mathcal{B}}$ can be computed in $\mathcal{O}(n)$ time using $\mathcal{O}(b)$ space. The output is correct with high probability.*

***Proof*** We assume $|\Sigma| \geq 2$, otherwise the problem has a trivial solution. Bollobás and Letzter [30, Theorem 4] showed that the maximum length of an LCE on $T$ is at most $2\log_{|\Sigma|} n + \log_{|\Sigma|}\log_{|\Sigma|} n$ with high probability. We bound this from above by $3\log n$ and amend PARAMETERIZED- ALGO as follows:

- *Case (a):* $b\log n < n$. We invoke MAIN- ALGO by setting $j_{\text{start}}$ to the smallest integer such that $2^{j_{\text{start}}} \geq 2\lfloor \log n \rfloor$, which gives $\ell = 2\lfloor \log n \rfloor \cdot 2 - 1 = 4\lfloor \log n \rfloor - 1$. After the $\mathcal{O}(n)$-time preprocessing of Lemma 1, computing the KR fingerprints takes $\mathcal{O}(b)\cdot 4(\mathcal{O}(\frac{\log n}{1}) + \mathcal{O}(\frac{\log n}{2}) + \mathcal{O}(\frac{\log n}{4}) + \ldots) = \mathcal{O}(b\log n)$ time. Hashing the fingerprints takes $\mathcal{O}(b)$ time per iteration with high probability, and so $\mathcal{O}(b\log n)$ total time. Merge sort takes $\mathcal{O}(b\log b)$ time. Since $\ell > 3\log n$, all suffixes of $T_{\mathcal{B}}$ will be fully sorted from the first invocation of MAIN- ALGO. If $b' = \mathcal{O}(b/\log b)$ suffixes are still unsorted after the first invocation, these will be fully sorted in the second invocation of MAIN- ALGO in $\mathcal{O}(n)$ time (Theorem 2). If $b' = \omega(b/\log b)$, we output incorrect arrays. The total time complexity is $\mathcal{O}(n + b\log n) = \mathcal{O}(n)$.
- *Case (b):* $b\log n \geq n$. Assume that we have $\mathcal{O}(s)$ space to sort the $b$ suffixes; we can do it efficiently using radix sort because it suffices to sort all prefixes of them of length $\mathcal{O}(\log_\sigma n)$ by the Bollobas and Letzter's result, where $\sigma = |\Sigma|$ (otherwise, we output incorrect arrays). The $b$ prefixes are each of length at most $c\log_\sigma n$, for some $c = \mathcal{O}(1)$; so radix sort takes $\mathcal{O}((b+s)(c\cdot\log n/\log\sigma)\cdot(\log\sigma/\log s))$ time, because we have at most $(c\log n/\log\sigma)$ letters in every prefix, and each time we sort $b$ letters, one from each prefix, we use $(\log\sigma/\log s)$ rounds of counting sort. Conveniently, the $\log\sigma$ terms cancel out. Then, because we set $s = b$, and by the fact that we are in the case $b \geq n/\log n$, we have that $\log n/\log s = \mathcal{O}(1)$. The total time complexity is thus $\mathcal{O}(b+s) = \mathcal{O}(b)$. The total space used is $\mathcal{O}(s) = \mathcal{O}(b)$. By comparing adjacent suffixes we compute the SLCP array within the same complexities. □

## 5 Experiments

***Implementations.***
We have used the following C++ implementations that we make all available at https://github.com/lorrainea/SSA under GPL-3.0 license. Although these implementations are somewhat simplified for practical efficiency purposes, they are quite close to their original theoretical descriptions.

1. A simplified implementation of the algorithm of Prezza [20], written by the author.[8] It first constructs an LCE data structure in $\mathcal{O}(n)$ time in place: using exactly the same space as $T$ (plus $\mathcal{O}(1)$ machine words). Since the structure supports $\mathcal{O}(\log n)$-time LCE queries, the implementation uses LCE queries and `std::sort` to construct the SSA . We have amended this implementation to also compute the

[8] https://github.com/nicolaprezza/rk-lce/blob/master/sa-rk.cpp

SLCP in $\mathcal{O}(b \log n)$ extra time using the SSA and LCE queries. We denote this implementation by SSA- LCE. Note that, for example, the use of `std::sort` does not guarantee that sorting the suffixes is done in-place in practice.

2. A simplified implementation of MAIN- ALGO, written by us. It is simplified in the sense that for hashing we use the `unordered_dense` class.[9] Also if the number of KR fingerprints to be grouped is smaller than a predefined constant (set to 1.5M by default), then we resort to `std::sort` to achieve the same goal. Furthermore, we have implemented array $A$ and set $B$ by means of `std::vector`, and thus we neither had full control of the exact number of machine words used per vector, nor did we know when these became available upon memory deallocation. For KR fingerprints, we have used the class by Kempa.[10] We denote this implementation by MA.
3. An implementation produced by us of PARAMETERIZED- ALGO using the above simplified implementation of MAIN- ALGO. We denote this implementation by PA.

***Datasets.***
We have used the following datasets choosing suffixes uniformly at random in all cases. For the first set of experiments (*sparse instances*) we chose $b \in [n/10^7, n/10^3]$ and for the second one (*dense instances*) we chose $b \in [2 \cdot n/10^2, n/10]$:

1. A file of $n = 11,038,279,710$ bytes (11.04 GB), with $|\Sigma| = 64$, containing 689, 781 human DNA reads of total length 5, 455, 971, 336, along with their quality scores.[11] The reads have been sequenced using an Oxford Nanopore MinION device. We denote this dataset by FASTQ.
2. A file of $n = 10,225,926,270$ bytes (10.23 GB), with $|\Sigma| = 96$, containing 21,928,568 Amazon reviews (Home and Kitchen).[12] We denote this dataset by AMAZON.
3. A synthetically generated file of $n = 5,000,000,000$ bytes (5GB), with $|\Sigma| = 26$, containing a single string of ASCII codes. Every letter of this string was selected uniformly at random. We denote this dataset by RANDOM.

***Setup.***
The experiments ran on an Intel Xeon Gold 648 CPU at 2.5GHz with 256GB RAM. All programs were compiled with `g++` version `10.2.0` at the `-Ofast` optimization level.

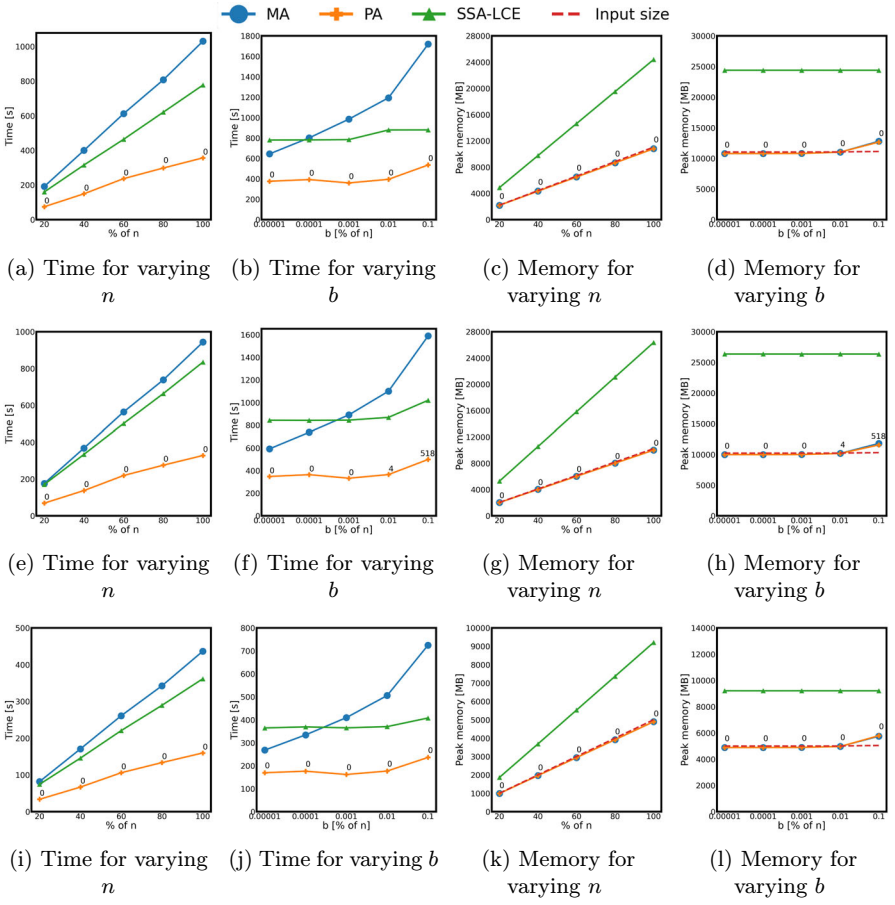
***Results (Sparse Instances).***
Figure 1 shows the results for the sparse instances; i.e., those where $b \in [n/10^7, n/10^3]$ is relatively small. By default, we used $b = 0.001\% \cdot n$ for the experiments that take prefixes of the whole datasets to vary the values of $n$; and we used the whole datasets for the experiments that use varying values of $b$. Figure 1a shows that: (i) PA is up to

[9] https://github.com/martinus/unordered_dense

[10] https://github.com/dominikkempa/lz77-to-slp/blob/main/src/karp_rabin_hashing.cpp

[11] https://github.com/nanopore-wgs-consortium/NA12878/blob/master/nanopore-human-genome/rel_3_4.md (FAF01132)

[12] https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/(Home and Kitchen)




(a) Time for varying $n$ (b) Time for varying $b$ (c) Memory for varying $n$ (d) Memory for varying $b$

(e) Time for varying $n$ (f) Time for varying $b$ (g) Memory for varying $n$ (h) Memory for varying $b$

(i) Time for varying $n$ (j) Time for varying $b$ (k) Memory for varying $n$ (l) Memory for varying $b$

**Fig. 1** Results for sparse instances of FASTQ (top row), AMAZON (middle row) and RANDOM (bottom row). The exact value of $b'$ is on top of the points of the PA curves to highlight the relevance of Theorem 2. Notably in all but two instances we have that $b' = 0$. By default, we used $b = 0.001\% \cdot n$ when varying $n$; and the whole dataset when varying $b$

2 times faster than SSA- LCE and over 2 times faster than MA; and (ii) the runtime of all implementations scales linearly with $n$. Figure 1b shows that: (i) PA is again much faster than the other two implementations; and (ii) PA and SSA- LCE are not substantially affected (if at all) by increasing $b$ as opposed to MA. In conjunction, the results of Figures 1a and 1b, show that PA runs in linear time because $b' = 0$ in all instances (see Theorem 2). Figure 1c shows that: (i) PA and MA consume about half the memory consumed by SSA- LCE; and (ii) the peak memory usage for all implementations scales linearly with $n$. Figure 1d shows that: (i) PA and MA are substantially more space-efficient than SSA- LCE; and (ii) PA and MA start off with the space required by the input and increase the peak memory usage only slightly when $b > 0.01\%$ of $n$. The former is explained by the fact that SSA- LCE is a simplified implementation of the algorithm in [20] (recall that no other implementation

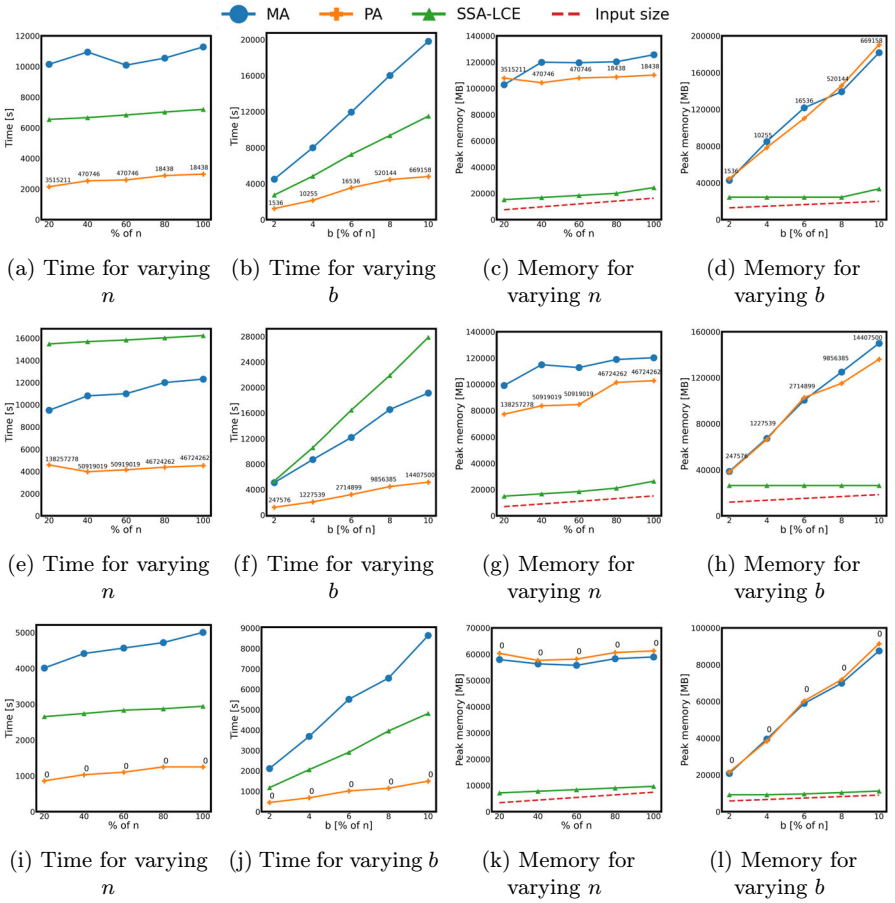
is available), which uses more than $\mathcal{O}(1)$ words of space to sort the $b$ suffixes. The results in Figures 1e-1h for AMAZON and in Figures 1i-1l for RANDOM are analogous.

The benefits of PA in terms of time compared to MA and SSA- LCE on these sparse instances are significant.[13] These benefits are explained by the fact that $b' = 0$ in all but two instances; even in the instance with the largest $b' > 0$, we still have that $b' = 518 < \lfloor b/\log b \rfloor = 439,149$ satisfying the condition in Theorem 2. These results *fully justify* the design goals of PA. Notably, in these sparse instances, SSA-LCE consumes more memory than PA and MA.

***Results (Dense Instances).***
Figure 2 shows the results for the dense instances, where $b \in [2 \cdot n/10^2, n/10]$. In these instances, $b$ is relatively large (from 2% to 10% of $n$). By default, we used $b = 6\% \cdot n$ for the experiments that take prefixes of the whole datasets to vary the values of $n$; and we used the whole datasets for the experiments that use varying values of $b$. Figure 2a shows that: (i) PA is up to 2 times faster than SSA- LCE and over 3 times faster than MA; and (ii) the runtime of all implementations increases only slightly with $n$. Figure 2b shows that: (i) PA is again many times faster than the other two implementations; and (ii) PA is not as substantially affected by increasing $b$ as SSA- LCE and MA are. In conjunction, the results of Figures 2a-2b show that, in these dense instances, increasing $b$ becomes considerably more costly than increasing $n$ (unlike in the sparse instances), due to the $\tilde{\mathcal{O}}(b)$ term in the time complexity of all algorithms being much more costly than the $\mathcal{O}(n)$ term. Figure 2c shows that: (i) PA and MA consume about 5 times more memory compared to SSA- LCE due to the large values of $b$ (as expected by the space analysis of the algorithms); and (ii) the peak memory usage for all implementations increases only slightly with $n$ (as expected due to the larger input string). Figure 2d shows that: (i) the peak memory usage of PA and MA scales linearly with $b$; and (ii) PA and MA consume up to 5 times more memory in comparison to SSA- LCE. Figures 2e-2f show that for AMAZON *both* PA and MA are faster than SSA- LCE with PA being over 2 times faster than SSA- LCE. This happens because SSA- LCE becomes considerably slower when $b'$ is large; i.e., when many pairs of suffixes have long LCP values. In such a case, comparing any two suffixes takes $\mathcal{O}(\log n)$ time for SSA- LCE instead of $\mathcal{O}(1)$ time, and so the $\tilde{\mathcal{O}}(b)$ term becomes a bottleneck. The peak memory usage results in Figures 2g-2h for AMAZON and Figures 2i-2l for RANDOM are analogous to those for FASTQ.

Notably, the benefits of PA compared to MA and SSA- LCE in terms of time remain significant even in these harder, dense instances, where the values of $b'$ increase compared to the $b'$ values in the analogous sparse instances (see Figure 1). Consider the instance for $b = 10\% \cdot n$ on AMAZON (Figure 2f). We still have that $b' = 14407500 < \lfloor b/\log b \rfloor = 34166616$. In these dense instances, SSA- LCE consumes much less memory than MA and PA; this is expected as SSA- LCE works in the restore model: it overwrites (and thus uses the space occupied by) $T$.

[13] Arguably, these sparse instances are the interesting instances for *sparse* suffix sorting.




(a) Time for varying $n$ (b) Time for varying $b$ (c) Memory for varying $n$ (d) Memory for varying $b$

(e) Time for varying $n$ (f) Time for varying $b$ (g) Memory for varying $n$ (h) Memory for varying $b$

(i) Time for varying $n$ (j) Time for varying $b$ (k) Memory for varying $n$ (l) Memory for varying $b$

**Fig. 2** Results for dense instances of FASTQ (top row), AMAZON (middle row) and RANDOM (bottom row). The exact value of $b'$ is on top of the points of the PA curves to highlight the relevance of Theorem 2. As expected, $b'$ decreases with increasing $n$ and increases with increasing $b$. By default, we used $b = 6\% \cdot n$ when varying $n$; and the whole dataset when varying $b$

**Acknowledgements** This work is partially supported by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively. Hilde Verbeek is supported by a Constance van Eeden Fellowship.

**Author Contributions** S.P.P. designed the study. All authors contributed to the design of algorithms. L.A.K.A., G.L., and S.P.P. wrote the code. All authors contributed in writing the manuscript. All authors reviewed the manuscript.

**Data Availability** The datasets used are publicly available at the provided links.

## Declarations

**Competing interests** The authors declare no competing interests.

## References

1. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Cai, J., Wong, C.K. (eds.) Computing and Combinatorics, Second Annual International Conference, COCOON '96, Hong Kong, June 17-19, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1090, pp. 219–230. Springer, (1996). https://doi.org/10.1007/3-540-61332-3_155
2. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings. Lecture Notes in Computer Science, vol. 2089, pp. 181–192. Springer, (2001). https://doi.org/10.1007/3-540-48194-X_17
3. Navarro, G., Prezza, N.: Universal compressed text indexing. Theor. Comput. Sci. **762**, 41–50 (2019). https://doi.org/10.1016/j.tcs.2018.09.007
4. Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. ACM Trans. Algorithms **17**(1), 8–1839 (2021). https://doi.org/10.1145/3426473
5. Grabowski, S., Raniszewski, M.: Sampled suffix array with minimizers. Softw. Pract. Exp. **47**(11), 1755–1771 (2017). https://doi.org/10.1002/spe.2481
6. Loukides, G., Pissis, S.P.: Bidirectional string anchors: A new string sampling mechanism. In: Mutzel, P., Pagh, R., Herman, G. (eds.) 29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference). LIPIcs, vol. 204, pp. 64–16421. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2021). https://doi.org/10.4230/LIPIcs.ESA.2021.64
7. Loukides, G., Pissis, S.P., Sweering, M.: Bidirectional string anchors for improved text indexing and top-$k$ similarity search. IEEE Trans. Knowl. Data Eng. **35**(11), 11093–11111 (2023). https://doi.org/10.1109/TKDE.2022.3231780
8. Ayad, L.A.K., Loukides, G., Pissis, S.P.: Text indexing for long patterns: Anchors are all you need. Proc. VLDB Endow. **16**(9), 2117–2131 (2023)
9. Ben-Nun, S., Golan, S., Kociumaka, T., Kraus, M.: Time-space tradeoffs for finding a long common substring. In: Gørtz, I.L., Weimann, O. (eds.) 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark. LIPIcs, vol. 161, pp. 5–1514. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2020). https://doi.org/10.4230/LIPIcs.CPM.2020.5
10. Bernardini, G., Fici, G., Gawrychowski, P., Pissis, S.P.: Substring complexity in sublinear space. In: Iwata, S., Kakimura, N. (eds.) 34th International Symposium on Algorithms and Computation, ISAAC 2023, December 3-6, 2023, Kyoto, Japan. LIPIcs, vol. 283, pp. 12–11219. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2023). https://doi.org/10.4230/LIPICS.ISAAC.2023.12
11. Bathie, G., Charalampopoulos, P., Starikovskaya, T.: Internal pattern matching in small space and applications. In: Inenaga, S., Puglisi, S. J. (eds.) 35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, Fukuoka, Japan, June 25–27, 2024, pp. 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPIcs.CPM.2024.4
12. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006). https://doi.org/10.1145/1217856.1217858
13. Bille, P., Fischer, J., Gørtz, I.L., Kopelowitz, T., Sach, B., Vildhøj, H.W.: Sparse text indexing in small space. ACM Trans. Algorithms **12**(3), 39–13919 (2016). https://doi.org/10.1145/2836166
14. I, T., Kärkkäinen, J., Kempa, D.: Faster sparse suffix sorting. In: Mayr, E.W., Portier, N. (eds.) 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France. LIPIcs, vol. 25, pp. 386–396. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2014). https://doi.org/10.4230/LIPIcs.STACS.2014.386

15. Gawrychowski, P., Kociumaka, T.: Sparse suffix tree construction in optimal time and space. In: Klein, P.N. (ed.) Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pp. 425–439. SIAM, (2017). https://doi.org/10.1137/1.9781611974782.27
16. Birenzwige, O., Golan, S., Porat, E.: Locally consistent parsing for text indexing in small space. In: Chawla, S. (ed.) Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pp. 607–626. SIAM, (2020). https://doi.org/10.1137/1.9781611975994.37
17. Kosolobov, D., Sivukhin, N.: Construction of sparse suffix trees and LCE indexes in optimal time and space. In: Inenaga, S., Puglisi, S.J. (eds.) 35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, Fukuoka, Japan, June 25–27, 2024, pp. 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2024). https://doi.org/10.4230/LIPIcs.CPM.2024.20
18. Chan, T.M., Munro, J.I., Raman, V.: Selection and sorting in the "restore" model. ACM Trans. Algorithms **14**(2), 11–11118 (2018). https://doi.org/10.1145/3168005
19. Fischer, J., I, T., Köppl, D.: Deterministic sparse suffix sorting in the restore model. ACM Trans. Algorithms **16**(4), 50–15053 (2020) https://doi.org/10.1145/3398681
20. Prezza, N.: Optimal substring equality queries with applications to sparse text indexing. ACM Trans. Algorithms **17**(1), 7–1723 (2021). https://doi.org/10.1145/3426870
21. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987). https://doi.org/10.1147/rd.312.0249
22. Ayad, L.A.K., Loukides, G., Pissis, S.P.: Sparse suffix and LCP array: simple, direct, small, and fast. In: Soto, J.A., Wiese, A. (eds.) LATIN 2024: Theoretical Informatics - 16th Latin American Symposium, Puerto Varas, Chile, March 18–22, 2024, Proceedings, Part I. Lecture Notes in Computer Science, pp. 162–177. Springer, (2024). https://doi.org/10.1007/978-3-031-55598-_11
23. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable (extended abstract). In: Kuich, W. (ed.) Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings. Lecture Notes in Computer Science, vol. 623, pp. 235–246. Springer, (1992). https://doi.org/10.1007/3-540-55719-9_77
24. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987). https://doi.org/10.1137/0216062
25. Bender, M.A., Conway, A., Farach-Colton, M., Kuszmaul, W., Tagliavini, G.: Iceberg hashing: Optimizing many hash-table criteria at once. J. ACM **70**(6) (2023) https://doi.org/10.1145/3625817
26. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: 51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA, pp. 787–796. IEEE Computer Society, (2010). https://doi.org/10.1109/FOCS.2010.80
27. Salowe, J.S., Steiger, W.L.: Simplified stable merging tasks. J. Algorithms **8**(4), 557–571 (1987). https://doi.org/10.1016/0196-6774(87)90050-2
28. Katajainen, J., Pasanen, T., Teuhola, J.: Practical in-place mergesort. Nord. J. Comput. **3**(1), 27–40 (1996)
29. Franceschini, G., Muthukrishnan, S., Puatracscu, M.: Radix sorting with no extra space. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4698, pp. 194–205. Springer, (2007). https://doi.org/10.1007/978-3-540-75520-3_19
30. Bollobás, B., Letzter, S.: Longest common extension. Eur. J. Comb. **68**, 242–248 (2018). https://doi.org/10.1016/j.ejc.2017.07.019