Linear Work Suffix Array Construction

Juha Kärkkäinen* Peter Sanders† Stefan Burkhardt‡

Abstract

Suffix trees and suffix arrays are widely used and largely interchangeable index structures on strings and sequences. Practitioners prefer suffix arrays due to their simplicity and space efficiency while theoreticians use suffix trees due to linear-time construction algorithms and more explicit structure. We narrow this gap between theory and practice with a simple linear-time construction algorithm for suffix arrays. The simplicity is demonstrated with a C++ implementation of 50 effective lines of code. The algorithm is called DC3, which stems from the central underlying concept of *difference cover*. This view leads to a generalized algorithm, DC, that allows a space-efficient implementation and, moreover, supports the choice of a space–time tradeoff. For any $v \in [1, \sqrt{n}]$, it runs in $\mathcal{O}(vn)$ time using $\mathcal{O}(n/\sqrt{v})$ space in addition to the input string and the suffix array. We also present variants of the algorithm for several parallel and hierarchical memory models of computation. The algorithms for BSP and EREW-PRAM models are asymptotically faster than all previous suffix tree or array construction algorithms.

1 Introduction

The *suffix tree* [60] of a string is the compact trie of all its suffixes. It is a powerful data structure with numerous applications in computational biology [28] and elsewhere [26]. It can be constructed in linear time in the length of the string [16, 19, 48, 57, 60]. The *suffix array* [24, 45] is the lexicographically sorted array of the suffixes of a string. It contains much the same information as the suffix tree, although in a more implicit form, but is a simpler and more compact data structure for many applications [2, 7, 24, 45]. However, until recently, the only linear-time construction algorithm was based on a lexicographic traversal of the suffix tree.

*Department of Computer Science, P.O.Box 68 (Gustaf Hällströmin katu 2b) FI-00014 University of Helsinki, Finland, juha.karkkainen@cs.helsinki.fi. Supported by the Academy of Finland grant 201560.

†Universität Karlsruhe, 76128 Karlsruhe, Germany, sanders@ira.uka.de.

‡Google Inc, 1600 Amphitheatre Parkway, 94043 Mountain View, CA, USA, Burkhardt@Google.com.

Due to a more explicit structure and the direct linear-time construction algorithms, theoreticians tend to prefer suffix trees over suffix arrays. This is evident, for example, in text books, including recent ones [13, 56]. Practitioners, on the other hand, often use suffix arrays, because they are more space-efficient and simpler to implement. This difference of theoretical and practical approaches appears even within a single paper [50].

We address the gap between theory and practice by describing the first direct linear-time suffix array construction algorithm, elevating suffix arrays to equals of suffix trees in this sense. Independently and simultaneously to our result, which originally appeared in [36], two different linear-time algorithms were introduced by Kim et al. [40], and Ko and Aluru [41]. In this paper, we will also introduce several extensions and generalizations of the algorithm, including space-efficient, parallel and external memory variants.

Linear-time algorithm. Many linear-time suffix tree construction algorithms are truly linear-time only for *constant alphabet*, i.e, when the size of the alphabet is constant [48, 57, 60]. Farach [16] introduced the first algorithm to overcome this restriction; it works in linear-time for *integer alphabet*, i.e., when the characters are integers from a linear-sized range. This is a significant improvement, since a string over any alphabet can be transformed into such a string by sorting the characters and replacing them with their ranks. This preserves the structure of the suffix tree and the order of the suffixes. Consequently, the complexity of constructing the suffix tree of a string is the same as the complexity of sorting the characters of the string [19].

Whereas the algorithms requiring a constant alphabet are incremental, adding one suffix or one character at a time to the tree, Farach's algorithm takes the following half-recursive divide-and-conquer approach:

1. Construct the suffix tree of the suffixes starting at odd positions. This is done by reduction to the suffix tree construction of a string of half the length, which is solved recursively.

2. Construct the suffix tree of the remaining suffixes using the result of the first step.

3. Merge the two suffix trees into one.

The crux of the algorithm is the merging step, which is an intricate and complicated procedure.

The same structure appears in some parallel and external memory suffix tree construction algorithms [17, 18, 19] as well as the direct linear-time suffix array construction algorithm of Kim et al. [40]. In all cases, the merge is a very complicated procedure. The linear-time suffix array construction algorithm of Ko and Aluru [41]

also uses the divide-and-conquer approach of first sorting a subset or *sample* of suffixes by recursion. However, its choice of the sample and the rest of the algorithm are quite different.

We introduce a linear-time suffix array construction algorithm following the structure of Farach's algorithm but using 2/3-recursion instead of half-recursion:

1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.

2. Construct the suffix array of the remaining suffixes using the result of the first step.

3. Merge the two suffix arrays into one.

Surprisingly, the use of two thirds instead of half of the suffixes in the first step makes the last step almost trivial: simple comparison-based merging is sufficient. For example, to compare suffixes starting at i and j with $i \bmod 3 = 0$ and $j \bmod 3 = 1$, we first compare the initial characters, and if they are the same, we compare the suffixes starting at $i + 1$ and $j + 1$, whose relative order is already known from the first step.

Space-efficient algorithms. All the above suffix array construction algorithms require at least n pointers or integers of *extra space* in addition to the n characters of the input and the n pointers/integers of the suffix array. Until recently, this was true for all algorithms running in $\mathcal{O}(n \log n)$ time. There are also so-called *lightweight* algorithms that use significantly less extra space [47, 4], but their worst-case time complexity is $\Omega(n^2)$. Manzini and Ferragina [47] have raised the question of whether it is possible to achieve $\mathcal{O}(n \log n)$ runtime using sublinear extra space.

The question was answered positively by Burkhardt and Kärkkäinen [6] with an algorithm running in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n/\sqrt{\log n})$ extra space. They also gave a generalization running in $\mathcal{O}(n \log n + nv)$ time and $\mathcal{O}(n/\sqrt{v})$ extra space for any $v \in [3, n]$. In this paper, combining ideas from [6] with the linear-time algorithm, we improve the result to $\mathcal{O}(nv)$ time in $\mathcal{O}(n/\sqrt{v})$ extra space, leading to an $o(n \log n)$ time and $o(n)$ extra space algorithm.

To achieve the result, we generalize the linear-time algorithm so that the sample of suffixes sorted in the first step can be chosen from a family, called the *difference cover samples*, that includes arbitrarily sparse samples. This family was introduced in [6] and its name comes from a characterization using the concept of *difference cover*. Difference covers have also been used for VLSI design [39], distributed mutual exclusion [44, 10], and quantum computing [5].

An even more space-efficient approach is to construct compressed indexes [21, 27, 42, 31, 32]. However, for constant v our algorithm is a factor of at least $\Theta(\log \log \sigma)$ faster than these algorithms, where σ is the alphabet size.

Table 1: Suffix array construction algorithms. The algorithms in [16, 17, 18, 19] are *indirect*, i.e., they actually construct a suffix *tree*, which can be then be transformed into a suffix array

model of computation	complexity	alphabet	source
RAM	$\mathcal{O}(n \log n)$ time	general	[45, 43, 6]
	$\mathcal{O}(n)$ time	integer	[16, 40, 41],DC
External Memory [59] D disks, block size B, fast memory of size M	$\mathcal{O}(\frac{n}{DB} \log_{\frac{M}{B}} \frac{n}{B} \log n)$ I/Os $\mathcal{O}(n \log_{\frac{M}{B}} \frac{n}{B} \log n)$ internal work	integer	[12]
	$\mathcal{O}(\frac{n}{DB} \log_{\frac{M}{B}} \frac{n}{B})$ I/Os $\mathcal{O}(n \log_{\frac{M}{B}} \frac{n}{B})$ internal work	integer	[19],DC
Cache Oblivious [22] M/B cache blocks of size B	$\mathcal{O}(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} \log n)$ cache faults	general	[12]
	$\mathcal{O}(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ cache faults	general	[19],DC
BSP [58] P processors h-relation in time $L + gh$	$\mathcal{O}(\frac{n \log n}{P} + (L + \frac{gn}{P}) \frac{\log^3 n \log P}{\log(n/P)})$ time	general	[17]
	$\mathcal{O}(\frac{n \log n}{P} + L \log^2 P + \frac{gn \log n}{P \log(n/P)})$ time	general	DC
$P = \mathcal{O}(n^{1-\epsilon})$ processors	$\mathcal{O}(n/P + L \log^2 P + gn/P)$ time	integer	DC
EREW-PRAM [33]	$\mathcal{O}(\log^4 n)$ time, $\mathcal{O}(n \log n)$ work	general	[17]
	$\mathcal{O}(\log^2 n)$ time, $\mathcal{O}(n \log n)$ work	general	DC
arbitrary-CRCW-PRAM [33]	$\mathcal{O}(\log n)$ time, $\mathcal{O}(n)$ work (rand.)	constant	[18]
priority-CRCW-PRAM [33]	$\mathcal{O}(\log^2 n)$ time, $\mathcal{O}(n)$ work (rand.)	constant	DC

Advanced models of computation. Since our algorithm is constructed from well studied building blocks like integer sorting and merging, simple direct suffix array construction algorithms for several models of computation are almost a corollary. Table 1 summarizes these results. We win a factor $\Theta(\log n)$ over the previously best direct external memory algorithm. For BSP and EREW-PRAM models, we obtain an improvement over *all* previous results, including the first linear work BSP algorithm.

Overview. The paper is organized as follows. Section 3 explains the basic linear time algorithm DC3. We then use the concept of a difference cover introduced in Section 4 to describe a generalized algorithm called DC in Section 5 that leads to a space efficient algorithm in Section 6. Section 7 explains implementations of the DC3 algorithm in advanced models of computation. The results together with some open issues are discussed in Section 8.

2 Notation

We use the shorthands $[i,j] = \{i,\ldots,j\}$ and $[i,j) = [i,j-1]$ for ranges of integers and extend to substrings as seen below.

The **input** of a suffix array construction algorithm is a *string* $T = T[0,n) = t_0 t_1 \cdots t_{n-1}$ over the alphabet $[1,n]$, that is, a sequence of n integers from the range $[1,n]$. For convenience, we assume that $t_j = 0$ for $j \geq n$. Sometimes we also assume that $n+1$ is a multiple of some constant v or a square to avoid a proliferation of trivial case distinctions and $\lceil\cdot\rceil$ operations. An implementation will either spell out the case distinctions or pad (sub)problems with an appropriate number of zero characters. The restriction to the alphabet $[1,n]$ is not a serious one. For a string T over any alphabet, we can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1,n]$ by renaming the characters of T with their ranks. Since the renaming is order preserving, the order of the suffixes does not change.

For $i \in [0,n]$, let S_i denote the *suffix* $T[i,n) = t_i t_{i+1} \cdots t_{n-1}$. We also extend the notation to sets: for $C \subseteq [0,n]$, $S_C = \{S_i \mid i \in C\}$. The goal is to sort the set $S_{[0,n]}$ of suffixes of T, where comparison of substrings or tuples assumes the lexicographic order throughout this paper. The **output** is the *suffix array* $SA[0,n]$ of T, a permutation of $[0,n]$ satisfying $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n]}$.

3 Linear-time algorithm

We begin with a detailed description of the simple linear-time algorithm, which we call DC3 (for Difference Cover modulo 3, see Section 4). A complete implementation in C++ is given in Appendix A. The execution of the algorithm is illustrated with the following example

$$\begin{array}{rl} & \texttt{0 1 2 3 4 5 6 7 8 9 10 11} \\ T[0,n) = & \texttt{y a b b a d a b b a d o} \end{array}$$

where we are looking for the suffix array

$$SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)\ .$$

Step 0: Construct a sample. For $k = 0, 1, 2$, define

$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and S_C the set of *sample suffixes*.

Example. $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, i.e., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Step 1: Sort sample suffixes. For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}]$$

whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of R_k is always unique because $t_{\max B_k+2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. Then the (nonempty) suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \ldots$ corresponds to S_i. The correspondence is order preserving, i.e., by sorting the suffixes of R we get the order of the sample suffixes S_C.

Example. $R = [\texttt{abb}][\texttt{ada}][\texttt{bba}][\texttt{do0}][\texttt{bba}][\texttt{dab}][\texttt{bad}][\texttt{o00}]$.

To sort the suffixes of R, first radix sort the characters of R and rename them with their ranks to obtain the string R'. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of R' using Algorithm DC3.

Example. $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

Example.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Step 2: Sort nonsample suffixes. Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \le S_j \iff (t_i, rank(S_{i+1})) \le (t_j, rank(S_{j+1})).$$

The pairs $(t_i, rank(S_{i+1}))$ are then radix sorted.

Example. $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(\texttt{0}, 0) < (\texttt{a}, 5) < (\texttt{a}, 7) < (\texttt{b}, 2) < (\texttt{y}, 1)$.

Step 3: Merge. The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$\begin{aligned} i \in B_1: &\quad S_i \le S_j \iff (t_i, rank(S_{i+1})) \le (t_j, rank(S_{j+1})) \\ i \in B_2: &\quad S_i \le S_j \iff (t_i, t_{i+1}, rank(S_{i+2})) \le (t_j, t_{j+1}, rank(S_{j+2})) \end{aligned}$$

Note that the ranks are defined in all cases.

Example. $S_1 < S_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$ and $S_3 < S_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

The time complexity is established by the following theorem.

Theorem 1. *The time complexity of Algorithm DC3 is $\mathcal{O}(n)$.*

Proof. Excluding the recursive call, everything can clearly be done in linear time. The recursion is on a string of length $\lceil 2n/3 \rceil$. Thus the time is given by the recurrence $T(n) = T(2n/3) + \mathcal{O}(n)$, whose solution is $T(n) = \mathcal{O}(n)$. □

4 Difference cover sample

The sample of suffixes in DC3 is a special case of a *difference cover sample*. In this section, we describe what difference cover samples are, and in the next section we give a general algorithm based on difference cover samples.

The sample used by the algorithms has to satisfy two *sample conditions*:

1. The sample itself can be sorted efficiently. Only certain special cases are known to satisfy this condition (see [37, 3, 9, 41] for examples). For example, a random sample would not work for this reason. Difference cover samples can be sorted efficiently because they are *periodic* (with a small period). Steps 0 and 1 of the general algorithm could be modified for sorting any periodic sample of size m with period length v in $\mathcal{O}(vm)$ time.

2. The sorted sample helps in sorting the set of all suffixes. The set of difference cover sample positions has the property that for any $i, j \in [0, n - v + 1]$ there is a small ℓ such that both $i + \ell$ and $j + \ell$ are sample positions. See Steps 2–4 in Section 5 for how this property is utilized in the algorithm.

The difference cover sample is based on difference covers [39, 10].

Definition 1. A set $D \subseteq [0, v)$ is a *difference cover* modulo v if

$$\{(i - j) \bmod v \mid i, j \in D\} = [0, v) \ .$$

Definition 2. A v-periodic sample C of $[0, n]$ with the period D, i.e.,

$$C = \{i \in [0, n] \mid i \bmod v \in D\} ,$$

is a *difference cover sample* if D is a difference cover modulo v.

By being periodic, a difference cover sample satisfies the first of the sample conditions. That it satisfies the second condition is shown by the following lemma.

Lemma 1. *If D is a difference cover modulo v, and i and j are integers, there exists $\ell \in [0, v)$ such that $(i + \ell) \bmod v$ and $(j + \ell) \bmod v$ are in D.*

Proof. By the definition of difference cover, there exists $i', j' \in D$ such that $i' - j' \equiv i - j \pmod{v}$. Let $\ell = (i' - i) \bmod v$. Then

$$\begin{aligned} i + \ell &\equiv i' \in D \pmod{v} \\ j + \ell &\equiv i' - (i - j) \equiv j' \in D \pmod{v} . \end{aligned}$$ □

Note that by using a lookup table of size v that maps $(i - j) \bmod v$ into i', the value ℓ can be computed in constant time.

The size of the difference cover is a key parameter for the space-efficient algorithm in Sections 6. Clearly, $\sqrt{v}$ is a lower bound. The best general upper bound that we are aware of is achieved by a simple algorithm due to Colbourn and Ling [10]:

Lemma 2 ([10]). *For any v, a difference cover modulo v of size at most $\sqrt{1.5v} + 6$ can be computed in $\mathcal{O}(\sqrt{v})$ time.*

The sizes of the smallest known difference covers for several period lengths are shown in Table 2.

Table 2: The size of the smallest known difference cover D modulo v for several period lengths v. The difference covers were obtained from [44] ($v \leq 64$) and [6] ($v = 128, 256$), or computed using the algorithm of Colbourn and Ling [10] ($v \geq 512$). For $v \leq 128$, the sizes are known to be optimal

v	3	7	13	21	31	32	64	128	256	512	1024	2048
$\lvert D\rvert$	2	3	4	5	6	7	9	13	20	28	40	58

5 General algorithm

The algorithm DC3 sorts suffixes with starting positions in a difference cover sample modulo 3 and then uses these to sort all suffixes. In this section, we present a generalized algorithm DC that can use any difference cover D modulo v.

Step 0: Construct a sample. For $k \in [0, v)$, define

$$B_k = \{i \in [0, n] \mid i \bmod v = k\}.$$

The set of sample positions is now $C = \bigcup_{k \in D} B_k$. Let $\bar{D} = [0, v) \setminus D$ and $\bar{C} = [0, n] \setminus C$.

Step 1: Sort sample suffixes. For $k \in D$, construct the strings

$$R_k = [t_k t_{k+1} \dots t_{k+v-1}][t_{k+v} t_{k+v+1} \dots t_{k+2v-1}] \dots [t_{\max B_k} \dots t_{\max B_k + v - 1}].$$

Let $R = \bigodot_{k \in D} R_k$, where $\bigodot$ denotes a concatenation. The (nonempty) suffixes of R correspond to the sample suffixes, and they are sorted recursively (using any period length from 3 to $(1 - \epsilon)v^2/|D|$ to ensure the convergence of the recursion).

Let $rank(S_i)$ be defined as in DC3 for $i \in C$, and additionally define $rank(S_{n+1}) = rank(S_{n+2}) = \dots = rank(S_{n+v-1}) = 0$. Again, $rank(S_i)$ is undefined for $i \in \bar{C}$.

Step 2: Sort nonsample suffixes. Sort each S_{B_k}, $k \in \bar{D}$, separately. Let $k \in \bar{D}$ and let $\ell \in [0, v)$ be such that $(k + \ell) \bmod v \in D$. To sort S_{B_k}, represent each suffix $S_i \in S_{B_k}$ with the tuples $(t_i, t_{i+1}, \dots, t_{i+\ell-1}, rank(S_{i+\ell}))$. Note that the rank is always defined. The tuples are then radix sorted.

The most straightforward generalization of DC3 would now merge the sets S_{B_k}, $k \in \bar{D}$. However, this would be a $\Theta(v)$-way merging with $\mathcal{O}(v)$-time comparisons giving $\mathcal{O}(nv \log v)$ time complexity. Therefore, we take a slightly different approach.

Step 3: Sort by first v characters. Separate the sample suffixes S_C into sets S_{B_k}, $k \in D$, keeping each set ordered. Then we have all the sets S_{B_k}, $k \in [0, v)$, as sorted sequences. Concatenate these sequences and sort the result stably by the first v characters.

For $\alpha \in [0, n]^v$, let S^α be the set of suffixes starting with α, and let $S^\alpha_{B_k} = S^\alpha \cap S_{B_k}$. The algorithm has now separated the suffixes into the sets $S^\alpha_{B_k}$, each of which is correctly sorted. The sets are also grouped by α and the groups are sorted. For $v = 3$, the situation could look like this:

$S^{aaa}_{B_0}$	$S^{aaa}_{B_1}$	$S^{aaa}_{B_2}$	$S^{aab}_{B_0}$	$S^{aab}_{B_1}$	$S^{aab}_{B_2}$	$S^{aac}_{B_0}$	$\cdots$

Step 4: Merge. For each $\alpha \in [0, n]^v$, merge the sets $S^\alpha_{B_k}$, $k \in [0, v)$, into the set S^α. This completes the sorting.

The merging is done by a comparison-based v-way merging. For $i, j \in [0, n]$, let $\ell \in [0, v)$ be such that $(i + \ell) \bmod v$ and $(j + \ell) \bmod v$ are both in D. Suffixes S_i and S_j are compared by comparing $rank(S_{i+\ell})$ and $rank(S_{j+\ell})$. This gives the correct order because S_i and S_j belong to the same set S^α and thus $t_i t_{i+1} \dots t_{i+\ell-1} = t_j t_{j+1} \dots t_{j+\ell-1}$.

Theorem 2. *The time complexity of Algorithm DC is $\mathcal{O}(vn)$.*

6 Lightweight algorithm

We now explain how to implement algorithm DC using only $\mathcal{O}(n/\sqrt{v})$ space in addition to the input and the output. We will however reuse the space for the output array $a[0, n]$ as intermediate storage.

Step 0:. The sample can be represented using their $\mathcal{O}(n/\sqrt{v})$ starting positions.

Step 1:. To sort the sample suffixes, we first (non-inplace) radix sort the v-tuples that start at sample positions. This is easy using two arrays of size $\mathcal{O}(n/\sqrt{v})$ for storing starting positions of samples and $n + 1$ counters (one for each character) stored in the output array $a[0, n]$. This is analogous to the arrays `R`, `SA12` and `c` used in Appendix A. Renaming the tuples with ranks only needs the $\mathcal{O}(n/\sqrt{v})$ space for the recursive subproblem. The same space bound applies to the suffix array of the sample and the space needed within the recursive call.

Step 2:. Sorting nonsample suffixes for a particular class S_{B_k}, $k \in \bar{D}$ amounts to radix sorting $(n + 1)/v$ many tuples of up to v integers in the range $[0, n]$. Similar to Step 1, we need only space $\mathcal{O}(n/v)$ for describing these tuples. However, we now arrange the sorted tuples in $a[k(n+1)/v, (k+1)(n+1)/v)$ so that the output array is not available for counters as in Step 1. We solve this problem by viewing each character as two subcharacters in the range $[0, \sqrt{n+1})$.

Step 3: Sort by first v characters. Scan the suffix array of the sample and store the sample suffixes S_{B_k}, $k \in C$ in $a[k(n+1)/v, (k+1)(n+1)/v)$ maintaining the order given by the sample within each class S_{B_k}. Together with the computation in Step 2, the output array now stores the desired concatenation of all sorted sets S_{B_k}. We now sort *all* suffixes stably by their first v characters. Using a counter array of size $\mathcal{O}(\sqrt{n})$ we can do that in $2v$ passes, total time $\mathcal{O}(vn)$, and additional space $\mathcal{O}(n^{3/4})$ by applying the almost inplace distribution sorting algorithm from Theorem 5 in the appendix with $k = \sqrt{n+1}$. Note that for $v = \mathcal{O}(\sqrt{n})$, $n^{3/4} = \mathcal{O}(n/\sqrt{v})$.

Step 4: Merge. Suppose S^α is stored in $a[b, b')$. This array consists of v consecutive (possibly empty) subarrays that represent $S^\alpha_{B_k}$, $k \in [0, v)$ respectively. We can merge them with $\mathcal{O}(\sqrt{|S^\alpha|v})$ additional space using the almost inplace merging routine (see Theorem 6 in the appendix). Note that for $v = \mathcal{O}(\sqrt{n})$, $\sqrt{|S^\alpha|v} = \mathcal{O}(n/\sqrt{v})$.

Theorem 3. *For $v = \mathcal{O}(\sqrt{n})$, algorithm DC can be implemented to run in time $\mathcal{O}(vn)$ using additional space $\mathcal{O}(n/\sqrt{v})$.*

The upper bound for v can be increased to $\mathcal{O}(n^{2/3})$ by using the comparison based algorithm from [6] when $v = \omega(\sqrt{n})$.

7 Advanced models

In this section, we adapt the DC3 algorithm for several advanced models of computation. We first explain the main ideas and then bundle the results in Theorem 4 below.

The adaptation to memory hierarchies is easy since all operations can be described in terms of scanning, sorting, and permuting sequences of tuples using standard techniques. Since scanning is trivial and since permuting is equivalent to sorting, all we really need is a good external sorting algorithm. The proof therefore concentrates on bounding the internal work associated with integer sorting.

Parallelization is slightly more complicated since the scanning needed to find the ranks of elements looks like a sequential process on the first glance. However, the technique to overcome this is also standard: Consider a sorted array $a[0, n]$. Define $c[0] = 1$ and $c[i] = 1$ if $c[i-1] \neq c[i]$ and $c[i] = 0$ otherwise for $i \in [1, n]$. Now the *prefix sums* $\sum_{i \in [0,j]} c[i]$ give the rank of $a[j]$. Computing prefix sums in parallel is again a well studied problem.

Theorem 4. *The DC3 algorithm can be implemented to achieve the following performance guarantees on advanced models of computation:*

model of computation	complexity	alphabet
External Memory [59] D disks, block size B, fast memory of size M	$\mathcal{O}(\frac{n}{DB} \log_{\frac{M}{B}} \frac{n}{B})$ I/Os $\mathcal{O}(n \log_{\frac{M}{B}} \frac{n}{B})$ internal work	integer
Cache Oblivious [22]	$\mathcal{O}(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ cache faults	general
BSP [58] P processors h-relation in time $L + gh$	$\mathcal{O}(\frac{n \log n}{P} + L \log^2 P + \frac{gn \log n}{P \log(n/P)})$ time	general
$P = \mathcal{O}(n^{1-\epsilon})$ processors	$\mathcal{O}(n/P + L \log^2 P + gn/P)$ time	integer
EREW-PRAM [33]	$\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n \log n)$ work	general
priority-CRCW-PRAM [33]	$\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n)$ work (randomized)	constant

Proof. **External memory:** Step 1 of the DC3 algorithm begins by scanning the input and producing tuples $([t_{3i+k} t_{3i+k+1} t_{3i+k+2}], 3i + k)$ for $k \in \{1, 2\}$ and $3i + k \in [0, n]$. These tuples are then sorted by lexicographic order of the character triples. The results are scanned producing rank position pairs $(r_{3i+k}, 3i + k)$. Constructing a recursive problem instance then amounts to sorting using the lexicographic order of (k, i) for comparing positions of the form $3i + k$. Similarly, assigning ranks to a sample suffix j at position i in the suffix array of the sample amounts to sorting pairs of the form (i, j).

Step 2 sorts triples of the form $(t_i, rank(S_{i+1}), i)$. Step 3 represents S_{3i} as $(t_{3i}, t_{3i+1}, rank(S_{3i+1}), rank(S_{3i+2}), 3i)$, S_{3i+1} as $(t_{3i+1}, rank(S_{3i+2}), 3i+1)$, and S_{3i+2} as $(t_{3i+2}, t_{3i+3}, rank(S_{3i+4})), 3i+2)$. This way all the information needed for comparisons is available. These representations are produced using additional sorting and scanning passes. A more detailed description and analysis of external DC3 is given in [14]. It turns out that the total I/O volume is equivalent to the amount I/O needed for sorting $30n$ words of memory plus the I/O needed for scanning $6n$ words.

All in all, the complexity of external suffix array construction is governed by the effort for sorting objects consisting of a constant number of machine words. The keys are integers in the range $[0, n]$, or pairs or triples of such integers. I/O optimal deterministic[1] parallel disk sorting algorithms are well known [53, 52]. We have to make a few remarks regarding internal work however. To achieve optimal internal work for all values of n, M, and B, we can use radix sort where the most significant digit has $\lfloor \log M \rfloor - 1$ bits and the remaining digits have $\lfloor \log M/B \rfloor$ bits. Sorting then starts with $\mathcal{O}(\log_{M/B} n/M)$ data distribution phases that need linear work each and can be implemented using $\mathcal{O}(n/DB)$ I/Os using the same I/O strategy as in [52]. It remains to stably sort the elements by their $\lfloor \log M \rfloor - 1$ most significant bits. This is also done using multiple phases of distribution sorting similar to [52] but we can now afford to count how often each key appears and use this information to produce splitters that perfectly balance the bucket sizes (we may have large buckets with identical keys but this is no problem because no further sorting is required for them). Mapping keys to buckets can use lookup tables of size $\mathcal{O}(M)$.

Cache oblivious: These algorithms are similar to external algorithms with a single disk but they are not allowed to make explicit use of the block size B or the internal memory size M. This is a serious restriction here since no cache oblivious integer sorting with $\mathcal{O}(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache faults and $o(n \log n)$ work is known. Hence, we can as well go to the comparison based alphabet model. The result is then an immediate corollary of the optimal comparison based sorting algorithm [22].

EREW PRAM: We can use Cole's merge sort [11] for parallel sorting and merging. For an input of size m and P processors, Cole's algorithm takes time $\mathcal{O}((m \log m)/P + \log P)$. The i-th level or recursion has an input of size $n(2/3)^i$ and thus takes time $(2/3)^i \mathcal{O}((n \log n)/P + \log P)$. After $\Theta(\log P)$ levels of recursion, the problem size has reduced so far that the remaining subproblem can be solved in time $\mathcal{O}((n/P \log(n/P))$ on a single processor. We get an overall execution time of $\mathcal{O}((n \log n)/P + \log^2 P)$.

BSP: For the case of many processors, we proceed as for the EREW-PRAM algorithm using the optimal comparison based sorting algorithm [25] that takes time $\mathcal{O}((n \log n)/P + (gn/P + L)\frac{\log n}{\log(n/P)})$.

[1] Simpler randomized algorithms with favorable constant factors are also available [15].

For the case of few processors, we can use a linear work sorting algorithm based on radix sort [8] and a linear work merging algorithm [23]. The integer sorting algorithm remains applicable at least during the first $\Theta(\log\log n)$ levels of recursion of the DC3 algorithm. Then we can afford to switch to a comparison based algorithm without increasing the overall amount of internal work.

CRCW PRAM: We employ the stable integer sorting algorithm [55] that works in $\mathcal{O}(\log n)$ time using linear work for keys with $\mathcal{O}(\log\log n)$ bits. This algorithm can be used for the first $\Theta(\log\log\log n)$ iterations for constant input alphabets. Then we can afford to switch to the algorithm [29] that works for keys with $\mathcal{O}(\log n)$ bits at the price of being inefficient by a factor $\mathcal{O}(\log\log n)$. Comparison based merging can be implemented with linear work and $\mathcal{O}(\log n)$ time using [30]. □

The resulting algorithms are simple except that they may use complicated subroutines for sorting to obtain theoretically optimal results. There are usually much simpler implementations of sorting that work well in practice although they may sacrifice determinism or optimality for certain combinations of parameters.

8 Conclusion

The main result of this paper is DC3, a simple, direct, linear time algorithm for suffix sorting with integer alphabets. The algorithm is easy to implement and it can be used as an example for advanced string algorithms even in undergraduate level algorithms courses. Its simplicity also makes it an ideal candidate for implementation on advanced models of computation. For example in [14] we describe an external memory implementation that clearly outperforms the best previously known implementations and several other new algorithms.

The concept of difference covers makes it possible to generalize the DC3 algorithm. This generalized DC algorithm allows space efficient implementation. An obvious remaining question is how to adapt DC to advanced models of computation in a space efficient way. At least for the external memory model this is possible but we only know an approach that needs I/O volume $\Omega(nv^{2.5})$.

The space efficient algorithm can also be adapted to sort an arbitrary set of suffixes by simply excluding the *nonsample* suffixes that we do not want to sort in the Steps 2–4. Sorting a set of m suffixes can be implemented to run in $\mathcal{O}(vm+n\sqrt{v})$ time using $\mathcal{O}(m + n/\sqrt{v})$ additional space. Previously, the only alternatives were string sorting in $\mathcal{O}(mn)$ worst case time or sorting all suffixes using $\mathcal{O}(n)$ additional space. The space efficient Burrows–Wheeler transform in [35] relies on space efficient sorting of subsets of suffixes.

In many applications [1, 2, 34, 38, 45], the suffix array needs to be augmented with the longest common prefix array lcp that stores the length of the longest common prefix of SA_i and SA_{i+1} in lcp[i]. Once the lcp information is known it also

easy to infer advanced search data structures like suffix trees and string B-trees [20]. There are simple linear time algorithms for computing the lcp array from the suffix array [38, 46], but they do not appear to be suitable for parallel or external computation. Farach's algorithm [16] and the other half-recursive algorithms compute the lcp array at each level of the recursion since it is needed for merging. With a similar technique the DC algorithm can be modified to compute the lcp array as a byproduct: If $k = SA[i]$ and $j = SA[i+1]$ then find an ℓ such that $k+\ell$ and $j+\ell$ are both in the sample. If $T[k, k+\ell) \neq T[j, j+\ell)$ then lcp$[i]$ can be computed locally. Otherwise, lcp$[i] = \ell + \text{lcp}(S_{k+\ell}, S_{j+\ell})$. The lcp of $S_{k+\ell}$ and $S_{j+\ell}$ can be approximated within an additive term v from the lcp information of the recursive string R using range minima queries. All these operations can be implemented in parallel or for memory hierarchies using standard techniques.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *LNCS*, pages 449–463. Springer, 2002.

[2] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th Symposium on String Processing and Information Retrieval*, volume 2476 of *LNCS*, pages 31–43. Springer, 2002.

[3] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, 1997.

[5] A. Bertoni, C. Mereghetti, and B. Palano. Golomb rulers and difference sets for succinct quantum automata. *Int. J. Found. Comput. Sci.*, 14(5):871–888, 2003.

[6] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.

[8] A. Chan and F. Dehne. A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4):533–538, 1999.

[9] R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 70–82. Springer, 2003.

[10] C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1–2):9–12, July 2000.

[11] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[12] A. Crauser and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[13] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[14] R. Dementiev, J. Mehnert, and J. Kärkkäinen. Better external memory suffix array construction. In *Workshop on Algorithm Engineering & Experiments*, Vancouver, 2005.

[15] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 138–148. ACM, 2003.

[16] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[17] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 174–183. IEEE, 1998.

[18] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. 23th International Conference on Automata, Languages and Programming*, pages 550–561. IEEE, 1996.

[19] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[20] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[21] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–298. IEEE, 1999.

[23] A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27:809–822, 2001.

[24] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[25] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.

[26] R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. Rapporto di Ricerca CS-96-14, Università "Ca' Foscari" di Venezia, Italy, 1996.

[27] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. 32nd Annual Symposium on Theory of Computing*, pages 397–406. ACM, 2000.

[28] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[29] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd Annual Symposium on Foundations of Computer Science*, pages 628–637. IEEE, 1992.

[30] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW-PRAM. *Information Processing Letters*, 33:181–185, 1989.

[31] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation*, volume 2906 of *LNCS*, pages 240–249. Springer, 2003.

[32] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 251–260. IEEE, 2003.

[33] J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[34] J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In Z. Galil and E. Ukkonen, editors, *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, volume 937 of *LNCS*, pages 191–204. Springer, 1995.

[35] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. In *DIMACS Working Group on The Burrows–Wheeler Transform: Ten Years Later*, Aug. 2004. To appear.

[36] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Conference on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[37] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics*, volume 1090 of *LNCS*, pages 219–230. Springer, 1996.

[38] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[39] J. Kilian, S. Kipnis, and C. E. Leiserson. The organization of permutation architectures with bused interconnections. *IEEE Transactions on Computers*, 39(11):1346–1358, Nov. 1990.

[40] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 186–199. Springer, June 2003.

[41] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 200–210. Springer, June 2003.

[42] T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, volume 2387 of *LNCS*, pages 401–410. Springer, 2002.

[43] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, Sweden, 1999.

[44] W.-S. Luk and T.-T. Wong. Two new quorum based algorithms for distributed mutual exclusion. In *Proc. 17th International Conference on Distributed Computing Systems*, pages 100–106. IEEE, 1997.

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

[46] G. Manzini. Two space saving tricks for linear time LCP array computation. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

[47] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 698–710. Springer, 2002.

[48] E. M. McCreight. A space-economic suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[49] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[50] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[51] K. S. Neubert. The flashsort1 algorithm. *Dr. Dobb's Journal*, pages 123–125, February 1998.

[52] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th Annual Symposium on Parallel Algorithms and Architectures*, pages 120–129. ACM, 1993.

[53] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *J. ACM*, 42(4):919–933, 1995.

[54] S. Puglisi, W. Smyth, and A. Turpin. The performance of linear time suffix sorting algorithms. In *Proc. Data Compression Conference*, March 2005. to appear.

[55] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[56] B. Smyth. *Computing Patterns in Strings*. Pearson Addison–Wesley, 2003.

[57] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[58] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 22(8):103–111, Aug. 1990.

[59] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[60] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[61] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

A Source code

The following C++ file contains a complete linear time implementation of suffix array construction. The main purpose of this code is to "prove" that the algorithm is indeed simple and that our natural language description is not hiding nonobvious complications. It should be noted that there are now faster (more complicated) implementations of our algorithm [54]. A driver program can be found at `http://www.mpi-sb.mpg.de/~sanders/programs/suffix/`.

```
inline bool leq(int a1, int a2,   int b1, int b2) // lexicographic order
{ return(a1 < b1 || a1 == b1 && a2 <= b2); }                // for pairs
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3)
{ return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3)); }     // and triples

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                           // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;         // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;   // count occurrences
  for (int i = 0, sum = 0;  i <= K;  i++)   // exclusive prefix sums
  {  int t = c[i];  c[i] = sum;  sum += t; }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}

// find the suffix array SA of T[0..n-1] in {1..K}^n
// require T[n]=T[n+1]=T[n+2]=0, n>=2
void suffixArray(int* T, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* R  = new int[n02 + 3];  R[n02]= R[n02+1]= R[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* R0   = new int[n0];
  int* SA0  = new int[n0];

  //******* Step 0: Construct sample ********
  // generate positions of mod 1 and mod 2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) R[j++] = i;

  //******* Step 1: Sort sample suffixes ********
  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(R , SA12, T+2, n02, K);
  radixPass(SA12, R , T+1, n02, K);
  radixPass(R , SA12, T  , n02, K);
```

```
  // find lexicographic names of triples and
  // write them to correct places in R
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (T[SA12[i]] != c0 || T[SA12[i]+1] != c1 || T[SA12[i]+2] != c2)
    {  name++;  c0 = T[SA12[i]];  c1 = T[SA12[i]+1];  c2 = T[SA12[i]+2]; }
    if (SA12[i] % 3 == 1) { R[SA12[i]/3]      = name; } // write to R1
    else                  { R[SA12[i]/3 + n0] = name; } // write to R2
  }

  // recurse if names are not yet unique
  if (name < n02) {
    suffixArray(R, SA12, n02, name);
    // store unique names in R using the suffix array
    for (int i = 0;  i < n02;  i++) R[SA12[i]] = i + 1;
  } else // generate the suffix array of R directly
    for (int i = 0;  i < n02;  i++) SA12[R[i] - 1] = i;

  //******* Step 2: Sort nonsample suffixes ********
  // stably sort the mod 0 suffixes from SA12 by their first character
  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) R0[j++] = 3*SA12[i];
  radixPass(R0, SA0, T, n0, K);

  //******* Step 3: Merge ********
  // merge sorted SA0 suffixes and sorted SA12 suffixes
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ? // different compares for mod 1 and mod 2 suffixes
        leq(T[i],       R[SA12[t] + n0], T[j],       R[j/3]) :
        leq(T[i],T[i+1],R[SA12[t]-n0+1], T[j],T[j+1],R[j/3+n0]))
    {        // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
    } else { // suffix from SA0  is smaller
      SA[k] = j;  p++;
      if (p == n0)  // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
    }
  }
  delete [] R; delete [] SA12; delete [] SA0; delete [] R0;
}
```

B Almost inplace stable distribution sorting and multiway merging

For the lightweight implementation of the DC-algorithm, we need subroutines that are combinations of well known ideas. We outline them here to keep the paper self-contained.

The first idea is used for inplace yet instable distribution sorting (e.g., [49, 51]): The algorithm works similar to the `radixPass` routine in Appendix A yet it reuses the input array to allocate the output buckets. When character $a[i]$ is moved to its destination bucket at array entry j, $a[j]$ is taken as the next element to be distributed. This process is continued until an element is encountered that has already been moved. This order of moving elements decomposes the permutation implied by sorting into its constituent cycles. Therefore, the termination condition is easy to check: A cycle ends when we get back to the element that started the cycle. Unfortunately, this order of moving elements can destroy a preexisting order of keys with identical value and hence is instable.

The second idea avoids instability using a reinterpretation of the input as a sequence of blocks. For example, (almost) inplace multiway merging of files is a standard technique in external memory processing [61, Section 5.3].

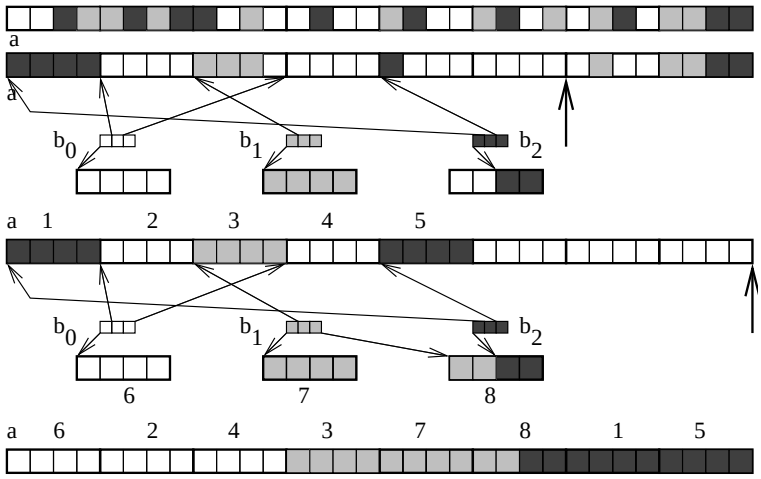

Figure 1: Example for the distribution algorithm for $k = 3$, $n = 32$, and $B = 4$. Four states are shown: Before distribution, when all but two blocks have been distributed, when all blocks are distributed, and the final sorted arrays. The numbers give the order of block moves in the final permutation. In this example, only three additional blocks are needed for temporary storage.

The synthesis of these two ideas leads to a "file-like" stable implementation of distribution sorting and multiway merging followed by an inplace permutation at

the block level that converts the file-representation back to an array representation. We work out the details in the proofs of the following two theorems.

Theorem 5. *An array $a[0,n)$ containing elements with keys in the range $[0,k)$, $k = \mathcal{O}(n)$, can be stably sorted in time $\mathcal{O}(n)$ using $\mathcal{O}(\sqrt{kn})$ additional space.*

Proof. Let $b_j = [a[i] : i \in [0,n), \text{key}(a[i]) = j]$ denote the j=th *bucket*, i.e., the sequence of elements with key j. Sorting a means to permute it in such a way that $b_j = a[\sum_{i\in[0,j)} |b_i|, \sum_{i\in[0,j]} |b_i|)$. We begin with a counting phase that computes the bucket sizes $|b_j|$.

Then we reinterpret a as a sequence of blocks of size $B = \Theta(\sqrt{n/k})$. For the time being, buckets will also be represented as sequences of blocks. For each key, we create an initially empty bucket that is implemented as an array of $\lceil |b_j|/B \rceil$ pointers to blocks. The first $(\sum_{i\in[0,j)} |b_i|) \bmod B$ elements of the first block of b_j are left empty in the distribution process. This way, elements are immediately moved to the correct position $\bmod B$. Buckets aquire additional blocks from a free list as needed. However, for the last block of a bucket b_j, $j < k-1$, the first block of bucket b_{j+1} is used. This way, only one partially filled block remains at the end: The last block of b_{k-1} is stored in another preallocated block outside of a. The free list is initially equipped with $2k+2$ empty blocks. The distribution process scans through the input sequence and appends an element with key j to bucket b_j. When the last element from an input block has been consumed, this block is added to the free list. Since at any point of time there are at most $2k+1$ partially filled blocks (one for the input sequence, one at the start of a bucket, and one at the end of a bucket), the free list never runs out of available blocks.

After the distribution phase, the blocks are permuted in such a way that a becomes a sorted array. The blocks can be viewed as the nodes of a directed graph where each nonempty block has an edge leading to the block in a where it should be stored in the sorted order. The nodes of this graph have maximum in-degree and out-degree one and hence the graph is a collection of paths and cycles. Paths end at empty blocks in a. This structure can be exploited for the permutation algorithm: In the outermost loop, we scan the blocks in a until we encounter a nonempty block $a[i, i+B)$ that has not been moved to its destination position j yet. This block is moved to a temporary space t. We repeatedly swap t with the block of a where its content should be moved until we reach the end of a path or cycle. When all blocks from a are moved to their final destination, the additionally allocated blocks can be moved to their final position directly. This includes the partially filled last block of b_{k-1}.

The total space overhead is $\mathcal{O}(kB) = \mathcal{O}(\sqrt{nk})$ for the additional blocks, $\mathcal{O}(\lceil n/B \rceil + k) = \mathcal{O}(\sqrt{nk})$ for representing buckets, and $\mathcal{O}(\lceil n/B \rceil + k) = \mathcal{O}(\sqrt{nk})$ for pointers that tell every block where it wants to go. □

Theorem 6. *An array $a[0,n)$ consisting of $k \le n$ sorted subarrays can be sorted in time $\mathcal{O}(n \log k)$ using $\mathcal{O}(\sqrt{kn})$ additional space.*

Proof. The algorithm is similar to the distribution algorithm from Theorem 5 so that we only outline the differences. We reinterpret the subarrays as sequences of blocks of a with a partially filled block at their start and end. Only blocks that completely belong to a subarray are handed to the free list. The smallest unmerged elements from each sequence are kept in a priority queue. Merging repeatedly removes the smallest element and appends it to the output sequence that is represented as a sequence of blocks aquired from the free list. After the merging process, it remains to permute blocks to obtain a sorted array. Except for space $\mathcal{O}(k)$ for the priority queue, the space overhead is the same as for distribution sorting. □