Algorithms **2010**, *3*, 145 - 167; doi:10.3390/a3020145

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Suffix-Sorting *via* Shannon-Fano-Elias Codes

Donald Adjeroh ⋆ **and Fei Nan**

Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-6109, USA; E-Mail: fnan@sta.samsung.com

⋆ Author to whom correspondence should be addressed; e-mail: don@csee.wvu.edu; Tel.: +1-304-293-9681.

Received: 23 December 2009; in revised form: 18 March 2010 / Accepted: 18 March 2010 / Published: 1 April 2010

Abstract: Given a sequence $T = t_0 t_1 \dots t_{n-1}$ of size $n = |T|$, with symbols from a fixed alphabet Σ, $(|\Sigma| \leq n)$, the suffix array provides a listing of all the suffixes of T in a lexicographic order. Given T, the suffix sorting problem is to construct its suffix array. The direct suffix sorting problem is to construct the suffix array of T directly *without* using the suffix tree data structure. While algorithims for linear time, linear space direct suffix sorting have been proposed, the actual constant in the linear space is still a major concern, given that the applications of suffix trees and suffix arrays (such as in whole-genome analysis) often involve huge data sets. In this work, we reduce the gap between current results and the minimal space requirement. We introduce an algorithm for the direct suffix sorting problem with worst case time complexity in $O(n)$, requiring only $(1\frac{2}{3} n \log n - n \log |\Sigma| + O(1))$ bits in memory space. This implies $5\frac{2}{3}n + O(1)$ bytes for total space requirment, (including space for both the output suffix array and the input sequence T) assuming $n \leq 2^{32}, |\Sigma| \leq 256$, and 4 bytes per integer. The basis of our algorithm is an extension of Shannon-Fano-Elias codes used in source coding and information theory. This is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem.

Keywords: suffix sorting; suffix arrays; suffix tree; Shannon-Fano-Elias codes; source coding

---

1. Introduction

The suffix array provides a compact representation of all the n suffixes of T in a lexicographic order. The sequence of positions in T of the first symbols from the sorted suffixes in this lexicographic order is the suffix array for the sequence. The suffix sorting problem is to construct the suffix array of T. Mamber and Myers [1] were the first to propose an $O(n \log n)$ algorithm to construct the suffix array with three to five times less space than the traditional suffix tree. Other methods for fast suffix sorting in $O(n \log n)$ time have been reported in [2], while memory efficient constructions were considered in [3]. Puglisi *et al.* [4] provide a detailed comparison of different recently proposed linear time algorithms for suffix sorting.

Suffix trees and suffix arrays have drawn a significant attention in recent years due to their theoretical linear time and linear space construction, and their logarithmic search performance. Gusfield [5] provides a detailed study on the use of suffix trees in the analysis of biological sequences. Suffix sorting is also an important problem in data compression, especially for compression schemes that are based on the Burrows-Wheeler Transform [6, 7]. In fact, it is known that the suffix sorting stage is a major bottleneck in BWT-based compression schemes[6–8]. The suffix array is usually favored over suffix trees due to its smaller memory footprint. This is important given that the applications of suffix trees and suffix arrays (such as in whole-genome sequence analysis) often involve huge data sets. An important problem therefore is to construct suffix arrays in linear time, while requiring space for the input sequence, in addition to $n \log n$ bits, the minimal space required to hold the suffix array. This translates to $5n$ bytes of total space requirment, assuming $n \leq 2^{32}, |\Sigma| = 256$, and 4 bytes per integer. Existing algorithms, such as the KS Algorithm [9] require $13n$ bytes, while the KA Algorithm [10] requires $10n$ bytes.

This paper presents a space-efficient linear time algorithm for solving the direct suffix sorting problem, using only $5\frac{2}{3}n$ bytes. The basis of the algorithm is an extension of the Shannon-Fano-Elias codes pupularly used in arithmetic coding. In the next section, we provide a background to the problem, and briefly describe related work. Section 3 presents our basic algorithm for suffix sorting. Section 4 improves the complexity of the basic algorithm using methods from source coding and information theory.

2. Background

The BWT [6, 7] performs a permutation of the characters in the input sequence, such that characters in lexically similar contexts will be near to each other. Thus, the BWT is very closely related to the suffix tree and suffix array - two important data structures used in pattern matching, analysis of sequence redundancy, and in data compression. The major link is the fact the BWT provides a lexicographic sorting of the contexts as part of the permutation of the input sequence. The recent book [7] provides a detailed treatment of the link between the BWT and suffix arrays and suffix trees. Apart from the BWT, other popular compression schemes have also been linked with the suffix tree data structure. For instance, the PPM* compression was implemented using suffix trees and suffix tries in [11]. The sequence decomposition step required for constructing the dictionary in LZ-based schemes can be performed by the use of tree structures, such as binary search trees, or suffix tries [12]. A detailed analysis on the use of suffix trees in data compression is provided in Szpankowski [13]. When decorated with the array of

longest common prefixes, the suffix array can be used in place of the suffix tree in almost any situation where a suffix tree is used [14]. This capability is very important for applications that require huge storage, given the smaller memory requirement of the suffix array. An important issue is how to devise efficient methods for constructing the suffix array, without the suffix tree.

Some methods for constructing suffix arrays first build the suffix tree, and then construct the suffix array by performing an inorder traversal of the suffix tree. Farach al.[15] proposed a divide and conquer method to construct the suffix tree for a given sequence in linear time. The basic idea is to divide the sequence into odd and even sequences, based on the position of the symbols. Then, the suffix tree is constructed recursively for the odd sequence. Using the suffix tree for the odd sequence, they construct the suffix tree for the even sequence. The final step merges the suffix tree from the odd and even sequences into one suffix tree using a coupled depth-first search. The result is a linear time algorithm for sufix tree construction. This divide and conquer approach is the precursor to many recent algorithms for direct suffix sorting.

Given the memory requirement and implementation difficulty of the suffix tree, it is desirable to construct the suffix array directly, without using the suffix tree. Also, for certain applications such as in data compression where only the suffix array is needed, avoiding the construction of the suffix tree will have some advantages, especially with respect to memory requirements. More importantly, direct suffix sorting without the suffix tree raises some interesting algorithmic challenges. Thus, more recently, various methods have been proposed to construct the suffix array directly from the sequence [9, 10, 16–18], without the need for a suffix tree.

Kim *et al.*[16] followed an approach similar to Farach's above, but for the purpose of constructing the suffix array directly. They introduce the notion of equivalent classes between strings, which they use to perform coupled depth-first searches at the merging stage. In [9] a divide and conquer approach similar to Farach's method was used, but for direct construction of the suffix array. Here, rather than dividing the sequence into two symmetric parts, the sequence was divided into two unequal parts, by considering suffixes that begin at positions $(i \mod 3 \neq 0)$ in the sequence. These suffixes are recursively sorted, and then the remaining suffixes are sorted based on information in the first part. The two sorted suffixes are then combined using a merging step to produce the final suffix array. Thus, a major difference is in the way they divided the sequences into two parts, and in the merging step.

Ko and Aluru [10] also used recursive partitioning, but following a fundamentally different approach to construct the suffix array in linear space and linear time. They use a binary marking strategy whereby each suffix in T is classified as either an S-suffix or an L-suffix, depending on its relative order with its next neighbor. Let $T'_i = t_i t_{i+1} t_{i+2} \ldots t_{n-1}$ denote the suffix T'_i of sequence T starting at position i. An S-suffix is a suffix that is lexicographically smaller than its right neighbor in T, while an L-suffix is one that is lexicographically larger than its right neighbor. That is, T'_i is an S-suffix if $T'_i \prec T'_{i+1}$, otherwise T'_i is an L-suffix. This classification is motivated by the observation that an S-suffix is always lexicographically greater than any L-suffix that starts with the same first character. The two types of suffixes are then treated differently, whereby the S-suffixes are sorted recursively by performing some special distance computations. The L-suffixes are then sorted using the sorted order of the S-suffixes. The classification scheme is very similar to the approach earlier used by Itoh and Tanaka [19]. But the algorithm in [19] runs in $O(n \log n)$ time on average.

Lightweight suffix sorting algorithms [3, 17, 20, 21] have also been proposed which pay more attention on the extra working space required. Working space excludes the space for the output suffix array. In general, the suffix array requires $n\lceil \log n \rceil$ bit space, while the input text requires another $n\lceil \log |\Sigma| \rceil$ bits, or in the worst case $n\lceil \log n \rceil$ bits. In [22], an algorithm that runs in $O(n \log n)$ worst case time, requiring $O(n \log n + (n/\sqrt{\log n}))$ working space was proposed, while Hon *et al.* [20] constructed suffix arrays for integer alphabets in $O(n \log n)$ time, using $O(n \log n)$-bit space. Nong and Zhang [17] combined ideas from the KS algorithm [9] and the KA algorithm [10] to develop a method that works in $O(n \log |\Sigma|)$ time, using $(n \log |\Sigma| + |\Sigma| \log n)$-bit working space (without the output suffix array). In [23, 24] they extended the method to use $2n + O(1)$ bytes of working space (or a total space of $7n + O(1)$ bytes, including space for the suffix array, and the original sequence), by exploiting special properties of the L and S suffixes used in the KA algorithm, and of the leftmost S-type substrings. In-place suffix sorting was considered in [25], where suffix sorting was performed for strings with symbols from a general alphabet using $O(1)$ working space in $O(n \log n)$ time. In some other related work [2, 26], computing the BWT was viewed as a suffix sorting problem. Okanohara and Sadakane [26] modified a suffix sorting algorithm to compute the BWT using a working space of $O(n \log |\Sigma| \log \log_{|\Sigma|} n)$ bits. There has also been various efforts on compressed suffix arrays and compressed suffix trees as a means to tackle the space problem with suffix trees and suffix arrays. (See [27–29] for examples). We do not consider compressed data structures in this work. A detailed survey on suffix array construction algorithms is provided in [4].

2.1. Main Contribution

We propose a divide-and-conquer sort-and-merge algorithm for direct suffix sorting on a given input string. Given a string of length n, our algorithm runs in $O(n)$ worst case time and space. The algorithm recursively divides an input sequence into two parts, performs suffix sorting on the first part, then sorts the second part based on the sorted suffix from the first. It then merges the two smaller sorted suffixes to provide the final sorted array. The method is unique in its use of Shannon-Fano-Elias codes in efficient construction of a global partial order for the suffixes. To our knowledge, this is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem.

Our algorithm also differs from previous approaches in the use of a simple partitioning step, and how it exploits this simple partitioning scheme for conflict resolution. The total space requirement for the proposed algorithm is $(\frac{5}{3}n\lceil \log n \rceil - n\lceil \log |\Sigma| \rceil) + O(1)$ bits, including the space for the output suffix array, and for the original sequence. Using the standard assumption of 4 bytes per integer, with $|\Sigma| \leq 256, n \leq 2^{32}$, we get a total space requirement of $5\frac{2}{3}n$ bytes, including the n bytes for the original string and the $4n$ bytes for the output suffix array. This is a significant improvement when compared with other algorithms, such as the $10n$ bytes required by the KA algorithm [10], or the $13n$ bytes required by the KS algorithm [9].

3. Algorithm I: Basic Algorithm

3.1. Notation

Let $T = t_0t_1t_2 \dots t_{n-1}$ be the input sequence of length n, with symbol alphabet $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_{|\Sigma|-1}\}$. Let $T'_i = t_it_{i+1}t_{i+2} \dots t_{n-1}$ denote the suffix of T starting at position $i (i = 0, 1, 2, \dots n-1)$. Let $T[i] = t_i$ denote the i-th symbol in T. For any two strings, say α and β, we use $\alpha \prec \beta$ to denote that the α preceeds β in lexicographic order. (Clearly, α and β could be individual symbols, from the same alphabet, i.e. $|\alpha| = |\beta| = 1$.) We use \$ as the end of sequence symbol, where $\$ \notin \Sigma$ and $\$ < \sigma, \forall \sigma \in \Sigma$. Further, we use SA to denote the suffix array of T, and S to denote the sorted list of first characters in the suffixes. Essentially, $S[i] = T[SA[i]]$. Given SA, we use SA' to denote its inverse. We define SA' as follows: $SA'[i] = k$ if $SA[k] = i; i, k = 0, 1, \dots, n-1$. That is, $S[k] = S[SA'[i]] = T[i]$. We use p_i to denote the probability of symbol $T[i]$, and P_i the probability of the substring $T[i \dots i+m-1]$, the m-length substring starting at position i.

3.2. Overview

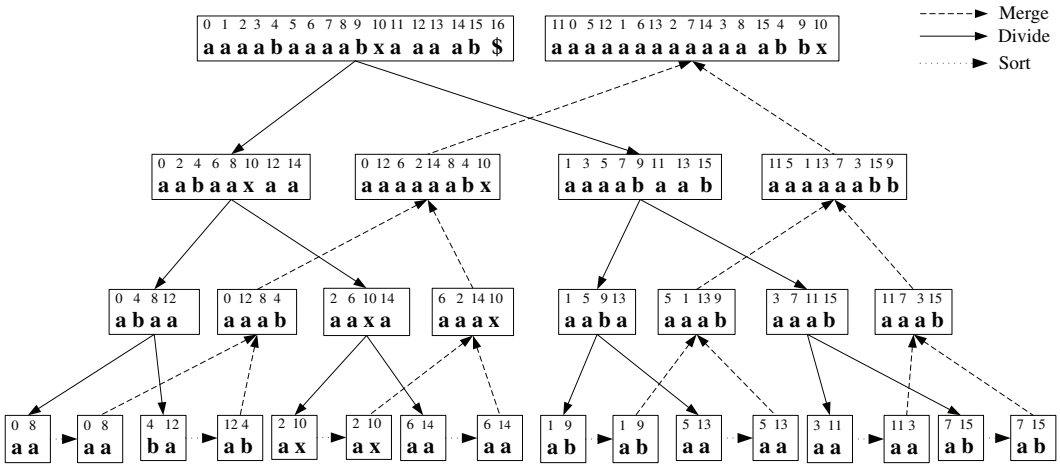
We take the general divide and conquer approach:

1. Divide the sequence into two groups;
2. Construct the suffix array for the first group;
3. Construct the suffix array for the second group;
4. Merge the suffix arrays from the two groups to form the suffix array for the parent sequence;
5. Perform the above steps recursively to construct the complete suffix array for the entire sequence.

Figure 1 shows a schematic diagram for the working of the basic algorithm using an example sequence, `T=aaaabaaaabxaaaab`. The basic idea is to recursively partition the input sequence in a top-down manner into two equal-length subsequences according the odd and even positions in the sequence. After reaching the subsequences with length ≤ 2, the algorithm then recursively merges and sorts the subsequences using a bottom-up approach, based on the partial suffix arrays from the lower levels of the recursion. Thus, the algorithm does not start the merging procedure until it reaches the last partition on a given branch.

Each block in the figure contains two rows. The first row indicates the position of the current block of symbols in the original sequence T. The second row indicates the current symbols. The current symbols are unsorted in the downstream dividing block and sorted in the upstream merging block. To see how the algorithm works, starting from the top left block follow the solid division arrow, the horizontal trivial sort arrow (dotted arrow), and then the dashed merge arrows. The procedure ends at the top right block. We briefly describe each algorithmic step in the following. Later, we modify this basic algorithm for improved complexity results.

Figure 1. Basic working of the proposed algorithm using an example. The original sequence is indicated at the top left. The sorted sequence and the suffix array are indicated in the top right box.



3.3. Algorithm I

3.3.1. Divide T

If $|T| \geq 2$, divide T into 2 subsequences, T_1 and T_2. T_1 contains all the symbols at the even positions of T. T_2 contains all the symbols at the odd positions of T. That is, $T_1 = \{T[j], j \in [0, |T|)|\ j \mod 2 = 0\}$, $T_2 = \{T[j], j \in [0, |T|)|\ j \mod 2 = 1\}$

3.3.2. Merge SA of T_1 and SA of T_2

Let SA_1 and SA_2 be the suffix array of T_1 and T_2 respectively. Let SA be the suffix array of T. If T_1 and T_2 have been sorted to obtain their respective suffix arrays SA_1 and SA_2, then we can merge SA_1 and SA_2 in linear time on average, to form the suffix array SA. Without loss in generality, we assume $SA_1 = a_0 a_1 a_2 \ldots a_u$, $SA_2 = b_0 b_1 b_2 \ldots b_v$ and $SA = c_0 c_1 c_2 \ldots c_{u+v}$ are the sorted indices of T_1, T_2 and T respectively. Given a_k, we use $\hat{a}_k$ to denote its corresponding position in T. That is, $T_1[SA_1[k]] = T_1[a_k] = T[\hat{a}_k]$. Similarly, for $\hat{b}_k$. $\hat{a}_k$ and $\hat{b}_k$ are easily obtained from a_k, based on the level of recursion, and whether we are on a left branch or a right branch. For $k = 0$ $(0 \leq k \leq u), l = 0$ $(0 \leq l \leq v)$ and $g = 0$ $(0 \leq g \leq u + v)$, we compare the partially ordered subsequences using a_k and b_l, viz.

$$\begin{aligned}
&\text{If } T[\hat{a}_k] \prec T[\hat{b}_l], && \begin{cases} SA[c_g] \leftarrow \hat{a}_k, S[c_g] \leftarrow T[\hat{a}_k]; k{+}{+}, g{+}{+} \end{cases} \\
&\text{If } T[\hat{b}_l] \prec T[\hat{a}_k], && \begin{cases} SA[c_g] \leftarrow \hat{b}_l, S[c_g] \leftarrow T[\hat{b}_l]; l{+}{+}, g{+}{+} \end{cases} \\
&\text{If } T[\hat{a}_k] = T[\hat{b}_l], && \begin{cases} c_x \leftarrow \text{ResolveConflict}(\hat{a}_k, \hat{b}_l) \\ SA[c_g] \leftarrow c_x, S[c_g] \leftarrow T[c_x]\ ; g{+}{+} \end{cases}
\end{aligned}$$

Whenever we compare two symbols, $T[\hat{a}_k]$ from T_1 and $T[\hat{b}_l]$ from T_2, we might get into the ambiguous situation whereby the two symbols are the same (*i.e.*, $T[\hat{a}_k] = T[\hat{b}_l]$). Thus, we cannot

easily decide which suffix precedes the other, (*i.e.*, whether $T'_{\hat{a}_k} \prec T'_{\hat{b}_l}$, or $T'_{\hat{b}_l} \prec T'_{\hat{a}_k}$), based on the individual symbols. We call this situation a **conflict**. The key to the approach is how efficiently we can resolve potential conflicts as the algorithm progresses. We exploit the nature of the division procedure, and the fact that we are dealing with substrings (suffixes) of the same string, for efficient conflict resolution. Thus, the result of the merging step will be a partially sorted subsequence, based on the sorted order of the smaller child subsequences. An important difference here is that unlike in other related approaches [9, 10, 16], our sort-order at each recursion level is global with respect to T, rather than being local to the subsequence at the current recursion step. This is important, as it significantly simplifies the subsequent merging step.

3.3.3. Recursive Call

Using the above procedure, we recursively construct the suffix array of T_1 from its two children T_{11} and T_{12}. Similarly, we obtain the suffix array for T_2 from its children T_{21} and T_{22}. We follow this recursive procedure until the base case is reached (when the length of the subsequence is ≤ 2).

3.4. Conflict Resolution

We use the notions of **conflict sequence** or **conflict pairs**. Two suffixes T'_i and T'_j form a conflict sequence in T if $T'_i[0 \ldots k] = T'_j[0 \ldots k]$, (that is, $T[i \ldots i+k] = T[j \ldots j+k]$) for some $k \geq 1$. Thus, the two suffixes can not be assigned a total order after considering their first k symbols. We say that the conflict between T'_i and T'_j is resolved whenever $T[i \ldots i+l_k-1] = T[j \ldots j+l_k-1]$, and $T[i+l_k] \neq T[j+l_k]$. Here, l_k is called the conflict length. We call the triple (T'_i, T'_j, l_k) a conflict pair, or conflict sequence. We use the notation $CP(i, j, l_k)$ to denote a conflict pair T'_i and T'_j with a conflict length of l_k. We also use $CP(i, j)$ to denote a conflict pair where the conflict length is yet to be determined, or is not important given the context of the discussion.

The challenge, therefore, is how to resolve conflicts efficiently given the recursive partitioning framework proposed. Obviously, $l_k \leq n$, the sequence length. Thus, the minimum distance from the start of each conflicting suffix to the end of the sequence ($\min\{n-i, n-j\}$), or the distance from an already sorted symbol can determine how long it will take to resolve a conflict. Here, we consider how the previously resolved conflicts can be exploited for a quick resolution of other conflicts. We maintain a traceback record on such previously resolved conflicts in a **conflict tree**, or a **conflict table**.

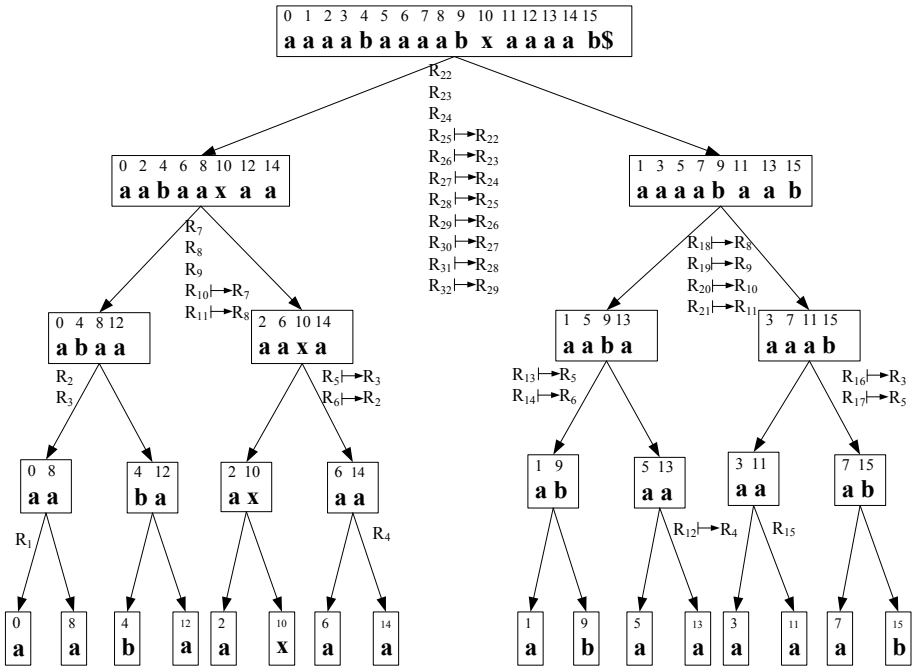
Figure 2 shows the conflict tree for the example sequence, `T=aaaabaaabxaaaab`. Without loss in generality and for easier exposition of the basic methodology, we assume that $n = 2^x$, for some positive integer x. Our conflict resolution strategy is based on the following properties of conflict sequences and conflict trees.

1. Given the even-odd recursive partitioning scheme, at any given recursion level, conflicts can only occur between specific sets of positions in the original sequence T. For instance, at the lowest level, conflicts can only occur between T_0, and $T_{n/2}$, or more generally, between T_i, and $T_{i+n/2}$. See Figure 2.

2. Given the conflict sequence, T'_i and T'_j the corresponding conflict pair $CP(i, j, l_k)$ is unique in T. That is, only one conflict pair can have the pair of start positions (i, j). Thus the conflict pairs $CP(i, j, l_k)$ and $CP(j, i, l_k)$ are equivalent. Hence, we represent both $CP(i, j, l_k)$ and $CP(j, i, l_k)$ as $CP(i, j, l_k)$, where, $i < j$.

3. Consider $CP(i, j)$ at level $h, 0 \leq h \leq \lceil \log n \rceil$ in the tree. We define its **neighbors** as the conflict pairs: $CP(i', j')$, with $(i' = i + 2^q, j' = j + 2^q)$, or $(i' = i - 2^q, j' = j - 2^q)$, where $q > 0$, with $q = h, h - 1, h - 2, \ldots, 1$, and $i', j' \leq n$. Essentially, neighboring conflicts are found on the same level in the conflict tree, from the leftmost node to the current node. For example, for $R_6 = CP(2, 14)$ at $h = 2$, the neighbor will be: $\{R_2 = CP(0, 12)\}$. We notice that, by our definition, not all conflicts in the same node are neighbors.

Figure 2. Conflict tree for the example sequence $T = aaaabaaaabxaaaab$. The original sequence is indicated at the root node. The notation $R_i \mapsto R_j$ indicates that conflict pair R_i is resolved by R_j after a certain number of steps, given that R_j has been resolved. Conflict pairs are labeled following a depth-first traversal on the tree.



Given the conflict tree, we can determine whether a given conflict pair $CP(i, j)$ can be resolved based on previous trace back information (*i.e.*, previously resolved conflicts). From the conflict tree, we see that this determination can be made by checking only a **fixed** number of neighboring conflict pairs. In fact, from the definition above, we see that the size of the neighborhood is at most 6. For $CP(i, j)$, the neighbors will be the conflict pairs in the set $CP(i', j')$, with $(i' = i - 2^h, j' = j - 2^h)$; $(i' = i + 2^h, j' = j + 2^h)$; $(i' = i - 2^{h-1}, j' = j - 2^{h-1})$; $(i' = i + 2^{h-1}, j' = j + 2^{h-1})$; $(i' = i - 2^{h-2}, j' = j - 2^{h-2})$; and $(i' = i + 2^{h-2}, j' = j + 2^{h-2})$. Some of these pair of indices may not be valid indices for a conflict pair. For example, using the previous example, $R_6 = CP(2, 14)$ has only one valid neighbor, $\{R_2 = CP(0, 12)\}$. Also, given that we resolve the conflicts based on the depth first traversal order, neighbors of $CP(i, j)$ that are located on a node to the right of $CP(i, j)$ in the conflict tree cannot be used to resolve conflict $CP(i, j)$. Therefore, in some cases, less than six probes may be needed.

Let $CP(i,j)$ be the current conflict. Let $CP(i',j',l'_k)$ be a previously resolved neighboring conflict. We determine whether $CP(i,j)$ can be resolved based on $CP(i',j',l'_k)$ using a simple check: Let $\Delta = (i'-i) = (j'-j)$. If Δ is negative, then $CP(i,j)$ can be resolved with $CP(i',j',l'_k)$ iff: $|\Delta| \leq l'_k - 1$ Conversely, if Δ is positive, then $CP(i,j)$ can be resolved with $CP(i',j',l'_k)$ iff: $l'_k \geq 0$ If any of the two conditions holds, we say that $CP(i,j)$ is resolved by $CP(i',j',l'_k)$, after Δ steps. Essentially, this means that after Δ comparison steps, $CP(i,j)$ becomes equivalent to $CP(i',j',l'_k)$. We denote this equivalence using the notation: $CP(i,j) \mapsto CP(i',j',l'_k)$. The following algorithm shows how we can resolve a given conflict pair, $CP(i,j)$.

RESOLVECONFLICT(i,j)
Generate neighbors for $CP(i,j)$
Remove invalid neighbors
Probe the valid neighbors for previously resolved conflict pairs
for each valid neighbor, compute the quantity $d = (l'_k - |\Delta|)$; **end for**
Select the neighbor $CP(i',j',l'_k)$ with maximum value for d
if ($\Delta < 0$ and $|\Delta| \leq l'_k$), then /* no extra work is needed */
    Compute $l_k = l'_k + \Delta - 1$.
else if ($\Delta > 0$ and $l'_k \geq 0$), then
    Perform at most $(\Delta - 1)$ extra comparison steps
    if (conflict is not resolved after the $\Delta - 1$ comparisons), then
        /*Resolve conflict using $CP(i',j',l'_k)$ */
        Compute $l_k = l'_k + \Delta$
    end if
  end if
else /* conflict cannot be resolved using earlier conflicts */
    Resolve conflict using direct symbol-wise comparisons
end if

The complexity of conflict resolution (when it can be resolved using previous conflicts) thus depends on the parameter Δ. Table 1 shows the conflict pairs for the example sequence used to generate the conflict tree of Figure 2. That table includes the corresponding Δ value where a conflict can be resolved based on a previous conflict.

The final consideration is how to store the conflict tree to ensure constant-time access to the resolved conflict pairs. Since for a given $CP(i,j)$, the (i,j) pair is unique, we can use these for direct access to the conflict tree. We can store the conflict tree as a simple linear array, where the positions in the array are determined based on the (i,j) values, and hence the height in the tree. To avoid the sorting that may be needed for fast access using the (i,j) indices, we use a simple hash function, where each hash value can be computed in constant time. The size of this array will be $O(n \log n)$ in the worst case, since there are at most $n \log n$ conflicts (see analysis below). The result will be an $O(1)$ time access to any given conflict pair, given its (i,j) index in T.

Table 1. Conflict pairs for T=aaaabaaaabxaaaab. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p.

R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ
R_1	$(0,8,0)$	–	–	R_9	$(6,8,1)$	–	–	R_{17}	$(3,7,1)$	R_5	-1	R_{25}	$(1,12,4)$	R_{22}	-1
R_2	$(0,12,3)$	–	–	R_{10}	$(2,8,1)$	R_7	-2	R_{18}	$(5,11,5)$	R_8	$+1$	R_{26}	$(1,6,4)$	R_{23}	-1
R_3	$(8,12,1)$	–	–	R_{11}	$(8,14,2)$	R_8	-2	R_{19}	$(5,7,2)$	R_9	$+1$	R_{27}	$(6,13,2)$	R_{24}	-1
R_4	$(6,14,1)$	–	–	R_{12}	$(5,13,2)$	R_4	$+1$	R_{20}	$(1,7,2)$	R_{10}	$+1$	R_{28}	$(2,13,3)$	R_{25}	-1
R_5	$(2,6,2)$	R_3	$+6$	R_{13}	$(1,5,3)$	–	–	R_{21}	$(7,13,3)$	R_{11}	$+1$	R_{29}	$(2,7,3)$	R_{26}	-1
R_6	$(2,14,1)$	R_2	-2	R_{14}	$(1,13,2)$	R_5	$+1$	R_{22}	$(0,11,5)$	–	–	R_{30}	$(7,14,1)$	R_{27}	-1
R_7	$(0,6,3)$	–	–	R_{15}	$(3,11,1)$	R_6	$+1$	R_{23}	$(0,5,5)$	–	–	R_{31}	$(3,14,2)$	R_{28}	-1
R_8	$(6,12,4)$	–	–	R_{16}	$(7,11,2)$	R_3	$+1$	R_{24}	$(5,12,3)$	–	–	R_{32}	$(3,8,2)$	R_{29}	-1

3.5. Complexity Analysis

Clearly, the complexity of the algorithm depends on the overall number of conflicts, the number of conflicts that require explicit symbol-wise comparisons, the number that can be resolved using earlier conflicts, and how much extra comparsions are required for those. We can notice that we require symbol-wise comparison mainly for the conflicts at the leftmost nodes on the conflict tree, and the first few conflicts in each node. For the sequence used in Figure 2 and Table 1, with $n = 16$, we have the following: 32 total number of conflict pairs; 20 can be resolved with only look-ups based on the neighbors (with no extra comparisons); 1 requires extra comparisons after lookup (a total of 6 extra comparisons); 11 cannot be resolved using lookups, and thus requires direct symbol-wise; comparisons (a total of 20 such comparisons in all). This gives 44 overall total comparisons, and 21 total lookups.

This can be compared with the results for the worst case sequence with the same length, $T = \mathtt{a}^{16} = \mathtt{aaaaaaaaaaaaaaaa}$. Here we obtain: 49 total number of conflict pairs; 37 can be resolved with only look-ups based on the neighbors (with no extra comparisons); 8 requires extra comparisons after lookup (a total of 12 extra comparisons); 4 cannot be resolved using lookups, and thus require direct symbol-wise; comparisons (a total of 19 such comparisons in all). This results in 41 overall total comparisons and 45 total lookups. While the total number of comparisons is similar for both cases, the number of lookups is significantly higher for $T = a^n$, the worst case sequence.

The following lemma establishes the total number of symbol comparisons and the number of conflicts that can be encountered using the algorithm.

Lemma 1: Given an input sequence $T = t_0, t_2 \ldots, t_{n-1}$, with length n, the maximum number of symbol comparisons and the maximum number of conflict pairs are each in $O(n \log n)$.

Proof. Deciding on whether we can resolve a given conflict based on its neigbors requires only constant time, using Algorithm RESOLVECONFLICT(). Assume we are sorting the worst case sequence, $T = a^n$. Consider level h in the conflict tree. First, we resolve the left-most conflict at this level, say conflict R_L. This will require at most $\frac{n}{2^{h_{max}-h}} = \frac{n}{2^h}$ symbol comparisons. For example, at $h = h_{max} - 1$, the level just before the lowest level, the leftmost conflict will be $R_1 = CP(0, \frac{n}{2})$, requiring $\frac{n}{2}$ comparsions. However, each conflict on the same level with R_L can be resolved with at

most $\frac{n}{2^{h_{max}-h+1}} = 2^{h-1}$ comparisons, plus constant time lookup using the earlier resolved neighbors. A similar analysis can be made for conflict pairs $CP(i,j)$, where both i and j are odd. There are are at most $(\frac{n}{2^h} - 1)$ conflicts for each node at level h of the tree. Thus, the total number of comparisons required to resolve all conflicts will be $\sum_{i=0}^{\lceil \log n \rceil} 2^i(\frac{n}{2^i} - 1) \leq n \log n$. In the worst case, each node in the tree will have the maximum $(\frac{n}{2^h} - 1)$ number of conflicts. Thus, similar to the overall number of comparisons, the worst case total number of conflicts will be in $O(n \log n)$. □

We state our complexity results in the following theorem:

Theorem 1: Given an input sequence $T = t_0, t_2 \ldots, t_{n-1}$, with symbols from an alphabet Σ, Algorithm I solves the suffix sorting problem in $O(n \log n)$ time and $O(n)$ space, for both the average case and the worst case.

Proof. The worst case result on required time follows from Lemma 1 above. Now consider the random string with symbols in Σ unifomly distributed. Here, the probablity of encountering one symbol is $\frac{1}{|\Sigma|}$. Thus, the probability of matching two substrings from T will decrease rapidly as the length of the substrings increase. In fact, the probability of a conflict of length l_k will be $Pr\{CP(i,j,l_k)\} = \frac{1}{|\Sigma|^{2l_k}}$. Following [30], the maximum value of l_k, the length of the longest common prefix for a random string is given by $l_{max} = O(\log_{|\Sigma|} n)$. Thus, the average number of matching symbol comparisons for a given conflict will be:

$$\eta_{compare} = \sum_{l_k=1}^{l_{max}} (l_k) Pr\{CP(i,j,l_k)\} = \frac{1}{|\Sigma|^{2.1}} + \frac{2}{|\Sigma|^{2.2}} + \frac{3}{|\Sigma|^{2.3}} + \ldots + \frac{l_{max}}{|\Sigma|^{2.l_{max}}}$$

This gives:

$$\eta_{compare} = \frac{1-|\Sigma|^{2.l_{max}}}{1-|\Sigma|} = \frac{1-\frac{1}{n^2}}{1-\frac{1}{|\Sigma|^2}} = \frac{n^2-1}{n^2} . \frac{|\Sigma|^2}{|\Sigma|^2-1} \leq 2$$

For each conflict pair, there will be exactly one mismatch, requiring one comparison. Thus, on average, we require at most 3 comparisons to resolve each conflict pair.

The probability of a conflict is just the probablity that any two randomly chosen symbols will match. This is simply $\frac{1}{|\Sigma|^2}$. However, we still need to make at least one comparison for each potential conflict pair. The number of this potential conflict pairs is exacty the same as the worst case number of conflicts. Thus, although the average number of comparisons per conflict is constant, the total time required to resolve all conflicts is still in $O(n \log n)$.

We can reduce the space required for conflict resolution as follows. A level-h conflict pair can be resolved based on only previous conflicts (its neighbors), all at the same level, h. Thus, rather than the current depth-first traversal, we change the traversal order on the tree, and use a bottom-up breadth-first traversal. Then, starting with the lowest level, $h = \lceil \log n \rceil$, we resolve all conflicts at a given level (starting from the leftmost), before moving up to the next level. Then, we re-use the space used for the lower levels in the tree. This implies a maximum of $n - 1$ entries in the hash table at any given time, or $O(n)$ space.

We make a final observation about the nature of the algorithm above. It may be seen that, in deed, we do not need to touch T_2, the odd tree at all, until the very last stage of final merging. Since we can sort T_1 to obtain SA_1 without reference to T_2, we can therefore use SA_1 to sort T_2 using radix sort, since positions in T_1 and T_2 are adjacent in T. This will eliminate consideration of the $(n - 1)$ worst case

conflicts at level $h = 0$, and all the conflicts in T_2. This will however not change the complexity results, since the main culprit is the $O(n \log n)$ total number of conflicts in the worst case. We make use of this observation in the next section, to develop an $O(n)$ time and space algorithm for suffix sorting.

4. Algorithm II: Improved Algorithm

The major problem with Algorithm I is the time taken for conflict resolution. Since the worst case number of conflicts is in $O(n \log n)$, an algorithm that performs a sequential resolution of each conflict can do no better than $O(n \log n)$ time in the worst case. We improve the algorithm by modifying the recursion step, and the conflict resolution strategy. Specifically, we still use binary partitioning, but we use a non-symmetric treatment of the two branches at each recursion step. That is, only one branch will be sorted, and the second branch will be sorted based on the sorted results from the first branch. This is motivated by the observation at the end of the last section. We also use a preprocessing stage inspired by methods in information theory to facilitate fast conflict resolution.

4.1. Overview of Algorithm II

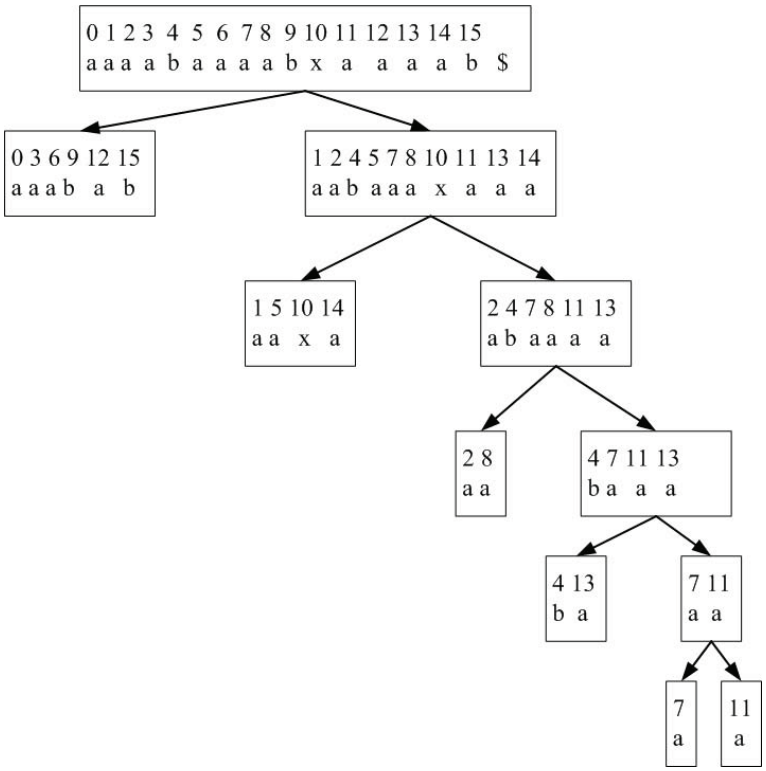
In Algorithm I, we perform symmetric division of T into two subsequences T_1 and T_2, and then merge their respective suffix arrays SA_1 and SA_2 to form SA, the suffix array of T. SA_1 and SA_2 in turn are obtained by recursive division of T_1 and T_2 and subsequent merging of the suffix arrays of their respective children. The improvement in Algorithm II is that when we divide T into T_1 and T_2, the division is no longer symetric. Similar to the KS Algorithm [9], here we form T_1 and T_2 as follows: $T_1 = \{T[j], j \in [0, |T|) |\ j \mod 3 = 0\}$, $T_2 = \{T[j], j \in [0, |T|) |\ j \mod 3 \neq 0\}$. This is a 1:2 asymetric partitioning. The division schemes are special cases of the general $1 : \eta$ partitioning, where $\eta = 2$ in the above, while $\eta = 1$ in the symetric partitioning scheme of Algorithm I. An important parameter is α, defined as: $\alpha = \frac{1+\eta}{\eta}$. In the current case, $\alpha = 1.5$.

Further, the recursive call is now made on only one branch of T, namely T_2. After we obtain the suffix array SA_2 for T_2, we radix sort T_1 based on the values in SA_2 to construct SA_1. This radix sorting step only takes linear time. The merging step remains similar to Algorithm I, though with some important changes to accomodate the above non-symetric partitioning of T. We also now use an initial ordering stage for faster conflict resolution.

4.2. Sort T_2 to Form SA_2

After dividing T into T_1 and T_2, we need to sort each child sequence to form its own smaller suffix array. Consider T_2. We form SA_2 by performing the required sorting recursively, using a non-symmetric treatment of each branch. T_2 is thus sorted by a recursive subdivision, and local sorting at each step. The suffix arrays of the two children are then merged to form the suffix array of their parent. Figure 3 shows this procedure for the example sequence $T =$ `aaaabaaaabxaaaab`. Merging can be performed as before (with changes given the non-symetric nature of T_1 and T_2). The key to the algorithm is how to obtain the sorted array of the left child from that of the right child at each recursion step.

Figure 3. Asymmetric recursive partitioning for improved algorithm, using $T = \texttt{aaaabaaaabxaaaaab}$. Recursive partitioning is performed on the right branch.



4.3. Sort T_1 by Inducing SA_1 from SA_2

Without loss in generality, we assume T_1 is unsorted and T_2 has been sorted to form SA_2. Given the partitioning scheme, we have that for any $t_k \in T_1$, $t_{k+1} \in T_2$. There must exist an ordering of $t_{k+1} \in SA_2$, since the indices in SA_2 are unique. For each k, we construct the set of pairs P, given by $P = \{\langle t_k, SA_2'[k+1])|t_k \in T_1\rangle\}$. Each pair in P is unique. Then, we radix sort P to generate the suffix array SA_1 of T_1. This step can thus be accomplished in linear time. This only works for the highest level (*i.e.*, obtaining SA_1 from SA_2). However, we can use a similar procedure, but with some added work, at the other levels of recursion.

Consider a lower level of recursion, say at level h. See Figure 3. We can append the values in the SA from the right tree so that we can perform successive bucket sorts. Thus, we use the SA from the right tree as the tie breaker, after a certain number of steps. In general, this will not change the number of comparisons at the lowest level of the tree. However, for the general case, this will ensure that at most α^{h-1} bucket sorts are performed, involving $\alpha^{h_{max}-h}$ symbols in each bucket sort, where $h_{max} = \lceil \log_\alpha n \rceil$ is the lowest level of recursion. For instance, using the example in Figure 3, at $h = 3$, we will have $T_1 = a^2a^8$, and $T_2 = b^4a^7a^{11}a^{13}$ (for convenience, we have used superscripts to denote the respective positions in the original sequence T). Assume T_2 has been sorted to obtain $SA_2 = [3\ 2\ 1\ 0]$. Using SA_2, we now form the sequences:

$$T[2]T[4] \circ 3$$
$$T[8]T[11] \circ 1$$

where symbol '$\circ$' denotes concatenation. The last symbol in each sequence is obtained from SA_2, the suffix array of the right tree. These sequences are then bucket-sorted to obtain SA_1. Since bucket sorting is linear in the number of input symbols, in the worst case, this will result in a total of $O(n)$ time at *each* level. Thus, the overall time for this procedure will still be in $O(n \log n)$ worst case.

We can also observe that with the asymetric partitioning, and by making the recursive call on only one partition at each level, we reduce both the overall number of conflicts and number of comparisons to $O(n)$. But the number of lookups could still be in $O(n \log n)$ worst case.

4.4. Improved Sorting - Using Shannon-Fano-Elias Codes

The key to improved sorting is to reduce the number of bucket sorts in the above procedure. We do this by pre-computing some information before hand, so that the sorting can be performed based on a small block of symbols, rather than one symbol at a time. Let m be the block size. With the precomputed information, we can perform a comparison involving an m-block symbol in $O(1)$ time. This will reduce the number of bucket sorts required at each level h from α^{h-1} to $\frac{\alpha^{h-1}}{m}$, each involving $\alpha^{h_{max}-h}$ symbols. By an appropriate choice of m, we can reduce the complexity of the overall sorting procedure. For instance, with $m = \log\log n$, this will lead to an overall worst case complexity in $O(\frac{n \log n}{\log\log n})$ time for determining the suffix array of T_1 from that of T_2. With $m = \log n$, this gives $O(n)$ time. We use $m = \log n$ in subsequent discussions.

The question that remains then is *how* to perform the required computations, such that *all* the needed block values can be obtained in linear time. Essentially, we need a pair-wise global partial ordering of the suffixes involved in each recursive step. First, we observe that we only need to consider the ordering between pairs of suffixes at the same level of recursion. The relative order between suffixes at different levels is not needed. For instance, using the example sequence of Figure 3, the sets of suffixes for which we need the pair-wise orderings will be those at positions: $\{\{1, 5, 10, 14\}; \{2, 8\}; \{4, 13\}\}$. Each subset corresponds to a level of recursion, from $h = 2$ to $h = h_{max} - 1$. Notice that we don't necessarily need those at level $h = 1$, as we can directly induce the sorted order for these suffixes from SA_2, after sorting T_2.

We need a procedure to return the relative order between each pair of suffix positions in constant time. Given that we already have an ordering from the right tree in SA_2, we only need to consider the prefixes of the suffixes in the left tree up to the corresponding positions in T_2, such that we can use entries in SA_2 to break the tie, after a possible $\frac{\alpha^{h-1}}{m}$ bucket sorts. Let Q_i be the m-length prefix of the suffix T'_i: $Q_i = T[i \ldots i + m - 1]$. We can use a simple hash function to compute a representative of Q_i, for instance using the polynomial hash function:

$$h(Q_i) = \textstyle\sum_{j=0}^{m-1} |\Sigma|^{m-1} f(Q_i[j]) \mod n'$$

where $f(x) = k$, if x is the k-th symbol in $\Sigma, k = 0, 1, \ldots, |\Sigma| - 1$, and n' is the nearest prime number $\geq n$. The problem is that the ordering information is lost in the modulus operation. Although order-preserving hash functions exist (see [31]), these run in $O(n)$ time on average, without much guarantees on their worst case. Also, with the m-length blocks, this may require $O(mn) = O(n \log n)$ time on average.

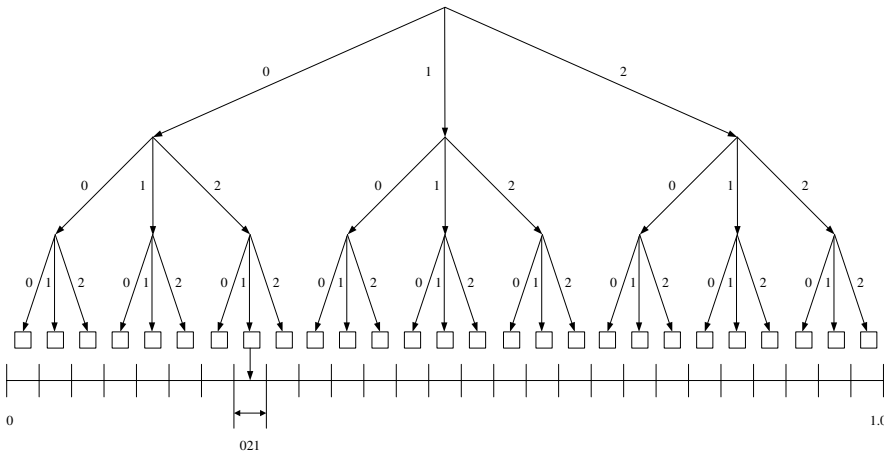
We use an information theoretic approach to determine the ordering for the pairs. We consider the required representation for each m-length block as a codeword that can be used to represent the block. The codewords are constrained to be order preserving: That is, $\mathcal{C}(Q_i) < \mathcal{C}(Q_j)$ iff $Q_i \prec Q_j$ and $\mathcal{C}(Q_i) = \mathcal{C}(Q_j)$ iff $Q_i = Q_j$, where $\mathcal{C}(x)$ is the codeword for sequence x. Unlike in traditional source coding where we are given *one* long sequence to produce its compact representation, here, we have a set of short sequences, and we need to produce their respective compact representations, and these representations must be order preserving.

Let P_i be the probability of Q_i, the m-length block starting at position i in T. Let p_i be the probability of symbol $t_i = T[i]$. If necessary, we can pad T with a maximum of $(m-1)$ '\$' symbols, to form a valid m-block at the end of the sequence. We compute the quantity: $P'_i = \prod_{k=i}^{i+m-1} p_k$. Recall that $t_i = T[i] \in \Sigma$, and $\sum_{j=0}^{|\Sigma|-1} Pr\{\sigma_j\} = 1$. For a given sequence T, we should have: $\sum_{i=0}^{n-1} P'_i = 1$. However, since T may not contain all the possible $m-$length blocks in Σ^m, we need to normalize the product of probabilities to form a probability space:

$$P_i = \frac{P'_i}{\sum_{i=0}^{n-1} P'_i} \tag{1}$$

To determine the code for Q_i, we then use the cumulative distribution function (cdf) for the P_i's, and determine the corresponding position for each P_i in this cdf. Essentially, this is equivalent to dividing a number line in the range [0 1], such that each Q_i is assigned a range proportional to its probability, P_i. See Figure 4. The total number of divisions will be equal to the number of unique m-length blocks in T. The problem then is to determine the specific interval on this number line that corresponds to Q_i, and to choose a tag q_i to represent Q_i.

Figure 4. Code assignment by successive partitioning of a number line.

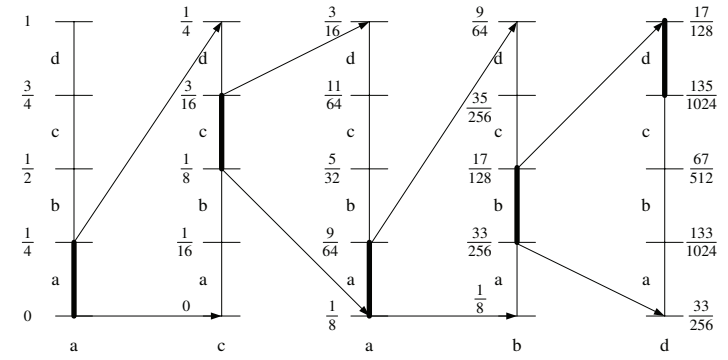


We use the following assignment procedure to compute the tag, q_i. First we determine the interval for the tag, based on which we compute the tag. Define the cumulative distribution function for the symbols in $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma|-1}\}$: $F_x(\sigma_j) = \sum_{v=0}^{j} Pr\{\sigma_v\}$. The symbol probabilities, $Pr\{\sigma_v\}$'s are simply obtained based on the p_i's. For each symbol σ_k in Σ, we have an open interval in the cdf: $[F_x(\sigma_{k-1}) \quad F_x(\sigma_k))$. Now, given the sequence $Q_i = s_1 s_2 \ldots s_k \ldots s_m$, $s_i \in \Sigma$, the procedure steps through the sequence. At each step $k, k = 1, 2, \ldots, m$ along the sequence, we can compute $U(k)$ and $L(k)$, the respective current upper and lower ranges for the tag using the following relations:

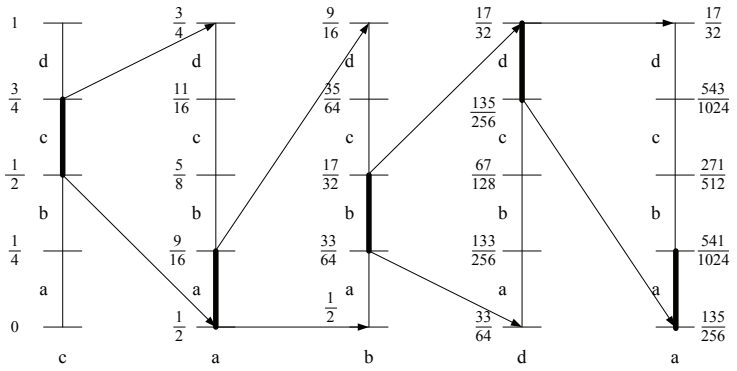
$$L(0) = 0; U(0) = 1$$
for $k = 1$ to m :
$$U(k) = L(k-1) + [U(k-1) - L(k-1)]F_x(s_k)$$
$$L(k) = L(k-1) + [U(k-1) - L(k-1)]F_x(s_k - 1)$$

The procedure stops at $k = m$, and the values of $U(k)$ and $L(k)$ at this final step will be the range of the tag, q_i. We can choose the tag q_i as any number in the range: $L(m) \leq q_i < U(m)$. Thus we chose q_i as the mid point of the range at the final step: $q_i = \frac{U(m)+L(m)}{2}$. Figure 5(a) shows an example run of this procedure for a short sequence: $Q_i = acabd$ with a simple alphabet, $\Sigma = \{a, b, c, d\}$, and where each symbol has an equal probability $p_i = \frac{1}{4}$. This gives $q_i = \frac{271}{2048}$.

Figure 5. Code assignment procedure, using an example sequence. The vertical line represents the current state of the number line. The current interval at each step in the procedure is shown with a darker shade. The symbol considered at each step is listed under their respective number lines (a) Using the sequence $Q_i = acabd$. (b) Evolution of code assignment procedure, after removing the first symbol (in previous sequence *acabd*), and bringing in a new symbol *a* to form a new sequence: $Q_{i+1} = cabda$.



(a)



(b)

Lemma 2: The tag assignment procedure results in a tag q_i that is unique to Q_i.

Proof. The procedure described can be seen as an extension of the Shanon-Fano-Elias coding procedure used in source coding and information theory [32]. Each tag q_i is analogous to an arithmetic coding sequence of the given m-length block, Q_i. The open interval defined by $[(L(m) \;\; U(m))$ for each m-length sequence is unique to the sequence.

To see this uniqueness, we notice that the final number line at step m represents the cdf for all m-length blocks that appeared in the original sequence T. Denote this cdf for the m-blocks as F_x^m. Given Q_i, the i-th m-block in T, the size of its interval is given by $(U(m) - L(m)) = P_i$. Since all probabilities are positive, we see that $F_x^m(Q_i) \neq F_x^m(Q_j)$ whenever $Q_i \neq Q_j$. Therefore, $F_x^m(Q_i)$ determines Q_i uniquely. Thus, $F_x^m(Q_i)$ serves as a unique code for Q_i. Choosing any number q_i within the upper and lower bounds for each Q_i define a unique tag for Q_i. Thus the chosen tag defined by the midpoint of this interval is unique to Q_i. □

Lemma 3: The tags generated by the assignment procedure are order preserving.

Proof. Consider the ordering of the tags for different m-length blocks. Each step in the assignment procedure uses the same fixed order of the symbols on a number line, based on their order in Σ. Thus, the position of the upper and lower bounds at each step depends on the previous symbols considered, and the position of the current symbol in the ordered list of symbols in Σ. Therefore the q_i's are ordered with respect to the lexicographic ordering of the Q_i's: $q_i < q_j \Leftrightarrow Q_i \prec Q_j$, and $q_i = q_j \Leftrightarrow Q_i = Q_j$. □

Lemma 4: Given $T = t_0 t_1 \ldots t_{n-1}$, all the required tags $q_i, \forall i$, can be computed in $O(n)$ time.

Proof. Suppose we have already determined P_i and q_i for the m-block Q_i as described above. For efficient processing, we can compute P_{i+1} and the tag q_{i+1} in the $[0\ 1]$ number line, using the previous values for P_i and q_i. This is based on the fact that Q_i and Q_{i+1} are consecutive positions in T (In practice, we need only a fraction of the positions in T, which will mean less time and space are required. But here we describe the procedure for the entire T since the complexity remains the same.). In particular, given $Q_i = T[i \ldots i+m-1]$, and $Q_{i+1} = T[i+1 \ldots i+m]$. We compute P'_{i+1} as:

$$P'_{i+1} = \frac{P'_i \cdot p_{i+m}}{p_i}$$

Then, we compute P_{i+1} using Equation (1). Thus, all the required P_i's can be computed in $O(n+m) = O(n + \log n) = O(n)$ time.

Similarly, given the tag q_i for Q_i, and its upper and lower bounds $U(m)$ and $L(m)$, we can compute the new tag q_{i+1} for the incoming m-block, Q_{i+1} based on the structure of the assignment procedure used to compute q_i, (see Figure 5(b)). We compute the new tag q_{i+1} by first computing it's upper and lower bounds. Denote the respective upper and lower bounds for q_i as: $U^i(m), L^i(m)$. Similarly, we use $U^{i+1}(m), L^{i+1}(m)$ for the respective bounds for q_{i+1}. Let $s = T[i] = Q_i[0]$ be the first symbol in Q_i. Its probability is given by p_i. Also, let $s_{new} = T[i+m]$ be the new symbol that is shifted in. Its probability is given by p_{i+m}, and we also know it's position in the cdf. We first compute the intermediate bounds at step $k = m-1$ when using Q_{i+1}, namely:

$$U^{i+1}(m-1) = [U(m) - F_x(s)](\tfrac{1}{p_i})$$
$$L^{i+1}(m-1) = [L(m) - F_x(s-1)](\tfrac{1}{p_i})$$

Multiplying by $(\frac{1}{p_i})$ changes the probability space from the previous range of $[F_x(s-1)\ F_x(s))$ to $[0\ 1]$. After the computations, we can then perform the last step in the assignment procedure to determine the final range for the new tag:

$$U^{i+1}(m) = L^{i+1}(m-1) + [U^{i+1}(m-1) - L^{i+1}(m-1)]F_x(s_{new})$$
$$L^{i+1}(m) = L^{i+1}(m-1) + [U^{i+1}(m-1) - L^{i+1}(m-1)]F_x(s_{new}-1)$$

The tag q_{i+1} is then computed as the average of the two bounds as before. The worst case time complexity of this procedure is in $O(n+m+|\Sigma|) = O(n+\log n+|\Sigma|)$. The $|\Sigma|$ component comes from the time needed to sort the unique symbols in T before computing the cdf. This can be performed in linear time using counting sort. Since $|\Sigma| \leq n$, this gives a worst case time bound of $O(n)$ to compute the required codes for all the $O(n)$ m-length blocks. □

Figure 5(b) shows a continuation of the previous example, with the old m-block: $Q_i = acabd$, and a new m-block $Q_{i+1} = cabda$. That is, the new symbol a has been shifted in, while the first symbol in the old block has been shifted out. We observe that the general structure in Figure 5(a) is not changed by the incoming symbol, except only at the first step and last step. For the running example, the new value will be $q_{i+1} = \frac{1051}{2048}$. Table 2 shows the evolution of the upper and lower bounds for the two adjacent m-blocks. The bounds are obtained from the figures.

Table 2. Upper and lower bounds on the current interval on the number line.

	a	c	a	b	d	c	a	b	d	a
U_k	1	$\frac{1}{4}$	$\frac{3}{16}$	$\frac{9}{64}$	$\frac{17}{128}$	1	$\frac{3}{4}$	$\frac{9}{16}$	$\frac{17}{32}$	$\frac{17}{32}$
L_k	0	0	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{33}{256}$	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{33}{64}$	$\frac{135}{256}$
$U_k - L_k$	1	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{64}$	$\frac{1}{256}$	1	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{64}$	$\frac{1}{256}$

Having determined q_i, which is fractional, we can then assign the final code for Q_i by mapping the tags to an integer in the range $[0\quad n-1]$. This can be done using a simple formula:

$$c_i = \mathcal{C}(Q_i) = \lfloor (n-1)\frac{q_i - q_{min}}{q_{max} - q_{min}} \rfloor$$

where $q_{min} = \min_i\{q_i\}$, and $q_{max} = \max_i\{q_i\}$. Notice that here, the c_i's computed will not necessarily be consecutive. But they will be ordered. Also, the number of distinct q_i's is at most n. The difference between c_i and c_{i+1} will depend on P_i and P_{i+1}. The floor function, however, could break down the lexicographic ordering. A better approach is to simply record the position where each Q_i fell on the number line. We then read off these positions from 0 to 1, and use the count at which each Q_i is encountered as its code. This is easily done using the cummulative count of occurence of each distinct Q_i. Since the q_i's are implicitly sorted, so are the c_i's. We have thus obtained an ordering of all the m-length substrings in T. This is still essentially a **partial ordering** of all the suffixes based on their first m symbols, but a total order on the distinct m-length prefixes of the suffixes.

We now present our main results in the following theorems:

Theorem 2: Given the sequence $T = t_0 t_1 \dots t_{n-1}$, from a fixed alphabet Σ, $|\Sigma| \leq n$, all the m-length prefixes of the suffixes of T can be ordered in linear time and linear space in the worst case.

Proof. The theorem follows from Lemmas 2 to 4. The correctness of the ordering follows from Lemma 2 and Lemma 3. The time complexity follows from Lemma 3. What remains is to prove is the space complexity. We only need to maintain two extra $O(|\Sigma|)$ arrays, one for the number line at each step, and the other to keep the cumulative distribution function. Thus, the space needed is also linear in $O(n + m + |\Sigma|) = O(n)$. □

Theorem 3: Given the sequence $T = t_0 t_1 \dots t_{n-1}$, with symbols from a fixed alphabet Σ, $|\Sigma| \leq n$, Algorithm II computes the suffix array of T in $O(n)$ worst case time and space.

Proof. At each iteration, the recursive call applies only to the $\frac{2}{3}n$ suffixes in T_2. Thus, the running time for the algorithm is given by the solution to the recurrence : $\varphi(n) = \varphi(\lceil \frac{2}{3}n \rceil) + O(n)$. This gives $\varphi(n) = O(n)$. Combined with Theorem 2, this establishes the linear time bound for the overall algorithm.

We improve the space requirement using an alternative merging procedure. Since we now have SA_1 and SA_2, we can modify the merging step by exploiting the fact that any conflict that can arise during the merging can be resolved by using only SA_2. To resolve a conflict between suffix T_{1_i} in T_1 and suffix T_{2_j} in T_2, we need to consider two cases:

- Case 1: If $j \bmod 3 = 1$, we compare $\langle T_1[i], SA_2[SA'_2[i+1]] \rangle$ versus $\langle T_2[j], SA_2[SA'_2[j+1]] \rangle$, since the relative order of both $T_{1_{i+1}}$ and $T_{2_{j+1}}$ are available from SA_2.
- Case 2: If $j \bmod 3 = 2$, we compare $\langle T_1[i], T_1[i+1], SA_2[SA'_2[i+2]] \rangle$ versus $\langle T_2[j], T_2[j+1], SA_2[SA'_2[j+1]] \rangle$. Again, for this case, the tie is broken using the triplet, since the relative order of both $T_{1_{i+2}}$ and $T_{2_{j+2}}$ are also available from SA_2.

Consider the step just before we obtain SA_1 from SA_2 as needed to obtain the final SA. We needed the codes for the m-blocks in sorting to obtain SA_2. Given the 1:2 non-symetric partitioning used, at this point, the number of such m-blocks needed for the algorithm will be $\frac{2}{3}.\frac{2}{3}n$. These require $\frac{4}{9}n$ integers to store. We need $\frac{2}{3}n$ integers to store SA_2. At this point, we also still have the inverse SA used to merge the left and right suffix arrays to form SA_2. This requires $\frac{4}{9}n$ integers for storage. Thus, the overall space needed at this point will be $1\frac{5}{9}n$ integers, in addition to the space for T. However, after getting SA_2, we no longer need the integer codes for the m-length blocks. Also, the merging does not involve SA'_1, so this need not be computed. Thus, we compute SA'_2, and re-use the space for the m-block codes. SA'_2 requires $\frac{2}{3}n$ integers. Further, since we are merging SA_1 and SA_2 from the same direction, we can construct the final SA in-place, by re-using part of the space used for the already merged sections of SA_1 and SA_2. (See for example [33, 34]). Thus, the overall space requirement in bits will be $(n \log |\Sigma| + n \log n + \frac{2}{3}n \log n) = (n \log |\Sigma| + 1\frac{2}{3}n \log n)$, where we need $n \log |\Sigma|$ bits to store T, $n \log n$ bits for the output suffix array, and $\frac{2}{3}n \log n$ bits for SA'_2. □

The above translates to a total space requirement of $7\frac{2}{3}n$ bytes, using standard assumptions of 4 bytes per integer, and 1 byte per symbol.

Though the space above is $O(n)$, the $\frac{2}{3}n\log n$ bits used to store $SA_2^{'}$ could further be reduced. We do this by by making some observations in the above two cases encountered during merging. We can notice that after obtaining SA_2 and SA_1, we do not really need to store the text T anymore. The key observation is that, merging of SA_2 and SA_1 proceeds in the same direction for each array, for instance, from the least to the largest suffix. Thus, at the k-th step, the symbol at position $i = SA_2^{'}[k]$ (that is, $T_2[i]$) can easily be obtained using SA_2, and two $O(|\Sigma|)$ arrays, namely, B_1: which stores the symbols in Σ in lexicographic order, and B_2 that stores the cummulative count for each symbol in T_2.

For $T_2[i+1]$, we compute $SA_2[SA_2^{'}[i]+1])$ and use the value as index into B_2. We then use the position in the B_2 arrray to determine the symbol value from B_1. Similarly we obtain the symbol $T_1[i] = T_1[SA_1^{'}[k]]$, using a second set of $O(|\Sigma|)$ arrays. For symbol $T_1[i+1]$ we do not have $SA_1^{'}$. However, we can observe that symbol $T_1[i+1]$ will be some symbol $T_2[j]$ in T_2. Hence, we can use SA_2 and $SA_2^{'}$ to determine the symbol, as described above.

Thus, we can now release the space currently used to store T and use this in part to store $SA_2^{'}$, and then merge SA_1 and SA_2 using $SA_2^{'}$ and the two sets of $O(|\Sigma|)$ B_1, B_2 arrays. The space saving gained in doing this will be: $(n\log|\Sigma| - 2(|\Sigma|\log|\Sigma| + |\Sigma|\log n)) \approx n\log|\Sigma|$ bits. Using this in the previous space analysis leads to a final space requirement of $(n\log n + \frac{2}{3}n\log n + 2|\Sigma|(\log|\Sigma| + \log n) - n\log|\Sigma|)$ $\approx (1\frac{2}{3}n\log n - n\log|\Sigma|)$ bits. This gives $5\frac{2}{3}n$ bytes, assuming $n \leq 2^{32}$, $|\Sigma| \leq 256$ at 4 bytes per integer.

Finally, since we do not need $SA_2^{'}$ anymore, we can release the space it is occupying. Compute a new set of B_1 and B_2 arrays (in place) for the newly computed SA. The second set of $O(|\Sigma|)$ arrays are no longer needed. Using SA and the new B_1 and B_2 arrays, we now recover the original sequence T, at no extra space cost.

5. Conclusion and Discussion

We have proposed two algorithms for solving the suffix sorting problem. The first algorithm runs in $O(n\log n)$ time and $O(n)$ space, for both the averge case and worst case. Using ideas from Shannon-Fano-Elias codes used in infomation theory, the second algorithm improved the first to an $O(n)$ worst case time and space complexity. The algorithms proposed perform direct suffix sorting on the input sequence, circumventing the need to first construct the suffix tree.

We mention that the proposed algorithms are generally independent of the type of alphabet, Σ. The only requirement is that Σ be fixed during the run of the algorithm. Any given fixed alphabet can be mapped to a corresponding integer alphabet. Also, since practically the number of unique symbols in $|T|$ cannot be more than n, the size of T, it is safe to say that $n \geq |\Sigma|$.

For practical implementation, one will need to consider the problem of practical code assignment, since the procedure described may involve dealing with very small fractions, depending on m, the block length, and $|\Sigma|$. This is a standard problem in practical data compression. With $n \leq 2^{32}$, $|\Sigma| = 256$, we have $m = 32$,and thus we may need to store values as small as $\frac{1}{|\Sigma|^{32}} = \frac{1}{256^{32}}$ while computing the tags. This translates to $\frac{1}{2^{256}}$, or about 1.158×10^{-77}. In most implementations, a variable of type `double` can store values as small as 1.7×10^{-308}. For the case of 1:2 asymetric partitioning used above, we need only only $\frac{4n}{9}$ of the m-blocks, and hence, the overall space needed to store them will still be n integers, since the size of double is typically twice that of integers. To use the approach for much larger sequences,

periodic re-scaling and re-normalization schemes can be used to address the problem. Another approach will be to consider occurrence counts, rather than occurrence probabilities for the m-blocks. Here, the final number line will be a cummulative count, rather than a cdf, with the total counts being n. Then, the integer code for each m-block can easily be read off this number line, based on its overall frequency of occurrence. Therefore, we will need space for at most $\frac{4n}{9} + |\Sigma|$ integers in computing the integer codes for the positions needed. Moffat *et al.* [35] provide some ideas on how to address practical problems in arithmetic coding.

Overall, by a careful re-use of previously allocated space, the algorithm requires $(1\frac{2}{3}n \log n - n \log |\Sigma|)$ bits, including the n bytes needed to store the original string. This translates to $5\frac{2}{3}n$ bytes, using standard assumptions of 4 byte per integer, and 1 byte for each symbol. This is a significant improvement over the $10n$ bytes required by the KA algorithm [10], or the $13n$ bytes required by the KS algorithm [9]. Our algorithm is also unique in its use of Shannon-Fano-Elias codes, traditionally used in source coding, for efficient suffix sorting. This is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem. We believe that this new idea of using an information theoretic approach to suffix sorting could shed a new light on the problems of suffix array construction, their analysis, and applications.

Acknowledgements

This work was partially supported by a DOE CAREER grant: No: DE-FG02-02ER25541, and an NSF ITR grant: IIS-0228370. A short version of this paper was presented at the 2008 IEEE Data Compression Conference, Snowbird, Utah.

References

1. Manber, U.; Myers, G. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* **1993**, *22*, 935–948.
2. Larsson, N.J.; Sadakane, K. Faster suffix sorting. *Theoret. Comput. Sci.* **2007**, *317*, 258–272.
3. Manzini, G.; Ferragina, P. Engineering a lightweight suffix array construction algorithm. *Algorithmca* **2004**, *40*, 33–50.
4. Puglisi, S.J.; Smyth, W.F.; Turpin, A. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.* **2007**, *39*, 1–31.
5. Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*; Cambridge University Press: Cambridge, UK, 1997.
6. Burrows, M.; Wheeler, D.J. *A Block-Sorting Lossless Data Compression Algorithm*; Research Report 124; Digital Equipment Corporation: Palo Alto, CA, USA, 1994.
7. Adjeroh, D.; Bell, T.; Mukherjee, A. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*; Springer-Verlag: New York, NY, USA, 2008.
8. Seward, J. On the performance of BWT sorting algorithms. In *Proceedings of IEEE Data Compression Conference*, Snowbird, UT, USA, March 28–30, 2000; Volume 17, pp. 173–182.
9. Kärkkäinen, J.; Sanders, P.; Burkhardt, S. Linear work suffix array construction. *J. ACM* **2006**, *53*, 918–936.

10. Ko, P.; Aluru, A. Space-efficient linear time construction of suffix arrays. *J. Discrete Algorithms* **2005**, *3*, 143–156.
11. Cleary, J.G.; Teahan, W.J. Unbounded length contexts for PPM. *Comput. J.* **1997**, *40*, 67–75.
12. Bell, T.; Cleary, J.; Witten, I. *Text Compression*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1990.
13. Szpankowski, W. Asymptotic properties of data compression and suffix trees. *IEEE Trans. Inf. Theory* **1993**, *39*, 1647–1659.
14. Abouelhoda, M.I.; Kurtz, S.; Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms* **2004**, *2*, 53–86.
15. Farach-Colton, M.; Ferragina, P.; Muthukrishnan, S. On the sorting-complexity of suffix tree construction. *J. ACM* **2000**, *47*, 987–1011.
16. Kim, D.K.; Sim, J.S.; Park, H.; Park, K. Constructing suffix arrays in linear time. *J. Discrete Algorithms* **2005**, *3*, 126–142.
17. Nong G.; Zhang, S. Optimal lightweight construction of suffix arrays for constant alphabets. In *Proceedings of Workshop on Algorithms and Data Structures*, Halifax, Canada, August 15–17, 2007; Volume 4619, pp. 613–624.
18. Maniscalco, M.A.; Puglisi, S.J. Engineering a lightweight suffix array construction algorithm. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 10–12, 2008.
19. Itoh, H.; Tanaka, H. An efficient method for in memory construction of suffix arrays. In *Proceedings of String Processing and Information Retrieval Symposium and International Workshop on Groupware*, Cancun, Mexico, September 22–24, 1999; pp. 81–88.
20. Hon, W.; Sadakane, K.; Sung, W. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, Cambridge, MA, USA, October 11–14, 2003.
21. Na, J.C. Linear-time construction of compressed suffix arrays using $O(n \log n)$-bit working space for large alphabets. In *Proceedings of 16th Annual Symposium on Combinatorial Pattern Matching 2005, LNCS*, Jeju Island, Korea, June 19–22, 2005; Volume 3537, pp. 57–67.
22. Burkhardt, S.; Kärkkäinen, J. Fast lightweight suffix array construction and checking. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, USA, January 12–14, 2003.
23. Nong, G.; Zhang, S.; Chan, W.H. Linear time suffix array construction using D-critical substrings. In *CPM*; Kucherov, G., Ukkonen, E., Eds.; Springer: New York, NY, USA, 2009; Volume 5577, pp. 54–67.
24. Nong, G.; Zhang, S.; Chan, W.H. Linear suffix array construction by almost pure induced-sorting. In *DCC*; Storer, J.A., Marcellin, M.W., Eds.; IEEE Computer Society: Hoboken, NJ, USA, 2009; pp. 193–202.
25. Franceschini, G.; Muthukrishnan, S. In-place suffix sorting. In *ICALP*; Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A., Eds.; Springer: New York, NY, USA, 2007; Volume 4596; pp. 533–545.
26. Okanohara, D.; Sadakane, K. A linear-time Burrows-Wheeler Transform using induced sorting. In *SPIRE*; Karlgren, J., Tarhio, J., Hyyrö, H., Eds.; Springer: New York, NY, USA, 2009; Volume 5721; pp. 90–101.

27. Ferragina, P.; Manzini, G. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Scienc*, Redondo Beach, CA, USA, November 12–14, 2000; pp. 390–398.
28. Grossi, R.; Vitter, J.S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, Baltimore, MD, USA, May 22–24, 2005.
29. Sirén, J. Compressed suffix arrays for massive data. In *SPIRE*; Karlgren, J., Tarhio, J., Hyyrö, H., Eds.; Springer: New York, NY, USA, 2009; Volume 5721, pp. 63–74.
30. Karlin, S.; Ghandour, G.; Ost, F.; Tavare, S.; Korn, L. New approaches for computer analysis of nucleic acid sequences. *Proc. Natl. Acad. Sci. USA* **1983**, *80*, 5660–5664.
31. Fox, E.A.; Chen, Q.F.; Daoud, A.M.; Heath, L.S. Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Syst.* **1991**, *9*, 281–308.
32. Cover, T.M.; Thomas, J.A. *Elements of Information Theory*; Wiley Interscience: Malden, MA, USA, 1991.
33. Symvonis, A. Optimal stable merging. *Comput. J.* **1995**, *38*, 681–690.
34. Huang, B.; Langston, M. Fast stable sorting in constant extra space. *Comput. J.* **1992**, *35*, 643–649.
35. Moffat, A.; Neal, R.M.; Witten, I.H. Arithmetic coding revisited. *ACM Trans. Inf. Syst.* **1995**, *16*, 256–294.