

In-Place Suffix Sorting

G. Franceschini¹ and S. Muthukrishnan²

¹ Department of Computer Science, University of Pisa

`francesc@di.unipi.it`

² Google Inc., NY

`muthu@google.com`

Abstract. Given string $T = T[1, \dots, n]$, the *suffix sorting* problem is to lexicographically sort the suffixes $T[i, \dots, n]$ for all i . This problem is central to the construction of suffix arrays and trees with many applications in string processing, computational biology and compression. A bottleneck in these applications is the amount of *workspace* needed to perform suffix sorting beyond the space needed to store the input as well as the output. In particular, emphasis is even on the constant c in the $O(n) = cn$ space algorithms known for this problem, Currently the best previous result [5] takes $O(nv + n \log n)$ time and $O(n/\sqrt{v})$ extra space, for any $v \in [1, \sqrt{n}]$ for strings from a general alphabet. We improve this substantially and present the first known in-place suffix sorting algorithm. Our algorithm takes $O(n \log n)$ time using $O(1)$ workspace and is optimal in the worst case for the general alphabet.

1 Introduction

Given string $T = T[1, \dots, n]$, the *suffix sorting* problem is to lexicographically sort the suffixes $T[i, \dots, n]$ for all i . Formally, the output is the array S such that if $S[j] = k$, $T[k, \dots, n]$ is the j th smallest suffix.³

This problem is central to many applications in string processing, computational biology and data compression. For instance, the array S is in fact the *suffix array* for string T and is directly applicable to many problems [3]. The classical suffix tree is a compressed trie in which the leaves comprise S . Finally, the beautiful Burrows-Wheeler transform uses S to compress T , and is a popular data compression method.

There are algorithms that will solve this problem using $O(n)$ *workspace*, i.e., space used in addition to the space needed to store T and S . However, in many applications, T is often very large, for example when T is a biological sequence, or large corpus. Therefore, for more than a decade, research in this area has been motivated by the fact that even constants in $O(n)$ matter. For example, the motivation to work with suffix arrays rather than suffix trees arose from decreasing the workspace used from roughly $11n$ to roughly $3n$. Since then the goal has been to minimize the extra workspace needed. This was explicitly posed as an open problem in [2].

³ As is standard, we will assume that the end of string is lexicographically smaller than all the other symbols in the string and hence, unequal strings can be compared in a well-defined way.

Currently the best previous result [5] takes $O(nv + n \log n)$ time and $O(n/\sqrt{v})$ extra space, for any $v \in [1, \sqrt{n}]$. Here we assume the general alphabet model in which the string elements can be compared pairwise.⁴ This has a variety of trade-offs: one is $O(n \log n)$ time and $O(n)$ space, and the other is $O(n^{3/2})$ time and $O(n^{1/4})$ space, depending on v .

Our main result is a substantial improvement over the above. In particular, we present the first known in-place suffix sorting algorithm, that is, our algorithm uses only $O(1)$ workspace. The running time of our algorithm is $O(n \log n)$ which is optimal in the general alphabet model since even sorting the n characters will take that much time in the worst case. Formally, we prove:

Theorem 1. *The suffixes of a text T of n characters drawn from a general alphabet Σ can be sorted in $O(n \log n)$ time using $O(1)$ locations besides the ones for T and the suffix array S of T .*

2 Preliminaries

We are given a text T of n characters drawn from a general alphabet Σ and an array S of n integers of $\lceil \log n \rceil$ bits each. A total order relation \leq is defined on Σ , the characters are considered atomic (no bit manipulation, hashing or word operations) and the only operations allowed on them are comparisons (w.r.t. \leq). Let T_i be the suffix of T starting with the character $T[i]$ (i.e. $T_i = T[i]T[i+1] \cdots T[n]$) and let integer i be the *suffix index* of T_i . The objective is to sort lexicographically the n suffixes of T . The result, consisting of the n suffix indices permuted according to the lexicographical order of the suffixes, is to be stored in S . Apart from accessing T (readonly) and S , we are allowed to use *only* $O(1)$ integers of $\lceil \log n \rceil$ bits each to carry out the computation.

In the following we denote with $<$ the lexicographical order relation. For any suffix T_i , we refer to T_{i-1} (T_{i+1}) as the *text-predecessor* (*text-successor*) of T_i . The terms *sequence* and *subarray* will have slightly different meanings. Even though they are both composed by contiguous elements of an array, a subarray is intended to be just a *static portion* of an array while sequences are *dynamic* and can be moved, exchanged, permuted etc. For any string A we denote with $A[i \dots j]$ the contiguous substring going from the i -th position to the j -th position of A . We extend the same notation to arrays and sequences.

2.1. A Space Consuming Approach

The strategy for sorting suffixes in our solution is based on the simple and elegant approach by Ko and Aluru in [6]. Even though their technique was originally used for the case where $\Sigma = \{1, \dots, n\}$, it can be extended to the comparison model. The result is a suffix sorting algorithm with an optimal $O(n \log n)$ time complexity but requiring $O(n)$ auxiliary locations in addition to S .

Let us recall Ko and Aluru's approach. The suffixes of T are classified as follows: a suffix T_i is an α -*suffix* (a β -*suffix*) if $T_i < T_{i+1}$ ($T_{i+1} < T_i$), that is if

⁴ The bounds in [5] are for strings with integer alphabet. The bound we have quoted is the best possible time bound they can achieve in the general alphabet model.

it is less (greater) than its text-successor w.r.t. the lexicographical order (T_n is classified as a β -suffix by convention). This classification has the following main property: *for any α -suffix T_i and any β -suffix T_j , if $T[i] = T[j]$ then $T_j \prec T_i$.*

Let us assume without loss of generality that the α -suffixes of T are fewer in number than the β -suffixes. An α -substring of T is a substring $A_i = T[i]T[i+1] \cdots T[i']$ such that (i) both T_i and $T_{i'}$ are α -suffixes and (ii) T_j is a β -suffix, for any $i < j < i'$. We need to sort the α -substring of T according to a variation of the lexicographical order relation, the *in-lexicographical order*, from which it differs in only one case: if a string s is a prefix of another string s' , then s follows s' . For any multiset \mathcal{M} (i.e. a collection allowing duplicates), for any total order $<$ defined on \mathcal{M} and for any $o \in \mathcal{M}$, the *bucket number of o* is the rank of o according to $<$ in the set $\mathcal{S}_{\mathcal{M}}$ obtained from \mathcal{M} by removing all the duplicates.

The Ko and Aluru's approach proceeds with the following three main steps.

First. We sort *in-lexicographically* the α -substrings of T .

Second. We build a string T' from T by replacing any α -substring with its bucket number (according to the *in-lexicographical order*). Then, we sort the suffixes of T' recursively, obtaining the corresponding array S' . Because of the main property of the α - β classification and by the definition of the *in-lexicographical order*, sorting the suffixes of T' is equivalent to sorting the α -suffixes of T .

Third. We distribute the suffixes into temporary buckets according to their first character. By the main property of the α - β classification we know that any α -suffix is greater than any β -suffix belonging to the same bucket. Therefore, for any bucket, we move all its α -suffixes to its right end and we dispose them in lexicographical order (known since the second step). Then, we move the suffixes from the temporary buckets to S (in the same order we find them in the temporary buckets). Finally, we take care of the β -suffixes. We scan S from left to right. Let T_i be the currently examined suffix. If the text-predecessor of T_i is an α -suffix, we ignore it (it is already in its final position in S). Otherwise, if T_{i-1} is a β -suffix, we exchange T_{i-1} with the leftmost β -suffix T_j having the same first character as T_{i-1} and not yet in its final position (if any). After the scanning process, the suffixes are in lexicographical order.

2.2. Obstacles

Before we proceed with the description of our algorithm, let us briefly consider some of the obstacles that we will have to overcome.

2.2.1 Input partitioning and simulated resources. A common approach for attacking space complexity problems consists of the following phases. First, the input set is partitioned into two disjoint subsets. Then, the problem is solved for the first subset using the second subset to simulate additional space resources. Usually these simulated resources are implemented by permuting the elements in the second subset in order to encode data or, if the model allows it, by compressing them in order to free some bits temporarily. After the problem has been solved for the first subset, the approach is applied recursively on the second one. Finally, the partial solutions for the two subsets are merged into one.

Unfortunately, this basic approach cannot be easily extended to the suffix sorting problem. This is due to the well-known fact that the suffixes of a sequence

cannot be just partitioned into generic subsets to be sorted separately and then merged efficiently. Only few specific types of partitionings are known to have this property and either they exploit some cyclic scheme (e.g. [4]), thus being too rigid for our purposes, or they need to be explicitly represented (e.g. [6]) thereby increasing the auxiliary memory requirements.

2.2.2 Auxiliary information needed in Ko and Aluru’s approach. Not only is the α - β partitioning unsuitable, but it also claims auxiliary resources. Clearly, in the first and third main steps of the Ko and Aluru’s approach, we need to be able to establish whether a particular suffix is an α -suffix or a β -suffix. The number of bits needed to represent the α - β partitioning can be reduced to n/c , for a suitably large integer constant c . We will employ various encoding schemes to maintain this information implicitly during the phases of the computation.

Let us consider the final scanning process of the third main step. For any suffix T_i considered, the positions in S of T_{i-1} and T_j (the leftmost β -suffix such that $T_j[1] = T_{i-1}[1]$ and not yet in its final position) must be retrieved efficiently. To this end, the algorithm in [6] explicitly stores and maintains the inverse array of S . Unlike the case of the α - β partitioning, it is clearly impossible to encode implicitly the necessary $n \lceil \log n \rceil$ bits. Therefore, we will devise an “on-the-fly” approach to the scanning process that will require neither the exchange step of T_{i-1} and T_j nor the use of any implicit encoding scheme.

3 Our Algorithm

In this section we present our optimal in-place suffix sorting algorithm for generic alphabets. We will assume without loss of generality that the α -suffixes of T are fewer in number than the β -suffixes (the other case is symmetric).

The α - β table. In [6] the information for classifying α and β -suffixes is calculated in linear time with a left to right scan of T . It is easy to see how this information can be gathered in linear time also by a right to left scan: by convention T_n is classified as α -suffix and T_{n-1} can be classified by comparing $T[n]$ and $T[n-1]$; for any $i < n-1$, let us assume inductively that the classification of T_{i+1} is known, if $T[i] \neq T[i+1]$ then T_i is classified by the result of the comparison between $T[i]$ and $T[i+1]$, otherwise T_i is of the same type of T_{i+1} . Therefore, to be able to classify any suffix T_i in $O(1)$ time there is no need to store a table with n entries of one bit each. For any integer constant c , we can use a table of n/c bits whose j -th entry represents the classification of T_{cj} . Any suffix T_i can be classified in $O(c)$ time: if the substrings $T_i[1 \dots c-i \bmod c]$ and $T_{i+1}[1 \dots c-i \bmod c]$ differ then T_i is classified by the result of their lexicographical comparison, otherwise T_i is of the same type of $T_{i+c-i \bmod c}$ (whose classification is in the $\frac{i+c-i \bmod c}{c}$ -th entry of the table).

We will refer to this smaller table as the α - β table. We will not be able to keep the α - β table explicitly stored or implicitly encoded all the time. Its information will be lost and recalculated multiple times during any execution.

3.1. Sorting the α -suffixes

In this section we show how to sort the α -suffixes of T . We have four phases.

3.1.1 First phase. We compute the α - β table and we store it in $S[1 \dots n/c]$ (that is, simply using one entry of S for any entry of the table). Let n_α be the number of α -suffixes. While we are scanning T to compute the α - β table we also find the α -substrings and we store the n_α pointers to them in $S[n - n_\alpha + 1, \dots, n]$. Since $n_\alpha \leq n/2$, in the following phases we can exploit $n(1/2 - 1/c)$ free locations in the first half of S (as we have seen, we can choose the constant c , defining the size of the α - β table, as large as we want). Let F denote the subarray of S containing the free locations.

3.1.2 Second phase. We divide the pointers to the α -substrings into d groups G_1, \dots, G_d of n_α/d contiguous pointers each, for a suitable constant d . Then, we sort each group *in-lexicographically* using the locations in the subarray F (and the α - β table to recognize the last position of any α -substring). As we can choose d as large as we want and given that the total length of the α -substrings is $O(n)$, each group can be sorted in $O(n \log n)$ time with any optimal string sorting algorithm operating in linear space (linear w.r.t. to the size of the group).

Now that the groups of α -substrings are *in-lexicographically* sorted we merge them. We first merge the first group with the second one, then the resulting sequence is merged with the third group and so forth. For any i , the i -th single binary merging step is performed with the help of the locations in F in the following way. Let G be the sequence to be merged with the i -th group G_i and let us assume that $|G| + |G_i| > |F|$ (at a certain point this will happen, since $n_\alpha > |F|$). Using the α - β table to recognize the ends of the α -substrings, we initially proceed like in a normal merging (we compare *in-lexicographically* the α -substrings pointed by $G[1]$ and $G_i[1]$ and move a pointer to $F[1]$ and so forth). Since $|G| + |G_i| > |F|$, after $|F|$ of these single string comparison steps F becomes full. At that point we slide what is left of G to the right so that it becomes adjacent with what is left of G_i . After the sliding we resume the merging but now we move the pointers to the subarray of $|F|$ positions that is right after F and that has become free after the sliding. We proceed in this fashion until G or G_i is empty. At that point we compact the sorted pointers with what is left of G or G_i . Finally, we slide the resulting sequence to the right to be adjacent with G_{i+1} (in case we need to).

3.1.3 Third phase. In this phase we build a sequence T' of n_α integers in the range $[1, n_\alpha]$ using the bucket numbers of the α -substrings (see Section 2.1). After the second phase the α -substrings are in *in-lexicographical* order and the pointers to them are permuted accordingly. Let us denote with P the subarray $S[n - n_\alpha + 1 \dots n]$ where the pointers are stored.

We start by modifying the allocation scheme for the α - β table. Up until now the n/c entries have been stored in the first n/c locations of S . We now allocate the α - β table so that the i -th entry is stored in the most significant bit of the $2i$ -th location of S , for any $1 \leq i \leq n/c$. Since we can choose c to be as large as we want, the n_α pointers residing in P will not be affected by the change.

Then, we associate to any pointer the bucket number of the α -substring it points to in the following way. We scan P from left to right. Let two auxiliary variables j and p be initially set to 1 and $P[1]$, respectively. In the first step of

the scanning we set $S[1]$ and $S[2]$ to $P[1]$ and 1, respectively. Let us consider the generic i -th step of the scanning, for $i > 1$.

1. We compare *in-lexicographically* the α -substrings pointed by p and $P[i]$ (using the α - β table to recognize the last positions of the two α -substrings).
2. If they are different we increment j by one and we set p to $P[i]$.
3. In any case, we set $S[2i - 1]$ and $S[2i]$ to $P[i]$ and j , respectively and we continue the scanning.

As we said, the scanning process depends on the α - β table for the substring comparisons. It might seem that writing the current value of j in the locations of S with even indices would destroy the α - β table. That is not the case. After the scanning process, the first $2n_\alpha$ locations of S contains n_α pairs $\langle p_i, b_i \rangle$, where p_i is a pointer to the i -th α -substring (in *in-lexicographical* order) and b_i is the bucket number of it. Since $n_\alpha \leq n/2$, the bits necessary to represent a bucket number for an α -substrings are no more than $\lceil \log(n/2) \rceil$ (and the locations of S have $\lceil \log n \rceil$ bits each). Therefore, the n/c entries of the α - β table and the bucket numbers of the first n/c pair $\langle p_i, b_i \rangle$ can coexist without problems.

After the scanning process we proceed to sort the n_α pairs $\langle p_i, b_i \rangle$ according to their first members (i.e. the pointers are the sorting keys). Since there can be as many as $n/2$ α -substrings, at the worst case we have that all the locations of S are occupied by the sequence of pairs. Therefore, for sorting the pairs we use mergesort together with an in-place, linear time merging like [8]. When the pairs are sorted, we scan them one last time to remove all the pointers, ending up with the wanted sequence T' stored in the first n_α locations of S .

3.1.4 Fourth phase. We start by applying our in-place algorithm recursively to the sequence T' stored in $S[1 \dots n_\alpha]$. At the worst case $|T'| = n_\alpha$ can be equal to $n/2$ and the space used by the algorithm to return the n_α sorted suffixes of T' is the subarray $S' = S[n - n_\alpha + 1 \dots n]$. Concerning the use of recursion, if before any recursive call we were to store explicitly $O(1)$ integer values (e.g. the value of n_α), we would end up using $O(\log n)$ auxiliary locations ($O(\log n)$ nested recursive calls). There are many solutions to this problem. We use the n most significant bits of S (the most significant bit in each of the n entries in S) to store the $O(1)$ integers we need for any recursive call. That is possible because starting from the first recursive call the alphabet of the text is not Σ anymore but $\{1, 2, \dots, n/2\}$ and the size of the input becomes at most $n/2$. Therefore, the n most significant bits of S are untouched during all the recursive calls.

After the recursive call, we have n_α integers of $\lceil \log(n/2) \rceil$ bits each stored in the subarray $S' = S[n - n_\alpha + 1 \dots n]$. They are the suffix indices of the sequence T' stored in the subarray $S[1 \dots n_\alpha]$ and they are permuted according to the lexicographical order of the suffixes of T' .

Finally, to obtain the lexicographical order of the α -suffixes of T we proceed as follows. First, scanning T as we do to compute the α - β table, we recover the indices of the α -suffixes and we store them in T' (we do not need the data in T' anymore). Then, for any $1 \leq i \leq n_\alpha$, we set $S'[i] = T'[S'[i]]$.

3.2. Sorting the suffixes

In this section we show how to sort the suffixes of T provided the α -suffixes are already sorted. Let us assume that the suffix indices for the α -suffixes are stored in $S[n - n_\alpha + 1 \dots n]$. Before we start let us recall that any two adjacent sequences U and V , possibly with different sizes, can be exchanged in-place and in linear time with three sequence reversals, since $UV = (V^R U^R)^R$.

We have six phases.

3.2.1 First phase. With a process analogous to the one used to compute the α - β table (which we do not have at our disposal at this time), we scan T , recover the $n - n_\alpha$ suffix indices of the β -suffixes and store them in $S[1 \dots n - n_\alpha]$.

Then, we sort the pointers stored in $S[1 \dots n - n_\alpha]$ according to the first character of their respective β -suffixes (i.e. the sorting key for $S[i]$ is $T[S[i]]$). We use mergesort together with the in-place, linear time merging in [8].

3.2.2 Second phase. Let us denote $S[1 \dots n - n_\alpha]$ and $S[n - n_\alpha + 1 \dots n]$ by S_β and S_α , respectively. After the first phase the pointers in S_β are sorted according to the first character of their suffixes and so are the ones in S_α .

Scanning S_β , we find the *rightmost location* j_β such that the following hold:

- (i) $S_\beta[j_\beta]$ is the leftmost pointer in S_β whose β -suffix has $T[S_\beta[j_\beta]]$ as first character.
 - (ii) $n - n_\alpha - j_\beta + 1 \geq 2n/c$, that is, the number of pointers in the subarray $S_\beta[j_\beta \dots n - n_\alpha]$ is at least two times the number of entries of the α - β table.
- The reason for this choice will be clear at the end of this phase.

Then, we find the leftmost position j_α in S_α such that $T[S_\alpha[j_\alpha]] \geq T[S_\beta[j_\beta]]$ (a binary search). Let us consider the pointers in S as belonging to four sequences B', B'', A' and A'' , corresponding to the pointers in $S_\beta[1 \dots j_\beta - 1]$, $S_\beta[j_\beta \dots n - n_\alpha]$, $S_\alpha[1 \dots j_\alpha - 1]$ and $S_\alpha[j_\alpha \dots n_\alpha]$, respectively. We exchange the places of the sequences B'' and A' .

For the sake of presentation, let us assume that the choice of j_β is *balanced* that is condition (ii) holds for subarray $S_\beta[1 \dots j_\beta - 1]$ too. The other case requires only minor, mainly technical, modifications to the final phases and we will discuss it in the full version of this paper.

In the final step of this phase, we calculate the α - β table by scanning T and while we are doing so we encode the table in B'' in the following way: if the i -th entry of the table is 0 (1) we exchange positions of the pointers $B''[2i - 1]$ and $B''[2i]$ so that they are in ascending (descending) order. It is important to point out that in this case we mean the relative order of *the two pointers themselves*, seen as simple integer numbers, and not the relative order of the suffixes pointed by them. Clearly, any entry of the table can be decoded in $O(1)$ time.

This basic encoding technique is known as odd-even encoding ([7]). Its main advantage w.r.t. other, more sophisticated, encoding techniques is its extreme simplicity. Its main drawback is that it introduces an $\omega(1)$ overhead if used to encode/decode a table with entries of $\omega(1)$ bits. Since we will use it only to encode the α - β table, the overhead will not be a problem.

At the end of the second phase the pointers in the S are divided into the four contiguous sequences $B'A'B''A''$ and the α - β table is implicitly encoded in B'' . For the rest of the paper let us denote with S_L and S_R the subarrays $S[1 \dots |B'| + |A'|]$ and $S[|B'| + |A'| + 1 \dots n]$, respectively (i.e. the two subarrays of S containing all the pointers in $B'A'$ and in $B''A''$).

3.2.3 Third phase. We start by merging *stably* the pointers in B' and A' according to the first character of their suffixes. So, the sorting key for pointer $B'[i]$ is $T[B'[i]]$ and the relative order of pointers with equal keys is maintained. For this process we use the stable, in-place, linear time merging in [8].

After the merging, the pointers in S_L are contained into m contiguous sequences $C_1C_2 \dots C_m$ where m is the cardinality of the set $\{T[S_L[i]] \mid 1 \leq i \leq |S_L|\}$ and for any j and $p', p'' \in C_j$, $T[p'] = T[p'']$. Let us recall that A' contained the pointers to the $|A'|$ lexicographically smallest α -suffixes and they were already in lexicographical order. Therefore, since we merged B' and A' stably, we know that any sequence C_j is composed by two contiguous subsequences, C_j^β followed by C_j^α , such that (i) C_j^β contains only pointers to β -suffixes and (ii) C_j^α contains only pointers to α -suffixes and they are already in lexicographical order.

Now we need to gather some elements from some of the sequences C_i into a sequence E . We process each sequence C_i starting from C_1 . Initially E is void. Let us assume that we have reached the location c_j in S_L where the generic subsequence C_j begins and let us assume that at this time E is located right before C_j . We process C_j with the following steps.

1. We find the ending locations of C_j^β and C_j^α , using the α - β table encoded in sequence B'' of S_R (e.g. with a linear scan of C_j).
2. If C_j^β contains at least two pointers we proceed as follows.
 - (a) We “mark” the second location of C_j^β by setting $C_j^\beta[2] = S_L[1]$. We can employ $S_L[1]$ as a “special value” since, by construction, it has to point to an α -suffix and so it is not among the values affected by this process.
 - (b) We enlarge E by one element at its right end (thus including the first element of C_j^β and shrinking C_j by one element at its left end).
3. We move E (which may have been enlarged by one element in step 2) past C_j in the following way. If $|E| \leq |C_j|$, we simply exchange them. Otherwise, if $|E| > |C_j|$, we exchange the first $|C_j|$ elements of E with C_j , thus “rotating” E . (If we just exchanged E and C_j all the times, the cost of the whole scanning process would be $O(n^2)$).

After this first scanning process, E resides at the right end of S_L . Moreover, $|E|$ sequences among $C_1C_2 \dots C_m$ had their first pointer to a β -suffix moved at the right end of S_L and their second pointer to a β -suffix overwritten with the “special value” $S_L[1]$.

We proceed with a second scanning of the sequences $C_1C_2 \dots C_m$ from left to right. This time we remove the “special values” $S_L[1]$ in every location we find it (except the first location of S_L) by compacting the sequences toward left. With this process $|E|$ locations are freed right before sequence E . (Clearly,

any sequence C_i that before the two scanings had $|C_i| = 2$ and contained only pointers to β -suffixes has now disappeared.)

Finally, we create a “directory” in the last $2|E|$ locations of S_L (the second scanning has freed the $|E|$ locations before sequence E). Let us denote with G_L and D_L the subarrays with the first $|S_L| - 2|E|$ and the last $2|E|$ locations of S_L , respectively. We proceed with the following steps.

1. We use mergesort with the in-place, linear time binary merging in [8] to sort the elements of E , for any $1 \leq i \leq |E|$ we use $T[E[i]]$ as sorting key.
2. We “spread” E through D_L , that is we move $E[1]$ to $D_L[1]$, $E[2]$ to $D_L[3], \dots$, $E[i]$ to $D_L[2i - 1]$ etc.
3. For any $1 \leq i \leq |E|$. We do a binary search for the character $t_i = T[D_L[2i - 1]]$ in G_L using the character $T[G_L[l]]$ as key for the l -th entry of G_L . The search returns the leftmost position p_i in G_L where t_i could be inserted to maintain a sorted sequence. We set $D_L[2i] = p_i$.

3.2.4 Fourth phase. In this phase we finalize the sorting of the $|S_L|$ lexicographically smallest suffixes of T (and their pointers will be stored in S_L).

We start the phase by scanning G_L from left to right. Let us remark that, by construction, $G_L[1]$ contains a pointer to an α -suffix, the lexicographically smallest suffix of T . For the generic i -th location of G_L we perform two main steps. First we will give a fairly detailed and formal description of these two steps and then we will give a more intuitive explanation of them.

1. We do a binary search for the character $T[G_L[i] - 1]$ (i.e. the first character of the text-predecessor of the suffix pointed by $G_L[i]$) in the directory in D_L . The binary search is done on the odd locations of D_L (the ones with pointers to T) by considering the character $T[D_L[2l - 1]]$ as the key for the l -th odd location of D_L .
2. If the binary search succeeded, let j be the (odd) index in D_L such that $T[G_L[i] - 1] = T[D_L[j]]$. Let p_j be the inward pointer stored in $D_L[j + 1]$. We use p_j to place the outward pointer to the text-predecessor of the suffix pointed by $G_L[i]$ in a position in S_L (not only in G_L). We have three cases:
 - (a) If $T[G_L[i] - 1] = T[G_L[p_j]]$ and $G_L[p_j]$ is a β -suffix (we verify this using the α - β table encoded in sequence B'' of S_R), we set $G_L[p_j]$ to $G_L[i] - 1$ and we increment $D_L[j + 1]$ (i.e. p_j) by one.
 - (b) If $G_L[p_j]$ is an α -suffix or $T[G_L[i] - 1] \neq T[G_L[p_j]]$, we set $D_L[j]$ to $G_L[i] - 1$ and $D_L[j + 1]$ (i.e. p_j) to $|G_L| + 1$.
 - (c) If $p_j > |G_L|$, we set $D_L[j + 1] = G_L[i] - 1$.

And now for the intuitive explanation. As we anticipated in Section 2.2.2, the scanning process of the third main step (see Section 2.1) needs to maintain the inverse array of S in order to find the correct position for the text-predecessor of T_i in its bucket. Obviously we cannot encode that much information and so we develop an “on-the-fly” approach. The directory in D_L is used to find an inward (i.e. toward S itself and not T) pointer to a location in one of the C_k sequences (which represent the buckets in our algorithm). So the first step simply

uses the directory to find the inward pointer. Unfortunately the directory itself is stealing positions from the sequences C_k that sooner or later in the scanning will be needed to place some outward pointer (i.e. toward T). For any sequence C_k that has a “representative”, that is a pair $\langle p_{out}, p_{in} \rangle$, in the directory (the sequences without representatives are already in lexicographical order as they contain only one β -suffix) we have three cases to solve.

In the first case (corresponding to step 2a), there is still space in C_k , that is the two lexicographically largest β -suffixes belonging to C_k have not yet been considered by the scanning (these are the suffixes of C_k whose space in S is stolen by the directory in D_L). In this case the inward pointer we found in the directory guides us directly to the right position in C_k .

In the second case (corresponding to step 2b) the space in C_k is full and the suffix whose pointer we are trying to place is the lexicographically second largest β -suffix belonging to C_k . To solve the problem, we place its outward pointer in the first location of the pair $\langle p_{out}, p_{in} \rangle$ in D_L corresponding to the bucket C_k . This overwrites the outward pointer p_{out} that we use in the binary search but this is not a problem, as the old one and the new one belong to the same bucket. The only problem is that we need to be able to distinguish this second case from the third case, when the last β -suffix belonging to C_k will be considered in the scan. To do so we set p_{in} to a value that cannot be a valid inward pointer for the phase (e.g. $|G_L| + 1$).

In the third case (corresponding to step 2c) the space in C_k is full, and the pointer to the lexicographically second largest β -suffix belonging to C_k has been placed in the first location of the pair $\langle p_{out}, p_{in} \rangle$ in D_L corresponding to C_k . When the largest β -suffix of C_k is finally considered in the scanning, we are able to recognize this case since the value of the p_{in} is an invalid one ($|G_L| + 1$). Then, we set p_{in} to be the pointer to the largest β -suffix of C_k and, for what concerns C_k , we are done.

After the scanning process, the pointers in G_L are permuted according to the lexicographical order of their corresponding suffixes. The same holds for D_L . Moreover, for any $1 \leq j \leq |D_L|/2$, the suffixes pointed by $D_L[2j - 1]$ and $D_L[2j]$ (a) have the same first character, (b) are both β -suffixes and (c) they are the second largest and largest β -suffixes among the ones with their same first character, respectively. Knowing these three facts, we can merge the pointers in G_L and D_L in two steps.

1. We merge G_L and D_L *stably* with the in-place, linear time binary merging in [8] using $T[G_L[i]]$ and $T[D_L[j]]$ as merging keys for any i, j .
2. Since we merged G_L and D_L *stably*, after step 1 any pair of β -suffixes with the same first character whose two pointers were previously stored in D_L , now *follow* immediately (instead of preceding) the α -suffixes with their same first character (if any). A simple right to left scan of S_L (using the encoded α - β table) is sufficient to correct this problem in linear time.

3.2.5 Fifth phase. We start the phase by sorting the pointers in the sequence B'' of S_R according to the first character of their corresponding suffixes. The

pointers in B'' were already in this order before we perturbed them to encode the α - β table. Hence, we can just scan B'' and exchange any two pointers $B[i]$ and $B[i+1]$ such that $T[B[i]] > T[B[i+1]]$ (since the pointers in B'' are for β -suffixes, we do not need to care about their relative order when $T[B[i]] = T[B[i+1]]$).

After that, we process S_R in the same way we processed S_L in the third phase (Section 3.2.3). Since α - β table was used in that process, we need to encode it somewhere in S_L .

Unfortunately, we cannot use the plain odd-even encoding on the suffix indices in S_L because they are now in lexicographical order w.r.t. to the suffixes they point to. If we encoded each bit of the α - β table by exchanging positions of two *consecutive* pointers $S_L[i]$ and $S_L[i+1]$ (like we did with sequence B'' in the second phase, Section 3.2.2), then, after we are done using the encoded table, in order to recover the lexicographical order in S_L we would have to compare n/c pairs of consecutive suffixes. At the worst case that can require $O(n^2)$ time.

Instead, we exploit the fact that pointers in S_L are in lexicographical order w.r.t. to their suffixes in the following way. Let Tab'_L and Tab''_L be the subarrays $S_L[1 \dots n/c]$ and $S_L[n/c+1 \dots 2n/c]$, respectively. We encode a smaller α - β table with only $n/2c$ entries in Tab''_L as follows (Tab'_L will be used in the sixth phase): if the i -th entry of the table is 0 (1) we exchange positions of the pointers $\text{Tab}''_L[i]$ and $\text{Tab}''_L[|\text{Tab}''_L| - i + 1]$ so that they are in ascending (descending) order (as before, in this case we mean the relative order of *the two pointers themselves*, seen as simple integer numbers).

Since the pointers in Tab''_L were in lexicographical sorted order before the pair swapping, the recovering of the order of all the pairs of Tab''_L can be achieved in $O(n)$ time in the following way. Let us denote with p_1, p_2, \dots, p_t be the pointers in Tab''_L after the encoding (where $t = |\text{Tab}''_L|$). We start lexicographically comparing the suffixes T_{p_1} and T_{p_t} . When we find the mismatch, let it be at the h -th pair of characters, we exchange p_1 and p_t accordingly. Then we proceed by comparing T_{p_2} and $T_{p_{t-1}}$ but we do not start comparing them by their first characters. Since Tab''_L was in lexicographical sorted order, we know that the first $h-1$ characters must be equal. Hence, we start comparing T_{p_2} and $T_{p_{t-1}}$ starting from their h -th characters. We proceed in this fashion until the last pair has been dealt with. Clearly this process takes $O(n)$ time at the worst case.

3.2.6 Sixth phase. In this phase we finalize the sorting of the remaining $|S_R|$ lexicographically largest suffixes of T (and their pointers will be stored in S_R).

The final sorting process for S_R is the same we used in the fourth phase (Section 3.2.4) for S_L but with one difference: the scanning process does not start from $G_R[1]$ but from $S_L[1]$ again. We proceed as follow.

1. We scan the first n/c pointers of S_L (they correspond to subarray Tab'_L , which has not yet been used to encode the α - β table). The scanning process is the same one of the fourth phase, except that we use D_R as directory and Tab''_L for the α - β table.
2. After the first n/c pointers of S_L have been scanned, we move the α - β table encoding from Tab''_L to Tab'_L and recover the lexicographical order of the pointers in Tab''_L . Then, we continue the scanning of the rest of S_L .

3. Finally, the scanning process arrives to G_R . After G_R is sorted we merge G_R and D_R as we merged G_L and D_L in the fourth phase and we are done.

Let us point out one peculiar aspect of this last process. Since during the scan we use D_R as directory, the suffixes whose pointers reside in S_L *will not be moved again*. That is because D_R has been built using pointers to β -suffixes whose first characters are always different from the ones of suffixes with pointers in S_L . For this reason, during the second scan of S_L , any search in D_R for a suffix whose pointer is in S_L *will always fail* and nothing will be done to the pointer for that suffix (correctly, since S_L is already in order).

To summarize, we get Theorem 1.

4 Concluding Remarks

We have presented first known inplace algorithm for suffix sorting, i.e., an algorithm that uses $O(1)$ workspace beyond what is needed for the input T and output S . This algorithm is optimal for the general alphabet. Ultimately we would like to see simpler algorithms for this problem. Also, an interesting case is one in which the string elements are drawn from an integer alphabet. Then we can assume that each $T[i]$ is stored in $\lceil \log n \rceil$ bits and $O(1)$ time bit operations are allowed on such elements. In that case, known suffix tree algorithms solve suffix sorting in $O(n)$ time and use $O(n)$ workspace in addition to T and S [1]. We leave it open to design inplace algorithms for this case in $o(n \log n)$ time and ultimately, in even $O(n)$ time.

References

1. M. Farach. Optimal Suffix Tree Construction with Large Alphabets. FOCS 1997: 137-143.
2. P. Ferragina and G. Manzini. Engineering a lightweight suffix array construction algorithm. *Proc. ESA*, 2002.
3. D. Gusfield. Algorithms on strings, trees and sequences: Computer Science and Computational Biology. Cambridge Univ Press, 1997.
4. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. *Int. Colloquium on Automata, Languages and Programming*, 2719:943–955, 2003.
5. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*. *In press*.
6. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 200–210. SpringerVerlag, 2003.
7. J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.
8. Jeffrey Salowe and William Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, December 1987.