Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Discrete Algorithms 3 (2005) 126–142
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

Constructing suffix arrays in linear time ☆

Dong Kyue Kim [a], Jeong Seop Sim [b], Heejin Park [c], Kunsoo Park [d,*]

[a] *School of Electrical and Computer Engineering, Pusan National University, Busan 609-735, South Korea*
[b] *School of Computer Science & Engineering, Inha University, Incheon 402-751, South Korea*
[c] *College of Information and Communications, Hanyang University, Seoul 133-791, South Korea*
[d] *School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea*

Available online 11 September 2004

Abstract

The time complexity of suffix tree construction has been shown to be equivalent to that of sorting: $O(n)$ for a constant-size alphabet or an integer alphabet and $O(n \log n)$ for a general alphabet. However, previous algorithms for constructing suffix arrays have the time complexity of $O(n \log n)$ even for a constant-size alphabet.

In this paper we present a linear-time algorithm to construct suffix arrays for integer alphabets, which do not use suffix trees as intermediate data structures during its construction. Since the case of a constant-size alphabet can be subsumed in that of an integer alphabet, our result implies that the time complexity of directly constructing suffix arrays matches that of constructing suffix trees.

Keywords: Index data structures; Suffix arrays

☆ This work was supported by MOST Grant M1-0309-06-0003, Korea Research Foundation Grant (KRF-2003-03-D00343), and Pusan National University Research Grant.

* Corresponding author.
E-mail addresses: dkkim@islab.ce.pusan.ac.kr (D.K. Kim), jssim@inha.ac.kr (J.S. Sim), hjpark@hanyang.ac.kr (H. Park), kpark@theory.snu.ac.kr (K. Park).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2004.08.019

1. Introduction

The suffix tree due to McCreight [20] is a compacted trie of all the suffixes of a string T. It was designed as a simplified version of Weiner's position tree [26]. The suffix array due to Manber and Myers [19] and independently due to Gonnet et al. [11] is basically a sorted list of all the suffixes of a string T. There are also some other index data structures such as suffix cactus [15] and suffix automata [3].

When we consider the complexity of index data structures, there are three types of alphabets from which string T of length n is drawn: (i) a constant-size alphabet, (ii) an integer alphabet where symbols are integers in the range $[0, n^c]$ for a constant c, and (iii) a general alphabet in which the only operations on string T are symbol comparisons.

The time complexity[1] of suffix tree construction has been shown to be equivalent to that of sorting [7]. Suffix trees can be constructed in linear time for a constant-size alphabet due to McCreight [20] and Ukkonen [25] or for an integer alphabet due to Farach-Colton, Ferragina, and Muthukrishnan [6,7]. For a general alphabet, suffix tree construction has time bound of $\Theta(n \log n)$.

Despite simplicity of suffix arrays among index data structures, the construction time of suffix arrays has been larger than that of suffix trees. Two known algorithms for constructing suffix arrays by Manber and Myers [19] and Gusfield [12] have the time complexity of $O(n \log n)$ even for a constant-size alphabet. Of course, suffix arrays can be constructed by way of suffix trees in linear time, but it has been an open problem whether suffix arrays can be constructed in $o(n \log n)$ time without using suffix trees.

In this paper we solve the open problem in the affirmative and present a linear-time algorithm to construct suffix arrays for integer alphabets. Since the case of a constant-size alphabet can be subsumed in that of an integer alphabet, we will consider only the case of an integer alphabet in describing our result.

We take the recent divide-and-conquer approach for our algorithm [6–8,14,24], i.e., (i) construct recursively a suffix array SA_o for the set of odd positions, (ii) construct a suffix array SA_e for the set of even positions from SA_o, and (iii) merge SA_o and SA_e into the final suffix array SA_T. The hardest part of this approach is the merging step and our main contribution is a new merging algorithm.

Our new merging algorithm is quite different from Farach-Colton et al.'s [6,7] that are designed for suffix trees. Whereas [6,7] use a coupled depth-first search in the merging, ours uses equivalence relations defined on factors of T [5,12] (and thus it is more like a breadth-first search). Also, Farach-Colton et al.'s algorithm goes back and forth between suffix trees and suffix arrays during its construction, while ours uses only suffix arrays during its construction.

The rest of this paper is organized as follows. In Section 2, we introduce some notations and definitions. In Section 3, we present the algorithm for constructing suffix arrays in linear time. We conclude with some remarks in Section 4.

[1] Throughout this paper, the model we consider is the RAM (random-access machine) where each word has $\Theta(\log n)$ bits and every word operation is done in constant time.

2. Preliminaries

2.1. Definitions and notations

We first give some definitions and notations that will be used in our algorithm. Consider a string T of length n over an alphabet Σ. Let $T[i]$ denote the ith symbol of string T and $T[i, j]$ the substring starting at position i and ending at position j in T. We assume that $T[n]$ is a special symbol # which is lexicographically smaller than any other symbol in Σ. We denote by S_i, $1 \leqslant i \leqslant n$, the suffix of T that starts at position i. The prefix of length k of a string α is denoted by $\mathtt{pref}_k(\alpha)$. We denote by $\mathtt{lcp}(\alpha, \beta)$ the longest common prefix of two strings α and β and by $\mathtt{lcp}_i(\alpha, \beta)$ the longest common prefix of $\mathtt{pref}_i(\alpha)$ and $\mathtt{pref}_i(\beta)$. When string α is lexicographically smaller than string β, we denote it by $\alpha \prec \beta$.

We define the suffix array $SA_T = (A_T, L_T)$ of string T as a pair of arrays A_T and L_T [7].

- The *sort array* A_T is the lexicographically ordered list of all suffixes of T. That is, $A_T[i]$ stores j such that S_j is the ith lexicographically smallest suffix among all suffixes $S_1, S_2, \ldots, S_n$ of T. The number i will be called the *index* of suffix S_j, denoted by $\text{index}(j) = i$.
- The *lcp array* L_T stores the length of the longest common prefix of two adjacent suffixes in A_T, i.e., $L_T[i] = |\mathtt{lcp}(S_{A_T[i]}, S_{A_T[i+1]})|$ for $1 \leqslant i < n$. We set $L_T[0] = L_T[n] = -1$.

We define odd and even arrays of a string T. *Odd suffixes* are suffixes beginning at odd positions in T. For example, S_1, S_3, and S_5 are odd suffixes. The *odd array* $SA_o = (A_o, L_o)$ is the suffix array of all odd suffixes. That is, the sort array A_o of SA_o is the lexicographically ordered list of all odd suffixes, and the lcp array L_o has the length of the longest common prefix of adjacent odd suffixes in A_o. *Even suffixes* are suffixes beginning at even positions in T, e.g., S_2, S_4, and S_6. The *even array* $SA_e = (A_e, L_e)$ is the suffix array of all even suffixes.

For a subarray $A[x, y]$ of sort array A, we define $\mathsf{P}_A(x, y)$ as the longest common prefix of the suffixes $S_{A[x]}$, $S_{A[x+1]}$, …, $S_{A[y]}$. If $x = y$, $\mathsf{P}_A(x, x)$ is defined as the suffix $S_{A[x]}$ itself. Lemma 1 gives some properties of P_A in a subarray of sort array A.

Lemma 1 [16]. *Given a suffix array* (A, L) *and* $x < y$,

(a) $\mathsf{P}_A(x, y) = \mathtt{lcp}(S_{A[x]}, S_{A[y]})$.
(b) $|\mathsf{P}_A(x, y)|$ *is equal to the minimum value in* $L[x, y-1]$.

In order to find $|\mathsf{P}_A(x, y)|$ efficiently, we define the following problem.

Definition 1 [1,2,10]. Given an array A of size n whose elements are integers in the range $[0, n-1]$ and two indices a and b $(1 \leqslant a < b \leqslant n)$ in array A, the *range-minimum query* $\text{MIN}(A, a, b)$ is to find the smallest index $a \leqslant j \leqslant b$ such that $A[j] = \min_{a \leqslant i \leqslant b} A[i]$.

This MIN query can be answered in constant time using a succinct data structure due to Sadakane [23]. This data structure requires $O(n)$ bits space and requires $O(n)$ time for construction. By a MIN query, we get the following lemma.

Lemma 2. *Given a suffix array* (A, L) *and* $x < y$, $\text{MIN}(L, x, y-1)$ *can be computed in constant time.*

An advantage of suffix trees is that *suffix links* are defined on suffix trees. When $\texttt{lcp}(S_i, S_j) = a\alpha$ for $a \in \Sigma$ and $\alpha \in \Sigma^*$, $\texttt{lcp}(S_{i+1}, S_{j+1}) = \alpha$. Suffix links enable us to find α from S_i and S_j. In suffix arrays this can be done by finding $\texttt{lcp}(S_{i+1}, S_{j+1})$ using a MIN query. This method will be used in Section 3.4 with the following lemma.

Lemma 3. *Let* i *and* j $(i < j)$ *be two positions in string* T. *If* $T[i]$ *and* $T[j]$ *match,* $|\texttt{lcp}(S_i, S_j)| = |\texttt{lcp}(S_{i+1}, S_{j+1})| + 1$; *otherwise,* $|\texttt{lcp}(S_i, S_j)| = 0$.

2.2. Equivalence classes

In this section, we will define equivalence relation E_l on sort arrays such as A_T, A_o, and A_e, and explain the relationship between equivalence classes of E_l on a sort array and subarrays of the sort array.

Let A be a sort array of size m and L be the corresponding lcp array. Equivalence relation E_l $(l \geqslant 0)$ on A is:

$$E_l = \left\{(i, j) \mid \texttt{pref}_l(S_{A[i]}) = \texttt{pref}_l(S_{A[j]})\right\}.$$

That is, i and j are in the same equivalence class of E_l on A if and only if two suffixes $S_{A[i]}$ and $S_{A[j]}$ have a common prefix of length l.

We describe the relationship between equivalence classes of E_l on A and subarrays of A. Since the integers in A are sorted in the lexicographical order of the corresponding suffixes, we get the following fact from the definition of E_l.

Fact 1. *Subarray* $A[p, q]$, $1 \leqslant p \leqslant q \leqslant m$, *is an equivalence class of* E_l, $0 \leqslant l \leqslant n$, *on* A *if and only if* $L[p-1] < l$, $L[q] < l$, *and* $L[i] \geqslant l$ *for all* $p \leqslant i < q$.

Example 1. Consider $A[2, 7]$ in Fig. 1. $A[2, 7]$ is an equivalence class of E_2 since $L[1] = 0$, $L[7] = 1$, and $L[i] \geqslant 2$ for all $2 \leqslant i < 7$.

We now describe how an equivalence class of E_l on A is partitioned into equivalence classes of E_{l+1}. Let $A[p, q]$ be an equivalence class of E_l. By Fact 1, $L[i] \geqslant l$ for all $p \leqslant i < q$. Let $p \leqslant i_1 < i_2 < \cdots < i_r < q$ denote all the indices such that $L[i_1] = L[i_2] = \cdots = L[i_r] = l$. Since $L[i] \geqslant l+1$ for $i \notin \{i_1, i_2, \ldots, i_r\}$ and $p \leqslant i < q$, $A[p, i_1]$, $A[i_1 + 1, i_2], \ldots, A[i_r + 1, q]$ are equivalence classes of E_{l+1}. We can find $i_1, i_2, \ldots, i_r$ in $O(r)$ time by Lemma 2 and we get the following lemma.

Lemma 4. *An equivalence class of* E_l *can be partitioned into equivalence classes of* E_{l+1} *in* $O(r)$ *time, where* r *is the number of the partitioned equivalence classes of* E_{l+1}.

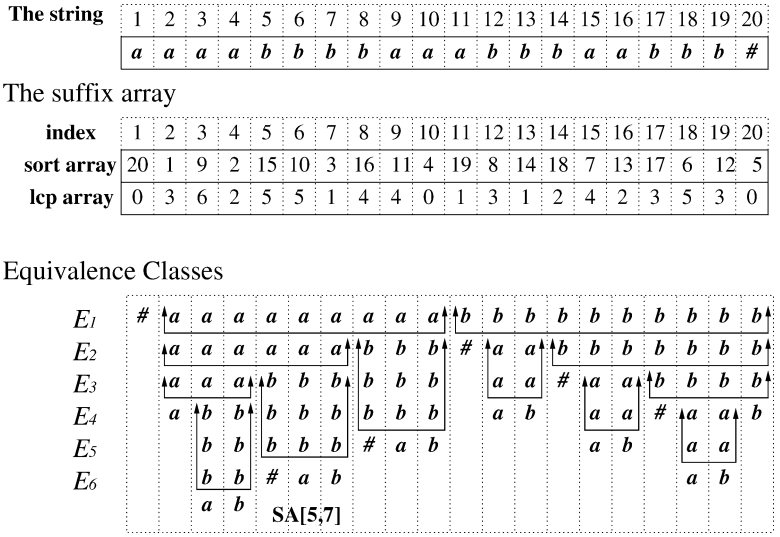


Fig. 1. Equivalence classes and subarrays of a sort array.

Example 2. In Fig. 1, equivalence class $A[2,7]$ of E_2 is partitioned into two equivalence classes, $A[2,4]$ and $A[5,7]$, of E_3.

An equivalence class of E_l can be an equivalence class of E_k for $k \neq l$. For example, $A[5,7]$ is an equivalence class of E_3, E_4, and E_5. In general, we have the following fact.

Fact 2. *A subarray $A[p,q]$ is an equivalence class of E_i for $a \leqslant i \leqslant b$ if and only if* $\max\{L[p-1], L[q]\} = a-1$ *and* $b = |\mathsf{P}_A(p,q)| (= \min_{p \leqslant i < q} L[i])$.

The integers a and b are called the *start stage* and *end stage* of equivalence class $A[p,q]$, respectively.

3. Linear-time construction

We present a linear-time algorithm for constructing suffix arrays for integer alphabets. Our construction algorithm follows the divide-and-conquer approach used in [6–8,14,24], and it consists of the following three steps.

1. Construct the odd array SA_o recursively. Preprocess L_o for range-minimum queries.
2. Construct the even tree SA_e from SA_o. Preprocess L_e for range-minimum queries.
3. Merge SA_o and SA_e to get the final suffix array SA_T.

The first two steps are essentially the same as those in [6–8] and our main contribution is a new merging algorithm in step 3.

3.1. Constructing odd array

Construction of the odd array SA_o is based on recursion and it consists of the following three steps. It takes linear time except for recursion.

1. Encode the given string T into a half-sized string T': We encode T into T' by replacing each pair of adjacent symbols $(T[2i-1], T[2i])$, $1 \leqslant i \leqslant n/2$, with a new symbol. How to encode T into T' is as follows.
   - Sort the pairs of adjacent symbols $(T[2i-1], T[2i])$ lexicographically and then remove duplicates: We use radix-sort to sort the pairs and we perform a scan on the sorted pairs to remove duplicates. Both the radix-sort and the scan take $O(n)$ time.
   - Map the ith lexicographically smallest pair of adjacent symbols into integer i: The integer i is in the range $[1, n/2]$ because the number of pairs is at most $n/2$.
   - Replace $(T[2i-1], T[2i])$ with the integer it is mapped into.

   Fig. 2 shows how to encode $T = aaaabbbbaaabbbaabbb\#$ of length 20. After we sort the pairs $(T[1], T[2]), (T[3], T[4]), \ldots, (T[19], T[20])$ and remove duplicates, we are left with 4 pairs which are aa, ab, $b\#$, and bb. We map aa, ab, $b\#$, and bb into integers of 1 to 4, respectively. Then, we get $T' = 1144124143$ of length 10.
2. Recursively construct the suffix array $SA_{T'}$ of T'.
3. Compute SA_o from $SA_{T'}$:
   - Sort array A_o: Since the ith suffix of T' corresponds to the $(2i-1)$st suffix of T, we get $A_o[k]$ by computing $2A_{T'}[k]-1$ for all k. For example, $A_o[2] = 2A_{T'}[2]-1 = 9$ in Fig. 2.

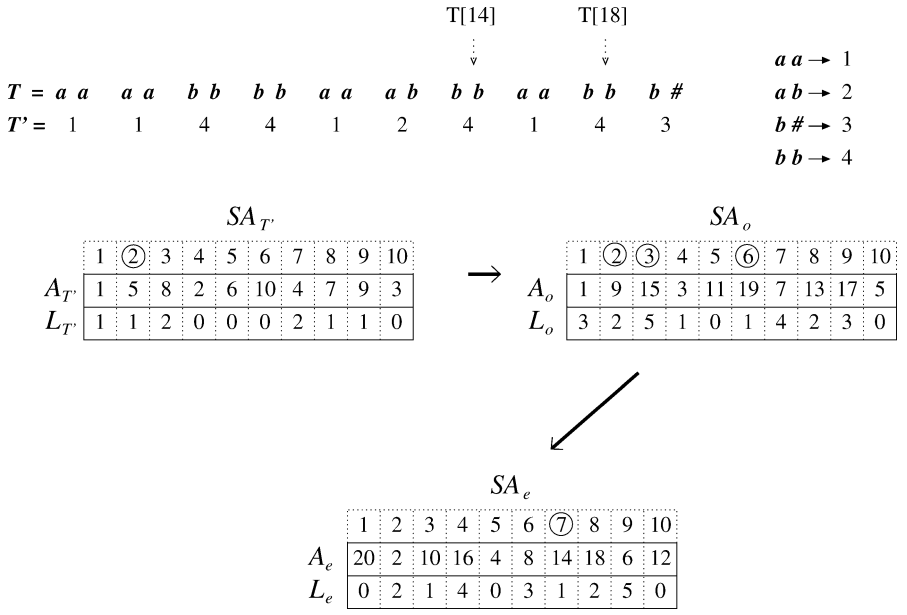


Fig. 2. Examples for constructing the odd array and the even array.

- Lcp array L_o: Since two symbols in T are encoded into one symbol in T', $L_o[i]$ is either $2L_{T'}[i]$ or $2L_{T'}[i]+1$. In Fig. 2, $L_o[2]=2$ when $L_{T'}[2]=1$, and $L_o[1]=3$ when $L_{T'}[1]=1$ because the first suffix and the fifth suffix of T' are 1144124143 and 124143, respectively, and 1 and 2 in T' are encodings of aa and ab, respectively.

3.2. Constructing even array

The even array SA_e is constructed from SA_o in linear time as follows.

- Sort array A_e: An even suffix is one symbol followed by an odd suffix. For example, S_8 of T is $T[8]$ followed by S_9 of T. We make tuples for even suffixes: the first element of a tuple is $T[2i]$ and the second element is suffix S_{2i+1} of T. First, we sort the tuples by the second elements (this result is given in A_o). Then we stably sort the tuples by the first elements and we get A_e.
- Lcp array L_e: Consider two even suffixes S_{2i} and S_{2j}. By Lemma 3, if $T[2i]$ and $T[2j]$ match, $|\mathtt{lcp}(S_{2i}, S_{2j})| = |\mathtt{lcp}(S_{2i+1}, S_{2j+1})| + 1$; otherwise, $\mathtt{lcp}(S_{2i}, S_{2j}) = 0$. We can get $|\mathtt{lcp}(S_{2i+1}, S_{2j+1})|$ from the odd array SA_o in constant time as follows. Let $x = \text{index}_o(2i+1)$ and $y = \text{index}_o(2j+1)$ in SA_o. By Lemma 1, $|\mathtt{lcp}(S_{2i+1}, S_{2j+1})| = |\mathsf{P}_{A_o}(x, y)|$, which is computed by a $\text{MIN}(L_o, x, y-1)$ query. For example, consider $L_e[7]$ in Fig. 2. Since $T[14] = T[18]$, $|\mathtt{lcp}(S_{14}, S_{18})| = |\mathtt{lcp}(S_{15}, S_{19})| + 1$. Since $\text{index}_o(15) = 3$ and $\text{index}_o(19) = 6$, $|\mathtt{lcp}(S_{15}, S_{19})| = \text{MIN}(L_o, 3, 5) = 0$. Thus, $L_e[7] = 1$.

3.3. Merging odd and even arrays

We will show how to obtain suffix array $SA_T = (A_T, L_T)$ from SA_o and SA_e in $O(n)$ time. The basic idea of this algorithm is that we first merge the odd and even suffixes coarsely using their prefixes of length 1 and then merge them more finely using their longer prefixes. This idea is illustrated in Fig. 3 which shows an example of merging A_o and A_e for $T = aaaabbbbaaabbbaabbb\#$. First, we merge the odd and even suffixes using their prefixes of length 1, which are #, a, and b. Then, the merging is based on the prefixes of length 2, length 3, etc. However, a direct reflection of this idea leads to an algorithm taking $O(n^2)$ time. Thus, we modify this idea so that we can merge A_o and A_e in $O(n)$ time.

We first introduce some abbreviations and notions related to equivalence classes. We will refer to equivalence classes of E_i as 'i-equivalence classes'. We will refer to equivalence classes on A_T as 'target equivalence classes', equivalence classes on A_o as 'odd equivalence classes', and equivalence classes on A_e as 'even equivalence classes'. Thus, an equivalence class of E_i on A_o is referred to as an odd i-equivalence class. *The prefix of an i-equivalence class* is defined as the common prefix of length i of the suffixes in the i-equivalence class. Now, we introduce the notions of *i-coupled* and *i-uncoupled* defined on odd and even i-equivalence classes. For brevity, we define them only on odd i-equivalence classes. (They are defined on even i-equivalence classes similarly.) An odd i-equivalence class X is i-coupled if there exists an even i-equivalence class Y such that the prefix of Y is the same as the prefix of X. We say X is i-coupled with Y. Otherwise (if X is not i-coupled with any even i-equivalence classes), X is i-uncoupled. If X is i-uncoupled, no

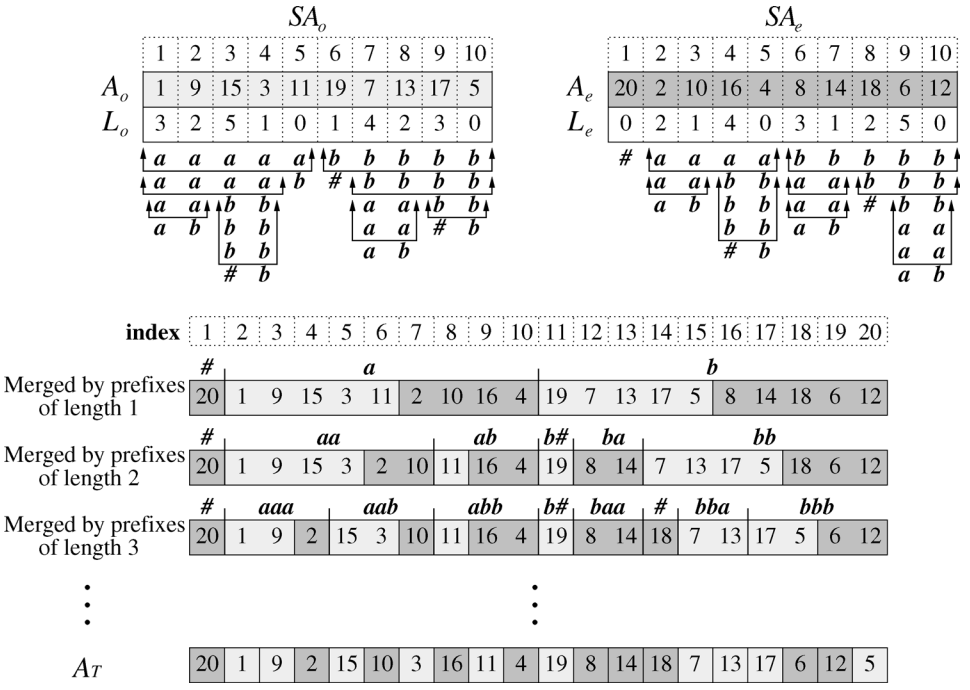


Fig. 3. An example of merging A_o and A_e for $T = aaaabbbbaaabbbaabbb\#$. Integers in lightly shaded boxes indicate odd suffixes and integers in darkly shaded boxes indicate even suffixes.

even suffixes have the same prefix as the prefix of X and thus the suffixes in X form a target i-equivalence class. Otherwise (if X is i-coupled with Y), the suffixes in X and Y form a target i-equivalence class. This fact is elaborated by the following lemma.

Lemma 5. *The suffixes in i-equivalence classes $A_o[w, x]$ and $A_e[y, z]$ that are i-coupled with each other form a target i-equivalence class $A_T[w + y - 1, x + z]$.*

Proof. Since $A_o[w, x]$ and $A_e[y, z]$ are i-coupled with each other, $\texttt{pref}_i(S_{A_o[a]})$, $w \leqslant a \leqslant x$, is lexicographically larger than $\texttt{pref}_i(S_{A_e[b]})$, $1 \leqslant b \leqslant y - 1$, and smaller than $\texttt{pref}_i(S_{A_e[c]})$, $z + 1 \leqslant c \leqslant n/2$. Similarly, $\texttt{pref}_i(S_{A_e[a]})$, $y \leqslant a \leqslant z$, is lexicographically larger than $\texttt{pref}_i(S_{A_o[b]})$, $1 \leqslant b \leqslant w - 1$, and smaller than $\texttt{pref}_i(S_{A_o[c]})$, $x + 1 \leqslant c \leqslant n/2$. Hence, all the suffixes in $A_o[w, x]$ and $A_e[y, z]$ form a target i-equivalence class $A_T[w + y - 1, x + z]$. □

We now explain the notion of a *coupled pair*, which is central in our merging algorithm. Consider an equivalence class $A_o[w, x]$ whose start stage is l_o and end stage is k_o and an equivalence class $A_e[y, z]$ whose start stage is l_e and end stage is k_e such that $l = \max\{l_o, l_e\} \leqslant k = \min\{k_o, k_e\}$ and $A_o[w, x]$ and $A_e[y, z]$ are l-coupled with each other. We call $C = \langle A_o[w, x], A_e[y, z]\rangle$ a *coupled pair*. Since $A_o[w, x]$ and $A_e[y, z]$ is l-coupled with each other, the suffixes in $A_o[w, x]$ and $A_e[y, z]$ form a target equiva-

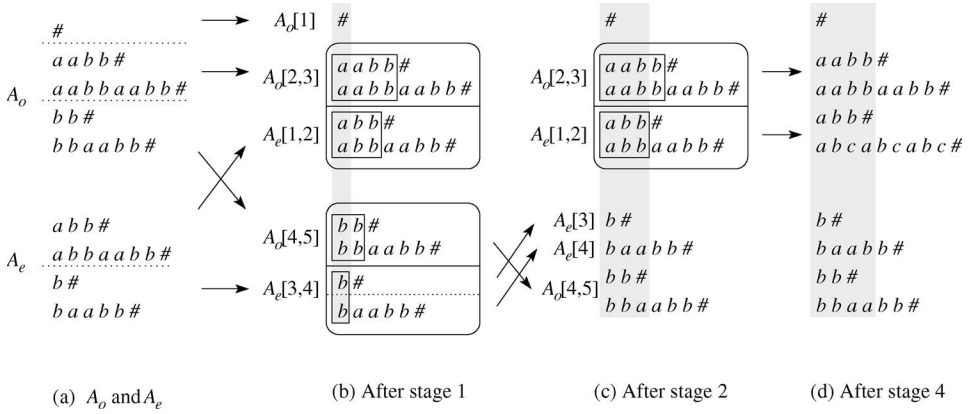


Fig. 4. An example of our merging algorithm.

lence class $A_T[w+y-1, x+z]$ by Lemma 5. We define the start stage and the end stage of coupled pair C as the start stage and the end stage of the target equivalence class $A_T[w+y-1, x+z]$. Since l is the smallest integer such that $A_o[w, x]$ is l-coupled with $A_e[y, z]$, l is the start stage of $A_T[w+y-1, x+z]$ and thus l is the start stage of C. Now we are interested in the end stage of C. Since one of $A_o[w, x]$ and $A_e[y, z]$ will be partitioned into several $(k+1)$-equivalence classes, $A_T[w+y-1, x+z]$ cannot be a target $(k+1)$-equivalence class. In the sense that the end stage of C cannot be larger than k, the value k is called the *limit stage* of C. The actual end stage of C is the value of $|\mathtt{lcp}(\mathsf{P}_{A_o}(w, x), \mathsf{P}_{A_e}(y, z))|$, and it is in the range of $[l, k]$.

Example 3. Consider a coupled pair $\langle A_o[2, 3], A_e[1, 2]\rangle$ in Fig. 4(b). Its start stage is 1 because the start stages of $A_o[2, 3]$ and $A_e[1, 2]$ are all 1 and $A_o[2, 3]$ and $A_e[1, 2]$ are 1-coupled with each other. Its limit stage is 3 because the end stages of $A_o[2, 3]$ and $A_e[1, 2]$ are 3 and 4, respectively. Its end stage is 1 because $|\mathtt{lcp}(\mathsf{P}_{A_o}(2, 3), \mathsf{P}_{A_e}(1, 2))| = 1$.

In our algorithm, we maintain coupled pairs in multiple queues $Q[k]$ for $0 \leqslant k < n$. Each queue $Q[k]$ contains coupled pairs whose limit stage is k.

Our merging algorithm consists of at most n stages, and it maintains the following invariants.

Invariant. At the end of stage $s \geqslant 0$, the odd array A_o and the even array A_e are partitioned into i-equivalence classes, $0 \leqslant i \leqslant s$, such that each i-equivalence class is either i-coupled or i-uncoupled, where i-coupled equivalence classes constitute coupled pairs whose start stages are at most s and limit stages are at least s and the suffixes in i-uncoupled equivalence classes are stored in correct places of A_T.

Since the limit stage of a coupled pair is at most $n-1$, this invariant guarantees that all equivalence classes are i-uncoupled for some $0 \leqslant i \leqslant n$ and the odd and even suffixes are stored in correct places of A_T after stage n. We will call an equivalence class whose suffixes are stored in correct places of A_T a *processed* equivalence class.

We describe the outline of stages. Initially, a coupled pair $\langle A_o[1, n/2], A_e[1, n/2]\rangle$ is stored in $Q[0]$. At stage $1 \leqslant s \leqslant n$, we do the following for each coupled pair $C = \langle A_o[w, x], A_e[y, z]\rangle$ stored in $Q[s-1]$. We first compute the end stage of C by solving the following coupled pair lcp problem. In the next section we will show how to solve the coupled pair lcp problem in O(1) time.

Definition 2 (*The coupled pair lcp problem*). Given a coupled pair $C = \langle A_o[w, x], A_e[y, z]\rangle$ whose limit stage is $s-1$, compute the end stage of C. Furthermore, if the end stage of C is less than $s-1$, determine whether $\mathsf{P}_{A_o}(w, x) \prec \mathsf{P}_{A_e}(y, z)$ or $\mathsf{P}_{A_o}(w, x) \succ \mathsf{P}_{A_e}(y, z)$.

After solving the coupled pair lcp problem for C, we have two cases depending on whether the end stage of C is $s-1$ or not.

- *Case* 1. If the end stage of C is $s-1$, $A_o[w, x]$ is $(s-1)$-coupled with $A_e[y, z]$. We first partition $A_o[w, x]$ and $A_e[y, z]$ into s-equivalence classes. Every partitioned s-equivalence class will be either s-coupled or s-uncoupled. The s-coupled equivalence classes constitute coupled pairs whose limit stages are at least s, and thus we store each coupled pair in $Q[k]$ for $s \leqslant k \leqslant n-1$, where k is the limit stage of the coupled pair. For the s-uncoupled equivalence classes, we store the suffixes in them into A_T.
- *Case* 2. If the end stage of C is smaller than $s-1$, $A_o[w, x]$ and $A_e[y, z]$ are $(s-1)$-uncoupled. We store the suffixes in them into A_T.

From the fact that every coupled pair generated in stage s has the limit stage at least s and every s-uncoupled equivalence class becomes a processed equivalence class in stage s, it is not difficult to see that the invariant is satisfied after stage s.

Example 4. Fig. 4 shows an example of merging the odd and the even arrays for $T = aabbaabb\#$. The odd and the even arrays are shown in Fig 4(a). Initially a coupled pair $\langle A_o[1, 5], A_e[1, 4]\rangle$ is stored in $Q[0]$. In stage 1, we perform operations on coupled pair $\langle A_o[1, 5], A_e[1, 4]\rangle$ stored in $Q[0]$. Since the limit stage of the coupled pair is 0, we partition $A_o[1, 5]$ and $A_e[1, 4]$ into 1-equivalence classes, $A_o[1]$, $A_o[2, 3]$, $A_o[4, 5]$, $A_e[1, 2]$, and $A_e[3, 4]$. Among them, $A_o[1]$ is 1-uncoupled and $A_o[2, 3]$ and $A_e[1, 2]$ are 1-coupled with each other and $A_o[4, 5]$ and $A_e[3, 4]$ are 1-coupled with each other. Let $C = \langle A_o[2, 3], A_e[1, 2]\rangle$ and $D = \langle A_o[4, 5], A_e[3, 4]\rangle$. We store the suffix in $A_o[1]$ into $A_T[1]$, C in $Q[3]$, and D in $Q[1]$ because the limit stages of C and D are 3 and 1, respectively. In stage 2, we perform operations on coupled pair D stored in $Q[1]$. Since the end stage of D is 1, we partition $A_o[4, 5]$ and $A_e[3, 4]$ into 2-equivalence classes $A_o[4, 5]$, $A_e[3]$, $A_e[4]$. They are all 2-uncoupled and thus we store the suffixes in them into A_T. In stage 4, we perform operations on coupled pair C stored in $Q[3]$. Since the end stage of C is 1 and $A_o[2, 3]$ and $A_e[1, 2]$ are 2-uncoupled, we store the suffixes in them into A_T.

Before describing stages in detail, we give an outline of computing L_T and introduce functions $\mathtt{fin}_o$ and $\mathtt{fin}_e$ and arrays $\mathtt{ptr}_o$ and $\mathtt{ptr}_e$. The invariant for computing L_T is as follows.

Invariant for L_T. At the end of stage s, $L_T[i]$ for $1 \leqslant i \leqslant n-1$ is computed if and only if $A_T[i]$ is either a suffix of the target equivalence class corresponding to a processed equivalence class, or the last suffix of the target equivalence class corresponding to a coupled pair stored in $Q[k]$ for some $s \leqslant k \leqslant n-1$.

Since all partitioned equivalence classes are processed after stage n, all $L_T[i]$'s for $1 \leqslant i \leqslant n-1$ are computed at the end of stage n. To satisfy this invariant, we do the following in stage s. Let A be a target equivalence class corresponding to an uncoupled equivalence class generated in stage s and B be a target equivalence class corresponding to a coupled pair generated in stage s. We compute $L_T[i]$ for i's such that $A_T[i]$ is either a suffix of A or the last suffix of B.

We introduce functions $\texttt{fin}_o$ and $\texttt{fin}_e$ and arrays $\texttt{ptr}_o$ and $\texttt{ptr}_e$, that are required to solve the couple-pair lcp problem in O(1) time. Since $\texttt{ptr}_e$ and $\texttt{fin}_e$ are similar to $\texttt{ptr}_o$ and $\texttt{fin}_o$, we explain $\texttt{ptr}_o$ and $\texttt{fin}_o$ only. At the end of stage s, the values stored in $\texttt{ptr}_o$ and $\texttt{fin}_o$ are as follows.

1. $\texttt{fin}_o[i]$ for $1 \leqslant i \leqslant n/2$ is defined if $A_o[i]$ is an entry of a processed equivalence class and it is the index of A_T where the suffix in $A_o[i]$ is stored.
2. $\texttt{ptr}_o[i]$ for $1 \leqslant i \leqslant n/2$ is defined if $A_o[i]$ is either the last entry of a coupled equivalence class or an entry of a processed equivalence class.
   - If $A_o[i]$ is the last entry of an equivalence class $A_o[a,b]$ (i.e., $i = b$) coupled with $A_e[c,d]$ (i.e., $\langle A_o[a,b], A_e[c,d]\rangle$ is stored in $Q[k]$ for some $s \leqslant k \leqslant n-1$), $\texttt{ptr}_o[b]$ stores d.
   - If $A_o[i]$ is an entry of a processed equivalence class $A_o[a,b]$:
     - If $A_o[i]$ is not the last entry of $A_o[a,b]$ (i.e., $a \leqslant i < b$), $\texttt{ptr}_o[i]$ stores b.
     - Otherwise, $\texttt{ptr}_o[b]$ stores β such that $A_e[\beta]$ is the last entry of a partitioned equivalence class $A_e[\alpha,\beta]$ and β satisfies $|\texttt{lcp}(S_{A_o[b]}, S_{A_e[\beta]})| \geqslant |\texttt{lcp}(S_{A_o[b]}, S_{A_e[\delta]})|$ for any other $1 \leqslant \delta \leqslant n/2$. In addition, $|\texttt{lcp}(S_{A_o[b]}, S_{A_e[\beta]})|$ is stored in $L_T[\texttt{fin}_o[b]]$ if $\texttt{fin}_o[b] < \texttt{fin}_e[\beta]$, and in $L_T[\texttt{fin}_e[\beta]]$ otherwise.

We describe stages in detail. Initially, we store a coupled pair $\langle A_o[1,n/2], A_e[1,n/2]\rangle$ in $Q[0]$ and initialize $\texttt{ptr}_o[n/2] = n/2$, $\texttt{ptr}_e[n/2] = n/2$, $L_T[0] = L_T[n] = -1$. In stage s, $1 \leqslant s \leqslant n$, we do nothing if $Q[s-1]$ is empty. Otherwise, for every coupled pair $C = \langle A_o[w,x], A_e[y,z]\rangle$ stored in $Q[s-1]$, we compute the end stage of C by solving the coupled pair lcp problem. We have two cases depending on whether the end stage of C is $s-1$ or not.

Case 1. If the end stage of C is $s-1$, $A_o[w,x]$ is $(s-1)$-coupled with $A_e[y,z]$. We first partition $A_o[w,x]$ and $A_e[y,z]$ into s-equivalence classes. Let C_o and C_e denote the set of equivalence classes into which $A_o[w,x]$ and $A_e[y,z]$ are partitioned, respectively. We denote odd s-equivalence classes in C_o by $A_o[w_i,x_i]$, $1 \leqslant i \leqslant r_1$, such that $\mathsf{P}_{A_o}(w_i,x_i) \prec \mathsf{P}_{A_o}(w_{i+1},x_{i+1})$ and even s-equivalence classes in C_e by $A_e[y_i,z_i]$, $1 \leqslant i \leqslant r_2$, such that $\mathsf{P}_{A_e}(y_i,z_i) \prec \mathsf{P}_{A_e}(y_{i+1},z_{i+1})$. Partitioning $A_o[w,x]$ and $A_e[y,z]$ into s-equivalence classes takes $O(r_1+r_2)$ time by Lemma 4.

```
Procedure MERGE(C_o, C_e)
 1: i ← 1 and j ← 1
 2: while i ⩽ r_1 or j ⩽ r_2 do
 3:    a_i ← the sth symbol of P_{A_o}(w_i, x_i)
 4:    b_j ← the sth symbol of P_{A_e}(y_j, z_j)
 5:    if a_i = b_j then          // A_o[w_i, x_i] and A_e[y_j, z_j] are s-coupled.
 6:       k ← min{|P_{A_o}(w_i, x_i)|, |P_{A_e}(y_j, z_j)|}
 7:       store ⟨A_o[w_i, x_i], A_e[y_j, z_j]⟩ into Q[k]
 8:       if i + j < r_1 + r_2 then L_T[x_i + z_j] ← s − 1 fi
 9:       if i < r_1 then ptr_o[x_i] ← z_j fi
10:       if j < r_2 then ptr_e[z_j] ← x_i fi
11:       i ← i + 1 and j ← j + 1
12:    else if a_i ≺ b_j then          // A_o[w_i, x_i] is s-uncoupled.
13:       fin_o[k] ← k + y_j − 1 for w_i ⩽ k ⩽ x_i
14:       Store A_o[k] into A_T[fin_o[k]] for w_i ⩽ k ⩽ x_i
15:       Store L_o[k] into L_T[fin_o[k]] for w_i ⩽ k < x_i
16:       if i + j < r_1 + r_2 then L_T[x_i + y_j − 1] ← s − 1 fi
17:       ptr_o[k] ← x_j for w_i ⩽ k < x_i
18:       if i < r_1 then ptr_o[x_i] ← z_j
19:       i ← i + 1
20:    else                          // A_e[y_j, z_j] is s-uncoupled.
21:       fin_e[k] ← k + w_i − 1 for y_j ⩽ k ⩽ z_j
22:       Store A_e[k] into A_T[fin_e[k]] for y_i ⩽ k ⩽ z_i
23:       Store L_e[k] into L_T[fin_e[k]] for y_i ⩽ k < z_i
24:       if i + j < r_1 + r_2 then L_T[w_i + z_j − 1] ← s − 1 fi
25:       ptr_e[k] ← z_j for y_j ⩽ k < z_j
26:       if j < r_2 then ptr_e[z_j] ← x_i fi
27:       j ← j + 1
28:    fi
29: od
end
```

Fig. 5. Procedure MERGE. We assume $a_{r_1+1} = b_{r_2+1} = \$$ where $\$ \succ a$ for any $a \in \Sigma$, $w_{r_1+1} = x_{r_1} + 1$, $x_{r_1+1} = x_{r_1}$, $y_{r_2+1} = z_{r_2} + 1$, and $z_{r_2+1} = z_{r_2}$.

We merge the partitioned s-equivalence classes in C_e and C_o according to the lexicographical order of their prefixes of length s. Since all the partitioned s-equivalence classes in C_e and C_o have the same prefix of length $s-1$, we merge the equivalence classes using only the sth symbols of their prefixes. Thus, merging the equivalence classes is basically the same as merging two sorted lists of integers. Procedure MERGE in Fig. 5 shows the details of merging the equivalence classes in C_o and C_e. If an s-equivalence class $A_o[w_i, x_i]$ in C_o is coupled with an s-equivalence class $A_e[y_j, z_j]$ in C_e, we store the coupled pair $\langle A_o[w_i, x_i], A_e[y_j, z_j]\rangle$ into Q (lines 5–11 of MERGE). Otherwise (if $A_o[w_i, x_i]$ or $A_e[y_j, z_j]$ is s-uncoupled), we store the suffixes in it into the appropriate places in A_T (lines 12–27 of MERGE).

For each equivalence class $A_o[w_i, x_i]$, we show that $\texttt{fin}_o[\alpha]$ and $\texttt{ptr}_o[\alpha]$ for $w_i \leqslant \alpha \leqslant x_i$ are computed correctly. (Similarly for $A_e[y_j, z_j]$.) We only show that $\texttt{ptr}_o[x_i]$ stores a correct value when $A_o[w_i, x_i]$ is s-uncoupled (so processed) because setting other values is trivial. From the description of procedure MERGE(C_o, C_e), $\texttt{ptr}_o[x_i]$ is z_j for some $1 \leqslant j \leqslant r_2$.

Claim. *z_j satisfies $|\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})| \geqslant |\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})|$ for $1 \leqslant \alpha \leqslant n/2$ and $|\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})|$ is stored in $L_T[\mathtt{fin}_o[x_i]]$ if $\mathtt{fin}_o[x_i] < \mathtt{fin}_e[z_j]$ and in $L_T[\mathtt{fin}_e[z_j]]$ otherwise.*

Proof of Claim. Let $lcp = |\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})|$. Since $A_o[w, x]$ and $A_o[y, z]$ is $(s-1)$-coupled and $A_o[w_i, x_i]$ is s-uncoupled, $lcp = s - 1$. Since $A_o[w_i, x_i]$ is s-uncoupled, $|\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})| \leqslant s - 1$ for $1 \leqslant \alpha \leqslant n/2$. Hence, z_j satisfies $lcp \geqslant |\mathtt{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})|$ for $1 \leqslant \alpha \leqslant n/2$. If $\mathtt{fin}_o[x_i] < \mathtt{fin}_e[z_j]$, $\mathtt{fin}_o[x_i] < x + z$ and thus $L_T[\mathtt{fin}_o[x_i]]$ is set to $s - 1$, which is *lcp*. Otherwise, $\mathtt{fin}_e[z_j] < x + z$ and thus $L_T[\mathtt{fin}_e[z_j]]$ is set to $s - 1$. □

Case 2. If the end stage of C is smaller than $s - 1$, $A_o[w, x]$ and $A_e[y, z]$ are $(s-1)$-uncoupled. Assume without loss of generality that $\mathtt{P}_{A_o}(w, x) \prec \mathtt{P}_{A_e}(y, z)$. We first store the suffixes in $A_o[w, x]$ and $A_e[y, z]$ into $A_T[w + y - 1, x + z]$. Since $\mathtt{P}_{A_o}(w, x) \prec \mathtt{P}_{A_e}(y, z)$, $\mathtt{fin}_o[i] = i + y - 1$ for $w \leqslant i \leqslant x$ and $\mathtt{fin}_e[i] = i + x$ for $y \leqslant i \leqslant z$. Thus, we store the suffixes $A_o[w, x]$ into $A_T[w + y - 1, x + y - 1]$ and those in $A_e[y, z]$ into $A_T[x + y, x + z]$, and we store the integers in $L_o[w, x - 1]$ into $L_T[w + y - 1, x + y - 2]$ and those in $L_e[y, z - 1]$ into $L_T[x + y, x + z - 1]$. We also set $\mathtt{ptr}_o[i] = x$ for $w \leqslant i < x$, $\mathtt{ptr}_e[i] = z$ for $y \leqslant i < z$, and $L_T[x + y - 1] = |\mathtt{lcp}(\mathtt{P}_{A_o}(w, x), \mathtt{P}_{A_e}(y, z))|$. We already set $\mathtt{ptr}_o[x]$ as z and $\mathtt{ptr}_e[z]$ as x and set $L_T[x + z]$ appropriately when we were storing C into $Q[s - 1]$ and the values stored in $\mathtt{ptr}_o[x]$, $\mathtt{ptr}_o[z]$, and $L_T[x + z]$ are still effective.

Consider the time complexity of the merging algorithm. Procedure MERGE (except `fin` and `ptr`) takes time proportional to the total number of odd and even partitioned equivalence classes. Since there are at most $n/2$ odd partitioned equivalence classes and at most $n/2$ even partitioned equivalence classes, MERGE takes $O(n)$ time. Since each entry of `fin` and `ptr` is set only once throughout stages, it takes $O(n)$ time overall. The rest of the merging algorithm takes time proportional to the total number of coupled pairs inserted into $Q[k]$. Since a coupled pair corresponds to a target equivalence class, the total number of coupled pairs is at most $n - 1$. Therefore, the time complexity of merging is $O(n)$.

3.4. The coupled pair lcp problem

Recall the coupled pair lcp problem: Given a coupled pair $C = \langle A_o[w, x], A_e[y, z]\rangle$ whose limit stage is $s - 1$, compute the end stage of C. And if the end stage of C is less than $s - 1$, determine whether $\mathtt{P}_{A_o}(w, x) \prec \mathtt{P}_{A_e}(y, z)$ or $\mathtt{P}_{A_o}(w, x) \succ \mathtt{P}_{A_e}(y, z)$. The problem is easy to solve when s is 1 or 2. When $s = 1$, $|\mathtt{P}_{A_o}(w, x)|$ and $|\mathtt{P}_{A_e}(y, z)|$ are 0 and thus the end stage of C is 0. When $s = 2$, the end stage of C is 1. From now on, we describe how to compute the end stage of C when $s \geqslant 3$. Assume without loss of generality that the end stage of $A_o[w, x]$ is $s - 1$.

We first show that when $s \geqslant 3$, the problem of computing the end stage of C (i.e., $|\mathtt{lcp}(\mathtt{P}_{A_o}(w, x), \mathtt{P}_{A_e}(y, z))|$) is reduced to the problem of computing the longest common prefix of two other suffixes.

$$\left|\mathtt{lcp}\left(\mathtt{P}_{A_o}(w, x), \mathtt{P}_{A_e}(y, z)\right)\right| = \left|\mathtt{lcp}_{s-1}\left(\mathtt{P}_{A_o}(w, x), \mathtt{P}_{A_e}(y, z)\right)\right|$$

$$= \left|\mathtt{lcp}_{s-1}(S_{A_o[w]}, S_{A_e[z]})\right|$$
$$= \left|\mathtt{lcp}_{s-2}(S_{A_o[w]+1}, S_{A_e[z]+1})\right| + 1.$$

The first equality holds because the end stage of $A_o[w, x]$ is $s-1$. The second equality holds because $\mathtt{pref}_{s-1}(\mathtt{P}_{A_o}(w, x)) = \mathtt{pref}_{s-1}(S_{A_o[w]})$ and $\mathtt{pref}_{s-1}(\mathtt{P}_{A_e}(y, z)) = \mathtt{pref}_{s-1}(S_{A_e[z]})$. The third equality holds because the start stage of the coupled pair is at least 1, which means that the first symbols of $S_{A_o[w]}$ and $S_{A_e[z]}$ are the same. From now on, let $w' = \text{index}_e(A_o[w] + 1)$, $x' = \text{index}_e(A_o[x] + 1)$, and $z' = \text{index}_o(A_e[z] + 1)$ for brevity.

We show how to compute $t = |\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[z']})|$ in O(1) time. We first define an index γ of A_o as follows.

Definition 3. Let γ be an index of array A_o such that $|\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[\gamma]})| \geqslant |\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[\delta]})|$ for any other index δ of A_o.

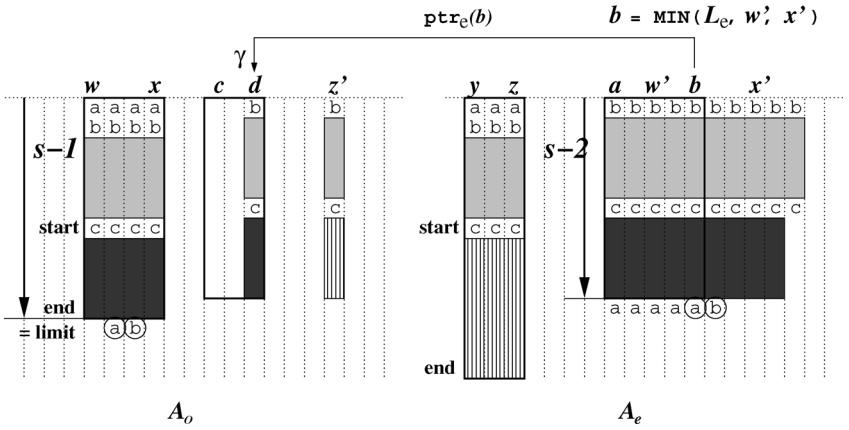
By definition of γ, t is the minimum of $t_1 = |\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[\gamma]})|$ and $t_2 = |\mathtt{lcp}_{s-2}(S_{A_o[\gamma]}, S_{A_o[z']})|$. To compute t, we first find γ and compute t_1. Let $A_e[a, b]$ be the partitioned equivalence class including $A_e[w']$ after stage $s-1$. We will show $\gamma = \mathtt{ptr}_e[b]$. There are two cases whether or not $A_e[a, b]$ constitutes a coupled pair stored in $Q[k]$ just after stage $s-1$.

If $A_e[a, b]$ constitutes a coupled pair stored in $Q[k]$ for $s-1 \leqslant k < n$, let $\langle A_o[c, d], A_e[a, b]\rangle$ denote the coupled pair. See Fig. 6(a).

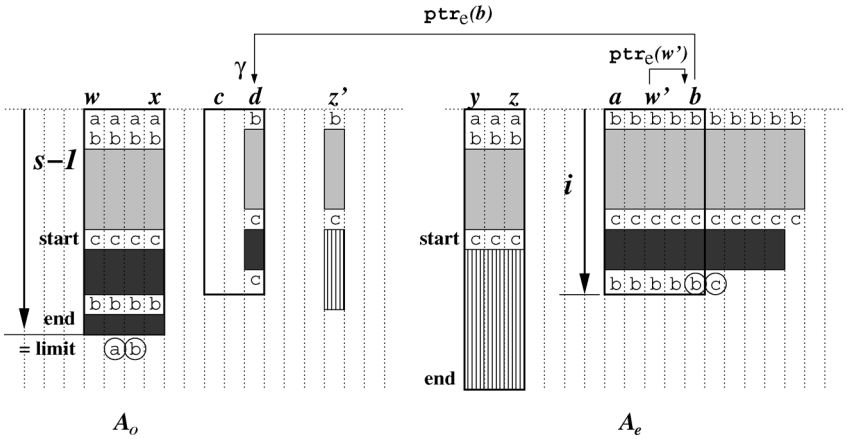
Lemma 6. *The start stages of $A_e[a, b]$ and $\langle A_o[c, d], A_e[a, b]\rangle$ are both $s-1$.*

Proof. The start stage of the coupled pair $C' = \langle A_o[c, d], A_e[a, b]\rangle$ is at most $s-1$ by the invariant. Since the start stage of C' is the maximum of the start stages of $A_o[c, d]$ and $A_e[a, b]$, the start stage of $A_e[a, b]$ is at most $s-1$. We show that the start stage of $A_e[a, b]$ is $s-1$ by showing that $A_e[a, b]$ is not an $(s-2)$-equivalence class. Since the end stage of $A_o[w, x]$ is $s-1$, it is easy to see $\mathtt{pref}_{s-2}(A_e[w']) = \mathtt{pref}_{s-2}(A_e[x'])$ and $\mathtt{pref}_{s-1}(A_e[w']) \neq \mathtt{pref}_{s-1}(A_e[x'])$. Since $\mathtt{pref}_{s-2}(A_e[w']) = \mathtt{pref}_{s-2}(A_e[x'])$, $A_e[w']$ and $A_e[x']$ are in the same $(s-2)$-equivalence class. However, $A_e[x']$ is not in $A_e[a, b]$ because $\mathtt{pref}_{s-1}(A_e[w']) \neq \mathtt{pref}_{s-1}(A_e[x'])$. Hence, $A_e[a, b]$ is not an $(s-2)$-equivalence class and the start stage of $A_e[a, b]$ is $s-1$. Since the start stage of $A_e[a, b]$ is $s-1$ and the start stage of C' is at most $s-1$ by the invariant, the start stage of C' is $s-1$. □

We show that γ is $\mathtt{ptr}_e[b] = d$ and t_1 is $s-2$. Since the start stage of C' is $s-1$ and $a \leqslant w' \leqslant b$, $|\mathtt{lcp}(S_{A_e[w']}, S_{A_o[d]})| \geqslant s-1$ and thus $|\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[d]})| = s-2$. Since $|\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[d]})|$ is at most $s-2$, γ in Definition 3 is d and $t_1 = |\mathtt{lcp}_{s-2}(S_{A_e[w']}, S_{A_o[\gamma]})| = s-2$. We have only to show how to find γ $(= d)$ in O(1) time. Since $A_e[w']$ and $A_e[x']$ are in the same $(s-2)$-equivalence class and $A_e[x']$ is not in $A_e[a, b]$ whose start stage is $s-1$, we can compute b from w' and x' in O(1) time by a $\text{MIN}(L_e, w', x')$ query. Once b is computed, we get d from $\mathtt{ptr}_e[b]$.



(a)



(b)

Fig. 6. Finding γ at stage s. (a) If $A_e[a, b]$ constitutes a coupled pair. (b) If $A_e[a, b]$ is processed.

If $A_e[a, b]$ is processed after stage $s-1$ (Fig. 6(b)), $A_e[a, b]$ is an i-uncoupled equivalence class for some $0 \leqslant i \leqslant s-1$ by the invariant. Since $A_e[a, b]$ is i-uncoupled, $\mathtt{pref}_i(S_{A_e[b]}) = \mathtt{pref}_i(S_{A_e[j]})$ and $\mathtt{pref}_i(S_{A_e[j]}) \neq \mathtt{pref}_i(S_{A_o[k]})$ for $a \leqslant j \leqslant b$ and $1 \leqslant k \leqslant n/2$ and thus $|\mathtt{lcp}(S_{A_e[w']}, S_{A_o[k]})| = |\mathtt{lcp}(S_{A_e[b]}, S_{A_o[k]})|$ for all $1 \leqslant k \leqslant n/2$. Hence, γ in Definition 3 is $\mathtt{ptr}_e[b]$ by definition of $\mathtt{ptr}_e$. We can compute γ in O(1) time because $\gamma = \mathtt{ptr}_e[b]$ and $b = \mathtt{ptr}_e[w']$ if $w' \neq b$ by definition of $\mathtt{ptr}_e$. We can also compute $|\mathtt{lcp}_{s-2}(S_{A_e[b]}, S_{A_o[\gamma]})|$ in O(1) time by definition of $\mathtt{ptr}_e$.

Finally, $t_2 = |\mathtt{lcp}_{s-2}(S_{A_o[\gamma]}, S_{A_o[z']})|$ is the minimum of $s-2$ and $|\mathtt{lcp}(S_{A_o[\gamma]}, S_{A_o[z']})|$, where $|\mathtt{lcp}(S_{A_o[\gamma]}, S_{A_o[z']})|$ can be obtained in O(1) time by the query $\mathrm{MIN}(L_o, \gamma, z'-1)$ or $\mathrm{MIN}(L_o, z', \gamma-1)$.

Therefore, we get the following lemma and theorem.

Lemma 7. *The coupled pair lcp problem can be solved in* O(1) *time.*

Theorem 1. *The odd and even arrays can be merged in* $O(n)$ *time and thus the suffix array can be constructed in* $O(n)$ *time.*

4. Concluding remarks

We have presented a linear-time algorithm to construct suffix arrays for integer alphabets, which do not use suffix trees as intermediate data structures during its construction. Since the case of a constant-size alphabet can be subsumed in that of an integer alphabet, our result implies that the time complexity of directly constructing suffix arrays matches that of constructing suffix trees. Recently, Kärkkäinen and Sanders [17] and Ko and Aluru [18] also proposed simple linear-time construction algorithms for suffix arrays. Burkhardt and Kärkkäinen [4] gave another construction algorithm that takes $O(n \log n)$ time using only $O(n/\sqrt{\log n})$ extra space.

Space reduction of a suffix array is an important issue [9,13,21,22] because the amount of text data is continually increasing. Grossi and Vitter [13] proposed the *compressed* suffix array of $O(n \log |\Sigma|)$-bits size and Sadakane [22] improved it by adding the *lcp* information. Since their compressions also exploit the odd-even divide-and-conquer approach used in this paper, our technique can be applied to building the compressed suffix array from a given string.

References

[1] M. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of LATIN 2000, in: Lecture Notes in Comput. Sci., vol. 1776, 2000, pp. 88–94.
[2] O. Berkman, U. Vishkin, Recursive star-tree parallel data structure, SIAM J. Comput. 22 (1993) 221–242.
[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1985) 31–55.
[4] S. Burkhardt, J. Kärkkäinen, Fast lightweight suffix array construction and checking, in: Symp. Combinatorial Pattern Matching, 2003, pp. 55–69.
[5] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Inform. Process. Lett. 12 (1981) 244–250.
[6] M. Farach, Optimal suffix tree construction with large alphabets, in: IEEE Symp. Found. Computer Science, 1997, pp. 137–143.
[7] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, J. Assoc. Comput. Mach. 47 (2000) 987–1011.
[8] M. Farach, S. Muthukrishnan, Optimal logarithmic time randomized suffix tree construction, in: Internat. Colloq. Automata Languages and Programming, 1996, pp. 550–561.
[9] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: IEEE Symp. Found. Computer Science, 2001, pp. 390–398.
[10] H.N. Gabow, J.L. Bentley, R.E. Tarjan, Scaling and related techniques for geometry problems, in: ACM Symp. Theory of Computing, 1984, pp. 135–143.
[11] G. Gonnet, R. Baeza-Yates, T. Snider, New indices for text: Pat trees and pat arrays, in: W.B. Frakes, R.A. Baeza-Yates (Eds.), Information Retrieval: Data Structures & Algorithms, Prentice Hall, 1992, pp. 66–82.
[12] D. Gusfield, An "Increment-by-one" approach to suffix arrays and trees, manuscript, 1990.
[13] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: ACM Symp. Theory of Computing, 2000, pp. 397–406.

[14] R. Hariharan, Optimal parallel suffix tree construction, J. Comput. Syst. Sci. 55 (1997) 44–69.
[15] J. Kärkkäinen, Suffix cactus: A cross between suffix tree and suffix array, in: Symp. Combinatorial Pattern Matching, 1995, pp. 191–204.
[16] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Symp. Combinatorial Pattern Matching, 2001, pp. 181–192.
[17] J. Kärkkäinen, P. Sanders, Simpler linear work suffix array construction, in: Internat. Colloq. Automata Languages and Programming, 2003, pp. 943–955.
[18] P. Ko, S. Aluru, Space-efficient linear time construction of suffix arrays, in: Symp. Combinatorial Pattern Matching, 2003, pp. 200–210.
[19] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–938.
[20] E.M. McCreight, A space-economical suffix tree construction algorithm, J. Assoc. Comput. Mach. 23 (1976) 262–272.
[21] J.I. Munro, V. Raman, S.S. Rao, Space efficient suffix trees, J. Algorithms 39 (2001) 205–222.
[22] K. Sadakane, Succinct representation of lcp information and improvement in the compressed suffix arrays, in: ACM-SIAM Symp. on Discrete Algorithms, 2002, pp. 225–232.
[23] K. Sadakane, Space-efficient data structures for flexible text retrieval systems, in: Internat. Symp. Algorithms and Computation, 2002, pp. 14–24.
[24] S.C. Sahinalp, U. Vishkin, Symmetry breaking for suffix tree construction, in: IEEE Symp. Found. Computer Science, 1994, pp. 300–309.
[25] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[26] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th IEEE Symp. Switching and Automata Theory, 1973, pp. 1–11.