Linear-Time Construction of Suffix Arrays
(Extended Abstract)

Dong Kyue Kim\textsuperscript{1*}, Jeong Seop Sim\textsuperscript{2}, Heejin Park\textsuperscript{3}, and Kunsoo Park\textsuperscript{3**}

\textsuperscript{1} School of Electrical and Computer Engineering, Pusan National University
dkkim1@pusan.ac.kr
\textsuperscript{2} Electronics and Telecommunications Research Institute, Daejeon 305-350, Korea
simjs@etri.re.kr
\textsuperscript{3} School of Computer Science and Engineering, Seoul National University
\{hjpark,kpark\}@theory.snu.ac.kr

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.

1 Introduction

The suffix tree due to McCreight \cite{McCreight76} is a compacted trie of all the suffixes of a string \(T\). It was designed as a simplified version of Weiner’s position tree \cite{Weiner}. The suffix array due to Manber and Myers \cite{ManberMyers93} and independently due to Gonnet et al. \cite{Gonnet} is basically a sorted list of all the suffixes of a string \(T\). There are also some other index data structures such as suffix automata \cite{Lecroq}. 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 of suffix tree construction has been shown to be equivalent to that of sorting \cite{Knuth}. For a general alphabet, suffix tree construction has time bound of \(\Theta(n \log n)\). And suffix trees can be constructed in linear time for a constant-size alphabet due to McCreight \cite{McCreight76} and Ukkonen \cite{Ukkonen} or for an integer alphabet due to Farach-Colton, Ferragina, and Muthukrishnan \cite{FarachColton}.

\* Supported by KOSEF grant R01-2002-000-00589-0.
\*\* Supported by BK21 Project and IMT2000 Project AB02.
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 [18] 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 [6,7,8,15,22], 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 is designed for suffix trees. Whereas Farach-Colton et al.’s uses 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.

Recently, Kärkkäinen and Sanders [16] and Ko and Aluru [17] 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 [20,13,21] 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 [21] improved it by adding the lcp information. Since their compressions also exploit the divide-and-conquer approach mentioned above, we can directly build the compressed suffix array from a given string in linear time by applying our proposed algorithm to their compressions.

2 Preliminaries

2.1 Definitions and Notations

We first give some definitions and notations that will be used in our algorithms. Consider a string $T$ of length $n$ over an alphabet $\Sigma$. Let $T[i]$ denote the $i$th 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 $\Sigma$. $S_i$, $1 \leq i \leq n$, denotes the suffix of
A minimum query MIN(

Given an array 

Definition 1. 

\[ T \] that starts at position \( i \). The prefix of length \( k \) of a string \( \alpha \) is denoted by \( \text{pref}_k(\alpha) \). We denote by \( \text{lcp}(\alpha, \beta) \) the longest common prefix of two strings \( \alpha \) and \( \beta \) and by \( \text{lcp}_1(\alpha, \beta) \) the longest common prefix of \( \text{pref}_1(\alpha) \) and \( \text{pref}_1(\beta) \). When string \( \alpha \) is lexicographically smaller than string \( \beta \), 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, \( A_T[i] = j \) if \( S_j \) is lexicographically the \( i \)th 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 adjacent suffixes in \( A_T \), i.e., \( L_T[i] = \text{lcp}(SA_T[i], SA_T[i+1]) \) for \( 1 \leq i < n \). We set \( L_T[0] = L_T[n] = -1 \).

We define odd and even arrays of a string \( T \). The odd array \( SA_o = (A_o, L_o) \) is the suffix array of all suffixes beginning at odd positions in \( T \). That is, the sort array \( A_o \) of \( SA_o \) is the lexicographically ordered list of all suffixes beginning at odd positions of \( T \), and the lcp array \( L_o \) has the length of the longest common prefix of adjacent odd suffixes in \( A_T \). Similarly, the even array \( SA_e = (A_e, L_e) \) is the suffix array of all suffixes beginning at even positions in \( T \).

For a subarray \( A[x, y] \) of sort array \( A \), we define \( P_A(x, y) \) as the longest common prefix of the suffixes \( S_{A[x]}, S_{A[x+1]}, \ldots, S_{A[y]} \). If \( x = y \), \( P_A(x, x) \) 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.** Let \( A[x, y] \) be a subarray of sort array \( A \) for \( x < y \) and \( L \) be a corresponding lcp array.

(a) \( P_A(x, y) = \text{lcp}(SA_{A[x]}, SA_{A[y]}) \).

(b) \( |P_A(x, y)| \) is equal to the minimum value in \( L[x, y - 1] \).

In order to find \( |P_A(x, y)| \) efficiently, we define the following problem.

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

This MIN query can be answered in constant time after linear-time preprocessing of array \( A \). The first solution [10] for the range minima problem constructed the cartesian tree [25] of array \( A \) and answered nearest common ancestor computations [14, 25] on the tree. In the Appendix, we described another solution for the problem, which is a modification of Berkman and Vishkin’s simple solution. We remark that this method uses only arrays without making any kinds of trees. Similar results were given by Bender and Farach-Colton [1] and Sadakane [21]. By a MIN query, we get the following theorem.
Theorem 1. Given a suffix array \((A, L)\) and \(x < y\), \(|P_A(x, y)|\) (i.e., the smallest index \(x \leq j < y\) such that \(L_T[j] = \min_{x \leq i < y} L_T[i]\)) can be computed in constant time.

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

Lemma 2. Let \(i \) and \(j\) \((i < j)\) be two positions in string \(T\). If \(T[i]\) and \(T[j]\) match, \(\lcp(S_i, S_j)\) \(= \lcp(S_{i+1}, S_{j+1}) + 1\); otherwise, \(\lcp(S_i, S_j)\) \(= 0\).

### 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 \(E_l\) \((l \geq 0)\) on \(A\) is:

\[
E_l = \{(i, j) \mid \text{pref}_l(S_{A[i]}) = \text{pref}_l(S_{A[j]})\}.
\]

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

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 suffixes they represent, we get the following fact from the definition of \(E_l\).
Fact 1 Subarray $A[p, q]$, $1 \leq p \leq q \leq m$, is an equivalence class of $E_l$, $0 \leq l \leq n$, on $A$ if and only if $L[p-1] < l$, $L[q] < l$, and $L[i] \geq l$ for all $p \leq i < q$.

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, $L[i] \geq l$ for all $p \leq i < q$. Let $p \leq i_1 < i_2 < \cdots < i_r < q$ denote all the indices such that $L[i_1] = L[i_2] = \cdots = L[i_r] = l$. Since $L[i] \geq l + 1$ for $i \not\in \{i_1, i_2, \ldots, i_r\}$ and $p \leq i < q$, $A[p, i_1], A[i_1+1, i_2], \ldots, A[i_r+1, q]$ are equivalence classes of $E_{l+1}$. We can find $i_1, i_2, \ldots, i_r$ in $O(r)$ time by Theorem 1 and we get the following lemma.

Lemma 3. 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 $E_{l+1}$.

An equivalence class of $E_l$ can be an equivalence class of $E_k$ for $k \neq l$. Consider $A[5, 7]$ in Fig. 1 where $L[i] \geq 5$ for $5 \leq i < 7$, $L[4] = 2$, and $L[7] = 1$. Then, $A[5, 7]$ 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 $a \leq i \leq b$ if and only if $\max\{L[p-1], L[q]\} = a - 1$ and $b = |P_A(p, q)| = \min_{p \leq i < q} L[i]$.

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

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, 7, 8, 15, 22], 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, 7, 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 takes linear time besides recursion.

1. Encode the given string $T$ into a string of a half size: We make pairs $(T[2i-1], T[2i])$ for every $1 \leq i \leq n/2$. Radix-sort all the pairs in linear time, and map the pairs into integers in the range $\{1, n/2\}$. If we convert the pairs in $T$ into corresponding integers, we get a new string of a half size, which is denoted by $T'$. 
2. Recursively construct suffix array \(SA_{T'}\) of \(T'\).
3. Compute \(SA_o\) from \(SA_{T'}\): We get \(A_o\) by \(A_o[i] = 2A_{T'}[i] - 1\) for all \(i\). Since two symbols in \(T\) are encoded into one symbol in \(T'\), we get \(L_o\) from \(L_{T'}\) as follows. If the first different symbols of \(T'\) in adjacent suffixes \(S_{A_o[i]}\) and \(S_{A_o[i+1]}\) have the same first symbol of \(T'\), then \(L_o[i] = 2L_{T'}[i] + 1\); otherwise, \(L_o[i] = 2L_{T'}[i]\).

### 3.2 Constructing Even Array

The even array \(SA_e\) is constructed from \(SA_o\) in linear time. We first compute the sort array \(A_e\) and then compute the lcp array \(L_e\) as follows.

1. Make the sort array \(A_e\): An even suffix is one symbol followed by an odd suffix. We make tuples for even suffixes: the first element of a tuple is \(T[2i]\) and the second element is suffix \(S_{2i+1}\). First, the tuples are sorted by the second elements (this result is given in \(A_o\)). Then we stably sort the tuples by the first elements and we get \(A_e\).
2. Compute the lcp array \(L_e\): Consider two even suffixes \(S_{2i}\) and \(S_{2j}\). By Lemma 2 if \(T[2i]\) and \(T[2j]\) match, \(|lcp(S_{2i}, S_{2j})| = |lcp(S_{2i+1}, S_{2j+1})| + 1\); otherwise, \(lcp(S_{2i}, S_{2j}) = 0\). We can get \(|lcp(S_{2i+1}, S_{2j+1})|\) from the odd array \(SA_o\) in constant time as follows. Let \(x = \text{index}(2i + 1)\) and \(y = \text{index}(2j + 1)\) in \(SA_o\). By Lemma 1 \(|lcp(S_{2i+1}, S_{2j+1})| = |P_{A_o}(x, y)|\), which is computed by a \(\text{MIN}(L_o, x, y)\) query.

### 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, where \(n\) is the length of \(T\). The main task in merging is to compute the sort array \(A_T\). The lcp array \(L_T\) is computed as a by-product during the merging. The target entry of an entry \(A_o[i]\) (resp. \(A_e[i]\)) is the entry of \(A_T\) that stores the integer in \(A_o[i]\) (resp. \(A_e[i]\)) after we merge \(A_o\) and \(A_e\). To merge \(A_o\) and \(A_e\), we first compute the target entries of entries in \(A_o\) and \(A_e\) and then store all the integers in \(A_o\) and \(A_e\) into \(A_T\). Hence, the problem of merging is reduced to the problem of computing target entries of entries in \(A_o\) and \(A_e\).

We first introduce some notions on equivalence classes of \(E_i\) on \(A_o\) and \(A_e\). For brevity, we define notions only on equivalence classes on \(A_o\). (They are defined on \(A_e\) similarly.) An equivalence class \(A_o[w,x]\) of \(E_i\) is \(i\)-coupled with an equivalence class \(A_e[y,z]\) of \(E_i\) if and only if all the suffixes represented by the integers in \(A_o[w,x]\) and \(A_e[y,z]\) have the common prefix of length \(i\), i.e., \(\text{pref}_i(P_{A_o}(w, x)) = \text{pref}_i(P_{A_e}(y, z))\). The integers in \(A_o[w,x]\) and \(A_e[y,z]\) (that are \(i\)-coupled with each other) form an equivalence class of \(E_i\) on \(A_T\) after we merge \(A_o\) and \(A_e\) because each odd suffix represented by an integer in \(A_o[w,x]\) and each even suffix represented by an integer in \(A_e[y,z]\) have the common prefix \(\text{pref}_i(P_{A_o}(w, x)) = \text{pref}_i(P_{A_e}(y, z))\) and the other odd or even suffixes do not have \(\text{pref}_i(P_{A_e}(w, x))\) as their prefixes.
Lemma 4. The integers in $A_o[w, x]$ and $A_e[y, z]$ that are $i$-coupled with each other form an equivalence class $A_T[w + y - 1, x + z]$.

An equivalence class $A_i[w, x]$ of $E_i$ is $i$-uncoupled if it is not $i$-coupled with any equivalence class of $E_i$ on $A_e$. If an equivalence class $A_o[w, x]$ of $E_i$ is $i$-uncoupled, no suffix represented by an integer in $A_e$ has pref($P_{A_o}(w, x)$) as its prefix and thus the integers in an $i$-uncoupled equivalence class $A_o[w, x]$ form an equivalence class of $E_i$ on $A_T$, which is $A_T[a + w, a + x]$ for some $a$, after we merge $A_o$ and $A_e$.

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 $l = \max\{l_o, l_e\} \leq k = \min\{k_o, k_e\}$ 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 integers in $A_o[w, x]$ and $A_e[y, z]$ form an equivalence class $A_T[w + y - 1, x + z]$ after we merge $A_o$ and $A_e$. We define the start stage and the end stage of coupled pair $C$ as the start stage and the end stage of 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 classes of $E_{k+1}$, $A_T[w + y - 1, x + z]$ cannot be an equivalence class of $E_{k+1}$. 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 $|\text{lcp}(P_{A_o}(w, x), P_{A_e}(y, z))|$, and it is in the range of $[l, k]$. In our algorithm, we maintain coupled pairs in multiple queues $Q[k]$ for $0 \leq k < n$. Each queue $Q[k]$ contains coupled pairs whose limit stage is $k$.

We now describe the outline of computing the target entries of entries in $A_o$ and $A_e$. We will compute target entries only for uncoupled equivalence classes on $A_o$ and $A_e$. Since all equivalence classes of $E_i$ on $A_o$ and $A_e$ will eventually be uncoupled as we increase $i$, we can find target entries of all entries in $A_o$ and $A_e$ in this way.

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

Invariant: At the end of stage $s \geq 0$, the equivalence classes that constitute coupled pairs whose start stages are at most $s$ and limit stages are at least $s$ are stored in $Q[i]$ for $s \leq i \leq n - 1$. For every $i$-uncoupled equivalence class for $0 \leq i \leq s$ that does not constitute such a coupled pair, the target entries for the equivalence class have been computed.

We will call an equivalence class for which target entries have been computed a processed equivalence class.

We describe the outline of stages. At stage $s$, 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 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 \( P_{A_o}(w,x) < P_{A_e}(y,z) \) or \( P_{A_o}(w,x) > P_{A_e}(y,z) \).

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

- 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 equivalence classes of \( E_o \). Every partitioned equivalence class will be either \( s \)-coupled or \( s \)-uncoupled. The \( s \)-coupled equivalence classes constitute coupled pairs whose start stages are at most \( s \) and limit stages are at least \( s \), and thus we store each coupled pair in \( Q[k] \) for \( s \leq k \leq n - 1 \), where \( k \) is the limit stage of the coupled pair. For the \( s \)-uncoupled equivalence classes, we find the target entries for them.

- 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 find the target entries for \( A_o[w,x] \) and \( A_e[y,z] \).

It is not difficult to see that the invariant is satisfied after stage \( s \).

In our merging algorithm, we will use four arrays \( p_{tr}, p_{fin}, f_{in}, \) and \( f_{fin} \). Since \( p_{tr} \) and \( f_{in} \) are similar to \( p_{tr} \) and \( f_{fin} \), we explain \( p_{tr} \) and \( f_{fin} \) only. At the end of stage \( s \), the values stored in \( p_{tr} \) and \( f_{fin} \) are as follows.

- \( f_{in} \) stores target entries for \( A_o \), i.e., \( f_{in}[i] \) for \( 1 \leq i \leq n_o \) is defined if \( A_o[i] \) is an entry of a processed equivalence class and it stores the index of the target entry of \( A_o[i] \).

- \( p_{tr}[i] \) for \( 1 \leq i \leq n_o \) 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 \( s \leq k \leq n - 1 \)), \( p_{tr}[b] \) 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., \( a \leq i < b \)), \( p_{tr}[i] \) stores \( b \).

    - Otherwise, \( p_{tr}[b] \) stores \( \beta \) such that \( A_o[\beta] \) is the last entry of a partitioned equivalence class \( A_o[a, \beta] \) and that \( \beta \) satisfies \( |lcp(S_{A_o[b]}, S_{A_e[\beta]})| \geq |lcp(S_{A_o[b]}, S_{A_e[\delta]})| \) for any other \( 1 \leq \delta \leq n_e \). In addition, \( |lcp(S_{A_o[b]}, S_{A_e[\beta]})| \) is stored in \( LT[f_{fin}[b]] \) if \( f_{fin}[b] < f_{fin}[\beta] \) and \( LT[f_{fin}[\beta]] \) otherwise.
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 equivalence classes of $E_o$. 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 equivalence classes in $C_o$ by $A_o[w_i, x_i]$, $1 \leq i \leq r_1$, such that $P_{A_o}(w_j, x_j) < P_{A_o}(w_k, x_k)$ if $j < k$ and equivalence classes in $C_e$ by $A_e[y_i, z_i]$, $1 \leq i \leq r_2$, such that $P_{A_e}(y_j, z_j) < P_{A_e}(y_k, z_k)$ if $j < k$. Partitioning $A_o[w, x]$ and $A_e[y, z]$ into equivalence classes of $E_o$ takes $O(r_1 + r_2)$ time by Lemma 3.

Each equivalence class in $C_o$ (resp. $C_e$) is either $s$-coupled or $s$-uncoupled. We find every coupled pair $(A_o[w_i, x_i], A_e[y_j, z_j])$, store it into $Q[\min\{|P_{A_o}(w_i, x_i)|, |P_{A_e}(y_j, z_j)|\}]$, set $\text{ptr}_o[x_i] = z_j$, $\text{ptr}_e[z_j] = x_i$, and compute $L_{T[x_i]}$ appropriately. For each $s$-uncoupled equivalence class $A_o[w_i, x_i]$, we find target entries for $A_o[w_i, x_i]$, store them in $\text{fin}_o[a]$ for $w_i \leq a \leq x_i$, and compute $\text{ptr}_o[k]$ for $w_i \leq k \leq x_i$ and $L_{T[\text{fin}_o[x_i]]}$. We perform a similar operation for each $s$-uncoupled equivalence class $A_e[y_j, z_j]$. The following procedure shows the operations in detail. (We assume $w_{r_1+1} = b_{r_2+1} = \$ $ where $\$ > a$ 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 $z_{r_2+1} = z_{r_2}$.)

Procedure MERGE($C_o, C_e$)
1: $i \leftarrow 1$ and $j \leftarrow 1$
2: while $i \leq r_1$ or $j \leq r_2$ do
3: \quad $a_i \leftarrow$ the $i$th symbol of $P_{A_o}(w_i, x_i)$
4: \quad $b_j \leftarrow$ the $j$th symbol of $P_{A_e}(y_j, z_j)$
5: \quad if $a_i = b_j$ then // $A_o[w_i, x_i]$ and $A_e[y_j, z_j]$ are $s$-coupled.
6: \quad \quad $k \leftarrow \min\{|P_{A_o}(w_i, x_i)|, |P_{A_e}(y_j, z_j)|\}$
7: \quad \quad store $(A_o[w_i, x_i], A_e[y_j, z_j])$ into $Q[k]$
8: \quad if $i + j < r_1 + r_2$ then $L_{T[x_i]} + z_j \leftarrow s - 1$ fi
9: \quad if $i < r_1$ then $\text{ptr}_o[x_i] \leftarrow z_j$ fi
10: \quad if $j < r_2$ then $\text{ptr}_e[z_j] \leftarrow x_i$ fi
11: \quad $i \leftarrow i + 1$ and $j \leftarrow j + 1$
12: else if $a_i < b_j$ then // $A_o[w_i, x_i]$ is $s$-uncoupled.
13: \quad if $i + j < r_1 + r_2$ then $L_{T[x_i + y_j - 1]} \leftarrow s - 1$ fi
14: \quad $\text{fin}_o[k] \leftarrow k + y_j - 1$ for $w_i \leq k \leq x_i$
15: \quad $\text{ptr}_o[k] \leftarrow z_j$ for $w_i \leq k \leq x_i$
16: \quad if $i < r_1$ then $\text{ptr}_o[x_i] \leftarrow z_j$
17: \quad $i \leftarrow i + 1$
18: else // $A_e[y_j, z_j]$ is $s$-uncoupled.
19: \quad if $i + j < r_1 + r_2$ then $L_{T[w_i + z_j - 1]} \leftarrow s - 1$ fi
20: \quad $\text{fin}_e[k] \leftarrow k + w_i - 1$ for $y_j \leq k \leq z_j$
21: \quad $\text{ptr}_e[k] \leftarrow z_j$ for $y_j \leq k \leq z_j$
22: \quad if $j < r_2$ then $\text{ptr}_e[z_j] \leftarrow x_i$ fi
23: \quad $j \leftarrow j + 1$
24: \quad fi
25: od
end
For each equivalence class $A_o[w_i, x_i]$, we show that $\text{fin}_o[\alpha]$ and $\text{ptr}_o[\alpha]$ for $w_i \leq \alpha \leq x_i$ store correct values. (Similarly for $A_e[y_j, z_j]$.) We only show that $\text{ptr}_o[x_i]$ 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($C_o, C_e$), $\text{ptr}_o[x_i]$ is $z_j$ for some $1 \leq j \leq r_2$.

Claim. $z_j$ satisfies $|\text{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})| \geq |\text{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})|$ for $1 \leq \alpha \leq n_e$ and $|\text{lcp}(S_{A_e[x_i]}, S_{A_o[z_j]})|$ is stored in $L_T[\text{fin}_o[x_i]]$ if $\text{fin}_o[x_i] < \text{fin}_e[z_j]$ and in $L_T[\text{fin}_e[z_j]]$ otherwise.

Proof of Claim: Since $A_o[w, x]$ and $A_o[y, z]$ is $(s - 1)$-coupled and $A_o[w_i, x_i]$ is $s$-uncoupled, $|\text{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})| = s - 1$. Since $A_o[w_i, x_i]$ is $s$-uncoupled, $|\text{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})| \leq s - 1$ for $1 \leq \alpha \leq n_e$. Hence, $z_j$ satisfies $|\text{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})| \geq |\text{lcp}(S_{A_o[x_i]}, S_{A_e[\alpha]})|$ for $1 \leq \alpha \leq n_e$. If $\text{fin}_o[x_i] < \text{fin}_e[z_j]$, $\text{fin}_o[x_i] = x + z$ and thus $L_T[\text{fin}_o[x_i]]$ is set to $s - 1$, which is $|\text{lcp}(S_{A_o[x_i]}, S_{A_e[z_j]})|$. Otherwise, $\text{fin}_e[z_j] < x + z$ and thus $L_T[\text{fin}_o[x_i]]$ 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 $P_{A_e}(w, x) < P_{A_e}(y, z)$. We first find the target entries for $A_o[w, x]$ and $A_e[y, z]$. We set $\text{fin}_o[i] = i + y - 1$ for $w \leq i \leq x$ and $\text{fin}_e[i] = i + x$ for $y \leq i \leq z$. We also set $\text{ptr}_o[i] = x$ for $w \leq i < x$, $\text{ptr}_e[i] = z$ for $y \leq i < z$, and $L_T[x + y - 1] = |\text{lcp}(P_{A_e}(w, x), P_{A_e}(y, z))|$. We already set $\text{ptr}_o[x]$ as $z$ and $\text{ptr}_e[z]$ as $x$ and set $L_T[x + z]$ appropriately when we were storing $C$ into $Q[s - 1]$ and the values stored in $\text{ptr}_o[x], \text{ptr}_e[z]$, and $L_T[x + z]$ are still effective.

Consider the time complexity of the merging algorithm. Procedure MERGE (except $\text{fin}$ and $\text{ptr}$) takes time proportional to the total number of partitioned equivalence classes in $A_o$ and $A_e$. Since there are at most $n_o$ partitioned equivalence classes in $A_o$ and at most $n_e$ classes in $A_e$, MERGE takes $O(n)$ time. Since each entry of $\text{fin}$ and $\text{ptr}$ 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 couple pair corresponds to an equivalence class on $A_T$, 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 $P_{A_o}(w, x) < P_{A_e}(y, z)$ or $P_{A_o}(w, x) \geq P_{A_e}(y, z)$. The problem is easy to solve when $s$ is 1 or 2. When $s = 1$, $P_{A_o}(w, x)$ and $P_{A_e}(y, z)$ 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 $s \geq 3$. Assume without loss of generality that the end stage of $A_o[w, x]$ is $s - 1$.

We first show that when $s \geq 3$, the problem of computing the end stage of $C$ (i.e., $|\text{lcp}(P_{A_o}(w, x), P_{A_e}(y, z))|$) is reduced to the problem of computing the longest common prefix of two other suffixes.
Theorem 5. Let \( \gamma \) be an index of array \( A_o \) such that \( |lcp(p_{A_o}(w, x), p_{A_o}(y, z))| \geq |lcp_{s-2}(S_{A_o}[w'], S_{A_o}[\gamma])| \) for any other index \( \delta \) of \( A_o \).

By definition of \( \gamma \), \( t \) is the minimum of \( t_1 = |lcp_{s-2}(S_{A_0}[w'], S_{A_0}[\gamma])| \) and \( t_2 = |lcp_{s-2}(S_{A_0}[\gamma], S_{A_0}[z'])| \). To compute \( t \), we first find \( \gamma \) and compute \( t_1 \). Let \( A_o[a, b] \) be the partitioned equivalence class including \( A_o[w'] \) after stage \( s-1 \). We will show \( \gamma = ptr_x[b] \). 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 \( s-1 \leq k < n \), let \( A_o[c, d] \) be the equivalence class coupled with \( A_e[a, b] \). See Fig. 2.

Lemma 5. The start stages of \( A_e[a, b] \) and \( (A_o[c, d], A_e[a, b]) \) are both \( s-1 \).

We show that \( \gamma = ptr_x[b] = d \) and \( t_1 \) is \( s-2 \). Since the start stage of \( O' \) is \( s-1 \) and \( a \leq w' \leq b \), \( |lcp(S_{A_0}[w'], S_{A_0}[d])| \geq s-1 \) and thus \( |lcp_{s-2}(S_{A_0}[w'], S_{A_0}[d])| = s-2 \). Since \( |lcp_{s-2}(S_{A_0}[w'], S_{A_0}[d])| \) is at most \( s-2 \), \( \gamma \) in definition 5 is \( d \) and \( t_1 = |lcp_{s-2}(S_{A_0}[w'], S_{A_0}[\gamma])| = s-2 \). We have only to show how to find \( \gamma (= d) \)

\[
|lcp(p_{A_o}(w, x), p_{A_o}(y, z))| = |lcp_{s-1}(p_{A_o}(w, x), p_{A_o}(y, z))|
= |lcp_{s-1}(S_{A_o}[w], S_{A_o}[z])|
= |lcp_{s-2}(S_{A_o}[w+1], S_{A_o}[z+1])| + 1
\]
in \( O(1) \) time. Since \( A_e[w'] \) and \( A_e[x'] \) are in the same equivalence class of \( E_s-2 \) 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 \( \text{ptr}_{s}[b] \).

If \( A_e[a,b] \) is processed after stage \( s-1 \), \( A_e[a,b] \) is an \( i \)-uncoupled equivalence class for some \( 0 \leq i \leq s-1 \) by the invariant. Since \( A_e[a,b] \) is \( i \)-uncoupled, \( \text{pref}_x(S_{A_e}[j]) = \text{pref}_x(S_{A_e}[j]) \) and \( \text{pref}_x(S_{A_e}[j]) \neq \text{pref}_x(S_{A_e}[k]) \) for \( a \leq j \leq b \) and \( 1 \leq k \leq n_e \) and thus \( |\text{lcp}(S_{A_e}[w'],S_{A_e}[b])| = |\text{lcp}(S_{A_e}[b],S_{A_e}[k])| \) for all \( 1 \leq k \leq n_o \). Hence, \( \gamma \) in definition 8 is \( \text{ptr}_{s}[b] \) by definition of \( \text{ptr}_{s} \). We can compute \( \gamma \) in \( O(1) \) time because \( \gamma = \text{ptr}_{s}[b] \) and \( b = \text{ptr}_{s}[w'] \) if \( w' \neq b \) by definition of \( \text{ptr}_{s} \). We can also compute \( |\text{lcp}_{s-2}(S_{A_e}[b],S_{A_e}[\gamma])| \) in \( O(1) \) time by definition of \( \text{ptr}_{s} \).

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

References

Appendix: Range Minima Problem

We define the range-minima problem as follows:

Given an array \( A = (a_1, a_2, \ldots, a_n) \) of integers \( 0 \leq a_i \leq n - 1 \), preprocess \( A \) so that any query \( \text{MIN}(A, i, j) \), \( 1 \leq i < j \leq n \), requesting the index of the leftmost minimum element in \( (a_i, \ldots, a_j) \), can be answered in constant time.

We first describe two preprocessing algorithms for the range-minima problem: algorithm E takes exponential time and algorithm L takes \( O(n \log n) \) time. Then, we present a linear-time preprocessing algorithm using both algorithms E and L. Finally, we describe how to answer a range-minimum query in constant time. Our algorithm is a modification of Berkman and Vishkin’s solution for the range minima problem [2].

Algorithm E: Since the elements of \( A \) are integers in the range \([0, n - 1]\), the number of possible input arrays of size \( n \) is \( n^n \). If we regard an array of size \( n \) as a string \( S \in \Sigma^n \) over an integer alphabet \( \Sigma = \{0, 1, \ldots, n - 1\} \), it is mapped to an integer \( k \) (\( 1 \leq k \leq n^n \)) such that \( S \) is lexicographically the \( k \)th string among the \( n^n \) possible strings. We make a table \( T_n(k, i, j) \) that stores the answer to query \( \text{MIN}(A, i, j) \), where \( A \) is mapped to \( k \). The size of table \( T_n \) is \( O(n^{n+2}) \) and it takes \( O(n^{n+2}) \) time to make \( T_n \).
Algorithm L: We now describe an $O(n \log n)$-time algorithm. We define the prefix and suffix minima as follows. The prefix minima of $A$ are $(c_1, c_2, \ldots, c_n)$ such that $c_i = \min\{a_1, \ldots, a_i\}$ for $1 \leq i \leq n$. Similarly, the suffix minima of $A$ are $(d_1, d_2, \ldots, d_n)$ such that $d_j = \min\{a_j, \ldots, a_n\}$ for $1 \leq j \leq n$. The prefix minima and suffix minima of $A$ can be computed in linear time. The preprocessing of algorithm L constructs a complete binary tree $T$ whose leaves are the elements of the input array $A$. Let $A_u$ be the list of the leaves of the subtree rooted at node $u$. Each internal node $u$ of $T$ maintains the prefix minima and suffix minima of $A_u$. It takes $O(n \log n)$ time to construct $T$. Since $T$ is a complete binary tree, it can be easily implemented by arrays.

Suppose that we are now given a range-minima query $\text{MIN}(A, i, j)$. Find the lowest common ancestor $u$ of two leaves $a_i$ and $a_j$ in $T$. Let $v$ and $w$ be the left and right children of $u$, respectively. Then, $[i, j]$ is the union of a suffix of $A_v$ and a prefix of $A_w$. The answer to the query is the minimum of the following two elements: the minimum of the suffix of $A_v$ and the minimum of the prefix of $A_w$. These operations take constant time using $T$.

We now describe a linear-time preprocessing algorithm for the range-minima problem.

- Let $m = \log \log n$. Partition the input array $A$ into $n/m$ blocks $A_i$ of size $m$. We map each block $A_i$ into an array $B_i$ whose elements are the rankings in the sorted list of $A_i$ (i.e., the elements of $B_i$ are integers in the range $[0, m - 1]$). We can sort $n/m$ blocks $A_i$ at the same time using $n$ buckets in $O(n)$ time. Apply algorithm E to all possible arrays of size $m$. Since $m^{m+2} = O(n)$, we can make table $T_m$ in $O(n)$ time using $O(n)$ space.

- Partition $A$ into $n/\log n$ blocks $A_i$ of size $\log n$ and find the minimum in each block. Apply algorithm L to an array of these $n/\log n$ minima. Also, we do the following for each block $A_i$. Partition $A_i$ into subblocks of size $\log \log n$, and find the minimum in each subblock. Apply algorithm L to these $\log n/\log \log n$ minima. The total time and space are $O(n)$.

When a query $\text{MIN}(A, i, j)$ is given, the range $[i, j]$ can be divided into at most five subranges, and the minimum in each subrange can be found in constant time by the preprocessing above. The answer to the query is the minimum of these five minima [2].