Efficient Algorithms for Finding a Longest Common Increasing Subsequence

Wun-Tat Chan* Yong Zhang† Stanley P. Y. Fung‡ Deshi Ye§
Hong Zhu¶

Abstract

We study the problem of finding a longest common increasing subsequence (LCIS) of multiple sequences of numbers. The LCIS problem is a fundamental issue in various application areas, including the whole genome alignment. In this paper we give an efficient algorithm to find the LCIS of two sequences in $O(\min(r \log \ell, n\ell + r) \log \log n + Sort(n))$ time where n is the length of each sequence and r is the number of ordered pairs of positions at which the two sequences match, ℓ is the length of the LCIS, and $Sort(n)$ is the time to sort n numbers. For m sequences where $m \geq 3$, we find the LCIS in $O(\min(mr^2, r \log \ell \log^m r) + m \cdot Sort(n))$ time where r is the total number of m-tuples of positions at which the m sequences match. The previous results find the LCIS of two sequences in $O(n^2)$ and $O(n\ell \log \log n + Sort(n))$ time. Our algorithm is faster when r is relatively small, e.g., for $r < \min(n^2/(\log \ell \log \log n), n\ell/\log \ell)$.

1 Introduction

Given m sequences of numbers, the *longest common increasing subsequence (LCIS)* is the longest sequence among all increasing sequences that are subsequences of each of the m sequences. The LCIS problem is a fundamental issue in various application areas, including the whole genome alignment [3, 4, 8].

A genome can be considered as a very long sequence of characters. Due to the huge size of the genomes (e.g., ≥ 4 Mb), using standard sequence alignment algorithms to align genomes is simply infeasible. Software for whole genome alignment has been developed

*Department of Computer Science, University of Hong Kong, Hong Kong, wtchan@cs.hku.hk
†Department of Computer Science, University of Hong Kong, Hong Kong, yzhang@cs.hku.hk
‡Department of Computer Science, University of Leicester, Leicester, United Kingdom, pyfung@mcs.le.ac.uk.
§Department of Computer Science, University of Hong Kong, Hong Kong, yedeshi@cs.hku.hk
¶Department of Computer Science and Engineering, Fudan University, China, hzhu@fudan.edu.cn

based on heuristics, in which the most popular one is called MUMmer [3]. The alignment process of MUMmer consists of several steps. Given two genomes, MUMmer identifies all maximal unique matching subsequences (MUM) in both genomes. An MUM is a subsequence that occurs exactly once in each genome and the MUM cannot be extended. Then a longest possible subset of MUMs that occur in the same order in each genome is extracted. MUMmer further constructs the alignment based on this subset of MUMs. One way of finding a longest possible subset of MUMs is to label the MUMs in one genome by the order of appearance. These labels corresponding to the MUMs that appear in the other genome form a permutation of $\{1, \ldots, n\}$ where n is the number of MUMs. Finding a longest possible subset of MUMs is equivalent to finding a longest increasing subsequence (LIS) of this permutation of $\{1, \ldots, n\}$. When we need to align three or more genomes [4], the step of finding a longest possible subset of MUMs becomes finding a longest common increasing subsequence (LCIS) of two or more permutations of $\{1, \ldots, n\}$.

Given a sequence $A = a_1 a_2 \ldots a_n$ of numbers, we say that A is an *increasing sequence* if $a_1 < a_2 < \cdots < a_n$. A sequence $S = s_1 s_2 \ldots s_\ell$ is a subsequence of A if for some integers $1 \leq i_1 < i_2 < \cdots < i_\ell \leq n$, $s_j = a_{i_j}$ for all $1 \leq j \leq \ell$. Given m sequences, $A^1, A^2, \ldots, A^m$, a sequence S is a *common increasing subsequence (CIS)* of the m sequences if S is an increasing sequence and S is a subsequence of A^i for all $1 \leq i \leq m$. The *longest common increasing subsequence (LCIS)* of $A^1, A^2, \ldots, A^m$ is the longest sequence among all CIS of $A^1, A^2, \ldots, A^m$. Let σ be the number of distinct numbers among the m sequences and $\sigma \leq n$. We can see that σ is also an upper bound on the length of the LCIS. Without loss of generality, we can assume that all m sequences are constructed from the alphabet set $\{1, 2, \ldots, \sigma\}$ because we can always map those σ distinct numbers (in ascending order) to $\{1, 2, \ldots, \sigma\}$.

The study of the LCIS problem originated from two classical subsequence problems, the *longest common subsequence (LCS)* and the *longest increasing subsequence (LIS)*. The LCS problem for two sequences of n elements can be solved easily in $O(n^2)$ time, by applying a simple dynamic programming technique. However, the only $o(n^2)$ result known so far was that by Masek and Paterson [10] which runs in $O(n^2/\log n)$ time. For some special cases, Hunt and Szymanski [6] gave a faster algorithm that runs in $O((r+n)\log n)$ time, where r is the total number of ordered pairs of positions at which the two sequences match. For finding the LIS of a sequence of n numbers, Schensted [11] and Knuth [7] gave an $O(n \log n)$ time algorithm. If the input sequence is a permutation of $\{1, 2, \ldots, n\}$, both the algorithms by Hunt and Szymanski [6], and Bespamyatnikh and Segal [1] run in $O(n \log \log n)$ time. The LCIS problem has not been studied until recently that Yang, Huang and Chao gave an $O(n^2)$ time algorithm [14] to find the LCIS of two sequences of n numbers. If the length of the LCIS, ℓ, is small, Brodal et al [2] gave a faster algorithm which runs in $O(n\ell \log \log n + Sort(n))$ time where $Sort(n)$ is the time to sort n numbers. For $m \geq 3$ sequences, Brodal et al [2] presented another algorithm that runs in $O(\min(mr^2, r \log^{m-1} r \log \log r) + m \cdot Sort(n))$ time where r is the number of m-tuples of positions at which the m sequences match.

Our results: Consider m sequences where each sequence consists of n numbers. Let r be the number of m-tuples of positions at which the m sequences match. We give a general approach in solving the LCIS problem for m sequences. For $m = 2$, we give two efficient implementations which run in $O(r \log \ell \log\log n + Sort(n))$ and $O((n\ell + r) \log\log n + Sort(n)))$ time where $Sort(n)$ is the time to sort n numbers. For $r < \min(n^2/(\log \ell \log\log n), n\ell/\log \ell)$, our algorithm has a time complexity better than the previous algorithms. For $m \geq 3$, we give a straightforward implementation which runs in $O(mr^2 + m \cdot Sort(n))$ time. In addition, we show that it is possible to improve the running time in some cases, by applying a dynamic data structure for the orthogonal range query problem [13], to $O(r \log \ell \log^m r + m \cdot Sort(n))$. Table 1 gives a summary of the results. Essentially, the LCIS problem of arbitrary number of sequences can be proved NP-complete by applying the same NP-complete proof for the LCS problem of arbitrary number of sequences [9]. Consider our algorithm for general m, it may run in exponential time as r may be as large as n^m. However, for the special cases that the number of matches between the m sequences is relatively small, e.g., when the m sequences are permutations of $\{1, 2, \ldots, n\}$, we have $r = n$ and our algorithms run in $O(n \log n \log\log n)$ time for $m = 2$ and $O(\min(mn^2, n \log^{m+1} n) + m \cdot Sort(n))$ time for $m > 2$.

The rest of the paper is organized as follows. Section 2 presents the algorithm to find the LCIS for two sequences of numbers. In particular, we give a general framework of our algorithm and provide two efficient implementations, with time complexities $O(r \log \ell \log\log n + Sort(n))$ and $O((n\ell + r) \log\log n + Sort(n)))$, respectively. Section 3 explains how the LCIS problem for m sequences of numbers can be solved using the same framework.

	Known results	Results of this paper
$m = 2$	$O(n^2)$ [14] $O(n\ell \log\log n + Sort(n))$ [2]	$O(r \log \ell \log\log n + Sort(n))$ $O((n\ell + r) \log\log n + Sort(n))$
$m \geq 3$	$O(\min(mr^2, r \log^{m-1} r \log\log r) + m \cdot Sort(n))$ [2]	$O(mr^2 + m \cdot Sort(n))$ $O(r \log \ell \log^m r + m \cdot Sort(n))$

Table 1: Summary of results of the LCIS problem for m sequences of n numbers, where ℓ is the length of the LCIS, r is the total number of m-tuples of positions at which the m sequences match, and $Sort(n)$ is the time to sort n numbers.

2 LCIS of Two Sequences of Numbers

Let $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_n$ be the two input sequences of n numbers, we give some definitions. For $1 \leq i \leq n$, denote A_i the prefix of A with the first i numbers, i.e., $A_i = a_1 a_2 \ldots a_i$. Similarly, $B_i = b_1 b_2 \ldots b_i$. We say that (i, j) is a *match* of A and B if $a_i = b_j$, and a_i (or b_j) is called the *match value* of (i, j). We call a match (i', j') *dominates*

another match (i, j) or that (i', j') is a *dominating match* of (i, j) if $a_{i'} < a_i$ and $i' < i$ and $j' < j$. It is a necessary condition for both $a_{i'}$ and a_i (or $b_{j'}$ and b_i) to appear in the LCIS. For every match (i, j), define the *rank* of the match, denoted by $R(i, j)$, to be the length of the LCIS of A_i and B_j such that a_i (and b_j) is the last element of the LCIS. If the length of the LCIS of A_i and B_j corresponding to match (i, j) is k which is greater than 1, then there must be an LCIS of $A_{i'}$ and $B_{j'}$ corresponding to a match (i', j') with length $k-1$ and $i' < i$, $j' < j$ and $a_{i'} = b_{j'} < a_i$. Therefore, $R(i, j)$ can be defined by the following recurrence equation.

$$R(i,j) = \begin{cases} 0 & \text{if } (i,j) \text{ is not a match,} \\ 1 & \text{if } (i,j) \text{ is a match and no match dominates } (i,j), \\ 1 + \max\{R(i',j') \mid (i',j') \text{ is a match dominating } (i,j)\} & \\ & \text{otherwise.} \end{cases} \tag{1}$$

It is easy to see that the length of the LCIS of A and B is $\ell = \max\{R(i, j)\}$.

First we give a straightforward algorithm to compute ℓ (see Algorithm 1 LCIS()). Then, based on the same framework we give an efficient implementation that runs in $O(\min(r \log \ell, n\ell + r) \log \log n + Sort(n))$ time where $Sort(n)$ is the time to sort n numbers. In order to locate all the matches, we first sort A and B individually and identify the matches in a sequential search on the sorted A and B. At the same time, we can list out the matches in ascending order of their values. This process takes $O(Sort(n) + r)$ time. Then, starting from the match with the smallest value to the match with the largest value, each match takes $O(r)$ time to compute its rank by checking all matches for the dominating matches. This straightforward implementation of Algorithm LCIS() which to compute the length of the LCIS takes $O(r^2 + Sort(n))$ times. Note that r can be as large as n^2, which implies a time complexity of $O(n^4)$. Moreover, to retrieve the LCIS we can record for each match its dominating match with the largest rank. Therefore, in the subsequent discussion for efficient implementation we will focus on computing the ranks of the matches only.

Algorithm 1 LCIS(): Find the length of the LCIS of two sequences, A and B.

```
Assume that we have all matches between A and B in ascending order of their values
for each match (i, j), in ascending order of their values do
    Find the match (i', j') that dominates (i, j) and with the largest rank
    Set R(i, j) = R(i', j') + 1
end for
Output max{R(i, j) | (i, j) is a match}
```

2.1 Efficient Implementations

In this section we give two efficient implementations of Algorithm LCIS(), which run in $O(r \log \ell \log \log n + Sort(n))$ and $O((n\ell + r) \log \log n + Sort(n))$ time. Theoretically, by run-

ning the two implementations in parallel, we actually have an $O(\min(r \log \ell, n\ell+r) \log \log n + Sort(n))$ time algorithm.

Both implementations speed up the step for finding a dominating match of a given match with the largest rank, which are based on a property described below. Consider the execution of Algorithm LCIS(). At any time, let M be the set of matches whose ranks have been computed so far. There are at most ℓ (disjoint) partitions of M, namely $M^1, M^2, \ldots, M^\ell$, where M^k contains the matches with rank k. Note that some of the partitions may be empty. Since the ranks of the matches are computed in ascending order of their values, when the rank of a match is to be computed, the ranks of all dominating matches of this match (if exist) should have been computed and those matches should appear in M. The ℓ partitions of M possess a kind of monotone property that helps locating the largest rank dominating match of a given match efficiently. The property is that if no dominating match of a given match exists in a partition, say M^k, then no partition corresponding to rank larger than k contains dominating match of the given match, and this property is shown in the following lemma.

Lemma 1. *Given a match t, if there is no match in M^k dominating t, there is no match in M^x dominating t for all $x \geq k$.*

Proof. If there is a match t' in M^x dominating t, i.e., t' is with rank x, then there is match t'' dominating t' with rank $x-1$, i.e., t'' in M^{x-1}, which also dominates t. The argument can be extended to have a match in M^k that dominates t, which is a contradiction. □

In view of Lemma 1, the search for the largest rank dominating match of a given match t can be carried out in a binary search on $M^1, M^2, \ldots, M^\ell$. First, we determine if there is a dominating match of t in $M^{\ell/2}$ (assuming that ℓ is a power of 2). If there is one in $M^{\ell/2}$, continue the binary search in $M^{\ell/2+1}, \ldots, M^\ell$. Otherwise, continue the binary search in $M^1, \ldots, M^{\ell/2-1}$. The search space can be reduced by half each time and the search terminates when there is only one partition left in the search space.

The framework of the efficient implementation is as follows. For each of the matches t in ascending order of their values, we compute the rank of t by finding the largest rank dominating match of t, using the binary search as described above. Suppose the largest rank among the dominating matches is k. The rank of t is then set to $k+1$ and t is inserted to M^{k+1}.

In order to achieve the desired running time, we need an efficient method to determine if a dominating match of a given match exists in a partition, which exploits more properties of the partitions. Suppose that the ranks of all matches of values no more than v have been computed and those matches appear in M. A match (i, j) in a partition is called a *critical match* if there is no match (i', j') of the same partition with $i' \leq i$ and $j' \leq j$. For each partition, we only consider the critical matches, and those non-critical matches can be removed. The following lemma shows that, to find in M the dominating match of a given match with value larger than v, it is sufficient to consider just the critical matches.

Lemma 2. *Suppose that M contains all matches of values no more than v. Given a match t of value larger than v, for any partition M^k, if there is a match in M^k dominating t, then there is also a critical match in M^k dominating t.*

Proof. Suppose that $t = (i, j)$ and there is a match (i', j') in M^k dominating t but (i', j') is not a critical match. Then $i' < i$ and $j' < j$ and there must be a critical match (i'', j'') in M^k with $i'' \leq i'$ and $j'' \leq j'$. It implies that (i'', j'') dominates t. □

For each partition, to determine if there is a match in the partition that dominates a given match (i, j), it is sufficient to verify if the critical match (i', j') of the partition with the maximum j' that $j' < j$ (or equivalently the maximum i' that $i' < i$) dominates (i, j) or not. This property is shown in the following lemma.

Lemma 3. *Suppose that M contains all matches of values no more than v. Given a match (i, j) of value larger than v, for any partition M^k, if there is a match in M^k dominating (i, j), then the critical match (i', j') in M^k with the maximum j' that $j' < j$ also dominates (i, j).*

Proof. We prove by contradiction. Suppose that there is a match (i^*, j^*) in M^k dominating (i, j) but the critical match (i', j') in M^k with the maximum j' that $j' < j$ does not dominate (i, j). Then we have $i^* < i \leq i'$ and $j^* \leq j' < j$ that implies (i', j') is not critical, which is a contradiction. □

In order to maintain only the critical matches in a partition, before we insert a new match (i, j) of rank k into M^k, we should remove in M^k those matches (i', j') that will become non-critical, i.e., $i \leq i'$ and $j \leq j'$. See Figure 1 for an example. The following lemma shows which particular critical match of the partition we should verify to see if the match will become non-critical or not.

Lemma 4. *Given a match (i, j), if there is any critical match (i', j') in M^k with $i \leq i'$ and $j \leq j'$, then the critical match (i^*, j^*) in M^k with the minimum j^* that $j^* \geq j$ satisfies the property that $i \leq i^*$ and $j \leq j^*$.*

Proof. We prove by contradiction. Suppose that $i \leq i'$ and $j \leq j'$ and the critical match (i^*, j^*) with the minimum j^* that $j^* \geq j$ has $i^* < i$. We can see from $i^* < i \leq i'$ and $j \leq j^* \leq j'$ that (i', j') is not a critical match, which is a contradiction. □

For each of M^k with $1 \leq k \leq \ell$, we employ the data structure *van Emde Boas (vEB) tree* [12] to store the critical matches in M^k. A vEB tree is a data structure that stores a set of integers, and supports all the operations insert, delete, and search in $O(\log \log n)$ time where n is the largest integer to be inserted in the vEB tree. In addition, the search operation can return for a given number its nearest numbers, i.e., for a given number j, the largest number $j' < j$ and the smallest number $j' \geq j$. In our implementation each critical match (i, j) is indexed by the value j in a vEB tree. We define four operations specific to our problem. Each operation can be implemented in $O(\log \log n)$ time.

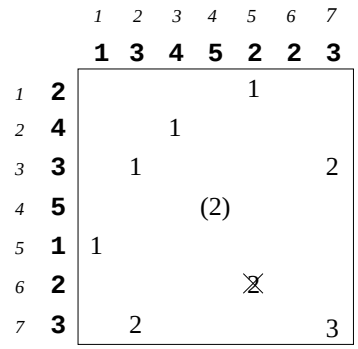

Figure 1: With two sequences 2435123 and 1345223, the ranks of the critical matches, before and after match $(4, 4)$ is considered, are shown. Match $(4, 4)$ finds two dominating matches $(2, 3)$ and $(3, 2)$, which are with the largest rank 1. Then match $(4, 4)$ has rank 2 and match $(6, 5)$ becomes non-critical.

- $FindRight(k, (i, j))$ returns the match (i', j') in the vEB tree corresponding to M^k with the minimum j' that $j' \geq j$;
- $FindLeft(k, (i, j))$ returns the match (i', j') in the vEB tree corresponding to M^k with the maximum j' that $j' < j$;
- $Insert(k, (i, j))$ inserts the match (i, j) in the vEB tree corresponding to M^k; and
- $Delete(k, (i, j)$ deletes the match (i, j) from the vEB tree corresponding to M^k.

In the following two sections we use the above operations to give two implementations of the Algorithm LCIS(), which run in $O(r \log \ell \log\log n + Sort(n))$ and $O((n\ell + r) \log\log n + Sort(n))$ time. Then we have the following theorem.

Theorem 5. *Our algorithm takes* $O(\min(r \log \ell, n\ell + r) \log\log n + Sort(n))$ *time to find the LCIS of two sequences of* n *numbers.*

2.1.1 Implementation I

In the first implementation of LCIS() (see Algorithm 2 LCIS-1()), we begin with σ empty vEB trees, namely T^k for $1 \leq k \leq \sigma$, to store the set of critical matches in M^k for $1 \leq k \leq \sigma$, respectively. Then all matches between A and B are identified. The matches are arranged in ascending order of their values. The matches (i, j) with the same value are arranged in descending order of j and then i. For each of the matches in this order, we find the largest rank dominating match by a binary search on the non-empty vEB trees as described before. To determine if there is a dominating match in T^k, we check if $FindLeft(k, (i, j))$ returns a match (i', j') with $i' < i$ and $j' < j$. After the rank of a match (i, j) is computed, we remove

every match (i', j') in $T^{R(i,j)}$ that becomes non-critical, i.e., $i \leq i'$ and $j \leq j'$. Then (i, j) is inserted to $T^{R(i,j)}$. The match (i, j) is a new critical match in $M^R(i, j)$ by the following lemma.

Lemma 6. *In the Implementation I, when (i, j) is inserted to $T^{R(i,j)}$, (i, j) must be a critical match of $M^{R(i,j)}$.*

Proof. Let $R(i, j) = k$. Suppose on the contrary that there is a match (i', j') in T^k with $i' \leq i$ and $j' \leq j$. If the value of (i', j') is less than that of (i, j), then the rank of (i, j) should be at least $k + 1$, which is a contradiction. If the value of (i', j') equals the value of (i, j), because of the order we consider the matches with the same value, either $j' > j$ or $i' > i$, which is also a contradiction. Finally, match value of (i', j') cannot be larger than that of (i, j) because we consider the matches in ascending order of their values. □

The following lemma proves the time complexity of the Implementation I.

Lemma 7. *The Implementation I runs in $O(r \log \ell \log \log n + Sort(n))$ time.*

Proof. The running time of the Implementation I consists of three parts. (1) It takes $O(Sort(n) + r)$ time to sort the two sequences A and B and identify the matches between A and B in ascending order of their values; (2) It takes $O(\log \ell \log \log n)$ time to find the largest rank dominating match of a given match in the binary search because the function $FindLeft$ is called $O(\log \ell)$ times and there are at most ℓ non-empty vEB trees. Hence it takes $O(r \log \ell \log \log n)$ time to find the ranks of all matches; (3) Each match can be inserted and deleted at most once from a vEB tree. Each of these operations takes $O(\log \log n)$ time, and hence it takes $O(r \log \log n)$ time for all matches. Altogether, it takes $O(r \log \ell \log \log n + Sort(n))$ time. □

2.1.2 Implementation II

In this section we give an implementation of LCIS() (see Algorithm 3 LCIS-2()) that runs in $O((n\ell + r) \log \log n + Sort(n))$ time. In Implementation II we do not apply binary search to find the ranks of individual matches. Rather, we use a sequential search to find the ranks of a group of matches. Later we show that at most n sequential searches are needed.

Similar to Implementation I, we begin with σ empty vEB trees, T^k for $1 \leq k \leq \sigma$, to store the set of critical matches in M^k for $1 \leq k \leq \sigma$, respectively. All matches between A and B are identified and arranged in ascending order of their values, so that we can find the ranks of the matches in this order. For the set of matches with the same value, say H, we divide H into groups such that a group consists of the matches (i, j) in H with the same value of j. Suppose a group consists of the matches $(i_1, j), (i_2, j), \ldots, (i_x, j)$ where $i_1 < i_2 < \cdots < i_x$. For any two of these matches (i_a, j) and (i_b, j) with $a < b$, we have $R(i_a, j) \leq R(i_b, j)$ because any dominating match of (i_a, j) is also a dominating match of

Algorithm 2 LCIS-1(): An implementation of LCIS() using binary search on the vEB trees.

Create σ vEB trees $T^1, \ldots, T^\sigma$ for the critical matches of $M^1, \ldots, M^\sigma$, respectively
Store all matches in ascending order of their values; for the matches (i, j) with the same value, order them in descending order of j and then i
for each match (i, j), in that order **do**
  Apply binary search on the non-empty vEB trees to find the largest k such that $FindLeft(k, (i, j))$ returns a match in T^k that dominates (i, j)
  If no such k, set $R(i, j) = 1$. Otherwise, set $R(i, j) = k + 1$.
  /* Clean up $T^{R(i,j)}$ and insert (i, j) */
  while $FindRight(R(i, j), (i, j))$ returns a match, say (i', j') **do**
    if $i' \geq i$ **then**
      $Delete(R(i, j), (i', j'))$
    else
      Terminate the while-loop
    end if
  end while
  $Insert(R(i, j), (i, j))$
end for
Output $\max\{R(i, j) \mid (i, j)$ is a match$\}$

(i_b, j). Therefore, we can apply a sequential search on T^1 to T^ℓ to determine the ranks of all the matches $(i_1, j), (i_2, j), \ldots, (i_x, j)$. Similarly, a different sequential search can be applied to each of all the other groups of H. After the ranks of the matches with the same value are computed, those matches are inserted to the corresponding vEB trees.

The following lemma proves the time complexity of the Implementation II.

Lemma 8. *The Implementation II runs in* $O((n\ell + r) \log \log n + Sort(n))$ *time.*

Proof. The running time of the Implementation II consists of three parts. Two of them are similar to those of Implementation I. (1) It takes $O(Sort(n) + r)$ time to sort the two sequences and identify all matches in ascending order of their values; (2) Each match can be inserted and deleted at most once to and from a vEB tree, respectively. Hence it takes $O(r \log \log n)$ time for these operations of all matches.

(3) For the remaining part, we analyze the time taken for the sequential searches for the ranks of all matches. Denote the set of matches with the same value v by H_v. Recall that for each H_v, we divide it into groups such that a group consists of the matches (i, j) in H_v with the same value of j. Let d_v be the number of groups in H_v. We have $\sum_{1 \leq v \leq \ell} d_v = n$ because matches (i, j) of different groups should either be with different match values or with different value of j. As we use a sequential search for each group, totally we need n sequential searches. For each sequential search we call $FindLeft$ at most $\ell + g$ times where g is the number of

matches in the group. Summing up for all sequential searches, it takes $O((n\ell + r) \log \log n)$ time. Altogether, the Implementation II takes $O((n\ell + r) \log \log n + Sort(n))$ time. □

Algorithm 3 LCIS-2(): An implementation of LCIS() using sequential search on the vEB trees.

```
Create σ vEB trees T^1, ..., T^σ for the critical matches of the σ partitions M^1, ..., M^σ
Store all matches in ascending order of their values
for v = 1 to σ do
    Let S be the set of matches (i, j) with value v, i.e., a_i = b_j = v
    Let D be the set of distinct values of j for all matches (i, j) in S
    for each y ∈ D do
        Let (i_1, y), ..., (i_x, y) with i_i < ··· < i_x be all matches (i, j) in S with j = y
        Set k = 1
        /* The sequence search */
        for z = 1 to x do
            while FindLeft(k, (i_z, y)) returns a dominating match of (i_z, y) do
                Set k = k + 1
            end while
            Set R(i_z, y) = k
        end for
    end for
    for each match (i, j) in S, in descending order of j and then i do
        while FindRight(R(i, j), (i, j)) returns a match, say (i', j') do
            if i' ≥ i then
                Delete(R(i, j), (i', j'))
            else
                Terminate the while-loop
            end if
        end while
        Insert(R(i, j), (i, j))
    end for
end for
Output max{R(i, j) | (i, j) is a match}
```

3 LCIS of m Sequences of Numbers

In this section we give algorithms to find the LCIS of m sequences of numbers, especially for $m \geq 3$. We first generalize the Algorithm LCIS() for two sequences to m sequences. Let the

m sequences be $A^1, A^2, \ldots, A^m$ where each of them has n elements. Suppose $A^k = a_1^k a_2^k \ldots a_n^k$. Again we can assume that the alphabet is $\{1, 2, \ldots, \sigma\}$ where $\sigma \leq n$. Let A_i^k denote the prefix of A^k with i elements, i.e., $a_1^k \ldots a_i^k$. We generalize the notations *match* and *dominating match* and *rank* as follows. A match is an m-tuple, $(i_1, \ldots, i_m)$ where $a_{i_1}^1 = a_{i_2}^2 = \ldots = a_{i_m}^m$. A match $(\delta_1, \ldots, \delta_m)$ is called a dominating match of another match $(\theta_1, \ldots, \theta_m)$ if $a_{\delta_1}^1 < a_{\theta_1}^1$ and $\delta_i < \theta_i$ for all $1 \leq i \leq m$. The rank of a match $\Delta = (\delta_1, \ldots, \delta_m)$, denoted by $R(\Delta)$, is the length of the LCIS of $A_{\delta_1}^1, A_{\delta_2}^2, \ldots, A_{\delta_m}^m$ such that $a_{\delta_1}^1$ is the last element of the LCIS. $R(\Delta)$ can be defined similar to Equation 1.

We generalize the Algorithm LCIS() to find the rank of each match. It takes $O(r + m{\cdot}Sort(n))$ time to sort the m sequences individually and identify all the matches. It takes $O(m)$ time to determine if a match dominates another match because a match is an m-tuple which contains m integers. For each match, it takes $O(mr)$ time to compute the rank of the match. Hence it takes $O(mr^2 + m{\cdot}Sort(n))$ time to compute the ranks of all matches and also the length of the LCIS of m sequences.[1]

It is possible to improve the running time with the binary search approach, by using data structures for *multidimensional orthogonal range queries* as described below. Given a set of N points in d-dimensional Euclidean space, each associated with a numerical value, design a data structure for the points that supports efficient insertion, deletion, and query of the form: return a point that falls within a hyper-rectangular region $[x_1, y_1] \times [x_2, y_2] \times \ldots \times [x_d, y_d]$. There is a data structure for the above problem that supports each update and query in $O(\log^d N)$ time, using space $O(N \log^{d-1} N)$ [13].

In our problem, a match is considered as a m-dimensional point. The task of finding a dominating match in a partition is in fact a multidimensional orthogonal range query. To determine if there is a dominating match of a match $(i_1, \ldots, i_m)$ in a partition, we could check if there is any match (or point) of the partition that falls within the hyper-rectangular region $[1, i_1 - 1] \times [1, i_2 - 1] \times \ldots \times [1, i_m - 1]$. Then, adopting the binary search approach in Section 2.1, we can find the LCIS in $O(r \log \ell \log^m r + m{\cdot}Sort(n))$ time.

Acknowledgement

We thank Chung-Keung Poon for introducing to us the orthogonal range query problem.

References

[1] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2):7–11, 2000.

[1] Bordal et al [2] improved the time complexity by storing the matches in the data structure by Gabow et al [5]. Then, they can find for each match the dominating match with the maximum rank in $O(\log^{m-1} r \log\log r)$ time [5]. Therefore, the LCIS can be found in $O(r \log^{m-1} r \log\log r + m{\cdot}Sort(n))$ time.

[2] G. S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz. Faster algorithms for computing longest common increasing subsequences. In *Proc. of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 330–341, 2006.

[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27:2369–2376, 1999.

[4] J. S. Deogun, J. Yang, and F. Ma. Emagen: An efficient approach to multiple whole genome alignment. In *Proc. of the Second Asia-Pacific Bioinformatics Conference (APBC)*, pages 113–122, 2004.

[5] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of the 16th ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[6] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[7] D. E. Knuth. *Sorting and Searching, The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[8] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[9] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978.

[10] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[11] C. Schensted. Longest increasing and decreasing subsequences. *Canad. J. Math.*, 13:179–191, 1961.

[12] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. of the 16th Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[13] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32(3):597–617, 1985.

[14] I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93(5):249–253, 2005.