Longest Common Increasing Subsequence: An Enumerative Solution

Chun Lin

October 19, 2020

Abstract

The problem of LCS (longest common subsequence) and LIS (longest increasing subsequence) are both well-solved; the former was proved to be solvable in $O(nm/log{n}) $ \cite{masek} and the later $O(nlog{log{n}})$ \cite{segal}. Recently, the problem LCIS (longest common increasing subsequence) was proposed. While it can be seen as a combination of the two aforementioned problems, it seems difficult to adapt their solutions to the LCIS problem. Most of the approaches to LCIS utilizes dynamic programming and sophisticated data structure to speed up, while we use a simple algorithm to attack the case where the alphabet size is extremely small comparing to the sequence lengths.

1 Introduction

1.1 definition

In this article, we present a solution to the **longest common increasing subsequence** problem: given two integer sequences a and b, with lengths m and n respectively, find any sequence s that satisfies the following two conditions and $|s|$ is the maximum possible: (1) the **common** condition: s is a subsequence of both a and b (2) the **increasing** condition: s is strictly increasing.

1.2 example

For example, consider the following sequences: $a = [2, 3, 4, 6, 8, 4]$, $b = [5, 1, 2, 6, 3, 4]$, the sequence $c = [6, 4]$ is a subsequence to both a and b, but it is not strictly increasing; the sequence $d = [2, 4]$ is a subsequence to both a and b, but it does not constitute a valid answer since there exists a longer sequence $e = [2, 3, 4]$ satisfying both the "common" and "increasing" conditions. One can easily verify that e constitutes a valid answer. In this case, e is one **longest common increasing subsequence** of a and b.

2 Input Restrictions

We assume that the input sequences a and b meet the following conditions:

(1) Each entry of a and b is an integer between $[0, z - 1]$, where z is given as input or a fixed integer.

(2) z is not longer than the word-size under our RAM model. That is, operations such as addition, subtraction or logical-OR between any two number can be treated as $O(1)$ in time-complexity.

Condition (1) can be ensured by preprocessing the input sequences with additional $O(m+n+z \log z)$ time cost, if there are no more than z different integers considering both sequences. This does not affect the overall complexity as we will see.

In short, we intend to efficiently solve the cases where the value set of the sequences is extremely small compared to the lengths of the sequences.

3 Sketch of the Algorithm

3.1 Overview

We will explain the algorithm in two parts: the main procedure **LCIS** and the subroutine **ENUMERATE-IS**. We shall first carefully explain ENUMERATE-IS since it is central to the whole algorithm.

3.2 Procedure: ENUMERATE-IS

3.2.1 Input

A sequence a consisting of integers ranging from 0 to $z-1$.

3.2.2 Output

An array Has of size 2^z that has the information of every **increasing subsequence** (abbreviated as **IS** from now on) of a.

3.2.3 Steps

1. Initialize a boolean array Has of size 2^z. Set $Has[0] \leftarrow true$, then for each integer i from 1 to 2^z-1, set $Has[i] \leftarrow false$. We can show a 1-1 correspondence of every possible increasing subsequence consisting of integers in $[0, z-1]$ to the integers in $[0, 2^z-1]$. That is, we may represent each increasing subsequence in a by an integer in $[0, 2^z-1]$. We intend to update Has such that $Has[i] = true$ if and only if the increasing subsequence represented by integer i has been found in a. It is straightforward to convert an increasing subsequence to its integer representation and back, as will be explained in the "analysis" section. Initially, only $Has[0]$ is set to true because we use 0 to represent an empty sequence, which is indeed an increasing subsequence of a.

2. Initialize z lists: $TODO[0], TODO[1], ..., TODO[z-1]$. Each list can be implemented using any data structure that supports $O(1)$ time insertion and $O(1)$ time deletion (e.g. a dynamic array). It does not matter how the elements within the lists are ordered. For each integer i from 0 to $z-1$, list $TODO[i]$ initially contains a single integer i. The purpose of each list, say $TODO[x]$, is to store every increasing sequence s such that s ends with x, and s is not found in a so far, but $s-x$ (s after deleting its last element) has already been found in a. These lists help us update Has efficiently.

3. The main for loop runs for each position of a. For the ith position, we have to first update the array Has: for each integer $s \in TODO[a_i]$, delete s from $TODO[a_i]$, and set $Has[s]$ to $true$. Then, for every integer $x > a_i$, insert $s + 2^x$ into $TODO[x]$.

4. After the main for-loop terminates, we return the whole array Has.

3.3 Procedure: LCIS

3.3.1 Input

Two sequences a, b consisting of integers ranging from 0 to $z-1$.

3.3.2 Output

An integer representing an LCIS of a, b. We will explain in the "analysis" section how to retrieve a sequence from its integer representation.

3.3.3 Steps

1. Call ENUMERATE-IS(a) and ENUMERATE-IS(b) and let the returned array be Has_a and Has_b respectively.

2. A common increasing subsequence exists if and only if its integer representation s is *true* in both Has_a and Has_b. So all that is left is to, using a simple for-loop, find any such sequence with the maximum possible length. To find the length of the sequence represented by integer s, we just need to calculate how many 1's there are in s's binary representation (This computation is short-handed as "bitcount(x)" in our pseudocode. For now, we assume this can be done in $O(1)$; for implementation details, see section "Analysis").

3. Return the integer representation of the LCIS of a, b that has been found (or one can, alternatively, convert it back to a sequence and return it).

4 Pseudocode

Algorithm 1 Longest Common Increasing Subsequence

procedure ENUMERATE-IS(a) ▷ input: a sequence a
  for $i \leftarrow 0...2^z - 1$ **do** ▷ initialize Has
    $Has[i] \leftarrow false$
  $Has[0] \leftarrow true$
  for $i \leftarrow 0...z - 1$ **do** ▷ initialize TODO
    $TODO[i] \leftarrow \varnothing$
    Push 2^i into $TODO[i]$
  for $i \leftarrow 1...|a|$ **do** ▷ main dynamic programming loop
    for every $s \in TODO[a_i]$ **do** ▷ make new IS using a_i
      Pop s from $TODO[a_i]$
      $Has[s] \leftarrow true$
      for $x \leftarrow (a_i + 1)...(z - 1)$ **do** ▷ update TODO
        Push $s + 2^x$ into $TODO[x]$
  return Has ▷ information of every IS in a
procedure LCIS(a, b) ▷ input: two sequences a, b of lengths m, n
  $Has_a \leftarrow ENUMERATE - IS(a)$ ▷ use the above procedure
  $Has_b \leftarrow ENUMERATE - IS(b)$
  $maxCISid \leftarrow 0$ ▷ records the longest CIS seen so far
  $maxCISlen \leftarrow 0$
  for $i \leftarrow 1...2^z - 1$ **do**
    if $Has_a[i] = true$ & $Has_b[i] = true$ & $bitcount(i) > maxCISlen$ **then**
      $maxCISlen \leftarrow bitcount(i)$
      $maxCISid \leftarrow i$
  return $maxCISid$

5 Analysis

5.1 Correctness

5.1.1 Lemma 1

Let S be the set of all possible increasing sequences consisting of only integers in $[0, z-1]$, where z is some positive integer. Then $|S| = 2^z$.

Proof We prove this theorem by directly establishing a bijection between the S and $[0, 2^z - 1]$. Let s be an increasing sequence consisting of only integers in $[0, z-1]$, and consider an integer r whose ith bit in binary representation is 1 if and only if $i - 1$ is contained in s. It is easy to see that this is indeed a bijection as any increasing sequence maps to exactly one integer, and one can easily map the integer back to the sequence by checking which bits in its binary representation are 1.

And it is also quite easy to see the minimum integer that can be mapped on is 0 while the maximum is $2^z - 1$, respectively representing an empty sequence and the sequence: $(0, 1, 2, ..., z - 1)$. ■

Example. The increasing sequence $(0, 1, 2)$, by our method, maps to 7; while an empty sequence maps to 0.

5.1.2 Theorem 1

The array Has returned by ENUMERATE-IS(a) has $Has[i] = true$ if and only if the increasing sequence that maps to integer i (by the method used in the proof of Lemma 1) is contained in a as a subsequence.

Proof We prove the following statement by induction: after $i \in \{0, ..., |a|\}$ step(s) of the for-loop at line 8, the array Has and each $TODO$ list contain the correct data considering prefix of length i of a.

First we prove the case $i = 0$. Considering the prefix of length 0 of a, which is an empty sequence. Only one increasing subsequence can be found in this prefix, that is the empty sequence - which is mapped to 0 by our method used in Lemma 1. Hence obviously, Has and each $TODO$ list contain the correct information.

Secondly we prove the induction case: assume the statement is true for every $i \in \{0...k\}$, where $|a| > k \geq 0$. Now, $Has[i] = true$ if and only if the increasing subsequence that maps to i is contained in prefix $a_1...a_k$, and for any integer s contained in $TODO[a_{k+1}]$, $Has[s - 2^{a_{k+1}}] = true$ and $Has[s] = false$. Now, we need to update Has and $TODO$ such that they are correct considering the prefix $a_1, ..., a_{k+1}$; and it is sufficient to to check if there is some increasing sequence that is *not* contained in prefix $a_1, ..., a_k$ but contained in $a_1, ..., a_{k+1}$.

Quite straightforwardly, every $s \in TODO[a_{k+1}]$ is a new increasing subsequence after considering one more entry a_{k+1}, by each $TODO$ list's definition. So naturally, for every $s \in TODO[a_{k+1}]$, we need to set $Has[s] \leftarrow true$ and for every $x \in \{a_{k+1} + 1, ..., z - 1\}$, insert $s + 2^x$ into $TODO[x]$. One can see that after such updates, the data structures Has and $TODO[x]$ for $x \in 0, ..., z - 1$ fit their corresponding definitions. ■

5.1.3 Theorem 2

The procedure LCIS(a, b) returns the integer representing some LCIS of a, b.

Proof By Theorem 1, some common increasing subsequence s of a, b exists if and only if its integer representation i is true in both Has_a and Has_b (see line 16, line 17 of Algorithm 1). By simply enumerating through every entry of Has_a and Has_b, one can find a longest common increasing subsequence for certain. ■

5.2 Time and Space Complexity

5.2.1 Theorem 3

The time and space complexity of ENUMERATE-IS(a) are $O(|a| + 2^z)$.

Proof Any of the possible 2^z increasing sequences is deleted from some $TODO$ list at most once: when it is at first found in a; and any sequence s is inserted into some $TODO$ list at most once: when the sequence made by deleting s's last entry is found in a.

This can be proved by contradiction: suppose during the course of the procedure, some sequence is pushed into some $TODO$ list twice, we name the first sequence to be pushed twice into some $TODO$ list s. s must have length greater than 1, because all possible sequence of length 1 is pushed only at the initialization stage. Deleting the last element of s gives another non-empty increasing sequence s'. Note that s' must be inserted into some $TODO$ list twice before s does, because the first time s is inserted, s' is deleted from the $TODO$ lists; thus, to insert s once again, s' must be inserted twice before s does. This obviously contradicts the assumption that s is the first such sequence to be inserted

twice. And it follows straightforwardly that if no sequences can be inserted twice, then no sequences can be deleted twice. Thus, we have proven the total time and space complexity on the operation of $TODO$ is $O(2^z)$.

Similarly, the time complexity on the updates of array Has must also be $O(2^z)$, because except for the empty sequence, every sequence s is deleted from some $TODO$ list when $Has[i]$ is set to $true$, where i is s's integer representation. So the total time and space complexity are both $O(|a| + 2^z)$. ■

5.2.2 Theorem 4

The time and space complexity of ENUMERATE-IS(a) are $O(|a| + |b| + 2^z)$.

Proof This procedure requires calling ENUMERATE-IS(a) and ENUMERATE-IS(b), which makes it already $O(|a| + |b| + 2^z)$ in time and space before the main for-loop at line 20. Now we prove that the main for-loop's time complexity is $O(2^z)$.

For the function *bitcount(x)*, one possible implementation is to simply tabulate the bit count of each integer $i \in [0, 2^z - 1]$ (for detail, see Algorithm 2). One can verify that for any positive integer x, the integer $(x\&(x-1))$ is exactly x without its least significant bit in binary representation ("&" represents binary AND operation). Initially, we record the bit count of 0 to be 0, then for each integer successively, the bit count can be obtained by simply calculating $bitcount(x\&(x-1)) + 1$.

Thus, we can spend $O(2^z)$ time and space in precomputation stage to store the bit count of each integer in $[0, 2^z - 1]$, that makes the main for-loop (between line 20 and 23) $O(2^z)$ in time. ■

We provide an implementation of *bitcount(x)* in pseudocode:

Algorithm 2 counting bits in binary representation

```
1: procedure TABULATE                          ▷ this function should be called before LCIS
2:     bits[0] ← 0                             ▷ this array should be accessible by function LCIS
3:     for i ← 1...2^z − 1 do
4:         bits[i] ← bits[i&(i − 1)] + 1
5: procedure BITCOUNT(x)                       ▷ input: a non-negative integer x
6:     return bits[x]
```

6 Alternative Algorithm

In this section, we provide another straightforward *dfs*-based LCIS algorithm. This algorithm may be advantageous in that it does not require an additional array to record the bit-count of each integer, and although its time complexity is still $O(|a|+|b|+2^z)$, it does not branch into sequences whose prefixes are not *all* common increasing subsequences of a, b.

In cases where there are only very few common increasing subsequences, this algorithm may run a lot faster than Algorithm 1 in that it does not iterate through all 2^z possible candidates.

Algorithm 3 Longest Common Increasing Subsequence

```
1: procedure DFS(pos, now, len)        ▷ input: position, current common increasing subsequence, current length
2:     if pos = z then
3:         if len > maxCISlen then maxCISlen ← len maxCISid ← now
4:         return
5:     if Has_a[now + 2^pos] = true and Has_b[now + 2^pos] = true then
6:         DFS(pos + 1, now + 2^pos, len + 1)
7:     DFS(pos + 1, now, len)
8: procedure LCIS(a, b)                ▷ input: two sequences a, b of lengths m, n
9:     Has_a ← ENUMERATE − IS(a)       ▷ use the above procedure
10:    Has_b ← ENUMERATE − IS(b)
11:    maxCISid ← 0                    ▷ records the longest CIS seen so far
12:    maxCISlen ← 0
13:    DFS(0, 0, 0)
14:    return maxCISid
```

7 Conclusion

In this article, we have presented an $O(|a| + |b| + 2^z)$ algorithm for the LCIS problem. This algorithm, when z is the order of $O(\log(|a| + |b|))$, is optimal in time complexity.