The Longest Common Increasing Subsequence Problem

Yongsheng Bai [1], Bob P. Weems [2]

[1] Department of Biology, The University of Texas at Arlington, Box 19498, Arlington, Texas 76019-0498, USA
[2] Department of Computer Science & Engineering, The University of Texas at Arlington, Box 19015, Arlington, TX 76019-0015, USA

Abstract

By reviewing various existing *Longest Increasing Subsequence* (*LIS*) and *Longest Common Subsequence* (*LCS*) methods, the *Longest Common Increasing Subsequence* (*LCIS*) problem is explored in a progressive way. A version of finding LCIS that uses quadratic time and space O(*mn*) is designed by taking the idea of ordinary dynamic programming with traceback. An optimal space efficient version, which runs in linear space O($m + n$), where m and n are the lengths of two sequences, is designed by using the recursive method and compared with the previous quadratic time and space version. A newer version (linear space with $m\log m$ cost for combining subsolutions) that uses both recursive and augmented tree structure technique to reduce the cost for combining subsolutions from O(m^2) to O(m log m) time in solving the LCIS problem is also designed and implemented.

Keywords: Longest increasing subsequence; Longest common subsequence; Longest common increasing subsequence; Dynamic programming

1. Introduction

Gries [4] stated precisely that after deleting m (not necessarily adjacent) values from the sequence $V_0, \ldots, V_{n-1}$, a *longest upsequence* or *Longest Increasing Subsequence* (*LIS*) of length $n - m$ is obtained if all values in this new sequence are in non-decreasing order and m is minimized. The problem of the LIS has been discussed by many researchers. Let L_n be the length of the LIS in a random permutation of $\{1, \ldots, n\}$, the expected value of L_n is known to be bounded as $(2 - o(1))\ n^{1/2} \le E[L_n] \le 2n^{1/2}$ [9]. In this formula, the lower bound was proved by Logan and Shepp [10]; Kerov and Vershick [8] derived the upper bound and a polynomial form ($n^{-1/6}$) in n for o(1). According to Rains [11], the distribution of the length of the LIS can be expressed exactly in terms of the moments of the trace of a random (uniformly distributed) unitary matrix. Erdos and Szekeres [2] proved that a sequence with n elements must have either an increasing or decreasing subsequence with $n^{1/2}$ elements. The running time complexity O(n log n) for the algorithm of computing the LIS of a sequence has been proved to be essentially optimal by Fredman [3].

A *Longest Common Subsequence* (*LCS*) is a common subsequence of maximum length. For example, if sequence X_1 is 1, 6, 7, 5, 7, 8 and sequence X_2 is 6, 4, 5, 8, 9; then 6, 5, 8 is the LCS for these two sequences. The classic dynamic programming method to solve LCS problem in quadratic time and space O(*mn*) is illustrated by Hirschberg [6], where m and n are the lengths of sequences A and B. Hunt and Szymanski [7] developed an efficient algorithm by examining only a sparse number of matches between the two sequences. Their algorithm takes O(($r + m$) log n) time where r is the total number of ordered pairs of positions at which the two sequences match. A linear space algorithm for computing the LCS was proposed by Hirschberg [5]. Furthermore, Hirschberg [6] developed two other algorithms for the LCS problem. In his first algorithm, O($pn + n$ log n) time is required where p is the length of the LCS. O($p(m + 1 - p)$ log n) time is obtained in his second algorithm.

A *Longest Common Increasing Subsequence* (*LCIS*) is a common subsequence that has the maximum length in the set of all common subsequences where the elements in each common subsequence are in non-decreasing order. For example, if sequence Y_1 is 1, 6, 2, 3, 5, 7, 8 and sequence Y_2 is 6, 2, 4, 5, 8, 9; then 2, 5, 8 is the LCIS for these two sequences. The LCIS of two given strings is not necessarily unique. Even though the LIS and LCS problems have been thoroughly explored, the LCIS problem has received little attention beyond some small applications in image processing.

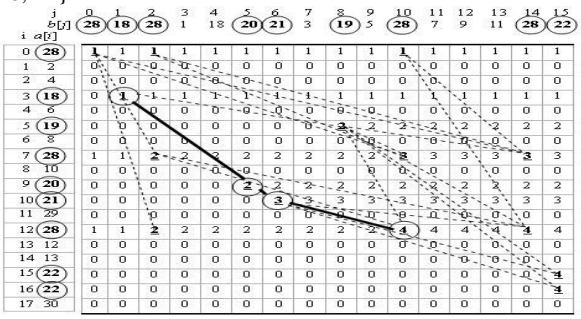
In this paper, a version of finding the LCIS, which takes quadratic time and space O(*mn*), is designed by taking the idea of ordinary dynamic programming with traceback to extract the matches. An optimal space efficient version, which runs in linear ($m + n$) space, where m and n are the length of two sequences

respectively, is designed by using the recursive method and compared with the previous quadratic time and space version. A newer version (linear space with O(m log m) cost for combining subsolutions) that uses both recursive and augmented tree structure technique to reduce the cost for combining subsolutions from O(m^2) to O(m log m) time in solving the LCIS problem is designed and implemented.

2. Space-Inefficient (*mn* space) version

Suppose the first sequence labels the rows of a matrix, and the second sequence labels the columns of a matrix. A row's *longest dominating chain* (*LDC*) value is defined as the maximum length of common increasing subsequence for matching the value in this row for the first sequence and possible matched value with the highest index in the column for the second sequence, essentially in the "folklore" version of LCS. The idea of this version is that each row's LDC value is updated as the column index is increased, that is, once the LDC value of a row is updated, the LDC values for the later instances of columns beyond current position in the same row will be copied from this updated LDC value. Specifically, through comparing the value in a row with the element in a column, the maximum length of common increasing subsequence for matching the value in a row and the element in a column, which is the LDC of the "slot" up to this intersection position, is recorded and traceback pointers are also saved in 2-dimensional arrays during the process. The new common pair value will be chosen as the candidate LCIS value only if it is greater than or equal to the previous selected common pair value and will make the common increasing subsequence up to this intersection position longest. Both time and space in this version cost O(mn). An example of this version is shown in Table 1.

Table 1. The LDC value table for sequence a {28, 2, 4, 18, 6, 19, 8, 28, 10, 20, 21, 29, 28, 12, 13, 22, 22, 30} and sequence b {28, 18, 28, 1, 18, 20, 21, 3, 19, 5, 28, 7, 9, 11, 28, 22}



Note: The connection of circled numbers is the LCIS path produced by the computer program.

3. Recursive (*m* + *n* space) version

This version uses the technique of recursively partitioning one problem into two subproblems to compute LCIS lengths. The pseudocode for generating left subproblems ("seam") of calculating the LCIS of the entire first sequence and the first n/2 elements of the second sequence is shown in Figure 1, where n is the length of the second sequence. The pseudocode for generating the right "seam" of calculating the LCIS of the entire first sequence and the last n/2 elements of the second sequence is similar to the pseudocode of generating the left subproblem presented previously besides the second (column) sequence is scanned from the rightmost (end) until the element with index n/2 is reached, and the first (row) sequence is scanned from the end to the beginning in this case. During the computation of generating the right subproblem, if the element in the first sequence is smaller than its compared element in the second sequence, the rest of the calculating steps in the current loop will be skipped, and the operation for the next element in the first sequence will start. At the end of processing, this version works through merging the "vertical seam" to get the LCIS length. The algorithm for merging seams and recursion is illustrated in Figure 2.

```
** a[0..m-1], b[0..n-1] are the input sequences. seam[i].LDC denotes the LDC value for row i. seam[i].CN denotes the correspond column number which has the matched value with the element in row i. supportLDC is the variable to temporarily store calculated LDC value. supportRow is the variable to temporarily store row number. supportColumn is the variable to temporarily store column number **
1: Initialize all rows' LDC values to 0;
2: for j ← 0 to n/2 − 1 do
   begin
     set supportLDC to 0; set supportRow to −1; set supportColumn to −1;
3:   for i ← 0 to m − 1 do
     begin
4:     if (a[i] ≤ b[j])
        { if (a[i] = b[j])
            { if (seam[i].LDC < supportLDC + 1)
                {set seam[i].LDC to the sum of supportLDC and 1; set seam[i].CN to j;}
              else if (seam[i].LDC > supportLDC)
                {set supportLDC to seam[i].LDC; set supportRow to i; set supportColumn to seam[i].CN;}
            }
          else if (seam[i].LDC > supportLDC)
            {set supportLDC to seam[i].LDC; set
```

```
supportRow to i; set supportColumn to seam[i].CN;}
        }
      end for
    end for
```

Fig. 1: Calculating the LCIS of the entire first sequence and the first n/2 elements of the second sequence.

```
** newLength is a temporary variable to store the length
value of LCIS. iSave is the index for the value in
sequence 1 which serves as the upper bound value of the
LCIS for row sequence in the new left subproblem. jSave
is the index for the value in sequence 1 which serves as
the lower bound value of the LCIS for the row sequence
in the new right subproblem. arrLeft[i].LDC denotes the
calculated row i's LDC value in the left subproblem.
arrRight[j].LDC denotes the calculated row j's LDC
value in the right subproblem. lcis[i] is the array to save
the elements of the LCIS **
1: Initialize newLength to 0, iSave and jSave to −1;
2: for i ← 0 to m − 1 do
   begin
3:   if (arrLeft[i].LDC > newLength)
       {set newLength to arrLeft[i].LDC; set iSave to i;
set jSave to −1;}
4:   if {arrRight[i].LDC > newLength)
       {set newLength to arrRight[i].LDC; set iSave to
−1; set jSave to i;}
   end for
5: for i ← 0 to m − 1 do
   begin
6:   for j ← i + 1 to m − 1 do
     begin
7:     if (a[i] < a[j] and arrLeft[i].LDC +
arrRight[j].LDC > newLength)
         {set newLength to the sum of arrLeft[i].LDC and
arrRight[j].LDC; set iSave to i; set jSave to j;}
     end for
   end for
8: if (iSave ≥ 0 and arrLeft[iSave].LDC ≥ 1)
     {set lcis[arrLeft[iSave].LDC − 1] to a[iSave]; copy
values in the row sequence for the left subproblem;
      copy values in the column sequence for the left
subproblem;
      recursively generate and merge seams in the left
subproblem;}
9: if (jSave ≥ 0 and arrRight[jSave].LDC ≥ 1)
     {copy values in the row sequence for the right
subproblem;
      copy values in the column sequence for the right
subproblem;
10:   if (iSave ≥ 0 and arrLeft[iSave].LDC ≥ 1)
        {set lcis[arrLeft[iSave].LDC] to a[jSave];
         recursively generate and merge seams in the
right subproblem;
        } else
        {set lcis[0] to a[jSave]; recursively generate and
merge seams;} }
```

Fig. 2: Merging seams and recursion.

The main ideas of the recursive version are shown in Figure 3. Specifically, 1) The LCIS of the first $n/2$ elements of the second sequence and the entire first sequence can be calculated using the usual NW to SE (↘) cost computation; 2) The LCIS of the last $n/2$ elements of the second sequence and the entire first sequence is calculated through the symmetric (SE to NW, ↖) cost computation; 3) The indices (*iSave*, *jSave*) with the maximum sum of their correspondent values will be used to retrieve LCIS elements, this sum is also the length of the LCIS; 4) The LCIS element can be found through scanning top-to-bottom across the left seam for the topmost entry index *i'Save* with the same correspondent value as index *iSave*; 5) Another LCIS element can be found through scanning bottom-to-top across the right seam for the bottommost entry index *j'Save* with the same correspondent value as index *jSave*; 6) Copy values in both sequences for the left subproblem. For the left subproblem, only the elements whose values are smaller than or equal to *a*[*iSave*] in both sequences need to be copied. The copied end element (position) stops at this *a*[*iSave*] for the row sequence or equivalent value element for the column sequence; 7) Copy values in both sequences for the right subproblem. For the right subproblem, only the elements whose values are greater than or equal to *a*[*jSave*] in both sequences need to be copied. The copied beginning element (position) starts from this *a*[*jSave*] for the row sequence or equivalent value element for the column sequence; 8) Recursively call for each of the two subproblems. This version runs in O(*mn*) time and O(*m* + *n*) space.

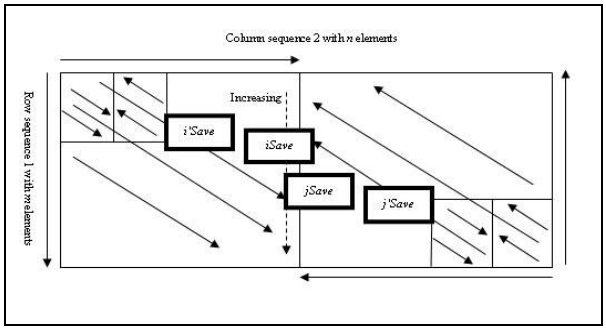


Fig. 3: The recursive version of LCIS.

4. Recursive and tree version

This version uses augmented binary tree ideas in [1] to speed-up merge from O(*mn*) to O(*m* log *m*).

Specifically, for each tree node, the element value, its index number (i), the LDC value for this element in the left partition ($arrLeft[i].LDC$), the maximum sum of $arrLeft[i].LDC$ and $arrRight[j].LDC$ (max_LDC), augmented subtree's maximum sum of the LDC values ($augmented\_max\_LDC$), and the correspondent element's index which determines this augmented subtree's maximum sum of LDC (max_i) are used to construct its fields. The binary search tree property is maintained by the element value $a[i]$, but the index i is used to break ties. Preorder tree walk method is used to load array-based tree. The index of the root is labeled as 1, the indices for parent, left child, and right child of the element at node i will be $i/2$, $2i$, and $2i+1$, respectively. The max_LDC and max_i for each subtree should be kept, and $a[i]$ from tree for right partition is deleted. For each deletion, the updating is done by taking the ideas of getting the max_LDC and max_i from left and right subtree nodes and continuing updating max values in ancestors. Finally, finding j such that $i < j$, $a[i] \le a[j]$, and sum of the LDC value for node i in the left partition and the LDC value for node j in the right partition is maximized by using $augmented\_max\_LDC$ value after searching the whole tree's nodes. The sample 3-node tree structure and the updating idea for this algorithm are illustrated in Figure 4. In this version, the cost of combining subsolutions (single seam processing) can be reduced to O(m log m). The whole procedure still takes O(mn) time since total $n/2$ combining operations are required.

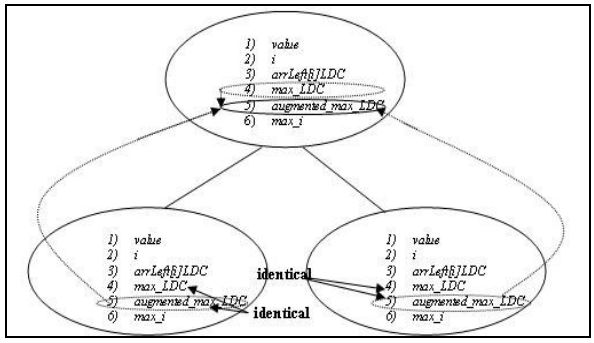


Fig. 4: The sample 3-node tree structure and the updating idea of recursive and tree version. The length of final LCIS is obtained by taking the maximum value of root's fourth field, its left child node's fifth field, and its right child node's fifth field.

5. Results

In mn space version, the LCIS is found using dynamic programming with traceback method, but in $m + n$ space version, the recursive technique of partitioning one problem into two subproblems and combining subsolutions is used. Our test machine is a DEC AlphaServer 4000 running Unix Tru64 (Digital Unix) with two 64-bit Alpha processors 21164 5/400 MHz and 1GB of RAM. Some sample test results for performance comparisons of different cases and different versions are listed in Figure 5 and Figure 6. The format for each test case is written as "(size for each sequence).(case number).dat." For example, *2000.1.dat* denotes the input sequence sizes are both 2000 where all elements have identical value; *2200.2.dat* denotes the input sequence sizes are both 2200 where elements have possible values between 0 and $m - 1$; *2400.3.dat* denotes the input sequence sizes are both 2400 where elements have possible values between 0 and $(m - 1)/10$; *2600.4.dat* denotes the input sequence sizes are both 2600 where elements have possible values between 0 and $(m - 1)/25$.

- *The total running time for test cases of $m + n$ space version using recursive technique of partitioning one problem into two subproblems to compute LCIS lengths are less than those of mn space version where the LCIS is found through dynamic programming with traceback method.* Since it is not necessary to save traceback pointers in the $m + n$ space version, the $m + n$ space version saves approximately 20~75% running time for the input sequences whose sizes range from 2000 to 5000
- *For the case that all sequence elements have identical value, test cases for both mn space and $m + n$ space version take more running time than other cases do (51~70% for mn space version and 28~40% for $m + n$ space version), but the $m + n$ space version saves approximately 26~48% running time for the case that all sequence elements have identical values compared to this version's contribution to other cases of the input sequences whose sizes range from 2000 to 5000*. The reason for this observation is because the worst-case number of traceback pointers is saved for mn version in the case that all sequence elements have identical value
- Because there was no significant difference in performance within the 5000 range, the version using both recursive method and augmented tree technique was tested only for input sequences whose sizes range from 5000 to 20000. *For the input sequences whose sizes range from 5000 to 20000*, t*he version using both recursive method and augmented tree technique significantly reduces approximately*

29% of the total running time over that of the version which only uses the recursive method. This is due to the reduction of time cost in doing seam processing in the version that uses both recursive method and augmented tree technique. The seam processing for this version is done by finding the new lower and upper value for two subproblems respectively by searching the tree. The time spent searching a tree node is O(log *m*) (the height of tree). So the seam processing time for the version that uses both recursive method and augmented tree technique is O(*m* log *m*)

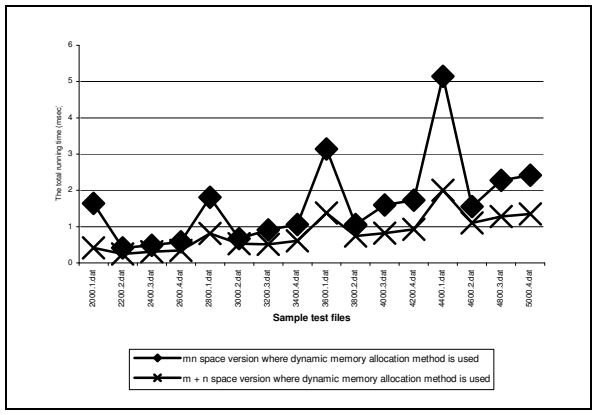


Fig. 5: The time cost results for some sample test cases for *m* + *n* space version with dynamic memory allocation vs. *mn* space version with dynamic memory allocation method.

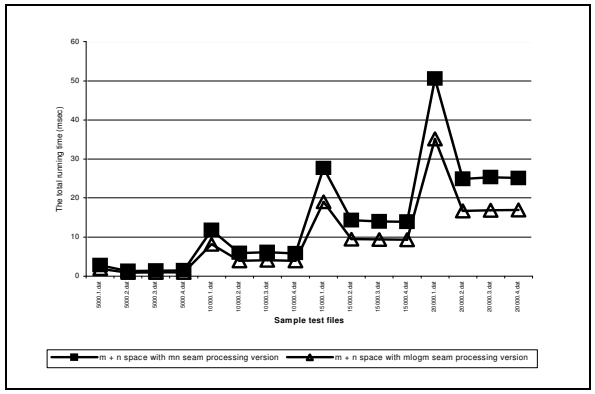


Fig. 6: The time cost results for some sample test cases for the version that only uses the recursive method and the version using both recursive and augmented tree technique.

6. Conclusions

A *mn* space version of finding the LCIS, which takes quadratic time, is designed by using the idea of dynamic programming with traceback to extract the matches. An optimal $m + n$ space (linear) version is designed by using the recursive method and compared with the previous *mn* space version. A newer version which takes O(*mn*) time and O($m + n$) space by using both recursive and augmented tree structure technique to reduce one single seam processing process from O(m^2) to O(m log m) time to solve the LCIS problem is designed and implemented. This version, combining the two techniques together, significantly reduces the merge time for seam processing when it is compared to the $m + n$ space version only using the recursive method; in that case, the seam processing time is O(*mn*). These results illustrate that the recursive method and the augmented tree technique can be used to solve the LCIS problem to achieve reduced time and space.

7. References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms (2nd ed.), *MIT Press, Cambridge, Mass.*, pp. 302–316, 2001.

[2] P. Erdos and G. Szekeres, "A combinatorial problem in geometry," *Composition Mathematica*, 2, pp. 463–470, 1935.

[3] M.L. Fredman, "On computing the length of longest increasing subsequences," *Discrete Math.*, 11, pp. 29–35, 1975.

[4] D. Gries, The Science of Programming, *Springer-Verlag, New York*, pp. 259–262, 1981.

[5] D.S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, 18 (6), pp. 341–343, 1975.

[6] D.S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, 24 (4), pp. 664–675, 1977.

[7] J.W. Hunt and T.G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, 20 (5), pp. 350–353, 1977.

[8] S. Kerov and A. Vershick, "Asymptotics of the plancherel measure of the symmetric group and the limiting form of young tableaux," *Dokl. Akad. Nauk SSSR*, 233, pp. 1024–1028, 1977.

[9] J.H. Kim, "On increasing subsequences of random permutations," *J. Comb. Theory*, A (76), pp. 148–155, 1996.

[10] B. Logan and L. Shepp, "A variational problem for random young tableaux," *Adv. Math.*, 26, pp. 206–222, 1977.

[11] E.M. Rains, "Increasing subsequences and the classical groups," *J. Combin.*, 5, 1998.