LIS using backtracking and branch-and-bound approaches

Article · December 2016

DOI: 10.1007/s40012-016-0108-x

2 authors, including:

Seema Rani
dronacharya govt. college, gurugram , india

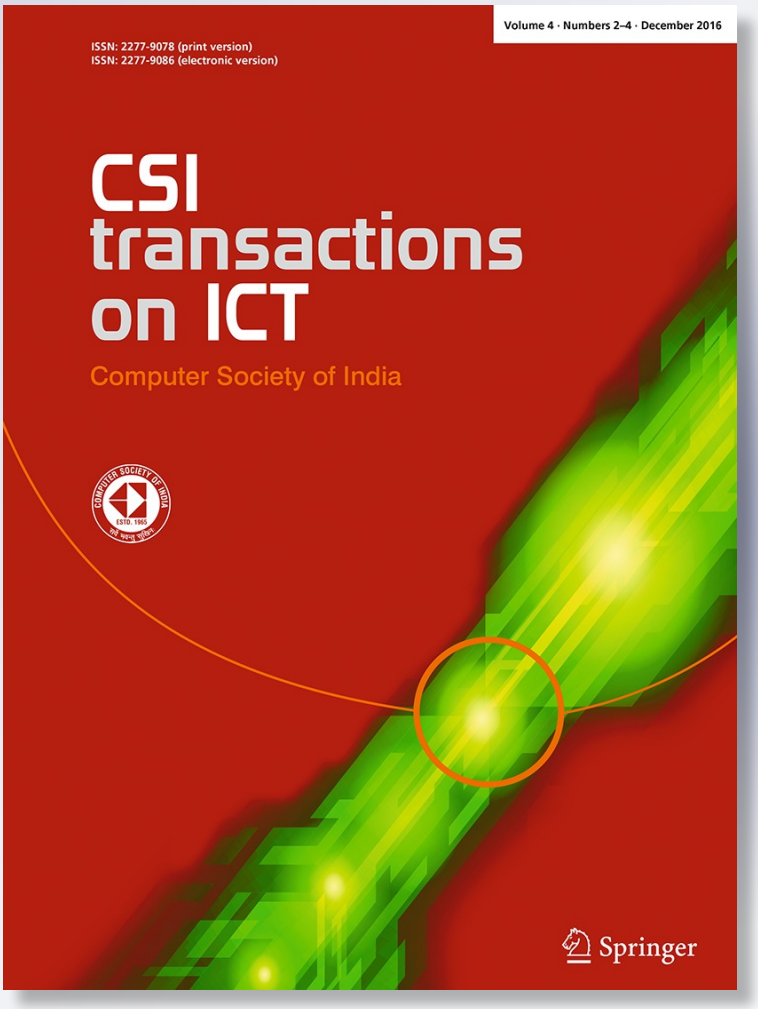
LIS using backtracking and branch-and-bound approaches

Seema Rani & Dharmveer Singh Rajpoot

CSI Transactions on ICT

ISSN 2277-9078
Volume 4
Combined 2-4

CSIT (2016) 4:87-93
DOI 10.1007/s40012-016-0108-x



Springer

CSIT (December 2016) 4(2–4):87–93
DOI 10.1007/s40012-016-0108-x

SPECIAL ISSUE REDSET 2016 OF CSIT

LIS using backtracking and branch-and-bound approaches

Seema Rani[1] · **Dharmveer Singh Rajpoot**[2]

Published online: 1 December 2016

Abstract Finding the longest increasing subsequence and its length from a sequence of finite integers is an NP-hard problem. Many significant efforts have been put to provide solutions to this problem with time complexity O(n log n) (n is the size of sequence), $O(n^2)$, O(n log log k), O(n) (using parallel processing) and more. In this paper we provide conceptual views of LIS and its solution using two approaches—backtracking and branch-and-bound. Its implementation using backtracking approach takes time $O(2^n)$ and the other solution based on the concept of branch-and-bound approach takes $O(n^2)$ time. Both solutions are efficient than the bruit force approach.

Keywords Backtracking · Branch-and-bound · Breadth first search (BFS) · Depth first search (DFS) · Longest increasing subsequence (LIS)

1 Introduction

Let $A = <X_1, X_2, X_3, \ldots X_n>$, then increasing subsequence of A is a sequence $<k_1, k_2, k_3, \ldots k_d>$ such that each $k_i < k_{i+1}$ and each k_i precedes k_{i+1} in the original A sequence. An increasing subsequence which has maximum length is LIS.

✉ Seema Rani
seema.fet@mriu.edu.in

Dharmveer Singh Rajpoot
dharmveer.rajpoot@jiit.ac.in

[1] Department of Computer Science, FET, Manav Rachna International University, Faridabad, India

[2] Department of Computer Science, Jaypee Institute of Information Technology University, Noida, India

This problem can be viewed in other way as finding the longest common subsequence (LCS) between A and its sorted sequence. That is, the solution to LIS problem can be used to find the LCS between two strings assuming one (which is having greater domain) string is defining the order of the symbols appearing in both strings. The applications of LCS include data compression, file comparison, and pattern recognition [1].

It has become scientific interest to completely sequence the genomes which are similar to other already sequenced genomes. The LIS is used in the process of whole-genome alignment to find the longest chain of MUMs (maximal unique match is a subsequence which exists exactly once in both sequences) appearing in both the genomes A (already sequenced) and B (to be sequenced) in the same order. Thus, by using LIS, it becomes easy to identify the gaps in B [2]. This problem is also related to Young Tableaux, which is useful in Schubert calculus and representation theory. Symmetric and general linear groups can be conveniently represented using Young Tableaux [3]. Thus, LIS has become practically important problem.

In this paper, we represent this problem using the concepts of Backtracking (DFS technique) and branch-and-bound (BFS technique) approaches. There exist certain problems which have multiple feasible solutions or a single optimal solution and are solved step by step. While solving such problems, we may reach at some state which may not lead to a solution or there may exist other feasible solutions and we feel there is a need to go back to the previous state and explore the solution from that previous sate. Such problems can be solved using backtracking approach. This is a DFS technique where one state is explored completely before exploring the next state. The other strategy which we are using for solving LIS is branch-and-bound. In this approach, the solution of a problem changes its state from

one state X to other multiple states where each state corresponds to one alternate from X. From all these states, only some of the states are explored further and rests are ignored (each problem has different strategy to find which states are to be explored and which are to be ignored). Throughout the paper we use n to denote the size of the sequence in question.

2 Related work

The LIS problem has been considered a standard example of dynamic approach [4]. Remarkable efforts have been made to solve this problem and its variants using different approaches. Many efficient optimal solutions (using single and multiple processors) and approximate solutions exist.

The LIS problem was first defined and addressed in 1961 by Schensted [5]. He proposed a solution with O(n log n) time. Fredman [6] gave another optimal solution using decision tree technique and proved that no more than (n − L) log L + O(n) comparisons are required to compute the LIS (L is the length of LIS). Alam and Rahman [7] provided a solution with O(n log n) time using a divide-and-conquer approach by maintaining a *parent* of each element. Bespamyathnik and Segal [8] improved Fredman's approach by using the data structure created by van Emde Boas [9] and provided a solution with O(n log log n) time. Crochemore and Porat [10] divided the sequence into m ($|LIS| \leq m$) blocks. And they provided a solution with O(n log log m) time by applying Fredman's approach sequentially on these blocks and renaming the elements.

Saks and Seshadhri provided an approximate solution with δn ($\delta < 0$) additive error having $(\log n)^c(1/\delta)^{o(1/\delta)}$ running time. The authors also output a $(1 + \Gamma)$-approximation ($\Gamma > 0$) to the distance to monotonicity $\in_f$ ($\in_f = 1 - |LIS|/n$) by selecting an appropriate splitter and then improving the approximation recursively [11].

There exist certain variants of this problem. Elmasry [12] provided a longest almost increasing subsequence (LaIS) in O(n log k) time. LaIS is a subsequence where each ith element is greater than X − c (X is the largest element among the first i − 1 elements, c is a small constant). Albert et al. [13] have found LIS for windows of fixed size w. They created a tableau and data structures for the first window and found the corresponding LIS. To find LIS for all other windows they simply maintained the previous tableau and data structures instead of creating the new ones. They achieved the solution in O(n log log n + OUTPUT) time (OUTPUT is the sum of the lengths of LISs of all the windows). Chen et al. [14] have also worked on this variant, but they also considered the windows of variable sizes. Windows of variable sizes were formed by either discarding element(s) from the front of the present window or by considering successive element(s) from the sequence as part of the next active window or both. The authors partitioned the elements on the basis of their heights. The height of any element h(e) is the size of the longest chain with e as an ending element. The elements having the same height h belong to an antichain L_h. They achieved the solution in O(n + OUTPUT + $\sum$Di) time (Di is the cost of discarding ith element) by creating and maintaining the predecessor and successor data structures for each element. Another variant of LIS is LICS, which is finding the LIS considering all the rotations of a given sequence. Albert et al. [15] achieved the solution in O(n3/2 log n) time. They identified a set Ź of elements and for each element of Ź, they computed the LICS. The LICS is computed by combing the LIS of the subsequence X (X is formed by ignoring the elements smaller than x (x ε Ź)) and the Longest Decreasing Subsequence (LDS) of the subsequence Y (from right to left) (Y is formed by ignoring the elements larger than x). Sebastian Deorowicz [16] proposed another solution to LICS problem in O(min (nl, n log n, l3 log n)) time, where l is the length of the LICS. Their solution was based on two important observations: (1) It is not necessary to work on all the rotations, and (2) combining the two precomputed covers is efficient than finding the cover from the scratch. Tseng et al. [17] have provided solutions to two other variants: (1) The LIS having minimum height (height is the difference between the first and the last elements of the LIS) with O(n log n) time, and (2) The LIS which contains a given sequence C as its subsequence with O(n log(n + |C|)) time.

3 Problem definition

If A is a finite sequence of integers, then an increasing subsequence (IS) K of A is a subsequence such that the elements in K are in increasing order and for each pair $<p, q>$ (p < q) of consecutive elements in K, p appears before q in A.

Let $A = <$ X1, X2, X3, … Xn$>$, then an increasing subsequence of A is a subsequence $K = <k_1, k_2, k_3, \ldots k_d>$ such that each $k_i < k_{i+1}$ and each k_i precedes k_{i+1} in the original A sequence. The longest increasing subsequence is an increasing subsequence having maximum length. For ex. If A = [48, 30, 34, 46, 39, 35, 36, 42, 45], then the possible increasing subsequences are:

- [48]
- [30, 34, 46]
- [30, 34, 39, 42, 45]
- [30, 34, 35, 36, 42, 45]

The last IS among the above ISs is the LIS.

4 Proposed work

In this paper, we represent this LIS problem using backtracking and branch-and-bound approach and provide the solutions.

4.1 Backtracking approach

Backtracking approach can be used to solve problems which search for a set of feasible solutions or for an optimal solution satisfying some criteria. The solution is generally expressed in the form of a sequence $<x_1, x_2, x_3, \ldots x_d>$ where the x_i's are chosen from some finite set depending on the problem being solved [18]. For example, the solution of n-queens problem is a sequence of n elements where each element is of the form (X, Y): the set of all first elements and that of all second elements from these pairs are both permutations of <1, 2, 3…n>.

Backtracking approach is based on DFS search, where in case if there exists different/multiple alternates for the current state of the problem being solved, then this approach explores one alternate completely (until a solution is found or it is confirmed that further exploration will not lead to a feasible/optimal solution) before proceeding to the next alternate. This process continues till an optimal solution is found or all the feasible solutions have been found. That is, while solving a problem, its state changes from one state to the next state and so on until one of the feasible solutions is found, an optimal solution is found, or a failure is detected. In case a feasible solution is found or a failure is detected, the concept of backtracking takes the problem one step back and further explores from this step. This process continues till all the alternate solutions are explored.

We conceptualize the LIS problem using backtracking approach. We want to find the desired solution such that (1) each element in the output must belong to the input sequence. (2a) And each element *a,* which is appearing before some other element *b* in the solution must be smaller than *b*. (2b) Also *a* must appear before *b* in the input sequence. We therefore, state the explicit constraint as the constraint (1) and implicit constraint as the constraints (2a) and (2b) mentioned above.

There are two possibilities, either an element from the input sequence is included in the solution or is excluded from the solution. An element X_h of the sequence is included if it is greater than the last element of the output sequence formed till now. Through this statement our (1) and (2a) constraints are met. To meet the constraint (2b), we will process the elements in the same order in which they are appearing in the input sequence.

An element X_h of the sequence is included if it is greater than the last element of the output sequence formed till now. It is not possible to find the desired LIS if we *always* follow this protocol. Since its inclusion may sometimes become an obstacle in finding larger increasing subsequence. So we consider both alternates (inclusion and exclusion) to generate all feasible ISs. We state the modified protocols—(a) An element X_h of the sequence becomes the next element (included) only if it is greater than the last element of the output sequence formed till now. (b) X_h is not included. That is, X_h is excluded whether it is smaller or greater than the last element of the partially formed output sequence. The exclusion of the element when it is greater than the last element of the partially formed solution generates the alternate solution set. These protocols are applied to each and every element of the input sequence one by one to generate all the possible increasing subsequences.

Suppose *K[1…p]* is one instance of longest increasing subsequence from first *t* elements of input sequence *A* of length n, then

$$\mathrm{K}[1\ldots\mathrm{p}+1],\ \text{if } \mathrm{K}[\mathrm{p}]<\mathrm{A}[\mathrm{t}+1]\ (\text{i.e. } \mathrm{K}[++\mathrm{p}]=\mathrm{A}[\mathrm{t}+1])$$
$$\text{and}\quad \mathrm{K}[1\ldots\mathrm{p}]\ (\text{whether } \mathrm{K}[\mathrm{p}]<\mathrm{A}[\mathrm{t}+1] \text{ or not})$$

are the next instances.

From each instance, at most two instances are being generated. An algorithm LIS_BT implements this approach. Initially beg = 1 and last = n.

The state space generated while executing this approach for data set A = [2, 9, 6, 8] is shown below in Fig. 1. There are five arguments: (1) original array, (2) specifies the first index of the remaining array, (3) specifies the last index of the remaining array, (4) output sequence partially formed so far and (5) length of the solution achieved so far. The left and right branches correspond to inclusion and exclusion of the element respectively. The state representing the exclusion of the last element is not considered since it is not required.

Through this approach we find all the increasing subsequences, then to find the LIS, we compare the length of all the solutions and a solution having the maximum length is actual LIS.

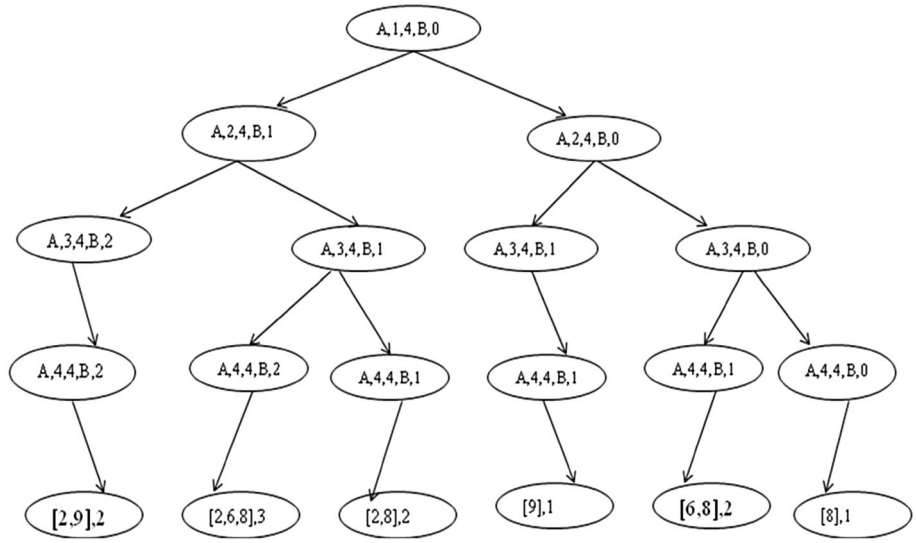


Fig. 1 State space A = <2, 9, 6, 8>

```
LIS_BT(A,  beg,  last,  C, len)
{
 //A is input sequence.
 //beg and  last are starting and last index of remaining
 //sequence respectively, C is an instance of feasible
 // solution of length len

  C[0] =0;
  static int  final =0;
  if ( beg == last)
     if  (A[last] > C[len])
        len = len +1;
        C[len] = A[last];
     if ( final =0 or final  <  len)
        final = len;
        copy entire C list into another B list;
  else
     if (C[len] < A[beg])
        C[++len] = A[beg];
        LIS_BT(A,  beg+1,  last,  C, len);
     LIS_BT(A,  beg+1 ,  last,  C, len);
  }
```

4.2 Improvement using branch-and-bound approach

The time required for backtracking (DFS) approach is $O(2^n)$. We can improve its time complexity by using BFS approach and by providing a bound function on each level. For each instance (state) of the problem, we generate left and right braches in the same way as we generated in the previous approach, i.e. the left and right branches correspond to inclusion and exclusion of the element respectively. We create the state space, where each state of the problem instance is represented by the following three components:

i. *min_len:* size of the IS achieved till now.
ii. *max_len:* the size of the longest IS possible by further exploring this instance, i.e. max_len = -min_len + the number of elements not processed yet.
iii. *last_val:* the last element of this instance.

An Algorithm LIS_BB implements branch- and-bound approach. We are considering each element of the input

one by one. From each instance at (i − 1)th level and the *i*th element of the input sequence, at most two new instances (states) are generated at *i*th level. At each level i, we maintain at most one state corresponding to each *l*, $0 \le l \le$ i. When there are more than one states corresponding to a particular length, then we consider the best among them and *ignore* the rest. We *discard* the states which can never lead to an LIS. The state space using this approach is shown in Fig. 2. The dotted ellipse represents *ignored* instance and dashed ellipse represents *discarded* instance. We record an index which leads to maximum length and the value of the maximum length achieved for each position *i* in the input sequence, where the *i*th element is the last element in the IS. These recorded values are accessed in reverse order to find the actual LIS. We are not actually exploring *all* the states at each level as we are ignoring and discarding some of the states.

```
LIS_BB(A,  1,  n)
{
  //A is input sequence of length n.
  //C[i] = 1 or 0, if an IS of length i exists or not
  //respectively. X[i].min_len, X[i].max_len and
  //X[i].last_val denote three components of state space
  //having IS of length i.
  //Max_Index is the position of the last element of LIS
  //(computed till now) in input sequence. Le[i] is the
  //length of the LIS having i^th element of input
  //sequence as the last element.

  C[0] =1;   max_length = 0;
  X[0].min_len=0;  X[0].max_len=n;  X[0].last_val=0;
  for i = 1 to n  do   C[i] = 0;
  for i = 1 to n  do
   k = max_length;
   for q = max_length to 0 step -1 do
     if (C[q] !=0)
        if (X[q].last_val < A[i])
           if ( C[q+1] = 0 or X[q+1].last_val > A[i])
              C[q+1] = 1;
              X[q+1].min_len  =  q+1;
              X[q+1 ].max_len  = q+1+n-i;
              X[q+1].last_val  = A[i];
           If ( k < q+1)
              k = q+1;
              Max_Index = i;
           Le[i]=q+1;
           Break;
        Else
           X[q].max_len  = X[q].max_len  -1 ;
   if (max_length < k)
      max_length = k;
   for  p = q to 0 do (decrease p)
      X[p ].max_len  = X[p ].max_len  -1 ;
      if (X[p ].max_len  ≤ X[max_length].min_len)
         C[p] = 0;
}
```

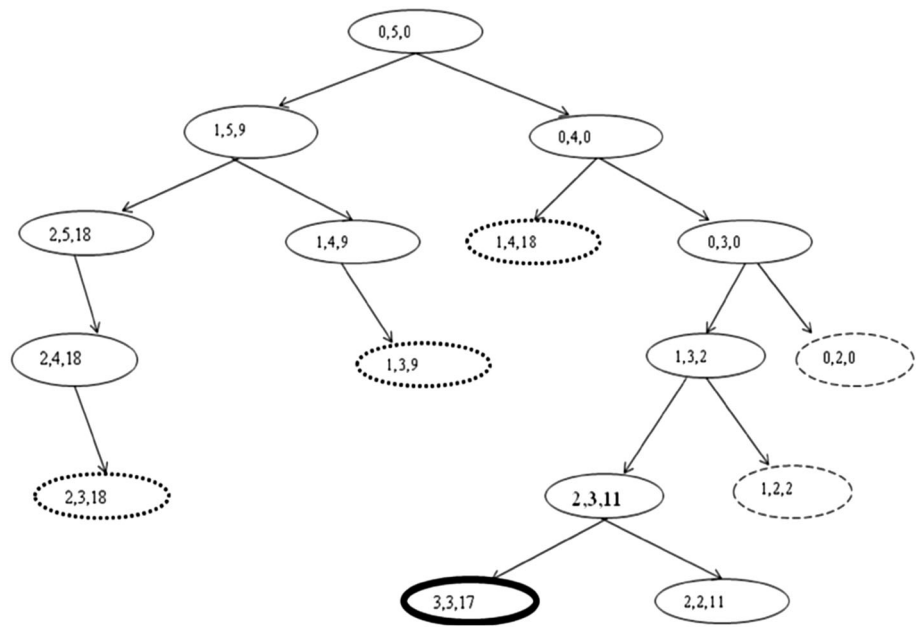


Fig. 2 State space A = <9, 18, 2, 11, 17>

Consider the instances at level (i − 1)th and *i*th element of the input sequence : Starting from the instance having the maximum value of the min_len, find an instance *X* which has the maximum value of min_len and whose last_val is less than the *i*th element. Update X.min_len = X.min_len + 1, X.last_val = A_i, and Y.max_len = Y.max_len − 1, ∀Y and Y ≠ X.

Suppose P and Q are two instances, the bounding function works as follows: If P.min_len = Q.min_len, then *ignore* P if P.last_val > Q.last_val otherwise *ignore* Q. Since, among the two instances of same min_len, the one having the smaller last_val is a better instance to explore further.

If for any instance P, the min_len achieved is the maximum length achieved so far and if ∃Q, such that Q.max_len ≤ P.min_len, then *discard* the Q instance. Because through instance P we have already achieved what we can achieve by exploring Q further in best situation.

5 Analysis and comparison

The solution using backtracking approach runs in $O(2^n)$ time in worst case when all the elements in the sequence are in increasing order because for each instance, two new instances will be generated at the next level. It takes O(n) time in best case when the elements are in decreasing sequence. In this situation only one new instance will be generated at the next level. The space requirement is (n) and (1) in worst and best cases respectively. In an average case, the time complexity of this approach is $O(2^n)$ and space requirement is (l) (l is the length of LIS). The worst and best cases are same for the BFS approach. The implementation using branch-and-bound approach improves the solution and it runs in $O(n^2)$ time in the worst case when all the elements in the sequence are in increasing order and in O(n) time in best case when the elements are in decreasing sequence. The space requirement for this approach is (nl) and (1) in worst and best cases respectively. In an average case, the time complexity of this approach is O(nl) and space requirement is (l) (l is the length of LIS). In best and average cases branch-and-bound approach is more efficient than dynamic approach. Dynamic approach takes $O(n^2)$ time in all the situations (best, worst and average).

6 Conclusion

The LIS problem has generally been viewed as an example of dynamic approach. In this paper we have conceptualized this problem using Backtracking (DFS technique) and branch-and-bound (BFS technique) approaches. By using backtracking approach, the implemented solution runs in $O(2^n)$ time in worst case when all the elements in the sequence are in increasing order and in O(n) time in best case when all the elements are in decreasing sequence. The space requirement is (n) and (1) in worst and best cases respectively. In an average case, the time complexity of this approach is $O(2^n)$ and space requirement is (l) (l is the length of LIS). The implementation using the branch-and-bound approach improves the solution which runs in $O(n^2)$ time in worst case when all the elements in the sequence are

in increasing order and in O(n) time in best case when all the elements are in decreasing sequence. The space requirement for this approach is (n) and (1) in worst and best cases respectively. In an average case, the time complexity of this approach is O(nl) and space requirement is (l) (l is the length of LIS). In best and average cases branch-and-bound approach is more efficient than dynamic approach. Dynamic approach takes $O(n^2)$ time in best, worst and average cases.

Although, better solutions using other approaches already exist, we believe our backtracking and branch-and-bound views of LIS are interesting and we hope that enthusiastic and experienced researchers will further explore the LIS problem from these angles.

References

1. Tsai YT (2003) The constrained longest common subsequence problem. Inf Process Lett 88(4):173–176
2. Delcher AL et al (1999) Alignment of whole genomes. Nucl Acids Res 27(11):2369–2376
3. Lascoux, A, Bernard L, Jean-Yves T (2002) The plactic monoid. In: Algebraic Combinatoric on Words, pp 10
4. Cormen TH (2009) Introduction to algorithms. MIT Press, Cambridge
5. Schensted C (1961) Longest increasing and decreasing subsequences. Can J Math 13(2):179–191
6. Fredman ML (1975) On computing the length of longest increasing subsequences. Discrete Math 11(1):29–35
7. Alam MR, Rahman MS (2013) A divide and conquer approach and a work-optimal parallel algorithm for the LIS problem. Inf Process Lett 113(13):470–476
8. Bespamyatnikh S, Segal M (2000) Enumerating longest increasing subsequences and patience sorting. Inf Process Lett 761:7–11
9. van Emde B (1977) Peter: preserving order in a forest in less than logarithmic time and linear space. Inf Process Lett 6(3):80–82
10. Crochemore M, Porat E (2010) Fast computation of a longest increasing subsequence and application. Inf Comput 208(9):1054–1059
11. Saks M, Seshadhri C (2010) Estimating the longest increasing sequence in polylogarithmic time. In: Foundations of computer science (FOCS), 51st Annual IEEE symposium on IEEE, pp 458–467
12. Elmasry A (2010) The longest almost-increasing subsequence. In: Computing and combinatorics. Springer, Berlin, pp 338–347
13. Albert MH et al (2004) Longest increasing subsequences in sliding windows. Theor Comput Sci 321(2):405–414
14. Chen E, Yang L, Yuan H (2007) Longest increasing subsequences in windows based on canonical antichain partition. Theor Comput Sci 378(3):223–236
15. Albert MH et al (2007) On the longest increasing subsequence of a circular list. Inf Process Lett 101(2):55–59
16. Deorowicz S (2009) An algorithm for solving the longest increasing circular subsequence problem. Inf Process Lett 109(12):630–634
17. Tseng CT, Yang CB, Ann HY (2009) Minimum height and sequence constrained longest increasing subsequence. J Internet Technol 10(2):173–178
18. Horowitz, Sahni, Rajasekaran (1998) Fundamentals of computer algorithms. Galgotia Publica tions Pvt. Ltd.