Journal of Discrete Algorithms 48 (2018) 17–26

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Improvised divide and conquer approach for the LIS problem

Seema Rani [a,b,*], Dharmveer Singh Rajpoot [c]

[a] Jaypee Institute of Information Technology University, Noida, India
[b] FET, Manav Rachna International University, Faridabad, India
[c] Computer Science and Engineering Deptt, Jaypee Institute of Information Technology University, Noida, India

ARTICLE INFO

Article history:
Received 17 November 2016
Received in revised form 13 October 2017
Accepted 29 January 2018
Available online 6 February 2018

Keywords:
Divide-and-Conquer (D&C)
Longest Increasing Subsequence (LIS)
Principal Row of Young Tableaux (PRYT)

ABSTRACT

There exist many optimal (using single and multiple processors) and approximate solutions to the longest increasing subsequence (LIS) problem. Through this paper, we present the enhancement to the divide-and-conquer approach presented in paper [1]. An improved D&C algorithmic solution is proposed which outputs optimal solution in all cases. The proposed algorithm takes $O(n \log n)$ time in best and average cases and $o(n \log^2 n)$ time in worst case. The portion of the proposed solution can run in parallel using multiprocessors.

1. Introduction

If X is a finite sequence of integers then Y is an increasing subsequence of X iff all elements in Y are in ascending order and for every pair of elements x and y which appear in X and Y, if x precedes y in Y then x also precedes y in X. The one which has maximum number of elements (longest) among all increasing subsequences is LIS.

The LIS problem plays a significant role in certain important fields. It is related to the Longest Common Subsequence (LCS) problem and can be restated as computing the LCS of a sequence and its sorted sequence. Any solution which solves the LIS problem can be applied to solve the LCS problem. While applying the LIS solution to LCS problem, it is to be assumed that one string which is having more distinct symbols defines the order of symbols. The LCS can be used in pattern recognition, file comparison and data compression techniques [17].

The interest in Genome study is increasing day by day. Scientists work on a variety of organisms to sequence the genome by comparing it with other similar genomes which are already sequenced. In whole-genome alignment process MUMs is identified using the LIS. MUMs (Maximum Unique Match) is a common subsequence which appears exactly once in each genome. Thus LIS helps to find the regions of the genome in question which are having gaps [9]. This problem has also been associated with Young Tableaux [14].

In this paper, we consider the D&C approach to solve the LIS problem. In D&C approach a problem of size n is divided into one/more smaller sub problems. These sub problems are solved individually and then the solutions of these sub problems are combined to solve the original problem [13]. In our approach we divide the problem of size n into two sub problems of equal size (if n is even) or one having size $\lfloor n/2 \rfloor$ and the other having size $\lfloor n/2 \rfloor + 1$. Elements of these sub sequences are processed independently to find the corresponding LISs. While combining the two solutions we use Knuth's mechanism to compute the Principal Row of Young Tableau (PRYT). Individual sub problems can be solved in parallel using different set of processors. However, the solution of two sub problems can be combined only after solving both sub problems. We discuss

* Corresponding author.
E-mail addresses: seema.fet@mriu.edu.in (S. Rani), dharmveer.rajpoot@jiit.ac.in (D.S. Rajpoot).

https://doi.org/10.1016/j.jda.2018.01.001
1570-8667/

the solution using single processor as well as multiple processors. Throughout the paper we consider – the given sequence as X, its size as n, X_1 and X_2 as first and second sub problems respectively, x_i as the ith element of X, length or size of a sequence refers to the number of elements in that sequence.

We are presenting an approach which has already been proposed earlier by Alam and Rahman [1]. It is really interesting. These authors are the first one to solve the LIS problem using D&C approach. In this approach, the data set is divided into two sub problems recursively until the size of each sub problem reduces to one. Each sub problem is solved individually. Then to find the solution of the corresponding problem, the solutions of both sub problems are combined by maintaining the principal row of Young Tableau. The problems with this paper [1] are: first, when sequence is exactly a permutation of $[1 \ldots n]$, it gives optimal solution for some of these permutations, but there are some permutations for which this approach fails to output optimal solution (a particular case is not handled correctly, discussed later in point c) of case 2); second, there is gap in the provided algorithmic solution and their proposed approach and third, the proposed approach is designed to work with the data sequences which are exactly permutations of $[1 \ldots n]$. In our paper, 1) we improved the approach addressed in [1] so that it works optimally with all permutations of $[1 \ldots n]$ and provide correct LIS, 2) we provide correct algorithm for this improved approach, and 3) this algorithm also works with general data sets, that is, whether a data sequence is a permutation of $[1 \ldots n]$ or not. Also we have discussed the sequences which contain multiple occurrences of some elements.

1.1. Principal row of Young tableaux

Knuth mechanism determines the principal row of Young Tableau as follows: A sequence of finite integers X is given. Another subsequence Y (PRYT) is generated. Y has initially one element (first element of X). For each element x starting from the second element of X, a least index of Y is found such that the element at this index is greater than x. Replace the element at the index found with x. In case if such an index is not found, place the element at the end of the subsequence. Ex. $X =< 2, 9, 6, 8, 3, 4, 7, 10 >$, the principal row (Y) is $< 2\ 34\ 7\ 10 >$. The position of x in Y indicates the |LIS| which ends with x. Ex. the LIS which ends with 4 is [9,14,16] and its length is 3 which is same as the position of 4 in Y. The data structure PRYT keeps track of the smallest element such that it is the last element of LIS of length m, $1 \leq m \leq n$.

2. Related work

Initially the LIS problem was addressed using dynamic approach which always outputs an optimal solution. Thereafter, this problem has been viewed and solved by many enthusiastic and experienced researchers using creative and intelligent methods.

In 1961, Schensted [16] has worked on multiple subsequences using the concept of Young Tableaux and addressed the increasing and decreasing subsequences. Fredman [12] proposed an approach on the basis of Knuth mechanism which determines the PRYT. He improved the maintenance of the principal row. For each element x (which is being processed), to find a least index in PRYT such that the element at this index is greater than x, the author reduced the number of uses of binary search. According to his approach, the element x is first compared with the last element of the PRYT, if it is not greater, only in that case, the approach uses the binary search method to find the desired index. The x replaces the element at the index found. Thus he proved an upper bound $((n - L) \log L + O(n))$ on the comparisons needed to find the LIS. Alam and Rahman [1] used divide-and-conquer approach. The problem is divided into two sub problems of almost equal sizes recursively such that the first sub problem contains the smaller elements from the original sequence and the rest form the second sub problem. After solving the sub problems individually, their solutions are combined using Fredman's approach to find the LIS. They proposed an algorithm that runs in $O(n \log n)$ time. Bespamyathnik and Segal [4] addressed this problem by combining Fredman's method with another efficient data structure formed by Boas [18]. Their proposed solution has the time complexity $O(n \log \log n)$. Crochemore and Porat [8] proposed another solution which takes $O(n \log \log m)$ time. According to this approach, the given sequence is divided into blocks of size m, $m \geq k$ ($k = |\text{LIS}|$). Each block needed to be sorted. But sorting each block individually is inefficient. In order to sort the elements of each block, the authors adopted a strategy. They represent each element of the given sequence by a pair <block_number, element>. Radix sort is applied to these pairs lexicographically. Thus all elements which belong to the first block are in ascending order and appear before all the elements which belong to any other block and so on. The elements of the first block are renamed according to their positions in the sorted sequence and LIS_1 is found using Fredman's approach for this block. The elements in the LIS_1 are merged with next block, they are renamed and LIS_2 is found using Fredman's approach. This process is repeated till the last block is processed.

Keeping in mind the *huge data sets* available today which need to be processed, Saks and Seshadhri proposed a solution which outputs an approximation to the length of LIS. To achieve a better approximation than the existing ones, the authors represented the data set in a 2-d plane such that x-axis represents the position of the element in the given data sequence and y-axis represents the actual value of the element. A suitable splitter (point in 2-d plane with x and y value) is found such that it belongs to the actual LIS. This 2-d plane is then divided into four parts by drawing two lines – one parallel to x-axis and other parallel to y-axis such that both intersect at the splitter. Then ignoring the left upper part and lower right part, the problem (given sequence) reduces in size. Further this process is carried recursively on the left lower part and upper right part to improve the approximation. The solution has δn ($\delta < 0$) additive error and it takes

$(\log n)^c(1/\delta)^{o(1/\delta)}$ running time. They also found the approximate solution to distance to monotonicity ε_f. The achieved output is $(1+\Gamma)$-approximation ($\Gamma > 0$) to ε_f ($\varepsilon_f = 1 - |\text{LIS}|/n$) [15].

Few important variants of LIS have been presented which are useful in practical life. Sometimes we do not need exactly increasing sequence but we want a sequence which is generally increasing. That is the requirement is to find a Longest almost Increasing Subsequence (LaIS) such that ith element is not necessarily greater than the $(i-1)$th element but its value should be close to the value of $(i-1)$th element. Elmasry [11] has found a solution to LaIS problem which runs in $O(n \log k)$ (k is |LaIS|) time. iElmasry's defines an LaIS as follows: suppose the elements from x_i to x_{i+j} (such that $x_i < x_{i+j}$) of X have been processed and the output sequence LaIS Y contains these elements. Also assume that there does not exist any element which is greater than x_i in Y between x_i and x_{i+j}. Then only those elements in X from x_{i+1} to x_{i+j-1} which are greater than $x_i - c$ (c is a small constant) appear between x_i and x_{i+j} in Y. This LaIS is achieved by finding an LaIS having largest element x_i and this subsequence is formed by finding an LaIS having largest element smaller than x_i and appending x_i to it. To make the process more efficient, the authors store one element corresponding to each length. That is after ith iteration, corresponding to every integer m, $m \leq i$, the authors maintain a record of the elements of the given sequence which is the largest element in the LaIS of length m till ith element. For an element x which is one of the largest elements appearing in the LIS being created, this record helps in identifying the number of successors of x in X (which are smaller than x) which can be placed in LIS after x. Albert et al. [3] provided the solution to LIS of every window having width w. They have used a data structure that maintains LISs for all suffixes of active window. According to the data structure, the ith element is inserted in the first row, if it replaces t, then t is inserted in the second row and so on. Then this data structure is improved by inserting the ith element in every row and by creating a new row with only this ith element. Certain information is analyzed from the data structure of the first window. This information is then used to maintain the data structures for subsequent windows. The time complexity of this approach is $O(n \log\log n + \text{OUTPUT})$ ($\text{OUTPUT} = \sum |\text{LIS}_s|$). Windows having different sizes is another variant. Chen et al. [5] worked on this variant. From an existing window another window is considered by either ignoring the front element(s) or by including next element(s). The authors worked by maintaining a linked list of all those elements which form the same height (height of e means the |LIS| which ends with e). Two data structures *Predecessor* and *Successor* are created and maintained for each element. Whenever window changes these data structures are modified accordingly. Thus the proposed solution takes $O(n + \text{OUTPUT} + \sum D_i)$ time. The time to ignore ith element is D_i). Albert et al. [2] proposed the solution to LICS i.e. finding LIS when the front and end of the sequence is not fixed and only relative ordering is known. The component, t_k^y identifies the last element in the LIS of length k when first y elements of the given sequence are processed. This approach maintains a record of t_k^y components. Using this strategy the authors found the LIS which ends not only with smaller elements but also with other larger elements, and they have used this strategy on two different parts of the sequence. That is, in one part the LIS of particular length ends with smallest possible element and in other part, another LIS starts with a larger element. This leads to better chances of merging the solutions. This approach takes $O(n^{3/2} \log n)$ time. Sebastian Deorowicz [10] also considered this same variant – LICS. The authors believe that LICS can be achieved by finding LISs for some rotations only. They also believe that precompiled covers can be merged more efficiently than finding the cover from the initial stage. Thus LICS is achieved by considering only certain rotations, by merging precompiled covers and comparing them. This approach runs in $O(\min(nl, n \log n, l^3 \log n))$ time. Tseng et al. [6] found the LIS which has minimum height. Minimum height means the sum of difference between every pair of consecutive elements is minimum. The authors achieved MHLIS in two steps: a) By finding the maximum length achievable with ith element as the last element (by using Fredman's approach). b) By maintaining a data structure (with binary search operation) for each length l such that it records the last element of all LISs having length l. Then among these elements, the one which is closest to the ith element with data structure having length one less than the length achieved in first step and (in a) is chosen as its predecessor. This approach takes $O(n \log n)$ time. The authors also output a LIS which contain a particular sequence T as its part. To find the LIS containing a particular subsequence, the authors find the LIS between each pair of consecutive elements from the given constrained sequence. These LISs are then merged to get the sequence constrained LIS. This approach has been improved by initially finding the LISs for all the subsequences by using Young Tableau. The proposed solution runs in $O(n \log(n + |C|))$ time.

3. Problem definition

Y is an increasing subsequence of a finite sequence of integers X such that the elements in Y are increasing and for each pair of elements a and b of Y, if $a < b$, then a appears before b in X.

Suppose X is a given sequence, $< x_1, x_2, x_3, \ldots x_n >$, then $Y =< y_1, y_2, y_3, \ldots y_d >$ is an increasing subsequence of X iff for each i, $1 \leq i < d$, $y_i < y_{i+1}$ and also each y_i appears before y_{i+1} in the given X sequence. An increasing subsequence which is longest among all ISs is LIS. If $X =< 69, 10, 24, 76, 38, 34, 35, 41, 44 >$, then its LIS is $< 10, 24, 34, 35, 41, 44 >$.

4. Proposed work

In this paper, we present the solution of LIS problem using divide-and-conquer approach.

4.1. Divide-and-conquer approach

We divide a problem into two equal size problems when n is even. When n is odd the size of the first sub problem is one less than the size of the second sub problem. The first sub problem contains the smaller elements from the original sequence and the rest form the second sub problem. This process of division continues recursively until the size of the sub problem reduces to one. When there is single element in the sub problem, its *predecessor* is initialized to 0. A *predecessor* is a list such that it stores best predecessor for each index in the given sequence. That is $\text{Pred}[p] = q$ means the predecessor of index p is index q (q is a best predecessor of p if $q < p$ and the length of the LIS which ends at pth position is one more than the length of the LIS which ends at qth position; in case if there are two or more such q, then the position which has smaller element is chosen as p's best predecessor). These p and q are global indexes, that is p and q are original positions in the input sequence. To combine the solutions of two sub problems, two components are considered – 1) their positions in the original given sequence and 2) Knuth's mechanism for maintaining PRYT. The predecessors and |LIS|s of only those indexes which form the second sub problem need updating. Throughout this complete process the position maintaining the longest length of increasing subsequence is recorded. And from this position the given sequence is processed backward to find the actual LIS.

Divide – The first $n/2$ smaller elements form the first sub problem X_1 and the rest form the second sub problem X_2. Each element in the sub problem is represented by two components – $<$ value, position $>$ (its value, and its position in the given sequence).

Now we consider three different types of data sets – 1) The given sequence of size n is exactly permutation of elements in the range $[1 \dots n]$. In this case since we know that the middle element is $\lfloor n/2 \rfloor$. We can simply divide these elements into X_1 and X_2. Each element x is processed one by one and if $x \leq \lfloor n/2 \rfloor$, then it is placed in X_1 otherwise it is placed in X_2. Ex. $X =< 4, 2, 5, 1, 3 >$ then here since $\lfloor 5/2 \rfloor = 2$, $X_1 =< (2, 2), (1, 4) >$ and $X_2 =< (4, 1)(5, 3), (3, 5) >$.

2) We might have data set which may not be exact permutation of range $[1 \dots n]$. Here, to divide the given sequence we need to know the middle value. So we need sorted sequence. From this sorted sequence the middle element is used to divide elements into two sub problems. But such data cases can have two different forms – a) $X =< 3012, 12, 5000, 200 >$ and b) $Y =< 300, 298, 301, 299 >$. Here X contains elements which vary in range $[12 \dots 5000]$. To sort such data which has large range, the best technique to use is a technique having time complexity $O(n \log n)$. Y contains elements where the elements are closer to each other. This situation is actually the practical one, where the values in data sets do not vary much. In such cases the counting sort can be used to sort data with slight modification. Counting sort takes $O(k)$ (when elements are represented such that they lie in the range $[1 \dots k]$).

3) The data set can have multiple occurrences of certain elements. In this case after sorting the elements using any stable technique, we find the value at middle position. But this middle element can have multiple occurrences and the situation is like first sub problem ends with this middle element and second sub problem starts with this middle element. Here we need to count the occurrences of middle element in the first half of the sorted sequence. If this count is b, then only first b occurrences of middle element are included in the first sub problem and rest are included in the second sub problem. For Ex. $X =< 2, 4, 8, 6, 2, 2, 2, 1, 10, 2 >$, $X_{\text{sorted}} =< 1, 2, 2, 2, 2, 2, 4, 6, 8, 10 >$, here the middle element at position 5 is 2. The number of 2s to be included in X_1 is first 4 2s. Thus $X_1 =< (2, 1), (2, 5), (2, 6), (2, 7), (1, 8) >$ and $X_2 =< (4, 2), (8, 3), (6, 4), (10, 9), (2, 10) >$.

During this recursive process of division, when only one element is left, its predecessor is initialized to 0. Thus $\text{Pred}[1, \dots, 10] = [0, \dots, 0]$.

Combine – The solved sub problems are combined to solve the original problem. While considering the whole problem, the order of elements remains same as in the given input sequence. The input to the combine procedure are – the given sequence of the form $<$ (value, ind), (value, ind) $\cdots >$, the middle element(m_e) which is used to divide the given sequence into two sub problems and computed predecessors for each global index in both sub sequences independently. Now when we consider the whole sequence, the predecessors of all those elements which belong to the second sub problem may need to be updated.

To update predecessors we maintain PRYT using Knuth's mechanism which is initially empty. We use a data structure *Knuth* such that it contains two elements – Knuth[i].val and Knuth[i].ind. Knuth[i].val indicates the element of X which is present at ith position in PRYT and Knuth[i].ind indicates the position of that element in X. During the combine phase, we associate with each global index a data structure *is_present* to keep track of its position in Knuth's PRYT. That is is_present[i] $= k$ indicates that ith (global index) element of X is present at kth position in PRYT. If an element is not there in PRYT or it is removed from PRYT then set is_present[i] $= 0$. For example the 4th element of X is 20 and it is present in PRYT at 2nd position, then Knuth[2].val = 20, Knuth[2].ind = 4, and is_present[4] = 2. We insert each element x of X in PRYT. If $x \leq m\_e$, it means $x \in X_1$ otherwise $x \in X_2$ (if m_e occurs more than once, then the number of times this m_e appears in X_1 depends on its occurrence in first half of X_{sorted}). Now since we know whether $x \in X_1$ or $x \in X_2$, both situations are handled in different ways. If an element x belongs to X_1 and predecessor of x is y, then remove the successor of y from PRYT and insert x after y in PRYT. If its predecessor is 0 (x has no predecessor) then replace the first element of PRYT by x. We maintain a variable *longest_LIS_len* (initially it is 0) locally to indicate the length of longest LIS (among all the LISs which end with elements belonging to X_1 and are processed till now). If an element x belongs to X_2 and predecessor of x is y, then remove the successor of y from PRYT and insert x after y in PRYT. If its predecessor is 0 (x has no predecessor) then replace (*longest_LIS_len* + 1)th element of PRYT by x. If x's predecessor is not present in PRYT, then find the largest

```
divide_LIS(X,i,j)
{
if(i==j)
   pred[i] = 0
if(j>i)
      n = j-i +1;
      x = ⌊((j-i+1)/2)⌋;
      m_e = X_sorted[i+x-1];
      for k =1 to n do
          if(x_k.val ≤ m_e)
                insert this element at the end in Y1
          else
                insert this element at the end in Y2
      divide_LIS( Y1 , i, i+x-1);
      divide_LIS( Y2 , x+i, j);
      combine_LIS(X, i, j, m_e);
}
```

Fig. 1. Algorithm divide_LIS.

index in PRYT such that it contains the element which is smaller than or equal to x using binary search. Search this index in PRYT, starting from the index which contains the element of X belonging to X_1 forming the longest LIS till the end of PRYT. Replace the element present at the index next to the found index and update the predecessor of x. In case if such an index is not found, replace the element present at the beginning of PRYT. In both cases whether $x \in X_1$ or $x \in X_2$, if x is inserted at the end of PRYT, then update the index, *max_ind* and maximum length, *max_len* forming the longest LIS globally.

Now consider we are currently at ith element x of X. Two sub problems are combined as follow:

Case 1. If $x \in X_1$: a) and predecessor of x is k which is present at jth position in PRYT. That is, we have Pred$[i] = k$ (i and k are global positions in X) and is_present$[k] = j$. Then replace $(j+1)$th element of PRYT by x (if $(j+1)$th element is not there then place x at $(j+1)$th position by extending PRYT). b) Otherwise if x's predecessor is 0, replace the first element of PRYT by x. Update certain data structures. That is, set is_present[Knuth$[j+1]$.ind$] = 0$ (to indicated the removal of element present already at $(j+1)$th position of PRYT), Knuth$[j+1]$.val $= x$, Knuth$[j+1]$.ind $= i$ and is_present$[i] = j+1$. If *longest_LIS_len* $\leq j+1$, then set *longest_LIS_len* $= j+1$.

Case 2. If $x \in X_2$: a) and predecessor of x is k which is present at jth position in PRYT. Then replace $(j+1)$th element of PRYT by x (if there is no element at $(j+1)$th position then place x at $(j+1)$th position by extending PRYT); b) and if x's predecessor is 0 (x has no predecessor) then replace (*longest_LIS_len*+1)th element of PRYT by x; c) and if x's predecessor (which is non zero) is not present in PRYT (the case which is handled incorrectly in paper [1] – according to the authors of paper [1], x's modified predecessor would be the largest element of X_1 and present in there in PRYT which is not correct), then find the largest index in PRYT (whether it belongs to X_1 or X_2) such that it contains the element which is smaller than or equal to x and replace the element at the index next to this index by x. If search fails to find such index, replace the first element of PRYT by x. For all the cases, update the data structures as we do in Case 1. Update the predecessor of i.

If insertion of x increases the length of PRYT, then update max_ind (to indicate the global index at which the actual LIS ends) and the max_len (indicates the |LIS|).

The implementation to divide a problem into smaller sub problems is shown in Fig. 1. This algorithm considers the general data set where data is not exactly permutation of range $[1 \ldots n]$. The combine approach is implemented through combine_LIS algorithm and is shown in Fig. 2. The state space in Fig. 3 shows the sub problems and their predecessors when the given sequence is $< 8, 9, 5, 2, 3, 7, 10, 4, 1, 6 >$.

Let us see the execution of the algorithms by taking an example. Suppose $X =< 8, 9, 5, 2, 3, 7, 10, 4, 1, 6 >$, during the division phase recursively when the size of sub problem reduces to 1, its predecessor is set to 0.
X is divided into $X_1 =< (5, 3), (2, 4), (3, 5), (4, 8), (1, 9) >$ and $X_2 =< (8, 1), (9, 2), (7, 6), (10, 7), (6, 10) >$.
X_1 is further divided into $X_{11} =< (2, 4), (1, 9) >$ and $X_{12} =< (5, 3), (3, 5), (4, 8) >$.

X_{11} is further divided into $X_{111} =< (1, 9) >$ and $X_{112} =< (2, 4) >$.
X_{12} is further divided into $X_{121} =< (3, 5), >$ and $X_{122} =< (5, 3), (4, 8) >$.
X_{122} is further divided into $X_{1221} =< (4, 8) >$ and $X_{1222} =< (5, 3) >$.

Similarly X_2 is divided.

Since the size of X_{111} is 1, its predecessor, Pred$[9] = 0$. For each sub problem of size one, its predecessor is set to 0. And thus we achieve Pred$[1 \ldots 10] = [0, \ldots 0]$.

```
combine_LIS(X, i, j, m_e)
{
   max_ind = 0;
   max_len = 0;
   longest_LIS_len =0;
   n = j-i+1;
   Knuth0.val=0;
   Knuth 0.ind=0;
   for r = 1 to n do
      k = Pred[Xr.pos];
      j=is_present[k];
      if(Xr.val ≤ m_e)
            if(longest_LIS.len ≤ j+1)
                  longest_LIS.len =j+1;
      else
            if ( k==0)
                  j= longest_LIS.len;
            else
                  if(j==0)
                     use binary search to find the largest index p such that it
                     contains the  element smaller than Xr.val in  PRYT from
                     longest_LIS_len to max_len;
                    j = p;
      is_present[Knuth j+1.ind] = 0;
      pred[Xr.pos] = Knuth j.ind;
      Knuth j+1.val =  Xr.val;
      Knuth j+1.ind =  Xr.pos;
      is_present[Xr.pos] = j+1;
      if(max_len  ≤ j +1)
            max_ind =  Xr.pos
            max_len = j+1;
}
```

Fig. 2. Algorithm combine_LIS.

$X_{11} =< (2,4),(1,9) >$ is solved by combining X_{111} and X_{112}. Since $2 \in X_{112}$, its Pred[4] = 0 and *longest_LIS_len* = 0, 2 is inserted at 1st position in PRYT. Pred[4] does not change. Second element is 1, which belongs to X_{111} and its predecessor is 0, replace 1st element by 1. PRYT $=< (1,9) >$ and Pred$[1 \ldots 10] = [0, \ldots 0]$.

For similar reasons, when X_{122} is solved, we get PRYT $=< (4,8) >$ and Pred$[1 \ldots 10] = [0, \ldots 0]$.

$X_{12} =< (5,3),(3,5),(4,8) >$ is solved by combining X_{121} and X_{122}. Since $5 \in X_{122}$, its predecessor, Pred[3] = 0 and *longest_LIS_ind* = 0, 5 is inserted at 1st position in PRYT. Pred[3] does not change. Next element, $3 \in X_{121}$ and Pred[5] = 0, replace 1st element by 3. Pred[5] does not change. Set *longest_LIS_len* = 1. Next element, $4 \in X_{122}$, Pred[8] = 0 and since *longest_LIS_len* = 1, insert 4 at 2nd position. Pred[8] = 5. Thus we get PRYT $=< (3,5),(4,8) >$ and Pred$[1 \ldots 10] =$ $[0, 0, 0, 0, 0, 0, 0, \underline{5}, 0, 0]$.

$X_1 =< (5,3),(2,4),(3,5),(4,8),(1,9) >$ is solved by combining X_{11} and X_{12}.

Iteration 1: PRYT $=< (5,3) >$ and no other change.
Iteration 2: PRYT $=< (2,4) >$, longest_LIS_len = 1 and no other change.
Iteration 3: PRYT $=< (2,4),(3,5) >$, Pred[5] = 4 and no other change.
Iteration 4: PRYT $=< (2,4),(3,5),(4,8) >$ and no other change.
Iteration 5: PRYT $=< (1,9),(3,5),(4,8) >$ and no other change.

Thus we get PRYT $=< (1,9),(3,5),(4,8) >$ and Pred$[1 \ldots 10] = [0, 0, 0, 0, \underline{4}, 0, 0, 5, 0, 0]$.

$X_{21} =< (7,6),(6,10) >$ is solved by combining X_{211} and X_{212}. We get PRYT $=< (6,10) >$ and Pred$[1 \ldots 10] =$ $[0, 0, 0, 0, 4, 0, 0, 5, 0, 0]$.

For similar reasons, while solving X_{222}, we get PRYT $=< (9,2),(10,7) >$ and Pred$[1 \ldots 10] = [0, 0, 0, 0, 4, 0, \underline{2}, 5, 0, 0]$.

We solve $X_{22} =< (8,1),(9,2),(10,7) >$ by combining X_{221} and X_{222}. We get PRYT $=< (8,1),(9,2),(10,7) >$ and Pred$[1 \ldots 10] = [0, \underline{1}, 0, 0, 4, 0, 2, 5, 0, 0]$.

Now we solve $X_2 =< (8,1),(9,2),(7,6),(10,7),(6,10) >$ by combining X_{21} and X_{22}.

Iteration 1: PRYT $=< (8,1) >$ and no other change.
Iteration 2: PRYT $=< (8,1),(9,2) >$ and no other change.
Iteration 3: PRYT $=< (7,6),(9,2) >$, longest_LIS_len = 1 and no other change.

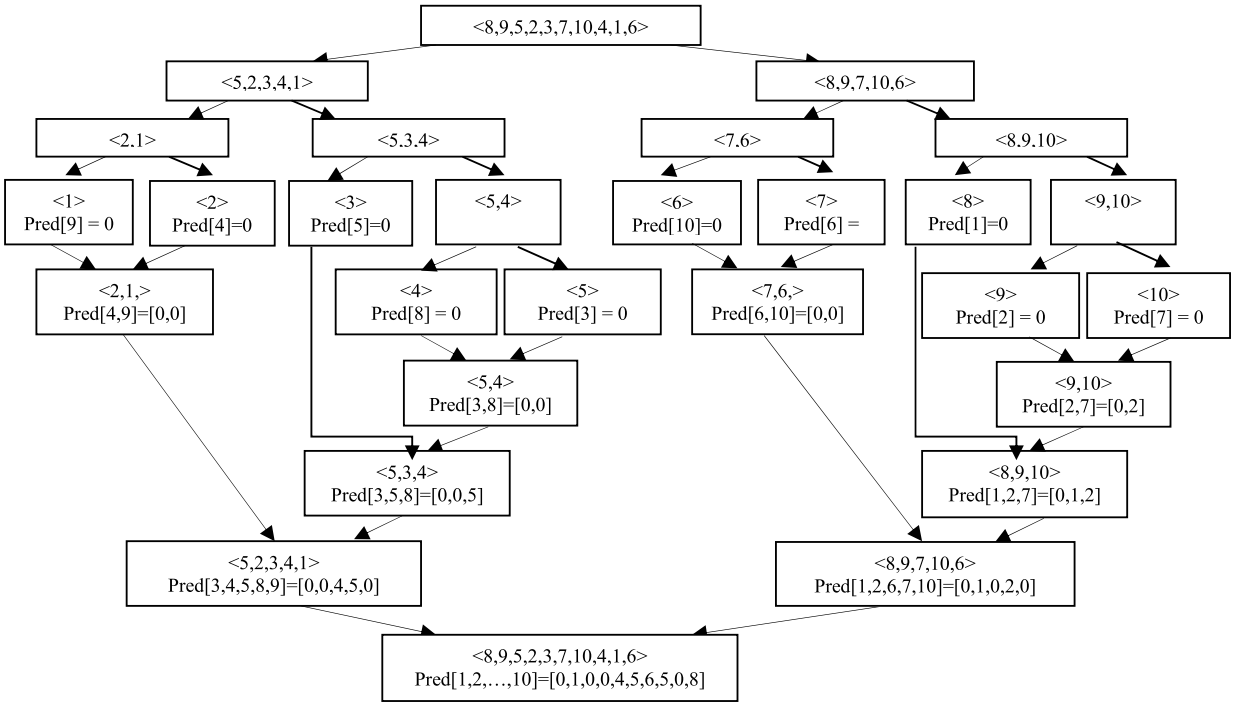

Fig. 3. State Space $A =< 8, 9, 5, 2, 3, 7, 10, 4, 1, 6 >$.

Iteration 4: PRYT $=< (7, 6), (9, 2), (10, 7) >$ and no other change.
Iteration 5: PRYT $=< (6, 10), (9, 2), (10, 7) >$ and no other change.

Thus we get PRYT $=< (6, 10), (9, 2), (10, 7) >$ and Pred$[1 \ldots 10] = [0, 1, 0, 0, 4, 0, 2, 5, 0, 0]$.

Now we solve $X =< 8, 9, 5, 2, 3, 7, 10, 4, 1, 6 >$

Iteration 1: PRYT $=< (8, 1) >$ and no other change.
Iteration 2: PRYT $=< (8, 1), (9, 2) >$ and no other change.
Iteration 3: PRYT $=< (5, 3), (9, 2) >$, longest_LIS_len $= 1$ and no other change.
Iteration 4: PRYT $=< (2, 4), (9, 2) >$ and no other change.
Iteration 5: PRYT $=< (2, 4), (3, 5) >$, longest_LIS_len $= 2$ and no other change.
Iteration 6: Since pred$[6] = 0$ and longest_LIS_len $= 2$, 7 is inserted at 3rd position. Thus PRYT $=< (2, 4), (3, 5), (7, 6) >$ and Pred$[6] = 5$.
Iteration 7: Since 7th element belongs to X_2 and Pred$[7] = 2$ which is not there in PRYT (point c of case 2) so a largest index is searched such that it contains value which is less than 10 in PRYT from index 2(longest_LIS_len) to index 3(max_len). This index is 3, so 10 is inserted at 4th position. Thus PRYT $=< (2, 4), (3, 5), (7, 6), (10, 7) >$ and Pred$[7] = 6$. But according to paper [1], Pred[7] is 5 and resulting PRYT $=< (2, 4), (3, 5), (10, 7) >$. So paper [1] does not handle this particular situation optimally. But to get the correct optimal solution for this case, we need to apply the binary search, and due to this binary search the complexity of the entire approach increases from $O(n \log n)$ to $o(n \log^2 n)$.

Iteration 8: PRYT $=< (2, 4), (3, 5), (4, 8), (10, 7) >$, longest_LIS_len $= 3$ and no other change.
Iteration 9: PRYT $=< (1, 9), (3, 5), (4, 8), (10, 7) >$ and no other change.
Iteration 10: Since Pred$[10] = 0$, so 6 is inserted at (longest_LIS_len $+ 1$)th position in PRYT. Thus PRYT $=< (1, 9), (3, 5), (4, 8), (6, 10) >$, and Pred$[10] = 8$.

Thus we get PRYT $=< (1, 9), (3, 5), (4, 8), (6, 10) >$, and Pred$[1 \ldots 10] = [0, 1, 0, 0, 4, \underline{5}, \underline{6}, 5, 0, \underline{8}]$.

Now we prove the correctness of *divide* and *combine* operations. It is important to divide the problem into two *equal* sub problems to achieve *efficiency*. The combine operation is responsible for optimal LIS.

Divide – If X is exactly a permutation of range $[1 \ldots n]$, then $\lfloor n/2 \rfloor$ is the *m_e*. Then all x_s which are less than or equal to $\lfloor n/2 \rfloor$ form X_1 and rest elements form X_2. If X is not a permutation of $[1 \ldots n]$, but it is a random collection of elements with some elements repeated, then to divide the problem into two equal sub problems we sort X and we consider the element at middle position as *m_e*. Also in case of multiple occurrences we count the number of times this *m_e* appears in first half of X_{sorted} and if this count is b, then during the divide process we include first b *m_e*(s) in X_1. And rest *m_e*(s) are included in X_2. Thus the *divide_LIS* procedure divides the data correctly into two sub problems.

Lemma 1. *During the combine phase if $x \in X_1$, then its predecessor does not change and if it exists it will always be present in PRYT.*

Proof. If $x \in X_1$, then its predecessor either does not exist or it is the element which is smaller than x and always belongs to X_1. But LIS for X_1 is already computed. Since the computation of LIS for X_1 is optimal, its predecessor remains same. □

Lemma 2. *During the combine phase if $x \in X_2$, then its predecessor may change and the combine_LIS procedure correctly computes the predecessor.*

Proof. If $x \in X_2$, then its predecessor may update – if it is 0, then other element which belongs to X_1 (since it is smaller) and which forms the *longest_LIS_len* becomes its predecessor. Otherwise there may be more than one candidate for being its predecessor. And combine operation correctly identifies the best among these. Suppose while solving X_2, there were three candidates (c_1, pos_1), (c_2, pos_2) and (c_3, pos_3) for being its predecessor such that $c_1 > c_2 > c_3$ and $\text{pos}_1 < \text{pos}_2 < \text{pos}_3$. Also suppose the |LIS| that ends with c_1 is L_1, the one that ends with c_2 is L_2, the one that ends with c_3 is L_3 and $L_1 > L_2 > L_3$. Thus the best candidates for being x's predecessor is (c_1, pos_1). But while combining the solutions the best predecessor may change due to presence of certain other elements which belong to X_1. The L_1, L_2 and(or) L_3 may change and our approach selects the best among these updated candidates. Let us consider an example. Suppose $X =< 10, 12, 1, 2, 3, 8, 14 >$. Since $m\_e = 3$, $X_1 =< (1, 3), (2, 4), (3, 5)$ and $X_2 =< (10, 1), (12, 2), (8, 6), (14, 7) >$.

For X_1, $\text{Pred}[3, 4, 5] = [0, 3, 4]$ and for X_2, $\text{Pred}[1, 2, 6, 7] = [0, 1, 0, 2]$, that is $Pred[1, 2, ..., 7] = [0, 1, 0, 3, 4, 0, 2]$.

While combining we have, $X =< 10, 12, 1, 2, 3, 8, 14 >$ and $\text{Pred}[1, 2, \ldots, 7] = [0, 1, 0, 3, 4, 0, 2]$, according to our combine approach each element of X is processed one by one. The contents of Pred at different iterations are as follows:

Iteration 1: PRYT = [(10,1)], Pred[1] = 0.
Iteration 2: PRYT = [(10,1), (12,2)], Pred[2] = 1.
Iteration 3: PRYT = [(1,3), (12,2)], Pred[3] = 0.
Iteration 4: PRYT = [(1,3), (2,4)], Pred[4] = 3.
Iteration 5: PRYT = [(1,3), (2,4), (3,5)], Pred[5] = 4.
Iteration 6: here since $8 \in X_2$ and Pred[6] = 0, then 8 replaces (*longest_LIS_len* + 1)th element in PRYT. But since there is no element present at (*longest_LIS_len* + 1)th position in PRYT, so we simply insert 8 at (*longest_LIS_len* + 1)th index by extending PRYT and hence $\text{PRYT} =< (1, 3), (2, 4), (3, 5)(8, 6) >$, Pred[6] = 5.
Iteration 7: here since $14 \in X_2$ and its predecessor, Pred[7] = 2 is not there in PRYT (point c of case 2), then 14 is inserted after the largest index in PRYT which contains the element smaller than 14. That is insert 14 after 8 and we get $\text{PRYT} = [(1, 3), (2, 4), (3, 5)(8, 6), (14, 7)]$, Pred[7] = 6. And thus we see that Pred[7] is another element which belonged to X_2. But according to paper [1], modified Pred[7] would be 5 (5th element is 3 and it belongs to X_1) and resulting $\text{PRYT} = [(1, 3), (2, 4), (3, 5), (14, 7)]$. □

4.1.1. Multiple occurrences

When some elements appear more than once, in that case we need to count the occurrences of middle element in first half of the sorted sequence so as to divide the data set into two equal size problems. The algorithm to divide a problem into two sub problems is given in Fig. 4. If middle element appears b times in first half of X_{sorted}, then while combining the solutions of two sub problems, we handled the first b middle elements as if they belong to X_1 and rest belong to X_2. When an element appears more than once in X, then LIS can have two variants – a strictly increasing longest subsequence such that an element appears at most once in the output and the other is non-decreasing longest subsequence such that the output sequence contains multiple occurrences of some elements.

5. Parallelism

In D&C approach, we divide the LIS problem into two subproblems which are solved independently and then these two solutions are combined to solve the base problem. So two processors can work independently simultaneously on two subproblems. When these two sub problems are solved, their solutions are combined. This parallelism can be implemented by a multithreaded approach discussed in [7]. The approach describes three keywords which support concurrent execution – *spawn, parallel* and *sync*. If there are two procedures which can execute in parallel, then this parallelism can be achieved by preceding the keyword *spawn* while calling those procedures. The next spawned procedure may not wait for the previous spawned procedure. If there is a normal procedure call after a spawned procedure, in that case the normal procedure may not use the output of the spawned procedure. If it wants to use the output of the spawned procedure, then a keyword *sync* must execute before calling the normal procedure. That is if statement *sync* executes, it means the statement after sync will only execute if all the spawned procedures called before the sync statement complete their execution [1].

6. Analysis and comparison

The presented D&C approach solves the LIS problem by dividing it into two equal sub problems, solving these sub problems independently and then combining the solutions to solve the original problem. There are two types of data sets:

```
divide_LISGen(X,i,j)
{
if(i==j)
   pred[i] = 0
if(j>i)
      n = j-i +1;
      x = ⌊((j-i+1)/2)⌋;
      m_e = X_sorted[i+x-1];
      for t = i+x-1 to  t ≥ i do step -1
          if (X_sorted[t] == m_e)
             a++;
          else
             break;
      for k =1 to n do
          if(x_k.val < m_e)
                insert this element at the end in Y1
          if(x_k.val == m_e && m<a)
                m++;
                insert this element at the end in Y1
          else
                insert this element at the end in Y2
      divide_LISGen(  Y1 , i, i+x-1);
      divide_LISGen(  Y2 , x+i, j);
      combine_LIS(X, i, j, m_e);
}
```

Fig. 4. Algorithm divide_LISGen.

When binary search is not at all executed (case 1) (e.g. $A =< 8, 9, 2, 10, 4, 5, 6, 1, 7 >$) and when binary search is required for certain cases (case 2) (e.g. $A =< 18, 19, 20, 8, 9, 10, 13, 14, 1, 2, 3, 4, 12, 11, 21 >$). Let a is the number of elements which require the binary search to find it predecessor. The recurrence equation for this approach is:

$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{case 1}$$

$$T(n) = 2T(n/2) + \Theta(n-a) + O(a \log n) \qquad \text{case 2}$$

The solution of the case 1 equation is $O(n \log n)$. The case 2 equation is treated as:

$$T(n) = 2T(n/2) + \Theta(n-a) + O(a \log n) \text{ and since}$$

$$2T(n/2) + \Theta(n-a) + O(a \log n) < 2T(n/2) + \Theta(n \log n).$$

The solution of $2T(n/2) + \Theta(n \log n)$ is $O(n \log^2 n)$ (big Oh, which signifies the bound which may or may not be asymptotically tight) so the case 2 equation has the solution $o(n \log^2 n)$ (little Oh, which signifies the bound which is not asymptotically tight). The presented approach handles all types of data sets optimally using the proposed algorithm. The presented approach is better than other sequential algorithms since it supports parallelism.

There already exist many other sequential solutions to the LIS problem which are mentioned under the Related Work. But none of the sequential solutions presented in [16,12,4,18,8] (optimal solution) or [15] (approximate solution) can achieve parallelism. Our approach outputs an optimal LIS faster than the backtracking and branch-and-bound approach.

7. Conclusion

In this paper we have presented the improvised solution to LIS problem using already existing D&C approach. Improvement in the approach and the proposed algorithm outputs optimal results for all types of data sets. This approach can be executed in parallel using multiple processors. The time complexity of this approach is $O(n \log n)$ in best and average. The worst case (when binary search is required for few cases) time complexity is $o(n \log^2 n)$. The space complexity of this approach is $O(n)$.

References

[1] M.R. Alam, M.S. Rahman, A divide and conquer approach and a work-optimal parallel algorithm for the LIS problem, Inf. Process. Lett. 113 (2013) 470–476.
[2] M.H. Albert, M.D. Atkinson, Doron Nussbaum, Jörg-Rüdiger Sack, Nicola Santoro, On the longest increasing subsequence of a circular list, Inf. Process. Lett. 101 (2007) 55–59.
[3] M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S.R. Rao, M.A. Safari, Longest increasing subsequences in sliding windows, Theor. Comput. Sci. 321 (2004) 405–414.

[4] S. Bespamyathnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inf. Process. Lett. 76 (1–2) (2000) 7–11.
[5] E. Chen, L. Yang, H. Yuan, Longest increasing subsequences in windows based on canonical antichain partition, Theor. Comput. Sci. 378 (3) (2007) 223–236.
[6] Chiou-Ting Tseng, Chang-Biau Yang, Hsing-Yen Ann, Minimum Height and Sequence Constrained Longest Increasing Subsequence.
[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3 ed., MIT Press, 2009.
[8] M. Crochemore, E. Porat, Fast computation of a longest increasing subsequence and application, Inf. Comput. 208 (9) (2010) 1054–1059.
[9] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, Steven L. Salzberg, Alignment of whole genomes, Nucleic Acids Res. 27 (11) (1999) 2369–2376.
[10] Sebastian Deorowicz, An algorithm for solving the longest increasing circular subsequence problem, Inf. Process. Lett. 109 (2009) 630–634.
[11] Arm Elmasry, The longest almost increasing subsequence, Inf. Process. Lett. 110 (16) (2010) 655–658.
[12] M.L. Fredman, On computing the length of the longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[13] E. Horowitz, S. Sahni, S. Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications Pvt. Ltd., 1998.
[14] B.L.A. Lascoux, J.-Y. Thibon, The plactic monoid, in: M. Lothaire (Ed.), Algebraic Combinatorics on Words, Cambridge University Press, Cambridge, UK, 2002, pp. 164–196.
[15] Michael Saks, C. Seshadhri, Estimating the longest increasing sequence in polylogarithmic time, in: 51st Annual IEEE Symposium on Foundations of Computer Science, 2010, pp. 458–467.
[16] C. Schensted, Longest increasing and decreasing subsequences, Can. J. Math. 13 (1961) 179–191.
[17] Yin-Te Tsai, The constrained longest common subsequence problem, Inf. Process. Lett. 88 (2003) 173–176.
[18] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (1977) 80–82.