Theoretical and Applied Informatics
ISSN 1896–5334
Vol.21 (2009), no. 3-4
pp. 135–148

On Some Variants of the Longest Increasing Subsequence Problem

SEBASTIAN DEOROWICZ [a]

[a]Silesian University of Technology

Received 20 August 2009, Revised 21 September 2009, Accepted 10 October 2009

Abstract: The problem of finding a longest increasing subsequence (LIS) is a well known task in sequence processing. There are many variants of the basic task. We discuss a recently introduced variant of LIS, a minimal height longest increasing subsequence problem and propose a new algorithm for it, which improves its time complexity. Moreover, we define a family of similar problems and introduce algorithms solving them.

Keywords: longest increasing subsequence

1. Introduction

The problem of finding a *longest increasing subsequence* (LIS)[9] in a sequence of integers is to find an increasing subsequence (subsequence is obtained for a sequence by removing zero or more symbols) of maximal possible length. It is known that the lower bound of worst-case time complexity for this problem is $\Omega(n \log n)$ (alternatively $\Omega(n \log \ell)$ when ℓ the length of the result) in the comparison model and such algorithms are known [9,2]. When the sequence is a permutation of integers from range $[1, n]$, assuming the RAM model, an algorithm of the worst-case time complexity $O(n \log \log n)$ using van Emde Boas trees [7] can be proposed.

The LIS problem has applications in many areas, e.g., as a tool to solve the longest common subsequence (LCS) problem [9], for finding a whole genome alignment [6].

There are many variants of the basic LIS problem. To name only a few:

- *heaviest increasing subsequence* (HIS) [10] – each symbol of sequence $A = a_1 a_2 \ldots a_n$ is accompanied by a weight and the goal is to find an increasing subsequence of maximal sum of weights,
- *longest increasing circular subsequence* (LICS)[1] – a longest increasing subsequence is looked for among all cyclic shifts of A,

- *longest increasing subsequence in a sliding window* (LISW)[3] – a longest increasing subsequence is looked for among all windows of length w in A, i.e., among $a_i a_{i+1} \dots a_{i+w-1}$ for all $1 \leq i \leq n - w + 1$,
- *slope-constrained longest increasing subsequence* (SCLIS)[12] – successive symbols in the resulting sequence must differ by a specified amount m, i.e., if a_i and a_j are successive symbols of the result ($i < j$), then $\frac{a_j - a_i}{j-i} \geq m$,
- *minimal height longest increasing subsequence* (MinHLIS [1]) [11] – such an LIS $A' = a'_1 a'_2 \dots a'_\ell$ is looked for that minimises the difference $a'_\ell - a'_1$.

These are only examples, but they show that the field of LIS-related problems is explored intensively. In this paper, we propose a family of algorithms solving the following problems (most of them are introduced in this paper):

- *minimal height LIS* (MinHLIS) – our algorithm offers better worst-case time complexities than presented in [11],
- *maximal height LIS* (MaxHLIS) – an LIS that has the largest possible height is requested, i.e., $a'_\ell - a'_1$ should be maximal possible,
- *minimal width LIS* (MinWLIS) – an LIS that has the minimal possible width is requested, i.e., if $a'_1 = a_i$ and $a'_\ell = a_j$, then $j - i$ should be minimal possible,
- *maximal width LIS* (MaxWLIS) – similar as above, but $j - i$ should be maximal possible,
- *minimal sum LIS* (MinSLIS) – an LIS that has the minimal sum of elements is requested,
- *maximal sum LIS* (MaxSLIS) – similar as above, but LIS with maximal sum of elements is requested.

The paper is organised as follows. In Section 2. we define the necessary terms. Section 3. contains description of the only existing (to the best of our knowledge) algorithm solving one of the problems considered in the paper. A family of algorithms together with an analysis of their time ans space complexities are discussed in Section 4. The last section concludes the paper.

2. Definitions

Let $A = a_1 a_2 \dots a_n$ be a sequence composed of unique symbols over an integer alphabet Σ and $1 \leq a_i \leq \sigma$ for $1 \leq i \leq n$. A sequence A' is a *subsequence* of A if it can

[1] In the original paper, the abbreviation MHLIS is used, but since the abbreviation for some other problem considered in this paper would be the same, we decided to use such a less compact name.

be obtained from A by removing zero or more symbols. A subsequence $A' = a'_1 a'_2 \dots a'_k$ is increasing if $a'_i < a'_{i+1}$ for all $1 \leq i < k$. A *longest increasing subsequence* (LIS) of A is an increasing subsequence that has the maximal possible length. There can be many LIS of A since only its length is unique. A *rank* of the symbol a_i is the LIS length of sequence $a_1 a_2 \dots a_i$. The LIS length of A is denoted by ℓ. A *height* of sequence $A' = a'_1 a'_2 \dots a'_k$ is $a'_k - a'_1$. A *width* of sequence $A' = a'_1 a'_2 \dots a'_k$ such that $a'_1 = a_i$ and $a'_k = a_j$ is $j - i$.

3. Background

To the best of our knowledge, there is only one paper on MinHLIS problem [11] and no works on other problems considered in this paper. The algorithm solving the MinHLIS problem works as follows. Forest F of balanced binary search trees T_1, T_2, $\dots$, T_ℓ is maintained – one tree for symbols of one rank, i.e., during the algorithm work, tree T_k contains all already processed symbols of rank k. There is also a balanced binary search tree M containing the minimal elements of $T_1, T_2, \dots, T_\ell$ and as auxiliary data for each symbol an index of the tree from F the symbol belongs to. When symbol a_i is to be processed: (*i*) its successor a'_i in M is determined, (*ii*) a_i has the same rank as a'_i, so it is added to the tree containing a'_i, and (*iii*) a'_i is replaced by a_i in M. If no successor in M exists: (*i*) the rank of a_i is the size of M increased by 1, (*ii*) tree $T_{|M|+1}$ is updated by a_i, and (*iii*) a_i is inserted to M. After symbol a_i is inserted to T_y its predecessor in T_{y-1} is determined and noted. Moreover, for each T_k a minimal height LIS ending at some symbol from it is stored. Then, after processing the whole sequence A, according to the links to predecessors an MinHLIS is found.

The worst-case time complexity of the algorithm is $O(n \log n)$ due to $O(n)$ operations on balanced binary search trees of size $O(n)$.

4. Proposed algorithms

4.1. Common Ideas

In this section, we propose algorithms for all the problems introduced in Section 1. The algorithms are based on a common concept of a greedy cover representation of a sequence. A *greedy cover* C is an ordered set of decreasing lists of the following properties:

1. the position of the list in the ordered set which each symbol belongs to is the length of an LIS ending at this symbol,
2. it is unique, i.e., there is no other greedy cover satisfying property (1).

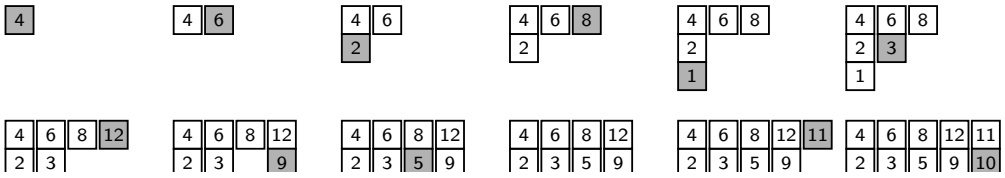

Fig. 1. Example of the algorithm finding the greedy cover of $A = 4,6,2,8,1,3,12,9,5,7,11,10$. The just placed elements are gray. The lists are vertical. Note that the most recent (the lowest) cells of each list are increasing.

From the above, the size of the cover is the LIS length of the sequence.

The algorithm computing the greedy cover [9, pp. 287–290] works as follows. For each successive symbol a_i, it founds the leftmost possible decreasing list of C ended by a symbol larger than a_i and appends a_i to it. If no such a list can be found, a new list containing a_i is appended to C. An example of the algorithm in work is presented in Fig. 1.

To locate the list to extend we use some priority queue Q supporting the operations: *insert*, *remove*, *successor*. The choice of the priority queue (PQ) type determines the time complexity of the algorithms and this subject will be discussed in Section 4.8. Q stores the smallest elements of each list of C (i.e, the elements ending the lists) accompanied with the pointers to the lists the elements belong to.

In all algorithms introduced in the paper, we maintain in fact a cover augmented by some other data. For each symbol a_i located in $C[k]$ ($C[k]$ means the kth list of C) we store a 4-tuple $\langle a_i, i, p_i, d_i\rangle$, where p_i is a pointer to some smaller symbol in $C[k-1]$ (if $k > 1$) or **nil** (if $k = 1$), and d_i is some data field (to be specified later).

A general scheme of all the algorithms is similar (Fig. 2).

4.2. Minimal Height LIS

In the algorithm solving the Minimal Height LIS problem, data fields are set as $d \leftarrow a_i$ (Fig. 3, line 10). Pointer p points to the largest element in $C[k-1]$ smaller than a_i (Fig. 3, line 12). Since, each list of the cover is decreasing, the search can be stopped when the first smaller element than a_i is noticed. Moreover, a_i is smaller than the ending element of $C[k]$, so the search can be started from the element of $C[k-1]$ pointed by the p field of the last element of $C[k]$. Only, when the actual element starts a new cover list, $C[k-1]$ is browsed from the beginning.

To compute the final result (Fig. 3, line 15), it suffices to find in $C[|C|]$ the element $\langle a_i, i, p_i, d_i\rangle$ of minimal value $a_i - d_i$. The difference $a_i - d_i$ is the height of a MinHLIS, and the pointers allow to obtain a MinHLIS itself. An example of the annotated cover produced by that algorithm is shown in Fig. 4.

**LIS()

```
01  for i ← 1 to n do
02      (s, k) ← Q.succ(a_i)      {s – a successor of a_i, k – list number of C containing s}
03      if s = nil then
04          k ← |C| + 1
05      else
06          Q.remove(s)
07      Q.insert(a_i, k)
08      if k = 1 then
09          p ← nil
10          Determine d in some way (depends on the algorithm)
11      else
12          Determine p in some way (depends on the algorithm)
13          d ← p^.d      {d gets the value of field d of 4-tuple pointed by p}
14      C[k].append(a_i, i, p, d)
15  Compute the final result on C
```

Fig. 2. A general scheme of the algorithms solving MinHLIS, MaxHLIS, MinWLIS, MaxWLIS problems

MinHLIS()

```
    {Lines 01–07 are the same as in Fig. 4.1.}
08      if k = 1 then
09          p ← nil
10          d ← a_i
11      else
12          p ← pointer to the largest element of C[k − 1] smaller than a_i
13          d ← p^.d      {d gets the value of field d of 4-tuple pointed by p}
14      C[k].append(a_i, i, p, d)
15  return Pointer to the element of C[|C|] with smallest value a_i − d_i
```

Fig. 3. An algorithm solving MinHLIS problem

Lemma 1. *The data fields d_i in each list of C are in a non increasing order if set as in Fig. 3.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $d_i = a_i$ and a_i's are decreasing, so also d_i's are decreasing. Let now $k > 1$ and some a_i is to be appended to $C[k]$. The data field of the ending symbol a_j of $C[k]$ equals to the data field of the largest symbol of $C[k-1]$ smaller than a_j. Since $a_i < a_j$, the largest symbol of $C[k-1]$ smaller than a_i cannot be larger than the symbol pointed by p_j. Therefore, $d_i \leq d_j$. □

Lemma 2. *For each 4-tuple $\langle a_i, i, p_i, d_i \rangle$, the difference $a_i - d_i$ is the height of a MinHLIS ending at a_i.*

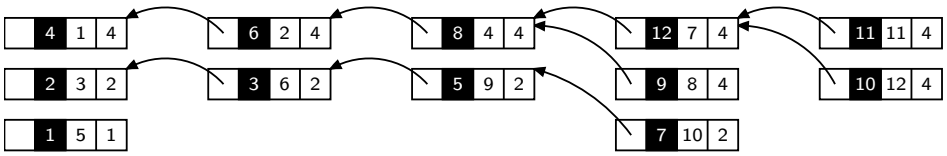

Fig. 4. Annotated greedy cover produced by MinHLIS algorithm for $A = 4,6,2,8,1,3,12,9,5,7,11,10$. The fields are shown in order: pointer p_i, symbol a_i (blacked), index i, data d_i.

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $a_i - d_i = 0$, so it is obvious. Let now $k > 1$ and $\langle a_i, i, p_i, d_i\rangle$ is some 4-tuple of $C[k]$. Concerning all 4-tuples $\langle a_{j'}, j', p_{j'}, d_{j'}\rangle$ of $C[k-1]$ such that $a_{j'} < a_i$, the value $a_i - d_{j'}$ is minimal (according to Lemma 1) if $a_{j'}$ is as large as possible (we denote it by a_j), i.e., the jth 4-tuple is pointed by p_i and $d_i = d_j$. Moreover, since p_i points a_j, then $j < i$, so the MinHLIS ending at a_j of length $k-1$ can be extended by a_i to obtain the MinHLIS of length k ending at a_i. □

Corollary 3. *The algorithm presented in Fig. 3 computes a MinHLIS.*

Proof. All the symbols of ranks equal to the LIS length of A are in $C[|C|]$, so it suffices to browse this list to find the symbol ending an LIS of the smallest height (Lemma 2). The pointers allow to obtain a MinHLIS itself. □

The time complexity is determined by two terms: the time necessary to make operations on a priority queue and on a cover. There are at most n operations of each kind (insert, remove, successor) on a priority queue. The choice of a PQ type will be discussed later, and now we only assume that the total cost of operations on a PQ is t_{PQ}.

There are n constant-time operations of appending the symbols to the cover lists. Concerning the cost of determining the pointer (Fig. 3, line 12), we can notice that browsing each list of the cover we never go backward. Once we started browsing the list we always go forward or stay at the same element. Since, the search is performed $O(n)$ times, the total number of elements in cover C is $O(n)$, and the number of lists in cover C is $O(n)$, the total cost of searching the cover lists is $O(n)$. Other operations in the main loop take also constant time per symbol ($O(n)$ in total).

To collect the result, list $C[|C|]$ containing $O(n)$ elements must be browsed and then $O(\ell)$ elements according to the pointers must be visited. Therefore, the cost of the final computation of the result (Fig. 3, line 15) is $O(n)$. Summing it up, the total worst-case time complexity of the algorithm is $t_{\mathrm{PQ}} + O(n)$.

4.3. Maximal Height LIS

In the algorithm solving the Maximal Height LIS problem, data fields are set as: $d \leftarrow a_i$ (Fig. 5, line 10). Pointer p points to the element ending $C[k-1]$ at the moment a_i is processed (Fig. 5, line 12).

MaxHLIS()

```
    {Lines 01–07 are the same as in Fig. 4.1.}
08      if k = 1 then
09          p ← nil
10          d ← a_i
11      else
12          p ← pointer to element ending C[k − 1]
13          d ← p^.d      {d gets the value of field d of 4-tuple pointed by p}
14      C[k].append(a_i, i, p, d)
15  return Pointer to the element of C[|C|] with largest value a_i − d_i
```

Fig. 5. An algorithm solving MaxHLIS problem

To compute the final result, it suffices to find in the last list of C the element $\langle a_i, i, p_i, d_i\rangle$ of maximal value $a_i - d_i$ (Fig. 5, line 15). The difference $a_i - d_i$ is the height of a MaxHLIS, and the pointers allow to obtain a MaxHLIS itself.

Lemma 4. *The data fields d_i in each list of C are in a non increasing order if set as in Fig. 5.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $d_i = a_i$ and a_i's are decreasing, so also d_i's are decreasing. Let now $k > 1$ and some a_i is to be appended to $C[k]$. The pointer field of the ending 4-tuple $\langle a_j, j, p_j, d_j\rangle$ of $C[k]$ points to some symbol of $C[k-1]$ that cannot be smaller than the symbol actually ending $C[k-1]$ (the lists are decreasing). Therefore, $d_i \leq d_j$. □

Lemma 5. *For each 4-tuple $\langle a_i, i, p_i, d_i\rangle$, the difference $d_i - a_i$ is the height of a MaxHLIS ending at a_i.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $d_i - a_i = 0$, so it is obvious. Let now $k > 1$ and $\langle a_i, i, p_i, d_i\rangle$ is some 4-tuple of $C[k]$. Concerning all 4-tuples $\langle a_{j'}, j', p_{j'}, d_{j'}\rangle$ of $C[k-1]$ such that $j' < i$ the value $a_i - d_{j'}$ is maximal (according to Lemma 4) if $a_{j'}$ is as small as possible (we denote it by a_j), i.e., the jth 4-tuple was at the end of $C[k-1]$ when a_i was added to the cover. Therefore, a_j is pointed by p_i and $d_i = d_j$. Moreover, since p_i points a_j, then $j < i$, so a MaxHLIS ending at a_j of length $k-1$ can be extended by a_i to obtain the MaxHLIS of length k ending at a_i. □

Corollary 6. *The algorithm presented in Fig. 5 computes a MaxHLIS.*

Proof. All the symbols of ranks equal to LIS of A are in $C[|C|]$, so it suffices to browse this list to find the symbol ending an LIS of largest height (Lemma 5). The pointers allow to obtain a MaxHLIS itself. □

MinWLIS()

```
    {Lines 01–07 are the same as in Fig. 4.1.}
08      if k = 1 then
09          p ← nil
10          d ← i
11      else
12          p ← pointer to element ending C[k − 1]
13          d ← p^.d      {d gets the value of field d of 4-tuple pointed by p}
14      C[k].append(a_i, i, p, d)
15  return Pointer to the element of C[|C|] with smallest value i − d_i
```

Fig. 6. An algorithm solving MinWLIS problem

The analysis of the worst-case time complexity for the MaxHLIS computing algorithm is very similar to that in Section 4.2. The only difference is the case of line 12 in the pseudocode. Here, it needs $O(1)$ time per element. Therefore, the total worst-case time complexity of the algorithm is $t_{\mathrm{PQ}} + O(n)$.

4.4. Minimal Width LIS

In the algorithm solving the Minimal Width LIS problem data fields are set as: $d \leftarrow i$ (Fig. 6, line 10). Pointer p points to the element ending $C[k-1]$ at the moment a_i is processed (Fig. 6, line 12).

To compute the final result, it suffices to find in the last list of C such 4-tuple $\langle a_i, i, p_i, d_i\rangle$ that has the minimal value $i - d_i$ (Fig. 6, line 15). The value $i - d_i$ is the width of a MinWLIS, and pointers p_i allows to obtain a MinWLIS itself.

Lemma 7. *The data fields d_i in each list of C are in a non-decreasing order if set as in Fig. 6.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $d_i = i$ and the elements that are more far in the list were added later, so d_i values are increasing. Let now $k > 1$ and some a_i is to be appended to $C[k]$. The pointer field of the ending 4-tuple $\langle a_j, j, p_j, d_j\rangle$ of $C[k]$ points to some element of $C[k-1]$ that was added not later to $C[k-1]$ than the symbol actually ending $C[k-1]$. Therefore, $d_i \geq d_j$. □

Lemma 8. *For each 4-tuple $\langle a_i, i, p_i, d_i\rangle$, the difference $i - d_i$ is the width of a MinWLIS ending at a_i.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $i - d_i = 0$, so it is obvious. Let now $k > 1$ and $\langle a_i, i, p_i, d_i\rangle$ is some 4-tuple of $C[k]$. Concerning all 4-tuples $\langle a_{j'}, j', p_{j'}, d_{j'}\rangle$ of $C[k-1]$ such that $j' < i$ the value $i - d_{j'}$ is minimal (according to Lemma 7) if j' is as large as possible (we denote it by j), i.e., the jth

MaxWLIS()

```
    {Lines 01–07 are the same as in Fig. 4.1.}
08      if k = 1 then
09          p ← nil
10          d ← i
11      else
12          p ← pointer to the largest element of C[k − 1] smaller than a_i
13          d ← p^.d      {d gets the value of field d of 4-tuple pointed by p}
14      C[k].append(a_i, i, p, d)
15  return Pointer to the element of C[|C|] with smallest value a_i − d_i
```

Fig. 7. An algorithm solving MaxWLIS problem

4-tuple was at the end of $C[k-1]$ when a_i was added to the cover. Therefore, a_j is pointed by p_i and $d_i = d_j$. Moreover, since p_i points a_j, then $j < i$, so a MinWLIS ending at a_j of length $k-1$ can be extended by a_i to obtain the MinWLIS of length k ending at a_i. □

Corollary 9. *The algorithm presented in Fig. 6 computes a MinWLIS.*

Proof. All the symbols of ranks equal to the LIS length of A are in $C[|C|]$, so it suffices to browse this list to find the symbol ending an LIS of the smallest width (Lemma 8). The pointers allow to obtain a MinWLIS itself. □

The analysis of the worst-case time complexity for the MinWLIS computing algorithm is identical to that in Section 4.3. Therefore, the total worst-case time complexity of the algorithm is $t_{\mathrm{PQ}} + O(n)$.

4.5. Maximal Width LIS

In the algorithm solving the Maximal Width LIS problem data fields are set as: $d \leftarrow i$ (Fig. 7, line 10). Pointer p points to the largest element in $C[k-1]$ smaller than a_i (Fig. 7, line 12). Since each list of the cover is decreasing, the search can be stopped when the first smaller element than a_i is noticed. Moreover, a_i is smaller than the last element of $C[k]$, so the search can be started from the element of $C[k-1]$ pointed by the p field of the last element of $C[k]$. Only, when the actual element starts a new cover list, $C[k-1]$ is browsed from the beginning.

To compute the final result, it suffices to find in $C[|C|]$ the element $\langle a_i, i, p_i, d_i \rangle$ of maximal value $i - d_i$ (Fig. 7, line 15). The difference $i - d_i$ is the width of a MaxWLIS, and the pointers allow to obtain a MaxWLIS itself.

Lemma 10. *The data fields d in each list of C are in a non-decreasing order if set as in Fig. 7.*

MinSLIS()

{*Lines 01–14 of Fig. 4.3.*}
15 **return** Pointer to ending element of $C[|C|]$

Fig. 8. An algorithm solving MinSLIS problem

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $d_i = i$ and the elements that are more far in the list were added later, so d_i are increasing. Let now $k > 1$ and some a_i is to be appended to $C[k]$. The data field of the ending symbol a_j of $C[k]$ is the data field of the largest symbol of $C[k-1]$ smaller than a_j. Since $a_i < a_j$, the largest symbol of $C[k-1]$ smaller than a_i cannot be larger than the symbol pointed by p_j. Therefore, data $d_i \geq d_j$. □

Lemma 11. *For each 4-tuple $\langle a_i, i, p_i, d_i \rangle$, the difference $i - d_i$ is the height of the MaxWLIS ending at a_i.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$, $i - d_i = 0$, so it is obvious. Let now $k > 1$ and $\langle a_i, i, p_i, d_i \rangle$ is some 4-tuple of $C[k]$. Concerning all 4-tuples $\langle a_{j'}, j', p_{j'}, d_{j'} \rangle$ of $C[k-1]$ such that $a_{j'} < a_i$, the value $i - d_{j'}$ is maximal (according to Lemma 10) if $a_{j'}$ is as large as possible (we denote it by a_j), i.e., the jth 4-tuple is pointed by p_i and $d_i = d_j$. Moreover, since p_i points a_j, then $j < i$, so the MaxWLIS ending at a_j of length $k-1$ can be extended by a_i to obtain MaxWLIS of length k ending at a_i. □

Corollary 12. *The algorithm presented in Fig. 7 computes a MaxWLIS.*

Proof. All the symbols of ranks equal to the LIS length of A are in $C[|C|]$, so it suffices to browse this list to find the symbol ending an LIS of largest width (Lemma 11). The pointers allow to obtain a MaxWLIS itself. □

The analysis of the worst-case time complexity for the MaxWLIS computing algorithm is identical to that in Section 4.2. Therefore, the total worst-case time complexity of the algorithm is $t_{\mathrm{PQ}} + O(n)$.

4.6. Minimal Sum LIS

The cover produced by the MaxHLIS algorithm (Fig. 5) can be also used for other purposes. In Fig. 8, an algorithm computing an LIS of minimal sum of elements is presented. As we can see, the difference between MaxHLIS and MinSLIS algorithms is only in the computation of final result.

MaxSLIS()

{*Lines 01–14 of Fig. 4.2.*}
15 **return** Pointer to starting element of $C[|C|]$

Fig. 9. An algorithm solving MaxSLIS problem

Lemma 13. *For any 4-tuples $\langle a_i, i, p_i, d_i\rangle$ and $\langle a_j, j, p_j, d_j\rangle$ belonging to the same cover list $C[k]$ such that $a_i < a_j$, the sum of elements of increasing subsequence (IS) ending at a_i (obtained be following the pointers) are less than the sum of elements of IS ending at a_j (obtained in a similar way).*

Proof. Let S_x denotes the sum of elements of increasing subsequence ending at a_x obtained by following the links. We proceed by recurrence on k, the list index in C. For $k = 1$, the only elements belonging to the two specified increasing sequences are a_i and a_j, so it is obvious that $S_i < S_j$. Let now $k > 1$. Since $a_i < a_j$ and both elements belong to the same list of C, $i > j$, i.e., the ith element had to be added to the list later. The pointer p_j points $a_{j'}$, the smallest element of $C[k-1]$ such that $j' < j$. The pointer p_i points $a_{i'}$, the smallest element of $C[k-1]$ such that $i' < i$. Thus $a_{i'} \leq a_{j'}$ and $i' \geq j'$. As $S_{i'} \leq S_{j'}$ and $a_i < a_j$, then $S_i < S_j$. □

Lemma 14. *A minimal sum LIS ending at a_i can be obtained by following the links from a_i.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$ it is obvious. Let now $k > 1$. From Lemma 13 it is known that the sums of increasing subsequences that can be obtained by following the links from all the symbols a_j of $C[k-1]$ are in a decreasing order. Therefore, the MinSLIS ending at the smallest symbol a_j of $C[k-1]$ such that $j < i$ extended by a_i is the minimal sum LIS ending at a_i, and this symbol (a_j) is pointed by p_i. □

Corollary 15. *From Lemmas 13 and 14, the minimal sum LIS can be obtained by following the links from the symbol ending $C[|C|]$.*

The time complexity analysis is almost identical to the one presented in Section 4.3.

4.7. Maximal Sum LIS

The cover produced by the MinHLIS algorithm (Fig. 3) can be also used for other purposes. In Fig. 9, an algorithm computing an LIS of maximal sum of elements is presented. As we can see the difference between MinHLIS and MaxSLIS algorithms is only in the computation of final result.

Lemma 16. *For any 4-tuples $\langle a_i, i, p_i, d_i\rangle$ and $\langle a_j, j, p_j, d_j\rangle$ belonging to the same cover list $C[k]$ such that $a_i < a_j$, the sum of elements of IS ending at a_i (obtained be following the pointers) are less than the sum of elements of IS ending at a_j (obtained in a similar way).*

Proof. Let S_x denotes the sum of elements of increasing subsequence ending at a_x obtained by following the links. We proceed by recurrence on k, the list index in C. For $k = 1$, the only elements belonging to the two specified increasing sequences are a_i and a_j, so it is obvious that $S_i < S_j$. Let now $k > 1$. Since $a_i < a_j$ and both elements belong to the same list of C, $i > j$, i.e., the ith element had to be added to the list later. The pointer p_j points $a_{j'}$, the largest element of $C[k-1]$ smaller than a_j and $j' < j$. The pointer p_i points $a_{i'}$, the largest element of $C[k-1]$ smaller than a_i and $i' < i$. Thus $a_{i'} \leq a_{j'}$ and $i' \geq j'$. As $S_{i'} \leq S_{j'}$ and $a_i < a_j$, then $S_i < S_j$. □

Lemma 17. *A maximal sum LIS ending at a_i can be obtained by following the links from a_i.*

Proof. We proceed by recurrence on k, the list index in C. For $k = 1$ it is obvious. Let now $k > 1$. From Lemma 16 it is known that the sums of increasing subsequences that can be obtained by following the links from all the symbols a_j of $C[k-1]$ are in a decreasing order. Therefore, the MaxSLIS ending at the largest symbol a_j of $C[k-1]$ such that $a_j < a_i$ extended by a_i is the maximal sum LIS ending at a_i, and this symbol (a_j) is pointed by p_i. □

Corollary 18. *From Lemmas 16 and 17, the maximal sum LIS can be obtained by following the links p from the symbol at the front of $C[|C|]$.*

The time complexity analysis is almost identical to the one presented in Section 4.2.

4.8. Complexity Analysis

The worst case time complexity is identical for all the proposed algorithms and is $t_{\mathrm{PQ}} + O(n)$, where t_{PQ} is the total cost of operations on a priority queue (PQ): $O(n)$ inserts, $O(n)$ removals, $O(n)$ successor queries. Since, the second term of the algorithm time complexity is only linear, the proper choice of a priority queue is crucial.

There are many variants of PQ that can be used here. A balanced binary search tree, e.g., red-black tree, handle all this operations in time $O(\log n)$, which can be also stated as $O(\log \ell)$ if we allow to express the result in terms of the output size. This leads to $t_{\mathrm{PQ}} = O(n \log \ell)$. This is the best we can do if the alphabet is unbounded. Often, however, alphabet is bounded (in fact, we assume that in the paper), or even A is a permutation of integers $[1, n]$. In this case, a van Emde Boas tree [7] is the best

possible candidate. It handles each of the necessary operations in time $O(\log\log\sigma)$, so $t_{\mathrm{PQ}} = O(n\log\log\sigma)$.

Summing it up, the total worst-case time complexities of this algorithms are:

$$O(n\log\ell) \qquad \text{or} \qquad O(n\log\log\sigma).$$

Concerning the space complexity, the cover needs $O(n)$ words of space, and the space necessary for a PQ depend on the choice we made. Red-black tree needs $O(n)$ words of space, while van Emde Boas tree $O(\sigma)$ bits. Therefore, the space complexities of the algorithms are:

$$O(n) \text{ words} \qquad \text{or} \qquad O(n) \text{ words} + O(\sigma) \text{ bits}.$$

The time complexity of the only published algorithm solving the MinHLIS problem is $O(n\log n)$, so our proposal outperforms it. The space complexity of our method is the same.

5. Conclusions

We presented a family of algorithms solving six variants of the longest increasing subsequence problem, i.e., LIS of minimal/maximal height/width/sum of elements. Only one of the problems handled in this paper was solved to date and our algorithm outperforms it in terms of the worst-case time complexity.

The worst-case time complexities of all the presented algorithms are the same as the lower bound for the worst-case time complexity for the LIS problem. Since all the algorithms compute some LIS, it is known that lower bound for the worst-case time complexity of the LIS problem holds also for them and our algorithms are optimal.

6. Acknowledgements

The author thanks Szymon Grabowski for reading the preliminary versions of the paper and suggesting improvements.

References

1. M.H. Albert, M.D. Atkinson, D. Nussbaum, J.-R. Sack, N. Santoro: On the longest increasing subsequence of a circular list, Information Processing Letters 101, pp. 55–59, 2007.
2. D. Aldous, P. Diaconis: Longest increasing subsequences: from patience sorting to the Baik–Deift–Johansson theorem, Bulletin of the AMS 36, pp. 413–432, 1999.

3. M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S.S. Rao, M.A. Safari: Longest increasing subsequences in sliding windows, Theoretical Computer Science 321, pp. 405–414, 2004.
4. E. Chen, L. Yang, H. Yuan: Longest increasing subsequences in windows based on canonical antichain partition, Theoretical Computer Science 378(3), pp. 223–236, 2007.
5. S. Deorowicz: An algorithm for solving the longest increasing circular subsequence problem, Information Processing Letters 109(12), pp. 630–634, 2009.
6. A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg: Alignment of while genomes, Nucleic Acids Research 27(11), pp. 2369–2376, 1999.
7. P. van Emde Boas, R. Kaas, E. Zijlstra: Design and implementation of an efficient priority queue, Mathematical Systems Theory 10, pp. 99–127, 1977.
8. M.L. Fredman: On computing the length of longest increasing subsequences, Discrete Mathematics 11, pp. 29–35, 1975.
9. D. Gusfield: Algorithms on strings, trees, and sequences, Cambridge University Press, USA, 1999.
10. G. Jacobson, K.-P. Vo: Heaviest increasing/common subsequence problems, Lecture Notes in Computer Science 644, pp. 52–66, 1992.
11. C.-T. Tseng, C.-B. Yang, and H.-Y. Ann: Minimal Height and Sequence Constrained Longest Increasing Subsequence, Journal of Internet Technology 10(2), pp. 173–178, 2009.
12. I-H. Yang, Y-C. Chen: Fast algorithms for the constrained longest increasing subsequence problems, In Proceedings of the 25th Workshop on Combinatorial Mathematics and Computation Theory, Hsinchu Hsien, Taiwan, April 25–26, pp. 226–231, 2008.

O pewnych wariantach problemu wyznaczania najdłuższego rosnącego podciągu

Streszczenie

Problem wyznaczania najdłuższego rosnącego podciągu (ang. longest increasing subsequence, LIS) jest jednym z problemów w przetwarzaniu ciągów. W artykule dyskutowany jest jeden z jego wariantów, a mianowice problem wyznaczania najdłuższego rosnącego podciągu o minimalnej wysokości. Zaprezentowany algorytm dla tego problemu oferuje złożoność czasową lepszą niż najszybszy opisany do tej pory w literaturze algorytm. Ponadto w pracy zdefiniowano rodzinę podobnych problemów, dla których zaproponowano optymalne algorytmy ich rozwiązywania.