

An Efficient Algorithm for the Longest Tandem Scattered Subsequence Problem

Adrian Kosowski

Department of Algorithms and System Modeling
Gdańsk University of Technology, Poland
kosowski@sphere.eti.pg.gda.pl

Abstract. The paper deals with the problem of finding a tandem scattered subsequence of maximum length (LTS) for a given character sequence. A sequence is referred to as tandem if it can be split into two identical sequences. An efficient algorithm for the LTS problem is presented and is shown to have $O(n^2)$ computational complexity and linear memory complexity with respect to the length n of the analysed sequence. A conjecture is put forward and discussed, stating that the complexity of the given algorithm may not be easily improved. Finally, the potential application of the solution to the LTS problem in approximate tandem substring matching in DNA sequences is discussed.

1 Introduction

A *perfect single repeat tandem sequence* (referred to throughout this article simply as a *tandem sequence*) is one which can be expressed as the concatenation of two identical sequences. Tandem sequences are well studied in literature. The problem of finding the longest tandem substring (the longest subsequence composed of consecutive elements) of a given sequence was solved by Main and Lorentz [7], who showed an $O(n \log n)$ algorithm, later improved to $O(n)$ complexity by Kolpakov and Kucherov [3]. A lot of attention has also been given to finding longest approximate tandem substrings of sequences, where the approximation criterium of the match is given either in terms of the Hamming distance or the so called edit distance between the substring and the sequence.

This paper deals with a related problem, concerning determining the longest tandem subsequence (which need not be a substring) of a given sequence (the so called *LTS* problem). A formal definition of *LTS* is in Subsection 2.1 and an efficient algorithm which solves *LTS* in $O(n^2)$ using $O(n)$ space is outlined in subsections 2.1 and 2.2. This result is a major improvement on the hitherto extensively used naive algorithm, which reduces the solution *LTS* to n iterations of an algorithm solving the longest common subsequence problem (*LCS*), yielding $O(n^3)$ computational complexity.

Finally, in Section 3 we consider the application of *LTS* as a relatively fast (but not always accurate) criterium for finding approximate tandem substrings of sequences and judging how well they match the original sequence.

2 An Efficient Algorithmic Approach to the *LTS* Problem

2.1 Notation and Problem Definition

Throughout the paper, the set of nonnegative integers is denoted by \mathbb{N} . Sets of consecutive elements of \mathbb{N} are referred to as *discrete intervals* and denoted by the symbol $\langle i, j \rangle$, which is equivalent to $\{i, i+1, \dots, j\}$.

Definition 1. A character sequence s of length n over nonempty alphabet Ω is a function $s : \langle 1, n \rangle \rightarrow \Omega$. The length $|s|$ of sequence s is the number of elements of the sequence, $|s| = n$. The symbol s_i , where $1 \leq i \leq n$, is used to denote $s(i)$, the i -th element of sequence s .

Sequence s is expressed in compact form as $s = [s_1 s_2 \dots s_n]$.

Definition 2. Sequence s of length $|s| = n$ is called a tandem sequence if n is an even number and $\forall_{1 \leq i \leq n/2} s_i = s_{i+n/2}$.

Definition 3. Sequence t , $|t| = k$, is called a subsequence of sequence s , $|s| = n$, if it is possible to indicate an increasing function $h : \langle 1, k \rangle \rightarrow \langle 1, n \rangle$, such that $\forall_{1 \leq i \leq k} t_i = s_{h(i)}$. This relation between sequences t and s is written in the form $t \subseteq s$.

Definition 4. The Longest Tandem Subsequence problem (*LTS* for short) for a given sequence s is the problem of finding a tandem sequence t such that $t \subseteq s$ and the length of sequence t is the maximum possible.

The suggested approach to the *LTS* problem reduces *LTS* for sequence s to the problem of determining the longest common subsequence of two sequences not longer than s .

Definition 5. The longest common subsequence $LCS(p, q)$ of sequences p and q is a sequence t , such that $t \subseteq p$ and $t \subseteq q$, of the maximum possible length.

The *LTS* problem for sequence $s = [s_1 s_2 \dots s_n]$ can be solved by means of an algorithm consisting of the following two stages.

Algorithm 1. Longest Tandem Subsequence

1. Determine an index l , $1 \leq l < n$ for which $|LCS([s_1 \dots s_l], [s_{l+1} \dots s_n])|$ takes the maximum possible value.
2. Compute sequence $t = LCS([s_1 \dots s_l], [s_{l+1} \dots s_n])$ and return as output.

The computational time and memory complexity of Algorithm 1 is dependent on the implementation of Stages 1 and 2. Both these steps will be analysed individually and shown to be solvable in $O(n^2)$ time using $\Theta(n)$ memory.

Stage 2 of Algorithm 1 can be implemented using Hirschberg's approach [1], who presented an algorithm which, given two character sequences p and q ($|p| = m$, $|q| = k$), computes the sequence $LCS(p, q)$ in $\Theta(mk)$ time and requires

$\Theta(m+k)$ memory. The strings whose longest common subsequence is determined in Stage 2 of Algorithm 1 have a total length of n , which closes the analysis of the complexity of this stage of the algorithm.

An efficient approach to Stage 1 of the algorithm is the subject of consideration in the following subsection. Since the problem of finding the index l for sequence s is of some significance and may even in certain applications be considered separately from the *LTS* problem (i.e. related to DNA sequencing, Section 3), it is useful to call it by its own name, referring to it as the *LTSsplit* problem.

2.2 A $\Theta(n^2)$ Time Algorithm for the *LTSsplit* Problem

Definition 6. *LTSsplit* is the problem in which, given a sequence s ($|s| = n$), we have to determine an index l , $1 \leq l \leq n$, such that the length of the string $LCS([s_1 \dots s_l], [s_{l+1} \dots s_n])$ is the maximum possible.

The suggested algorithmic solution to the *LTSsplit* problem is based on dynamic programming. In order to describe the lengths of the analyzed subsequences, it is convenient to define the family of functions f_k , for $1 \leq k \leq n$. For a given k , function $f_k : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N}$ is given as follows:

$$f_k(i, j) := \begin{cases} |LCS([s_1 \dots s_i], [s_j \dots s_k])|, & \text{for } 1 \leq i < j \leq k \\ 0, & \text{in all other cases when } i, j \geq 0 \\ -1, & \text{when } i, j < 0 \end{cases} \quad (1)$$

Index l may be expressed in terms of the function f_k by the following set of conditions:

$$\begin{cases} f_n(l, l+1) = \max_{r: 1 \leq r < n} (f_n(r, r+1)) \\ 1 \leq l < n \end{cases} \quad (2)$$

The values of function $f_k(i, j)$, for $1 \leq i < j \leq k \leq n$, can be expressed using a simple recursive formula:

$$f_k(i, j) = \begin{cases} \max \{f_k(i-1, j), f_{k-1}(i, j)\} & \text{when } s_i \neq s_k \\ f_{k-1}(i-1, j) + 1 & \text{when } s_i = s_k \end{cases} \quad (3)$$

In order to express values of function f_k using values of function f_{k-1} it is helpful to introduce the index $\gamma_k(i)$, defined as the largest value of r such that $r \leq i$ and $s_r = s_k$, or 0 if no such value exists. From formula (3) we have

$$f_k(i, j) = \max \{f_{k-1}(\gamma_k(i) - 1, j) + 1, f_{k-1}(i, j)\} \quad (4)$$

Let us now consider the family of functions $d_k : \mathbb{N}^+ \times \mathbb{N} \rightarrow \{0, 1\}$, defined for $1 \leq k \leq n$ as follows: $d_k(i, j) = f_k(i, j) - f_k(i-1, j)$. For the range of arguments $1 \leq i < j \leq k$ the value of function f_k may be expressed as

$$f_k(i, j) = \sum_{r=1}^i d_k(r, j) \quad (5)$$

A convenient characterization of function d is given by the following property.

Property 1. Let $1 \leq i < j \leq k$. The following statements hold

1. Suppose that $d_{k-1}(i, j) = 1$.
Then $d_k(i, j) = 0$ iff $\forall_{\gamma_k(i-1) \leq r < i} d_{k-1}(r, j) = 0$.
2. Suppose that $d_{k-1}(i, j) = 0$.
Then $d_k(i, j) = 1$ iff $s_i = s_k$ and $\exists_{\gamma_k(i-1) \leq r < i} d_{k-1}(r, j) = 1$.

Proof. Both claims of the property are proven below.

(*Claim 1.*) Let us assume that $d_{k-1}(i, j) = 1$ and let $d = \sum_{r=\gamma_k(i-1)}^{i-1} d_{k-1}(r, j)$ and $w = f_{k-1}(\gamma_k(i-1) - 1, j)$. By using formulae (5) and (4) we obtain $f_{k-1}(i-1, j) = w + d$ and $f_k(i-1, j) = \max\{w+1, w+d\} = w + \max\{1, d\}$ respectively. Moreover, since $f_{k-1}(\gamma_k(i-1) - 1, j) + 1 \leq f_{k-1}(i-1, j) + 1 = f_{k-1}(i, j) = w + d + 1$, by formula (4) $f_k(i, j) = w + d + 1$. Therefore $f_k(i, j) = f_k(i-1, j)$ iff $d = 0$.

(*Claim 2.*) First, assume that $d_{k-1}(i, j) = 0$ and $s_i = s_k$. Let w and d be defined as in the proof of claim 1. Acting identically as last time, we get $f_{k-1}(i-1, j) = w + d$ and $f_k(i-1, j) = w + \max\{1, d\}$. By formula (3) $f_k(i, j) = f_{k-1}(i-1, j) + 1 = w + d + 1$. Therefore $f_k(i, j) = f_k(i-1, j) + 1$ iff $d > 0$. Let us now assume that $d_{k-1}(i, j) = 0$ and $s_i \neq s_k$. Suppose that $f_k(i, j) = f_k(i-1, j) + 1$. From formula (3) we have $f_k(i, j) = f_{k-1}(i, j) = f_{k-1}(i-1, j) \leq f_k(i-1, j)$, a contradiction.

At this point it is essential to notice that function d_k has another interesting property, which is useful in the construction of an efficient algorithm for *LTSplit*.

Property 2. Given i and k , $1 \leq i < k \leq n$, the value of $d_k(i, j)$ is equal to 1 iff $j \in \langle i+1, a_k(i) \rangle$, for some function $a_k : \mathbb{N} \rightarrow \mathbb{N}$.

For an illustration of the function a_k , see Fig. 1.

Proof. For given i , the observation that the set $S = \{j : i < j \leq k \wedge d_k(i, j) = 1\}$ is a discrete interval whose left end equals $i+1$ is a consequence of the property

$F_k = \{f_k(i, j)\}$								$D_k = \{d_k(i, j)\}$								$A_k = \{a_k(i)\}$	
i	j	1	2	3	4	5	6	i	j	1	2	3	4	5	6	i	
1 B		0	1	1	1	0	0	1 B		0	1	1	1	0	0	1 B	4
2 A		0	0	2	2	1	1	2 A		0	0	1	1	1	1	2 A	6
3 B		0	0	0	2	1	1	3 B		0	0	0	0	0	0	3 B	0
4 B		0	0	0	0	1	1	4 B		0	0	0	0	0	0	4 B	0
5 C		0	0	0	0	0	1	5 C		0	0	0	0	0	0	5 C	0
6 A		0	0	0	0	0	0	6 A		0	0	0	0	0	0	6 A	0

Fig. 1. Values of functions f_k , d_k and a_k for sequence s beginning with BABBCA and $k = 6$

of monotonicity. More formally speaking, the proof proceeds by induction with respect to k .

If $k = 1$ then S is empty.

Let $k > 1$ and suppose that the inductive assumption holds for $k - 1$. It suffices to show that if $d_k(i, j) = 1$, then for an arbitrarily chosen t , $i < t < j$, we have $d_k(i, t) = 1$. We will consider two separate cases.

First, let $d_k(i, j) = 1$ and $d_{k-1}(i, j) = 1$. By claim 1 of Property 1, for some r , $\gamma_k(i - 1) \leq r < i$, we obtain $d_{k-1}(r, j) = 1$. By the inductive assumption we conclude that $d_{k-1}(r, t) = 1$ and $d_{k-1}(i, t) = 1$. The equality $d_k(i, t) = 1$ is a conclusion from claim 1 of Property 1.

Now, suppose $d_k(i, j) = 1$ and $d_{k-1}(i, j) = 0$. By claim 2 of Property 1 we have $s_i = s_k$ and for some r , $\gamma_k(i - 1) \leq r < i$, we have $d_{k-1}(r, j) = 1$. As in the previous case, $d_{k-1}(r, t) = 1$. The equality $d_k(i, t) = 1$ is a conclusion from either claim 1 or claim 2 of Property 1, depending on whether $d_{k-1}(i, t) = 1$ or $d_{k-1}(i, t) = 0$, respectively.

By definition of function d_k , $d_k(i, j) = 0$ when $j \leq i$, which completes the proof.

As a direct conclusion from Property 2, the values of function a_k uniquely determine all values of function d_k . It is possible to consider a unique representation of matrix $D_k = \{d_k(i, j)\}$ of dimension $n \times n$ ($1 \leq i, j \leq n$) in the form of the column of numbers $A_k = \{a_k(i)\}$ of dimension n ($1 \leq i \leq n$).

Theorem 1. *There exists an algorithm solving the $LTSplit$ problem for a given sequence s of length n in $O(n^2)$ time and using $\Theta(n)$ memory.*

Proof. The algorithm for solving the $LTSplit$ problem consists of the following steps:

1. For all k , $1 \leq k \leq n$, compute the column A_k by modifying column A_{k-1} , making use of Property 2.
2. Determine an $LTSplit$ index l from the values of column A_n using the following equation (directly inferred from the definitions of A_n , d_n , f_n):

$$f_n(i, i + 1) = |\{p : 1 \leq p \leq i \wedge a_n(p) \geq i + 1\}| \quad (6)$$

using condition (2) to guarantee the suitable choice of l .

The linear memory complexity of Steps 1 and 2 of the algorithm is evident. It is also obvious that Step 2 of the algorithm can be performed in $O(n^2)$ time (in fact, Step 2 can even be implemented with $O(n)$ running time, yet this is irrelevant to the proof). It now suffices to present a $O(n^2)$ approach to the problem of finding the column A_n in Step 1 of the algorithm.

To clarify this step, we will consider a geometrical presentation of column A_k as a set $P_k = \{p_1, \dots, p_k\}$ of k closed horizontal segments of the plane, where the segment p_i has vertical coordinate i , left horizontal coordinate 0 and right horizontal coordinate $a_k(i)$. For some k , consider a pair of values i, j , where $1 \leq i < j \leq k$. By definition of column A_k and set P_k , the value of $d_k(i, j)$ is 1

iff the point (i, j) belongs to some segment of P_k . We define the *visible section of segment p_i at height r* , $0 < r < i$, as a segment $q \subseteq p_i$ whose projection π_x to the horizontal axis fulfills the condition: $\pi_x(q) = \pi_x(p_i) \setminus \bigcup_{t=r}^{i-1} \pi_x(p_t)$. For the sake of completeness of the definition, the visible section of any segment at height 0 is assumed to be empty. The following corollary is a direct conclusion resulting from the analysis of Property 2.

Corollary 1. *Given set P_{k-1} , the set P_k may be constructed from P_{k-1} by performing the following transformations:*

1. *for all i , $1 \leq i \leq k$, such that $s_i = s_k$, remove segment p_i from the set and insert a segment with vertical coordinate i and horizontal coordinates 0 and k .*
2. *for all i , truncate the right part of segment p_i by removing the visible section of p_i at height $\gamma_k(i-1)$ of P_{k-1} from p_i .*

An example of the transformation of set P_{k-1} into set P_k is presented in Fig. 2. Since the operations described in Corollary 1 only modify the right endpoints of segments from P_k , the described procedure may be considered in terms of introducing appropriate modifications to the column A_k . The transformation from A_{k-1} to A_k can be performed in $O(k)$ time in two sweeps, once to detect the indices i for which $s_i = s_k$ and update the values as in Step 1 of the transformation, the other – to perform Step 2 of the transformation. Thus the column A_n can be obtained in $O(n^2)$ operations and the proof is complete.

In order to formalise the adopted approach, a complete implementation of both steps of the algorithm for *LTSSplit* is given below. To simplify the code, the two sweeps corresponding to Steps 1 and 2 of the transformation from A_{k-1} to A_k are performed in slightly modified order, which does not influence the correctness of the algorithm.

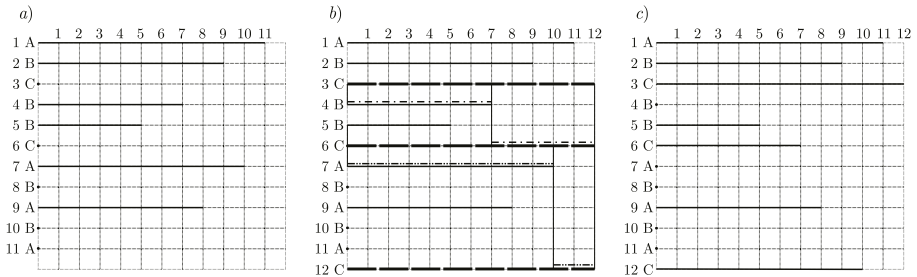


Fig. 2. An illustration of the transformation of set P_{k-1} into set P_k for a sequence beginning with ABCBBCABABAC ($k = 12$)

a) the set P_{k-1} b) the set after Step 1 of conversion (newly added segments are marked with a bold dashed line; segment fragments to be truncated are denoted by a dash-dot line) c) the set P_k

```

algorithm LTSplit ( $s$  : array 1.. $n$  of character) : integer;
var  $a, c, k, l, t$  : integer;
 $A := [0, \dots, 0]$  : array 1.. $n$  of integer;
begin
  {(*) Compute column  $A_k$  for  $k = 1, 2, \dots, n$ }
  for  $k$  in  $(1, 2, \dots, n)$  do
    for  $a$  in  $(k - 1, k - 2, \dots, 1)$  do
      if  $s[a] = s[k]$  then begin
         $t := A[a]$ ;
         $c := a + 1$ ;
        {Perform Step 2 of Corollary 1 using a downward sweep technique}
        while  $(t < k)$  and  $(c < k)$  do begin
          {Trim the segment corresponding to  $p_c$  to length  $t$ ,
            removing the section of it which is visible at height  $a$ }
           $(A[c], t) := (\min\{A[c], t\}, \max\{A[c], t\})$ ;
           $c := c + 1$ ;
        end;
         $A[a] := k$ ;
      end;
    {(**) Calculate index  $l$  using the column  $A_n$ }
     $c := 0$ ;
     $l := 1$ ;
    for  $k$  in  $(1, 2, \dots, n - 1)$  do begin
       $t := 0$ ;
      for  $a$  in  $(1, 2, \dots, k)$  do
        if  $A[a] > k$  then  $t := t + 1$ ;
      if  $t > c$  then begin
         $c := t$ ;
         $l := k$ ;
      end;
    end;
  return  $l$ ;
end.

```

2.3 Remarks on the Efficiency of the Algorithm for *LTS*

The approach to the *LTS* problem described in Algorithm 1 decomposes it into the *LTSplit* and *LCS* subproblems, both of which can be solved using $O(n^2)$ algorithms with a low coefficient of proportionality (similar for both algorithms on most system architectures).

The existence of faster algorithms for the problem appears unlikely, since no algorithm with $o(n^2)$ complexity is known for *LCS* in the case of general sequences. This may formally be stated as the following conjecture.

Conjecture 1. It is believed that the computational complexity of an algorithm solving the *LTS* problem is never lesser than the complexity of an optimal algorithm solving *LCS* for general sequences.

3 Final Remarks

One of the major issues of DNA string matching deals is the problem of finding the longest approximate tandem substring of a given DNA sequence. Formally speaking, a *longest approximate single tandem string repeat* in sequence s is defined as a substring $p \subseteq s$ (a subsequence composed of consecutive elements of s) of maximum possible length, which can be split into two similar substrings p_1, p_2 [4, 5]. The criterium of similarity may have varying degrees of complexity. Typically described criteria include the Hamming distance, the Levenshtein edit distance (elaborated on in [6]), as well as more complex criteria (expressing the distance in terms of weights of operations required to convert one sequence to the other, [2, 8]).

In some applications, the criterium used to describe the similarity of p_1 and p_2 is the length of $LCS(p_1, p_2)$. Given the sequence p , the *LTSsplit* algorithm can be applied to find the best point for splitting p so as to maximise $LCS(p_1, p_2)$. In consequence the output of the algorithm solving *LTS* directly leads to the answer to the two most relevant problems, namely whether p can be split into two similar fragments and, if so, what those fragments are.

Acknowledgement

The author would like to express his gratitude to Michał Małafiejski, Ph.D., from the Gdańsk University of Technology, for phrasing the *LTS* problem in its simplest form and for his kind and helpful contribution to the improvement of this paper.

References

1. Hirschberg D.S., A linear space algorithm for computing maximal common subsequences, *Information Processing Letters* (1975) **18**.
2. Kannan S.K., Myers E.W., An Algorithm for Locating Nonoverlapping Regions of Maximum Alignment Score. *SIAM Journal of Computing*, pp. 648–662 (1996) **25**.
3. Kolpakov R.M., Kucherov G., Finding Maximal Repetitions in a Word in Linear Time. *Symposium on Foundations of Computer Science FOCS'99*, New-York, pp. 596–604 (1999).
4. Kolpakov R.M., Kucherov G., Finding approximate repetitions under Hamming distance. *Theoretical Computer Science*, pp. 135–156 (2003) **303**.
5. Landau G.M., Schmidt J.P., An algorithm for approximate tandem repeats, *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching* pp. 120–133, Springer-Verlag (1993) **684**.
6. Levenshtein V.I., Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Phys. Dokl.*, pp. 707–710 (1966) **10**.
7. Main M.G., Lorentz R.J., An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, pp. 422–432 (1984) **5**.
8. Schmidt J.P., All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal of Computing*, pp. 972–992 (1998) **27**.