# New Tabulation and Sparse Dynamic Programming Based Techniques for Sequence Similarity Problems 

Szymon Grabowski<br>Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90-924 Łódź, Poland<br>sgrabow@kis.p.lodz.pl


#### Abstract

Calculating the length of a longest common subsequence (LCS) of two strings, $A$ of length $n$ and $B$ of length $m$, is a classic research topic, with many worstcase oriented results known. We present two algorithms for LCS length calculation with respectively $O\left(m n \log \log n / \log ^{2} n\right)$ and $O\left(m n / \log ^{2} n+r\right)$ time complexity, the latter working for $r=o(m n /(\log n \log \log n))$, where $r$ is the number of matches in the dynamic programming matrix. We also describe conditions for a given problem sufficient to apply our techniques, with several concrete examples presented, namely the edit distance, LCTS and MerLCS problems.


Keywords: sequence similarity, longest common subsequence, sparse dynamic programming, tabulation

## 1 Introduction

Measuring the similarity of sequences is an old research topic and many actual measures are known in the string matching literature. One classic example concerns the computation of a longest common subsequence (LCS) in which a subsequence that is common to all sequences and has the maximal possible length is looked for. A simple dynamic programming (DP) solution works in $O(m n)$ time for two sequences of length $n$ and $m$, respectively, but faster algorithms are known. The LCS problem has many applications in diverse areas, like version control systems, comparison of DNA strings, structural alignment of RNA sequences. Other related problems comprise calculating the edit (Levenshtein) distance between two sequences, the longest common transposition-invariant subsequence, or LCS with constraints in which the longest common subsequence of two sequences must contain, or exclude, some other sequence.

Let us focus first on the LCS problem, for two sequences $A$ and $B$. It is defined as follows. Given two sequences, $A=a_{1} \cdots a_{n}$ and $B=b_{1} \cdots b_{m}$, over an alphabet $\Sigma$ of size $\sigma$, find a longest subsequence $\left\langle a_{i_{1}}, a_{i_{2}}, \ldots, a_{i_{\ell}}\right\rangle$ of $A$ such that $a_{i_{1}}=b_{j_{1}}, a_{i_{2}}=$ $b_{j_{2}}, \ldots, a_{i_{\ell}}=b_{j_{\ell}}$, where $1 \leq i_{1}<i_{2}<\cdots<i_{\ell} \leq n$ and $1 \leq j_{1}<j_{2}<\cdots<j_{\ell} \leq$ $m$. The found sequence may not be unique. W.l.o.g. we assume $n \geq m$. To avoid uninteresting complications, we also assume that $m=\Omega\left(\log ^{2} n\right)$. Additionally, we assume that $\sigma=O(m)$. The case of a general alphabet, however, can be handled with standard means, i.e., we can initially map the sequences $A$ and $B$ onto an alphabet of size $\sigma^{\prime}=O(m)$, in $O\left(n \log \sigma^{\prime}\right)$ time, using a balanced binary search tree. We do not comprise this tentative preprocessing step in further complexity considerations.

Often, a simplified version of the LCS problem is considered, when one is interested in telling only the length of a longest common subsequence (LLCS).


## 2 Related work

A standard solution to the LCS problem is based on dynamic programming, and it is to fill a matrix $M$ of size $(n+1) \times(m+1)$, where each cell value depends on a pair of compared symbols from $A$ and $B$ (that is, only if they match or not), and its (at most) three already computed neighbor cells. Each computed $M[i, j]$ cell, $1 \leq i \leq$ $n, 1 \leq j \leq m$, stores the value of $\operatorname{LLCS}(A[1 \ldots i], B[1 \ldots j])$. A well-known property describes adjacent cells: $M(i, j)-M(i-1, j) \in\{0,1\}$ and $M(i, j)-M(i, j-1) \in\{0,1\}$

ars of research, surprisingly little can be said about the worstLCS. It is known that in the very restrictive model of unconarisons with equal/unequal answers only, the lower bound ched by a trivial DP algorithm. If the input alphabet is lower bound is simply or between $\leq$-comparisons are all und grows words, the gap betweer is and the huge. in 1977 by Hunt and $S \quad$ ae a milethe departure point f lgorithms nski (HS) algorithm is amic pro11 fraction isits only the matching cells of th cells. This kind of selective scan led sparse ing (SDP). We note that the number of all matches in $M$, denoted can be found in $O(n)$ time, and after this (negligible) preprocessing we can decide if the HS approach is promising to given data. More precisely, the brks in $O(n+r \log m)$ or even $O(n+r \log \log m)$ time. Note that in .e., for $r=\Theta(m n)$, this complexity is however superquadratic.
ymanski concept was an inspiration for a number of subsequent al'S calculation, and the best of them, the algorithm $O(D \log \log (\min (D, m n / D)))$ worst-case time (plus $D \leq r$ is the number of so-called dominant matches minant iff $M[i, j]=M[i-1, j]+1=M[i, j-1]+1)$. ( $m n$ ) for any value of $D$. A more recent algorithm, ent if the alphabet is very small (in particular, co time complexity is $O(m \sigma+\min (D \sigma, p(m-q))+n)$, where $p=L L$ $q=\operatorname{LLCS}(A[1 \ldots m], B)$.

A different approach is to divide the dynamic matrix into small bl the number of essentially different blocks is small enough to be precomputed before the main processing phase. In this way, the block may be processed in constant time
each, making use of a built loc first used to the LCS problem and refined by Bille and Fara pexities we space.
different a ic program technique was stant alphabet, alphabet. The $\log ^{2} n$ ), respecompute several There are a few such variants nces therein), all of them working in $O(\lceil m / w\rceil n)$ worst-case time, $m)$-time and $O(\sigma \quad$, where $w$ is the machine te of research con compressed form. CS algorithms for oressed inputs. We 78-factorization of $\log n$ ) time, where f two lution their computing their edit distance in $O$ ( (non-compressed) length.

Some other LCS-related results can be

## 3 LCS in $O\left(m n \log \log n / \log ^{2} n\right)$ time

In this section we modify the technique of Bille and Fa
improving its worst-case time by factor $\log \log n$, to achieve $C$ time complexity, with linear space. First we present the origina signal how our algorithm diverts from it. In the presentatio details of the BFC solution are changed, to make the descrip with our variant.

The dynamic programming matrix $M[0 \ldots n, 0 \ldots m]$ is divided into rectangular blocks with shared borders, of size $\left(x_{1}+1\right) \times\left(x_{2}+1\right)$, and the matrix is processed in horizontal stripes of $x_{2}$ rows. By "shared borders" we mean that e.g. the bottom row of some block being part of its output is also part of the input of the block below. Values inside each block depend on:
(i) $x_{1}$ corresponding symbols from sequence $A$,
(ii) $x_{2}$ corresponding symbols from sequence $B$,
(iii) the top row of the block, which can be encoded differentially in $x_{1}$ bits,
(iv) the leftmost column of the block, which can be encoded differentially in $x_{2}$ bits.

The output of each block will be found via a lookup table built in a preprocessing stage. The key idea of the BFC technique is alphabet remapping in superblocks of size $y \times y$. W.l.o.g. we assume that $x_{1}$ divides $y$ and $x_{2}$ divides $y$. Consider one superblock of the matrix, corresponding to the two substrings: $A\left[i^{\prime} y+1 \ldots\left(i^{\prime}+1\right) y\right]$ and $B\left[j^{\prime} y+1 \ldots\left(j^{\prime}+1\right) y\right]$, for some $i^{\prime}$ and $j^{\prime}$. For the substring $B\left[j^{\prime} y+1 \ldots\left(j^{\prime}+1\right) y\right]$ its symbols are sorted and $q \leq y$ unique symbols are found. Then, the $y$ symbols are remapped to $\Sigma_{B_{j^{\prime}}}=\{0 \ldots q-1\}$, using a balanced BST. Next, for each symbol from the snippet $A\left[i^{\prime} y+1 \ldots\left(i^{\prime}+1\right) y\right]$ we find its encoding in $\Sigma_{B_{j^{\prime}}}$, or assign $q$ to it if it wasn't found there. This takes $O(\log y)$ time per symbol, thus the substrings of $A$ and $B$ associated with the superblock are remapped in $O(y \log y)$ time. The overall alphabet remapping time for the whole matrix is thus $O((m n \log y) / y)$.


Figure 1. One horizontal stripe of the DP matrix, with 4 blocks of size $5 \times 5\left(x_{1}=x_{2}=4\right)$. The corresponding snippets from sequence $A$ and $B$ are abbea and fgadf, respectively. These snippets are translated to a new alphabet (the procedure for creating the new alphabet is not shown here) of size 6 , where the characters from $A$ are mapped onto the alphabet $\{0,1, \ldots, 4\}$ and value 5 is used for the characters from $B$ not used in the encoding of the symbols from $A$ belonging to the current superblock (the superblock is not shown here). The LCS values are stored explictly in the dark shaded cells. The white and dark shaded cells with arrows are part of the input, and their LCS values are encoded differentially, with regard to their left or upper neighbor. The diagonally shaded cells are the output cells, also encoded differentially. The bottom right corner (BR) is stored in three forms: as the difference to its left neighbor ( 0 or 1 ), as the difference to its upper neighbor ( 0 or 1 ) and the value of UL (upper-left corner) plus the difference between BR and UL. The difference between BR and UL is part of the LUT output for the current block.

This remapping technique allows to represent the symbols from the input components $(i)$ and $(i i)$ on $O(\log \min (y+1, \sigma))$ bits each, rather than $\Theta(\log \sigma)$ bits. It works because not the actual symbols from $A$ and $B$ are important for LCS computations, but only equality relations between them. To simplify notation, let us assume a large enough alphabet so that $\min (y+1, \sigma)=y+1$.

In this way, the input per block, comprising the components $(i)-(i v)$ listed above, takes $x_{1} \log (y+1)+x_{2} \log (y+1)+x_{1}+x_{2}$ bits, which cannot sum to $\omega(\log n)$ bits, otherwise the preprocessing time and space for building the LUT handling all possible blocks would be superpolynomial in $n$. Setting $y=x_{1}^{2}$ and $x_{1}=x_{2}=$ $\log n /(6 \log \log n)$, we obtain $O\left(m n(\log \log n)^{2} / \log ^{2} n\right)$ overall time, with sublinear LUT space.

Now, we present our idea. Again, the matrix is processed in horizontal stripes of $x_{2}$ rows and the alphabet remapping in superblocks of size $y \times y$ is used. The difference concerns the lookup table; instead of one, we build many of them. More precisely, for each (remapped) substring of length $x_{2}$ from sequence $B$ we build a lookup table for fast handling of the blocks in one horizontal stripe. Once a stripe is processed, its LUT is discarded to save space. This requires to compute the answers for all possible inputs in components (i), (iii) and (iv) (the component (ii) is fixed for a given stripe). The input thus takes $x_{1} \log (y+1)+x_{1}+x_{2}=x_{1} \log (2(y+1))+x_{2}$ bits.

The return value associated with each LUT key are the bottom and the right border of a block, in differential form (the lowest cell in the right border and the rightmost cell in the bottom border are the same cell, which is represented twice;


1) to its left neighbor in the bottom border and once as ts upper neighbor in the right border) and the difference bottom right and the top left corner (to know the explicit right corner), requiring $x_{1}+x_{2}+\log \left(\min \left(x_{1}, x_{2}\right)+1\right)$ bits
nd the output of an LUT fits a machine word, i.e., does not cess one block in constant time. Still, as in the original BFC ling costs also impose a limitation. More precisely, we are tal time of remapping the alphabet in all the superblocks, building all $O\left(m / x_{2}\right)$ LUTs and finally processing all the blocks, which is described by the formula:

$$
O\left(m \log y+(m n \log y) / y+\left(m / x_{2}\right) 2^{x_{1} \log (2(y+1))+x_{2}} x_{1} x_{2}+m n /\left(x_{1} x_{2}\right)\right)
$$

where $2^{x_{1} \log (2(y+1))+x_{2}}$ is the number of all possible LUT inputs and the $x_{1} x_{2}$ multiplier corresponds to the computation time per one LUT cell. Let us set $y=\log ^{2} n / 2$, $x_{1}=\log n /(4 \log \log n)$ and $x_{2}=\log n / 4$. In total we obtain $O\left(m n \log \log n / \log ^{2} n\right)$ time with $o(n)$ extra space (for the lookup tables, used one at a time, and alphabet remapping), which improves the Bille and Farach-Colton result by factor $\log \log n$. The improvement is achieved thanks to using multiple lookup tables (one per horizontal stripe). Formally, we obtain the following theorem.

Theorem 1. The length of the longest common subsequence (LCS) between two sequences, $A$, of length $n$, and $B$, of length $m$, where $n \geq m \geq \log ^{2} n$, both over an integer alphabet, can be computed in $O\left(m n \log \log n / \log ^{2} n\right)$ worst-case time. The algorithm needs o( $n$ ) words of space, apart for the two sequences themselves.

## 4 LCS in $O\left(m n / \log ^{2} n+r\right)$ time (for some $r$ )

In this algorithm we also work in blocks, of size $(b+1) \times(b+1)$, but divide them into two groups: sparse blocks are those which contain at most $K$ matches and dense blocks are those which contain more than $K$ matches. Obviously, we do not count possible matches on the input boundaries of a block.

We observe that knowing the left and top boundary of a block plus the location of all the matches in the block is enough to compute the remaining (right and bottom) boundaries. This is a nice property as it eliminates the need to (explicitly) access the corresponding substrings of $A$ and $B$.

The sparse block input will be encoded as:
(i) the top row of the block, represented differentially in $b$ bits,
(ii) the leftmost column of the block, represented differentially in $b$ bits,
(iii) the match locations inside the block, each in $\log \left(b^{2}\right)$ bits, totalling $O(K \log b)$ bits.

Each sparse block will be blocks, on the other hand, will handled with our algorithm fro the LUT build costs would be this algorithm would never be that $K=\Theta(\log n / \log \log n)$,


As this algorithm's worst-case time is $\Omega\left(m n / \log ^{2} n\right)$, it is easy to notice that the preprocessing costs for building required LUTs and alphabet mapping will not dominate. Each dense block is divided into smaller blocks of size $\Theta(\log n / \log \log n) \times$ $\Theta(b)$. Let the fraction of dense blocks in the matrix be denoted as $f_{d}$ (for example, if half of the $(b+1) \times(b+1)$ blocks in the matrix are dense, $\left.f_{d}=0.5\right)$. The total time complexity (without preprocessing) is then

$$
O\left(\left(1-f_{d}\right) m n / b^{2}+f_{d}(m n \log \log n /(b \log n))\right) .
$$

The fraction $f_{d}$ must be $o(1)$, otherwise this algorithm is not better in complexity than the previous one. This also means that $1-f_{d}$ may be replaced with 1 in further complexity considerations.

Recall that $r$ is the number of matches in the dynamic programming matrix. We have $f_{d}=O\left((r / K) /\left(m n / b^{2}\right)\right)=O\left(r b^{2} \log \log n /(m n \log n)\right)$. From the $f_{d}=o(1)$ condition we also obtain that $r b^{2}=o(m n \log n / \log \log n)$. If $r=o(m n /(\log n \log \log n))$, then we can safely use the maximum possible value of $b$, i.e., $b=\Theta(\log n)$ and obtain the time of $O\left(m n / \log ^{2} n\right)$.

Unfortunately, in the preprocessing we have to find and encode all matches in all sparse blocks, which requires $O(n+r)$ time. Overall, this leads to the following theorem.

Theorem 2. The length of the longest common subsequence (LCS) between two sequences, $A$, of length $n$, and $B$, of length $m$, where $n \geq m \geq \log ^{2} n$, both over an integer alphabet, can be computed in $O\left(\mathrm{mn} / \log ^{2} n+r\right)$ worst-case time, assuming $r=o(m n /(\log n \log \log n))$, where $r$ is the number of matching pairs of symbols between $A$ and $B$.

Considering to the presente than the result from the previo

On the other hand, it is ess the one from Eppstein et al. dominant matches $D$ is the $D$ complexity formula to obtain $O$ $n n /\left(\log ^{2} n\right.$ $\log ^{2} n \log l d$ in this section is
 e is yet another constraint. From the comparison to Sakai's algode that our algorithm needs $\sigma=\omega(\log \log \log n)$ to dominate for $\left.n /\left(\log ^{2} n \log \log \log n\right)\right)$.

## applications

The techniques presented in the two previous sections may be applied to any sequence similarity problem fulfilling certain properties. The conditions are specified in the following lemma.

Lemma
subsequer

[^0]similarity problem returning the length of a desired uces, $A$ of length $n$ and $B$ of length $m$, both over a is $\min \left(r, m^{2}\right)$, but it may matter, in complexity terms, only ich is a less interesting case.
common integer alphabet $\Sigma$ of size $\sigma=O(m)$. We assume that $1 \leq m \leq n$. Let $Q$ admit a dynamic programming solution in which $M(i, j)-M(i-1, j) \in\{-1,0,1\}$, $M(i, j)-M(i, j-1) \in\{-1,0,1\}$ for all valid $i$ and $j$, and $M(i, j)$ depends only on the values of its (at most) three neighbors $M(i-1, j), M(i, j-1), M(i-1, j-1)$, and whether $A_{i}=B_{j}$.

There exists a solution to problem $Q$ with $O\left(m n \log \log n / \log ^{2} n\right)$ worst-case time. There also exists a solution to $Q$ with $O\left(m n / \log ^{2} n+r\right)$ worst-case time, for $r=$ $o(m n /(\log n \log \log n))$, where $r$ is the number of symbols pairs $A_{i}, B_{j}$ such that $A_{i}=$ $B_{j}$. The space use in both solutions is $O(n)$ words.

Proof. We straightforwardly apply the ideas presented in the previous two sections.
 $s$ to allow a broader range of differences $(\{-1.0 .1\})$ between amic programming matrix. tant factor ly serves to calculate the ed e between the BFC technique was pre of the edit obtain the following theore distance between two seque and $B$, of length $m$, where $n \geq m \geq \log ^{2} n$, both over an int mputed in $O\left(m n \log \log n / \log ^{2} n\right)$ worst-case time. Alternatively, the distance can be found in $O\left(m n / \log ^{2} n+r\right)$ worst-case time, for $r=o(m n /(\log n \log \log n))$, where $r$ is the numbe such that $A_{i}=B_{j}$. The space use in both solutions is $O(n)$

Ar
quenc
$t)\left(s_{2}\right.$
spond

the longest common transposition-invariant subsewe look for a longest subsequence of all $s_{i}$ belong to $A$ (in increasing o , $B$ (in increasing order), and $t \in\{$ nsposition. This problem is motivat known results for LCTS are $O(m n$ $\left.)^{2} / \log ^{2} n\right)$ if the BFC technique is applied for a $\left.\sigma=O\left(\log ^{2} n /(\log \log n)^{2}\right)\right)$. Applying the fort ssible transpositions, gives immediately $O(m n \sigma]$ $=O\left(n^{1-\varepsilon}\right)$, for any $\varepsilon>0$, otherwise the LUT b
; the latter result requires more care. First we notice that the over all th p to $m n$, so $\Theta(m n)$ is the total et us div ato dense ones and sparse ones, are those $\quad \mathrm{g} \log n / \sigma$ matches. The number ms is thu: $\quad \sigma n)$. We handle dense transpoanique fro ones with the technique from ; us $O(m$ total tir $\quad \omega\left(\log n(\log \log n)^{2}\right)$, as this conthe numb sparse transposition limited to $)$ ), as red $\sigma=\omega\left(\log ^{2} n /(\log \log n)^{2}\right)$ and niche in which our algorithm is the first one to achieve $O(m n)$ total time.

Theorem 5. The length of the longest common transposition-invariant subsequence (LCTS) between two sequences, $A$, of length $n$, and $B$, of length $m$, where $n \geq m \geq$


Lemma 6. Let $Q$ be a scom returning the length of a desired subsequence, involving three sequences, $A$ of length $n, B$ of length $m$ and $P$ of length $u$, all over a common integer alphabet $\Sigma$ of size $\sigma=O(m)$. We assume that $1 \leq m \leq n$ and $u=\Omega\left(n^{c}\right)$, for some constant $c>0$. Let $Q$ admit a dynamic programming solution in which $M(i, j, k)-M(i-1, j, k) \in\{-1,0,1\}, M(i, j, k)-M(i, j-1, k) \in$ $\{-1,0,1\}$ and $M(i, j, k)-M(i, j, k-1) \in\{-1,0,1\}$, for all valid $i, j$ and $k$, and $M(i, j, k)$ depends only on the values of its (at most) seven neighbors: $M(i-1, j, k)$, $M(i, j-1, k), M(i-1, j-1, k), M(i, j, k-1), M(i-1, j, k-1), M(i, j-1, k-1)$ and $M(i-1, j-1, k-1)$, and whether $A_{i}=B_{j}, A_{i}=P_{k}$ and $B_{j}=P_{k}$.

There exists a solution to $Q$ with $O\left(m n u / \log ^{3 / 2} n\right)$ worst-case time. The space use is $O(n)$ words.

Proof. The solution works on cubes of size $b \times b \times b$, setting $b=\Theta(\sqrt{\log n})$ with an appropriate constant. Instead of horizontal stripes, 3D"columns" of size $b \times b \times u$ are now used. The LUT input consists of $b$ symbols from sequence $P$, encoded with respect to a supercube in $O(\log \log n)$ bits each, and three walls, of size $b \times b$ each, in differential representation. The output are the three opposite walls of a cube. The restriction $u=\Omega\left(n^{c}\right)$ implies that the overall time formula without the LUT build times is $\Omega\left(m n^{1+c} / \log ^{3 / 2} n\right)$, which is $\Omega\left(m n^{1+c^{\prime}}\right)$, for some constant $c^{\prime}, c \geq c^{\prime}>0$, e.g., for $c^{\prime}=c / 2$. The bu in be made $O\left(m n^{1+c^{\prime \prime}}\right)$, for any constant
 chosen appropriately. We now set $c^{\prime \prime}=c^{\prime}$ dominating.
the merged longest common subsequence sequences, $A, B$ and $P$, and its returned sequence of $P$ and can be split into two sequence of $A$ and $T^{\prime \prime}$ is a subsequence of he DP formula for this problem $M(i, j, k)$ hbors: $M(i-1, j, k), M(i, j-1, k)$ and
 volving three sequences, $A, B$ and $P$, of length respectively $n$, $m$ and $u$, where $m \leq n$ and $u=\Omega\left(n^{c}\right)$, for some constant $c>0$, all over an integer alphabet of size $\sigma$, can be computed in $O\left(m n u / \log ^{3 / 2} n\right)$ worst-case time.

## 6 Conclusions

On the example of the longest common subsequence problem we presented two algorithmic techniques, making use of tabulation and sparse dynamic programming paradigms, which allow to obtain competitive time complexities. Then we generalize the ideas by specifying conditions hose fulfilments lead to immediate applications of these techr lems considered here as applications comprise the edit distan

As a future work, we are going example improve the SEQ-EC-LCS re to improve the tabulation based resu
dencies, which may for research option is to try ences.

## Acknowledgments

The author wishes to thank Sebastian Deorowicz and an anonymous reviewer for helpful comments on a preliminary version of the manuscript.

## References

1. A. Apostolico: String editing and longest common subsequences, in Handbook of Formal Languages, vol. 2 Linear Modeling: Background and Application, Springer, 1997, ch. 8, pp. 361398.
2. L. Bergroth, H. Hakonen, and T. Raita: A survey of longest common subsequence algorithms, in SPIRE, IEEE Computer Society, 2000, pp. 39-48.
3. P. Bille and M. Farach-Colton: Fast and compact regular expression matching. Theoret. Comput. Sci., 409(3) 2008, pp. 486-496.
4. M. Crochemore, G. M. Landau, and M. Ziv-Ukelson: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput., 32(6) 2003, pp. 1654-1673.
5. S. Deorowicz: Speeding up transposition-invariant string matching. Inform. Process. Lett., 100(1) 2006, pp. 14-20.
6. S. Deorowicz and A. Danek: Bit-parallel algorithms for the merged longest common subsequence problem. Internat. J. Foundations Comput. Sci., 24(08) 2013, pp. 1281-1298.
7. S. Deorowicz and S. Grabowski: Subcubic algorithms for the sequence excluded LCS problem, in Man-Machine Interactions 3, 2014, pp. 503-510.
8. D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano: Sparse dynamic programming I: Linear cost functions. J. ACM, 39(3) 1992, pp. 519-545.
9. P. GaWrychowski: Faster algorithm for computing the edit distance between SLP-compressed strings, in SPIRE, 2012, pp. 229-236.
10. D. S. Hirschberg: An information-theoretic lower bound for the longest common subsequence problem. Inform. Process. Lett., 7(1) 1978, pp. 40-41.
11. K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng: Efficient algorithm for finding interleaving relationship between sequences. Inform. Process. Lett., 105(5) 2008, pp. 188-193.
12. J. W. Hunt and T. G. Szymanski: A fast algorithm for computing longest common subsequences. Comm. ACM, 20(5) 1977, pp. 350-353.
13. H. Hyyrö: Bit-parallel LCS-length computation revisited, in AWOCA, Australia, 2004, University of Sydney, pp. 16-27.
14. V. Mäkinen, G. Navarro, and E. Ukkonen: Transposition invariant string matching. Journal of Algorithms, 56(2) 2005, pp. 124-153.
15. W. Masek and M. Paterson: A faster algorithm computing string edit distances. J. Comput. Syst. Sci., 20(1) 1980, pp. 18-31.
16. G. Navarro, S. Grabowski, V. Mäkinen, and S. Deorowicz: Improved time and space complexities for transposition invariant string matching, Technical Report TR/DCC-2005-4, Department of Computer Science, University of Chile, 2005.
17. Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor: Efficient sparse dynamic programming for the merged LCS problem with block constraints. International Journal of Innovative Computing, Information and Control, 6(4) 2010, pp. 1935-1947.
18. Y. SAKAI: A fast on-line algorithm for the longest common subsequence problem with constant alphabet. IEICE Transactions, 95-A(1) 2012, pp. 354-361.
19. C. K. Wong and A. K. Chandra: Bounds for the string editing problem. J. ACM, 23(1) 1976, pp. 13-16.

[^0]:    ${ }^{1}$ A slightl if $m=o$

