# A Modified algorithm to find Longest Common Subsequences and optimizing space, time

**Abstract**

In the study of Longest Common Subsequence (LCS) is an important topic and an important component in the application of Computational Mathematics. The LCS problem is to find a subsequence which is common to at least two or more given sequences. The largest length subsequence is called as LCS. Due to high demands for computational time, power and memory, this paper introduces a new efficient modified algorithm to find the longest common subsequences in two different sequences X and Y. The sequences represented in memory in vertical and horizontal directions. An array is established where each sequence assigned in this array. A new node is added to it for every match between two sequences. If two or more matches in different locations in Sequence Y share the same in X, the corresponding node will construct the LCS in various ways. Continuing in this process we obtain a group of LCS between the sequences X and Y. The proposed modified algorithm has been implemented and tested using Matlab language. This algorithm shows very good speedups and indicated efficiently minimizing the space complexity and optimizing the time taken to execute and impressive improvements has been achieved.

## 1 Introduction

In the study of Longest common Subsequence (LCS) is an important topic and an important component in the application of Computer Science and Mathematics to apply DNA matching in Molecular Biology. Many Scientific tasks involve measuring the similarity between two data sets.

In a diverse array of applications ranging from automated spell-checking to alignment of DNA sequences, the data we wish to compare is naturally represented as a sequence of letters. In this context, two of the most foundational and popular measures sequence similarity are the Longest Common Subsequence.

A subsequence of a string is a sequence of letters appearing left to right in the original string. Hence, given two sequences A and B. Intuitively, two strings with a larger LCS value are more similar than two strings with a smaller LCS.
Consequently, much recent research on sequence similarity has involved finding faster algorithms for returning good approximations for LCS. There has been a long line of work obtaining better and better approximation algorithms running in near linear time.

## 2 Literature Review

A large number of research has been conducted in finding LCS between two sequences. The Needleman Wunsch [1] algorithm was the first application of dynamic programming which provide a global alignment between two sequences. This algorithm leads to the evolution of various efficient LCS algorithms. It is only suitable if the two sequences are similar of length. The Hirschberg [2] algorithm evolved from Needleman-Wunsch algorithm provides optimize version of Needleman- Wunsch. Another proposal for LCS proposed and optimized for [2] is proposed in [3]. Various parallel algorithms have been proposed in earlier to reduce the computation time and such algorithms are CREW PRAM model and Systolic arrays. In [4], they proposed fast LCS algorithm. Fast LCS efficiency which has been proposed in [4], further improved in [5]. A parallel LCS algorithm based on dynamic programming has also been proposed in [6].
More recently, A parallel formulation of the anti diagonal algorithm has been proposed [10] to the

LCS algorithm using GPU based model. Although these approaches offer faster in execution times and are still expensive in nature, hence only few computers are equipped with GPU.
It is noted in [11] on the efficient calculation of the LCS measures. Apart from the applications in computational biology the necessity to calculate this measure arises, for example in data compression[12]. For fixed $m$ polynomial algorithms based on the dynamic programming (DP) are known[13] to solve the LCs problem. Standard dynamic programming algorithm takes $O(n^m)$ time, where $n$ denotes the longest common subsequence. These exact methods become quickly impractical when $m$ increases and $n$ is not small.

## 3 Problem Description

LCS approach required a large amount of time for the deluge of genetic data which is represented billions of characters. Time for finding LCS can be reduced tremendously if we can be able to solve the problem using non -alignment based distributed algorithm. When a new LCS is found, it is important to know what other sequences it is most like that and find those sequences. Sequence comparison has been used successfully to establish the limit between genes and gene evolved in normal growth and development. One way of detecting the similarity of two or more sequences is found their LCS.

A subsequence of a given sequence of letters is just the given sequence with some letters (possibly none) left out. Generally, gives a sequence of letter $X = (x_1, x_2, \ldots\ldots, x_m)$, the sequence
$Z = (z_1, z_2, \ldots\ldots, z_k)$ is a subsequence of $X$ if there exists a strictly increasing sequence
$(z_1, z_2, \ldots\ldots, z_k)$ of indices of X such that for all $j = 1, 2, \ldots\ldots k$, we have $x_{ij} = z_j$. For example $Z = (B, C, D, B)$ is a subsequence of $X = (A, B, C, B, D, A, B)$ with corresponding index sequence $(2, 3, 5, 7)$.
Given two sequences $X$ and $Y$, we say that a sequence $Z$ is a common Subsequence of $X$ and $Y$ if $Z$ is a Subsequence of both $X$ and $Y$.
For example if $X = (A, B, C, B, D, A, B)$ and $Y = (B, D, C, A, B, A)$, the subsequence $(B, C, A)$ is a Common Subsequence of both $X$ and $Y$. The Sequence $(B, C, A)$ is not a longest Common Subsequence (LCS) of $X$ and $Y$, however, since it has length 3 and the sequence $(B, C, B, A)$ is LCS of $X$ and $Y$, as is the sequence $(B, D, A, B)$, since there is no common Subsequence of length 5 or greater.

## 4 Existing approach:

In the LCS problem, we are given two sequences $X = (x_1, x_2, \ldots\ldots, x_m)$ and $Y = (y_1, y_2, \ldots\ldots, y_n)$ and wish to find minimum length common sequence of X and Y.
In this approach to solve LCS problem is to enumerate all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the LCS found. Each subsequence of X corresponds to a subset of indices $(1, 2, \ldots\ldots.., m)$ of $X$, so this approach requires exponential time, making it impractical for long sequence.
The common and popular algorithm of finding the LCS between two strings is the well-known dynamic programming approach. A DNA of any organism is a linear sequence as a basic structure $x_1, x_2, x_3.., x_m$ of nucleotide. Each $x_i$ is recognized by the set of the four alphabets which are: $A, T, G, C$. Applications in the field of bioinformatics require comparing the DNA of various organisms. A sample of DNA comprises a sequence of molecules known as bases, which are possibly Adenine,Cystosine,Guanine and Thymine.ie $A, C, G, T$[7].

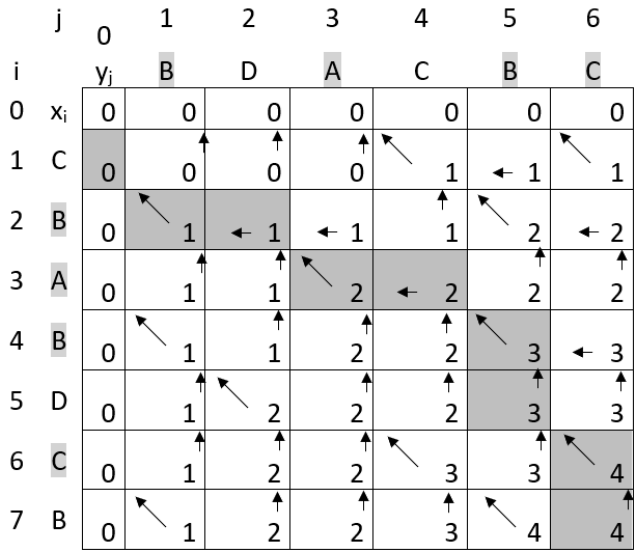


Figure 1: add the caption

Step1: Identifying a longest common subsequence.

**Theorem 4.1** The optimal substructure property of LCS problem.

Let, $X = (x_1, x_2, \ldots\ldots, x_m)$ and $Y = (y_1, y_2, \ldots\ldots, y_n)$ be sequences. Let $C = (c_1, c_2, \ldots\ldots, c_i)$ be any LCS of $X$ and$Y$. The theorem states three cases given below:

1. If $x_m = y_n$ , then $c_i = x_m = y_n$ and $C_{i-1}$ is an LCS of $x_{m-1}$ and$y_{n-1}$.
2. If $x_m \neq y_n$ , then $c_i \neq X_m$ means that $c$ is an LCS of $x_{m-1}$ and$Y$ .
3. If $x_m \neq y_n$ , then $c_i \neq y_n$ means that $C$ is an LCS of $X$ and $y_{n-1}$ .

Step 2: Generate a recursive loop for LCS solution

$$C[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0. \\ C[i-1, j-1] + 1, & \text{if } i, j > 0 \text{ and } a_i = b_j. \\ max(C[i, j-1], C[i-1, j]), & \text{if } i, j > 0 \text{ and } a_i \neq b_j \end{cases}$$

Step3: Calculating the length of LCS

The length of LCS of strings X and Y is denoted by $r(X, Y)$, or when the input strings are known, by $r$. Length of LCS $(r) = C[m-1, n-1]$.

Step4: Backtracking to construct an LCS.

In the two given sequences $X$ and $Y$, let's say $C$ be a common subsequence between two sequences if the subsequence C exists in both the strings [8]. In the given figure $X = (C, B, A, B, D, C, B)$ and $Y = (B, D, A, C, B, C)$, the sequence $(B, A, C)$ is common but not the longest common subsequence of $X$ and $Y$. However, since it has length 3 whereas the sequence $(B, A, B, C)$ is also exactly same in both $X$ and $Y$, with length 4. Hence the sequence $(B, A, B, C)$ is an LCS of $X$ and $Y$, same as the sequence $(B, D, C, B)$, since $X$ and $Y$ have no common subsequence of length more than 4.

It is noted that we are given two sequences X and Y we wish to have a maximum-length common subsequence which can be solved by dynamic programming.

with this procedure, we meet consequences of this LCS in reverse order by traversing the matrix by backtracking. In order to get an appropriate result, recursive methods is to be used to print the

LCS of $X$ and $Y$. In Figure1., this procedure prints $BABC$. The procedure takes time $O(m+n)$, since it has nested loop of $I$ and $j$ which decrease by 1 in each iteration.

# 5 Modified Approach

Steps for the Algorithm

1. Find the LCS:
   First, the algorithm identifies the longest common subsequence (LCS) between a given pair (or set) of sequences. The LCS problem is typically solved using dynamic programming. It finds the longest subsequence common to two sequences, which doesn't have to be contiguous but must appear in the same order.

2. Search for Similar Subsequences:
   After finding the LCS, the algorithm would search through other sequences to identify other subsequences of the same length. This step involves comparing the length of the LCS with subsequences in other sequences, and checking for the existence of a subsequence that matches the length.

   - Matching Condition: The subsequences in the other sequences could be similar to the LCS, meaning they share the same length and potentially some common elements or patterns.

3. Return Similar Sequences:
   After finding these subsequences of the same length in the other sequences, the algorithm would return the results, showing how the LCS appears in other sequences of similar length.

Example

For example, if the LCS found between two sequences is "ABC", the algorithm would then search through other sequences to find subsequences that also have the length of 3 (or the length of the LCS), such as "DEF", "XYZ", etc., that share a similar pattern of common subsequences.

Possible Extensions

- Similarity Metrics: You could define additional similarity measures beyond just length, such as edit distance or Jaccard similarity between subsequences, to capture more complex similarities.
- Multiple Sequence LCS: If working with multiple sequences (more than two), you could extend the LCS search to operate on more than two sequences at a time.
- LCS (Longest Common Subsequence): Given two sequences (strings, arrays, etc.), the LCS is the longest sequence that appears in both of them in the same order, but not necessarily consecutively.
- Algorithm: After finding the LCS, the algorithm could extend its search to find all other subsequences of the same length from the two sequences that match.

Step-by-step approach:

1. Find the LCS: The first step is the standard LCS algorithm, where you compute the longest common subsequence between two given sequences. This can be done using dynamic programming with a 2D table to store intermediate results, and the time complexity is typically $O(m \times n)$, where m and n are the lengths of the two sequences.

2. Identify Other Subsequences of the Same Length:

- Once the LCS is found, the next step is to find other subsequences of the same length from both sequences.
- This can be achieved by scanning both sequences and extracting all possible subsequences of the same length as the LCS.
- For each subsequence, you check if it appears in both sequences in the same order.
- Depending on the context, this may involve using additional data structures like sets or hash maps to store and compare subsequences efficiently.

3. Return Similar Subsequences: Once all matching subsequences of the same length are found, the algorithm can return them, either as a list of subsequences or as a set of unique subsequences, depending on the specific requirements.

Example: Let's consider two strings: String 1: "ABCBDAB" String 2: "BDCAB" The LCS between these two strings is "BDAB", which has a length of 4. The algorithm would:

- Find this LCS.
- Then search for all subsequences of length 4 that appear in both strings. For example, "BDAB", "BCAB", or other possible subsequences of the same length.

Efficiency Considerations:

- Computational Complexity: Searching for all subsequences of the same length can be computationally expensive. Extracting all subsequences from both strings involves generating combinations, which could be $O(2^m)$ for a string of length mm. This could be impractical for long sequences.
- Optimization: The process could be optimized by using dynamic programming or suffix trees/arrays for faster substring and subsequence search.

This algorithm is useful in applications such as bioinformatics (where finding common patterns in DNA or protein sequences is important), text comparison, or version control systems.

$LCS[i, j]$ is the length of LCS of $S[1, 2......, i]$ with $T[1, 2, ....., j]$. one we can solve for $LCS[i, j]$ interms of the LCS's problems.

Case I: If $S[i] \neq T[j]$, then the desired subsequence has to ignore one of S[i] or T[j], so $LCS[I, j] = \max(LCS[i-1, j-1], LCS[i, j-i])$.

Case II: If $S[i] = T[j]$ ,then the LCS of $S[1, 2......, i]$ and $T[1, 2, ....., j]$ might as well as match them up. $LCS[i, j] = 1 + LCS[i-1, j-1]$.

In this paper we have proposed modified algorithm in order to find LCS for sequence of letters . It is mainly used for protein sequence and DNA sequence. After examine this algorithm, we analyze that, this algorithm can efficiently be used for improving time and space complexity.
Some preliminary definitions are as follows: We represent the concatenation of sequences $X$ and $Y$ by $XIIY$.
We are given two sequences $X$ and $Y$ of length $n$ and $m$ respectively.

$$X = x_1, x_2, ..., x_n,$$

$$Y = y_1, y_2, ..., y_n,$$

where $x_i$ and $y_i$ are chosen from finite alphabets.
Eg. $X = \{A, B, Q, P\},\ Y = \{P, Q, R, S, T\}$,
$x_i \in \{A, B, Q, P\}, y_i \in \{P, Q, R, S, T\}$
The aim is to determine the length of all similar subsequences of $X$ and $Y$ as well as their locations. Achieving this objective, the modifies approach has been divided into two parts such as matching and space.
This comparision of two sequences for all possible identical matches as for matching. The modified algorithm aim to determine the length as well as location of all possible common subsequences between two sequences represented in $X$ and $Y$. For each matching location and length will be added to the current process. It performs continuously reading new pair of characters $x$ and $y$ are equal or terminate. The length and location of the subsequence which is obtained are considered as input to the process.The matching process yields the common subsequence and and place their locations and length in separate array which the space is minimized. This shows the optimum in using for space.
It can be illustrated as
(a) Prefixes are eliminated from both sequences of letters.
(b) LCS matrix is created with order$(m, n)$ instead of order$(m + 1, n + 1)$. It reduces the number of iterations.
(c)In the back tracking technique, the number of iterations are reduced.
(d)This modified algorithm optimizes the space for storing and number of iterations are reduced. It shows the optimal process time.

$p_{1i}$, represents the string $p_1, p_2, ....p_i$ (elements 1 through $i$ of string $P$). Similarly, the prefix of length $j$ of string $Q$ is represented by $Q_{1j}$.
We define $L(i, j)$ to be the length of the LCS of prefixes of lengths $i$ and $j$ of strings $P$ and $Q$, i e. the length of the LCS of $P_{1i}$ and $Q_{1j}$.

**Theorem 5.1** *For $n \geqslant 1$, $(i, j)$ is $n$-letters iff $L(i, j) \geqslant n$ and $p_i = q_j$. Thus, there is a common subsequence of length $n$ of $P_{1i}$ and $Q_{1j}$*

*Proof.* By induction on $n$. $(i, j)$ Is a 1-letter if and only if $a = b$ (by definition), in which case $L(i, j)$ necessarily is at least 1. Thus the theorem is true for $n = 1$.
Assume it is true for $n - 1$, consider $n$ if $(i, j)$ $n$-letter then there exists $i' < i$ and $j' < j$ such that $(i', j')$ is $n - 1$ letter sequence. By assumption there is a subsequence $D^i = d_1, d_2, ..., d_{n-1}$ of $P_{1i'}$ and $Q_{1j'}$.
Since $p_i = q_j((i, j)$ is a $n$-letter), $D = D^i$ is a common sequence of $P_{1i}$ and $Q_{1j}$.

Thus $L[i, j] \geqslant n$
Conversely $L[i, j] \geqslant n$ and $p_i = q_j$ then there exists $i' < i$ and $j' < j$ such that
$p'_i = q'_j$ and $L[i', j'] = L[i, j] \geqslant n - 1$,
$(i', j')$ is a $n - 1$ letter sequence (by Inductive hypothesis) and thus $i, j$ is a $n$-letter sequence.
The length of LCS is $p$, the maximum value of $n$ such that there exists a $n$-letter sequence. As we see, to recover an LCS, if suffices to maintain the sequence of a 0-letter, 1-letter,...$(p - 1)$-letter and a $n$-letter such that in this sequence $i$-letter generate the $i + 1$ letter for $0 \leqslant i < p$. □

**Lemma 5.2** *Let $x = (x_1, x_2)$ and $y = (y_1, y_2)$ be two $n$-candidates. $x_1 \geqslant x_2$ and $y_1 \geqslant y_2$ then we say that $y$ rules out $x$($x$ is a superfluous $n$-letter sequence) since any $(k+1)$- letter that could be generated by $x$ can also be generated by $y$. Thus, from the set of $n$-letters, we need consider only those that are minimal under the usual vector ordering. Note that if $x$ and $y$ are minimal elements then $x_1 < y_1$ if and only if $x_2 > y_2$.*

**Theorem 5.3** *For fix integers $m$ and $n$ with $m > n \geqslant 2$. Suppose that there is a optimized time algorithm $(T)$ that achieves a $\dfrac{1}{n - \alpha}$ approximation of the LCS of two sequences of length at most*

*k from an alphabet size n. Then, there is also a* $O((n+T))\binom{m}{n}$ *time algorithm that achieves as* $\frac{1}{(m(1-\alpha/n))}$ *approximation of the LCS of two sequences of length atmost n from an alphabet of size m.*

*Proof.* We how to come across the LCS for two sequences of length at most k over an $m$-any alphabet, to the LCS for two sequences of length at most $k$ over an l-length sequence for any $n < m$.
The reduction runs in $O(k\binom{m}{n})$ time and process $\binom{m}{n}$ instances of 1 length LCS.
Let $P$ and $Q$ be two sequences of length at most $n$ over an alphabet of size $m$. Then define $L$ to be the longest common subsequence of $P$ and $Q$ (It is not known the identity of $L$). In order that, sort the letter symbol according to their number of occurrences in $L$.

Let $a$ be the collection of n most frequent letter symbols in $L$. Let $L_a$ be the subsequence of $L$ obtained by restricting $L$ to the letters of alphabet of that contains the symbols of $a$.
Since $a$ has the $n$ most frequent symbols in $L$, La contained least an $\frac{n}{m}$ fraction of the characters in $L$.
Now let us illustrate our approach. Given $P$ and $Q$, we consider all subsets of the alphabet consisting of precisely $n$ symbols. Note that one of these subsets will be $a$.

For each such collection $b$, consider the subintervals of LCS problem formed by restricting the symbols of $y$. Let OPT be the optimal LCS for this interval. From that

$$|Optimized\ sequence\ of\ a| \geqslant L_a \geqslant \binom{m}{n}|C|.$$

So, we consider $b = a$, if we can efficiently obtain a $\frac{1}{n-\alpha}$ approximation for optimized $a$, we will get a common subsequence of $P$ and $Q$ of length at least

$$\frac{|Optimized\ (a)|}{n-\alpha} \geqslant \frac{|L|}{m(1-\alpha/n)}.$$

Which yields the desired approximation for the LCS of $P$ and $Q$. The running time is multiplied by $\binom{m}{n}$, which is simply a constant provided m is bounded above by a constant. By setting $A = \frac{n\beta m}{1+\beta m}$, we obtain a linear time reduction from obtaining a $\frac{1}{\beta+m}$ approximation for $m$-sequence of letters to a $\frac{1}{n+(\beta m)/n}$ approximation for $n$-sequence of letters. □

**Lemma 5.4** *Fix an integer* $m \geqslant 3$ *and a constant* $\beta > 0$. *The problem of obtaining a* $\frac{1}{m+\beta}$ *approximation for the LCS of two sequences of letters from an alphabet of size m can be reduced in linear time to the problem of obtaining a* $\frac{1}{2+(\beta m)/2}$ *approximation for the LCS of two sequences.*

*Proof.* This follows from Theorem 5.2 by taking $n = 2$. Note that the reduction has a multiple over head of $O(m^2)$, which is constant when $m$ is constant. □

To develop an algorithm that finds all Longest Common Subsequences (LCS) between two sequences when there are two or more of the same length, we can use dynamic programming. This involves building a matrix to calculate the LCS length and then backtracking to identify all subsequences of the maximum length. Here's a step-by-step MATLAB implementation: MATLAB Code

```
function lcsList = findAllLCS(seq1, seq2)
% Find all Longest Common Subsequences (LCS) of seq1 and seq2
% Input:
%   seq1 - First sequence (string or array)
%   seq2 - Second sequence (string or array)
% Output:
%   lcsList - Cell array of all LCSs of maximum length

% Convert input to row vectors
seq1 = seq1(:)';
seq2 = seq2(:)';

len1 = length(seq1);
len2 = length(seq2);

% Step 1: Build the LCS length table
L = zeros(len1 + 1, len2 + 1);
for i = 1:len1
for j = 1:len2
if seq1(i) == seq2(j)
L(i+1, j+1) = L(i, j) + 1;
else
L(i+1, j+1) = max(L(i+1, j), L(i, j+1));
end
end
end

% Step 2: Backtrack to find all LCS
lcsList = backtrackAll(L, seq1, seq2, len1, len2);
end

function lcsList = backtrackAll(L, seq1, seq2, i, j)
% Backtracking to find all LCS of maximum length
% Input:
%   L - LCS length table
%   seq1, seq2 - Input sequences
%   i, j - Current indices in seq1 and seq2
% Output:
%   lcsList - Cell array of all LCSs

if i == 0 || j == 0
% Base case: return an empty string/array
lcsList = {''};
return;
end

if seq1(i) == seq2(j)
% If characters match, include them in the LCS
lcsList = backtrackAll(L, seq1, seq2, i-1, j-1);
for k = 1:length(lcsList)
```

```
lcsList{k} = [seq1(i), lcsList{k}];
end
else
% If characters don't match, explore all paths
lcsList = {};
if L(i, j-1) == L(i, j)
lcsList = [lcsList, backtrackAll(L, seq1, seq2, i, j-1)];
end
if L(i-1, j) == L(i, j)
lcsList = [lcsList, backtrackAll(L, seq1, seq2, i-1, j)];
end
% Remove duplicates
lcsList = unique(lcsList);
end
end
```

# 6 Results

1. LCS Length Table: A dynamic programming matrix L is built to store the lengths of LCS up to each index pair (i, j).
2. Backtracking: After building the matrix, all possible LCS paths are traced by recursively exploring equal maximum-length paths, ensuring all LCS of the same length are found.
3. Handling Duplicates: The unique function ensures the list of LCS sequences doesn't contain duplicates.

Example Usage
seq1 = 'ABCBDAB';
seq2 = 'BDCAB';
lcsList = findAllLCS(seq1, seq2);
disp('All Longest Common Subsequences:');
disp(lcsList);
Example Output
For the input sequences seq1 = 'ABCBDAB' and seq2 = 'BDCAB', the output might be:
All Longest Common Subsequences:
'BCAB'
'BDAB'
This implementation ensures all LCS of maximum length are identified. Here are several large examples of sequences to test:
Example 1: Strings with Repeated Patterns
Sequence X: ABABACABABACABABACABABAC
Sequence Y: BACABABACABABACABACABABA
Expected LCSs (length = 12):

1. ABABACABABAC
2. ABACABABACAB
3. BACABABACABA

Example 2: Sequences with Gaps
Sequence X: ACGTGACGTAGCTGACTGACG
Sequence Y: GTCGACGTGACGTACTGCGAC
Expected LCSs (length = 9):

1. ACGTGACGA
2. GCGTGACGA
3. GTCGACGTA

Example 3: DNA-like Sequences
Sequence X: AGCTTAGCTAGGCTAAGCTAAGTACGTAAGCTAG
Sequence Y: GCTAGGCTTAGCTAACGTAGCTAAGCTTAGCTAAG
Expected LCSs (length = 18):

1. GCTAGGCTTAGCTAAGCT
2. GCTTAGCTAAGCTAAGCT

Example 4: Random Alphanumeric
Sequence X: A1B2C3D4E5F6G7H8I9J0KLMNOP
Sequence Y: B1C2A3D4E5F6G7H8I9KLMNOPQ
Expected LCSs (length = 14):

1. B2C3D4E5F6G7H8I9J0KLM
2. A1B2C3D4E5F6G7H8I9KLM

Example 5: Binary Sequences
Sequence X: 1101011101010011010111101010 1101
Sequence Y: 1011011101101010110011101101010 11
Expected LCSs (length = 16):

1. 1011101011010110
2. 1101011011010111

Time and Space Optimizations:

- Time Optimization: After computing the LCS length, instead of finding one LCS using backtracking, we compute all LCSs by exploring all paths from the DP table. This might involve adding more steps, but it helps to find all LCSs in the given sequences.
- Space Optimization: Use a rolling array technique to reduce the space complexity from $O(m \times n)$ to , where we maintain only two rows of the DP table at any given time.

# 7 Conclusion

In this paper we have proposed a modified algorithm in order to find all LCS for sequence of letters. After examine this algorithm, we analyze that this algorithm can efficiently be used for improving optimize the time and space complexity. Sequence matching is a fundamental upcoming area in Computational Mathematics. The modified algorithm proposed in this paper addressed not only the problem of finding longest common sub sequence in two different sequences but also finds exactly

other LCS along with their locations and length. It is based on the Computational Mathematics, we utilized Matlab package to overcome this issue. If two or more matches share the same location then space is optimized. The matching process yields all possible matches between sequences $X$ and $Y$. The derived results, compared to the existing approaches [9] presented more efficient speedups and indicated that impressive improvements has been achieved. This proposed modified algorithm can be more developed to locate LCS among multiple of sequences. Also it can be revisited to perform the process of alignment between two sequences. We feel that there are still some scope to improve the time complexity and performance of our approach through different data structure such as space complexity and finding equal length with different letters.

## References


[1] Needleman, Saul B., and Christian D. Wunsch. " A general method applicable to search for similarities in the amino acid sequence of two proteins", *Journal of Molecular biology*, 48.3,pp.443-453, 1970.

[2] Hirsberg and Daniel S. "A linear space algorithm for computing maximal common subsequences", *Communications of the ACM*, 18.6, pp 341-343, 1975.

[3] Hunt, James W., and Thomas G Szymanski, " A fast algorithm for computing longest common subsequences", *Communications of the ACM*, 20.5, pp 350-353, 1977.

[4] Chen, Yixin, Andrew Wan and Wei Liu., " A fast parallel algorithm for finding the longest common subsequence of multiple bio sequences", *BMC bio informatics*,7.4, 1, 2006.

[5] Eswaran S and RajaGopalan S.P, " An Efficient fast pruned parallel algorithm for finding LCS in Bio sequences", *Anale Seria Informatics*, Vol. VIII fasc.1, 2010.

[6] Dharaief, Amine, Raik Issaoui and Abdelfettah Belghith, " Parallel computing the longest common subsequence(LCS) on GPUs: efficiency and language stability" *The first international conference on Advanced communications and computation (INFOCOMP)*, 2011.

[7] Corman,T.H, Leiserson,R.L, Rivest, Stein,C " Introduction to Algorithm using Dynamic programming",*Trans. on Math software* Third Edition,ch.15,sec15.4,pp 390-397, 1985.

[8] Bergroth,L. Hakonen,H and Raita,T ,"A survey of longest common subsequence algorithms", *IEEE computer society*, DC,USA, 2000.

[9] Izzat Alsmadi, Maryam Nuser,"String Matching Evaluation Methods for DNA Comparison," *International Journal of Advanced Science and Technology*, Vol. 47, October, 2012, pp. 13-32.

[10] Li Z, Goyal A, Kimm H," Parallel Longest Common Sequence Algorithm on Multicore Systems Using OpenACC, OpenMP and OpenMPI", *IEEE 11th international symposium on embedded multicore/many-core systems-on-chip (MCSoC)*, 2017, pp. 158–165.

[11] R. Beal, T. Afrin, A. Farheen, D. Adjeroh, "A new algorithm for the LCS problem" with application in compressing genome resequencing data,*BMC Genomics* 17 (4) (2016) 544.

[12] J. Storer, "Data Compression", *Methods and Theory", Computer Science Press*, MD, USA, 1988.

[13] D. Gusfield, "Algorithms on Strings, Trees, and Sequences",*Computer Science and Computational Biology*, Cambridge University Press, 1997.