International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue IV, April 2018- Available at www.ijraset.com

A Revised Algorithm to find Longest Common Subsequence

Deena Nath[1], Jitendra Kurmi[2], Deveki Nandan Shukla[3]
[1, 2, 3]Department of Computer Science, Babasaheb Bhimrao Ambedkar University Lucknow

Abstract: The aim of this paper is to make a revised algorithm in order to fetch the longest common subsequence from the given two strings 'X' and 'Y' for DNA matching in molecular biology etc. The revised algorithm prominence on the running time optimization and reduces the space and time complexity. The analysis of comparison between revised and current dynamic LCS has also performed. Moreover, the anticipated significance of this algorithm is to apply this algorithm in multiple sequences.
Keywords: Sequence matching, DNA sequence, Dynamic Programming, Longest Common Subsequence, Pattern Matching.

I. INTRODUCTION

String comparison is one of the basic activity performed in order to get useful information while studying the relationship between the various types of organisms and the genetic resemblance among them. In molecular biology we study the nucleotides sequences. These finite sequences represent organic molecules which forms the nucleic acid (DNA and RNA). Attempts are made to find matches between such organic molecules to retrieve genetic information so as to find how much they resemble and homologous to each other.

This process of information processing in the field of computer science is a vital part of DNA Matching which helps to gather information regarding genetic disorder, medication, genetic parenthood etc.

Wagner and Fischer proposed this concept in the year 1974. The problem of longest common subsequence is to fetch the longest common values of X and Y by deleting zero or more characters from any one or both the stings[4].

II. LITERATURE REVIEW

The research on literature of LCS algorithms renders a comprehensive range of opportunities to attune the system to enhance the overall performance.

A. The Levenshtein distance is a method to find the edit distance between two sequences by counting the number of insertions, deletions, and substitutions needed to makes valuable changes in the string. Wagner and Fischer in the year 1974 introduced an algorithm using the concept of a matrix in order to get the solution for this problem with dynamic programming. This algorithm just gives the length of LCS as a result but not the LCS.

B. Using divide-and-conquer strategy and dynamic approach altogether, Hirschberg in 1975 introduced a concept to find LCS. The problem is divided recursively into smaller and easier subproblems and solved individually and then combining their solution to solve the whole problem. Firstly, the matrix is traversed in the forward direction and then it is traversed in the reverse direction. The time and space complexity of this algorithm is O(mn) and O(m) respectively[2].

C. Dominant matches was another approach given by Hirschberg in 1977. The complexity of this algorithm is O (rn+nlogn) here r is the total number of ordered doublets at which the matching of two strings is performed.

D. J.W. Hunt and T.G. Szymanski's in the year 1977, stated that computing LCS from two strings is equivalent to finding the longest monotonically increasing path in the graph, where xi = yj. Hence introduced an algorithm on the same concept used in determining the longest increasing path.

E. In this case, we select consecutive symbols from X and traverse Y in reverse order in order to search for matches until all the elements of Y has been traversed. The solutions to these subprobelms are then merged in order to get the Longest Common Subsequence as a result. The process gets terminated as we get the optimal solution. The time complexity of this algorithm is O(n (m - r)). This method is used for fast execution to find the LCS when the given strings are of a large length and are suitable for similar texts.

F. Another algorithm was published by Apostolico, A. & Guerra in 1987, an alternative to support the framing of the LCS. The time complexity of this algorithm is O(mlogn+dlog (2mn/d)) where d is the smallest distance of the next nearest common elements. The closest occurrence of elements in Y is a representation of each symbol.

G. In the year 1990 Wu ET puts an effort to minimize the edit distance problem to reduced edit distance and apply it to find LCS. This reduced edit distance in spite of calculating the actual edit distance, it is directed to reduce the number of deletion. Hence, this algorithm is suitable for small strings. The time complexity of this algorithm is O(n (m-r)).

H. Rick in 1955 introduced an algorithm based on advancing from contour to contour which was compared with other existing algorithms and proved to be better among them. Contour is a region where the matching values are found & it was bounded by broken lines. The concept of dominant matches is used in this algorithm achieved by a class of dualization. The time complexity of this algorithm is O(min {(rm), r (n-r)}).

III.THE EXISTING APPROACH

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 of x1, x2, x3... xm 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, Cytosine, Guanine, and Thymine i.e. {A, C, G, T}[3].

Step 1: Identifying a longest common subsequence.

Theorem: The optimal substructure property of LCS problem.

Let, X= (x_1, x_2, x_3... x_m) and Y= (y_1, y_2, y_3…, y_n) be sequences. Let C= (c_1, c_2, c_3…, c_i) be any LCS of X & 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

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

Step 3: Calculating the length of LCS

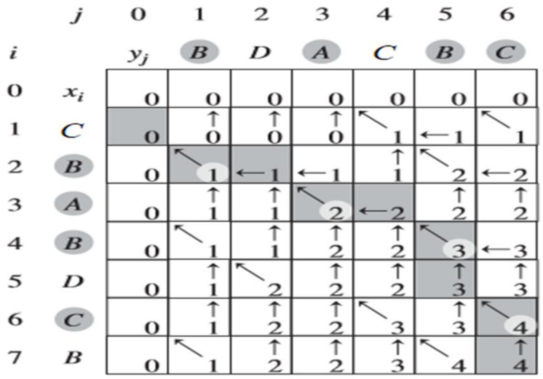
The length of LCS of strings X and Y is denoted by r (X, Y), or when the input strings are known, by r.

Length of LCS (r)= C[m-1,n-1].

Step 4: Backtracking to construct an LCS

In the two given strings X and Y, let's say C be a common subsequence between the two strings if the subsequence C exists in both the strings[5]. 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.

We are given two sequences X and Y and we wish to have a maximum-length common subsequence which can be solved using dynamic programming.



	j	0	1	2	3	4	5	6
i		y_j	B	D	A	C	B	C
0	x_i	0	0	0	0	0	0	0
1	C	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	A	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	C	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Fig. 1 LCS matrix of X and Y

With this procedure, we meet the elements of this LCS in reverse order by traversing the matrix using backtracking. In order to get an appropriate result, recursive method is to be used to print the LCS of X and Y.
In Figure. 1, 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.

IV. REVISED APPROACH

A. In this approach we have eliminated the prefixes (an empty subsequence) which are added to both the sequences or strings X and Y.

In the well-known dynamic LCS method, we used to add prefix to both the sequences.
For example, if X= {C, B, A, B, D, C, B} and Y = {B, D, A, C, B, C}.
We were supposed to add Prefix to both the Strings
i.e. X= {Φ, C, B, A, B, D, C, B} and Y = {Φ, B, D, A, C, B, C}.
But, in this revised Algorithm there is no need to add any prefixes to any of the Strings.

B. In this approach we will create an LCS matrix of (m, n) instead of (m+1, n+1) in order to reduce the iterations by eliminating the first row and first column which is used to initialize it by 0.

We add prefixes to both the sequences in order to get the initial values as zero (0) to start tracing both the sequence and find the common elements in both. Hence we have to take the matrix of (m+1, n+1) size.
But, in the revised Algorithm we don’t need to initialize the first row and column by 0 separately and hence we can remove the first row and column and reduce the size of the matrix.

C. In this approach we have used only one matrix ‘C’ instead of two. We don’t need an extra matrix ‘B’ for back tracking, which was just used to hold the directions to retrace the common subsequence which is the longest among all. Back tracking can easily be done with the help of one matrix i.e. C which hold the value of common subsequence matched.

D. In the LCS algorithm, for instance, if we drop the b table. Each c[i,j] entry in a cell of table c, rely on c[i-1,j -1], c[i-1,j], and c[i,j-1] entries of the c table. Given the value of c[i,j], we can obtain the value of c[i,j] in O(1) time using three values without examining table b. Thus, we don't need b table and we can redesign an algorithm for LCS in O(m+n) time.

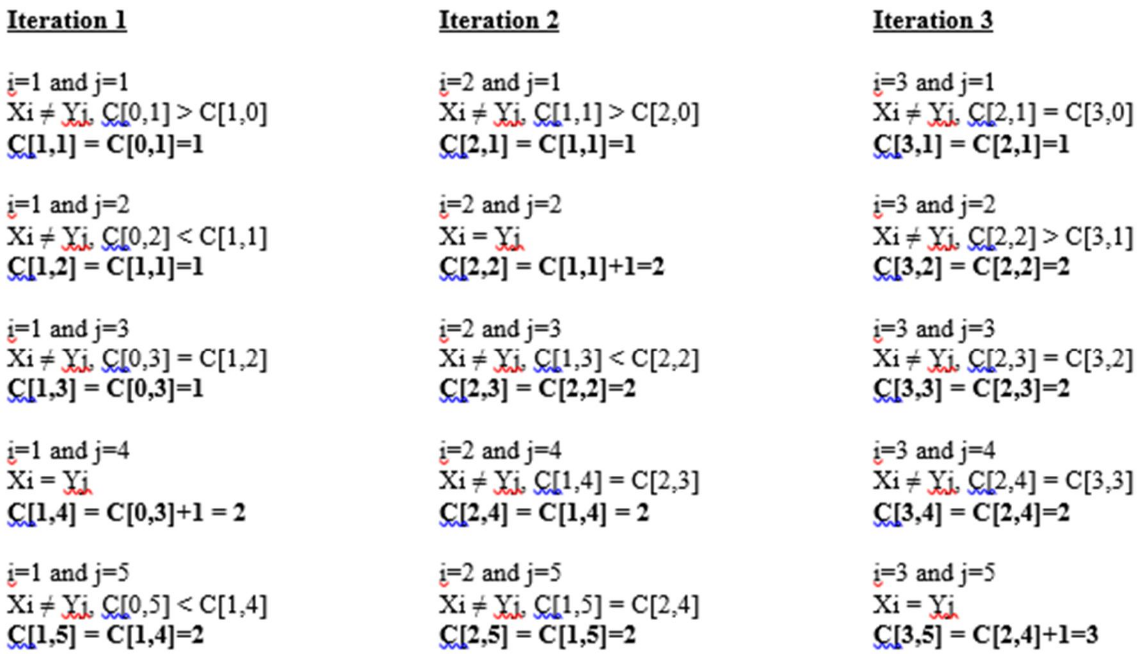
Although we save Θ(mn) space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need Φ(mn) space for the c table anyway.
Let’s take an example to go through the well-known dynamic approach.
Consider two strings X and Y, where X= {B, C, D, A, C, B, A} and Y = {A, B, D, B, C, A}.
if m and n are the size of String X and String Y, take a matrix ‘C’ of size (m x n).
here m=7 and n=6, hence matrix ‘C’ is of size (7,6).
i= 0 to m-1 => i= 0 to 6 and j= 0 to n-1 => j= 0 to 5

1) Step 1

If X[0] is equal to Y[j], then C[0][j]=1 else C[0][j]=0
C[0][0] = 0
C[0][1] = 1
C[0][2] = 0
C[0][3] = 1
C[0][4] = 0
C[0][5] = 0
Similarly if Y[0] is equal to X[i], then C[i][0]=1 else C[i][0]=0
C[1][0] = 0
C[2][0] = 0
C[3][0] = 1
C[4][0] = 0
C[5][0] = 0
C[6][0] = 1
Note: Here i will start from 1 since C[0][0] is already initialised

2) Step 2

if $X_i=Y_j$ then C[i,j] = C[i-1,j-1]+1
else if C[i-1,j]>= C[i,j-1] then C[i,j] = C[i-1,j]
else C[i,j] = C[i,j-1]



Iteration 1

i=1 and j=1
Xi ≠ Yi, C[0,1] > C[1,0]
C[1,1] = C[0,1]=1

i=1 and j=2
Xi ≠ Yi, C[0,2] < C[1,1]
C[1,2] = C[1,1]=1

i=1 and j=3
Xi ≠ Yi, C[0,3] = C[1,2]
C[1,3] = C[0,3]=1

i=1 and j=4
Xi = Yi
C[1,4] = C[0,3]+1 = 2

i=1 and j=5
Xi ≠ Yi, C[0,5] < C[1,4]
C[1,5] = C[1,4]=2

Iteration 2

i=2 and j=1
Xi ≠ Yi, C[1,1] > C[2,0]
C[2,1] = C[1,1]=1

i=2 and j=2
Xi = Yi
C[2,2] = C[1,1]+1=2

i=2 and j=3
Xi ≠ Yi, C[1,3] < C[2,2]
C[2,3] = C[2,2]=2

i=2 and j=4
Xi ≠ Yi, C[1,4] = C[2,3]
C[2,4] = C[1,4] = 2

i=2 and j=5
Xi ≠ Yi, C[1,5] = C[2,4]
C[2,5] = C[1,5]=2

Iteration 3

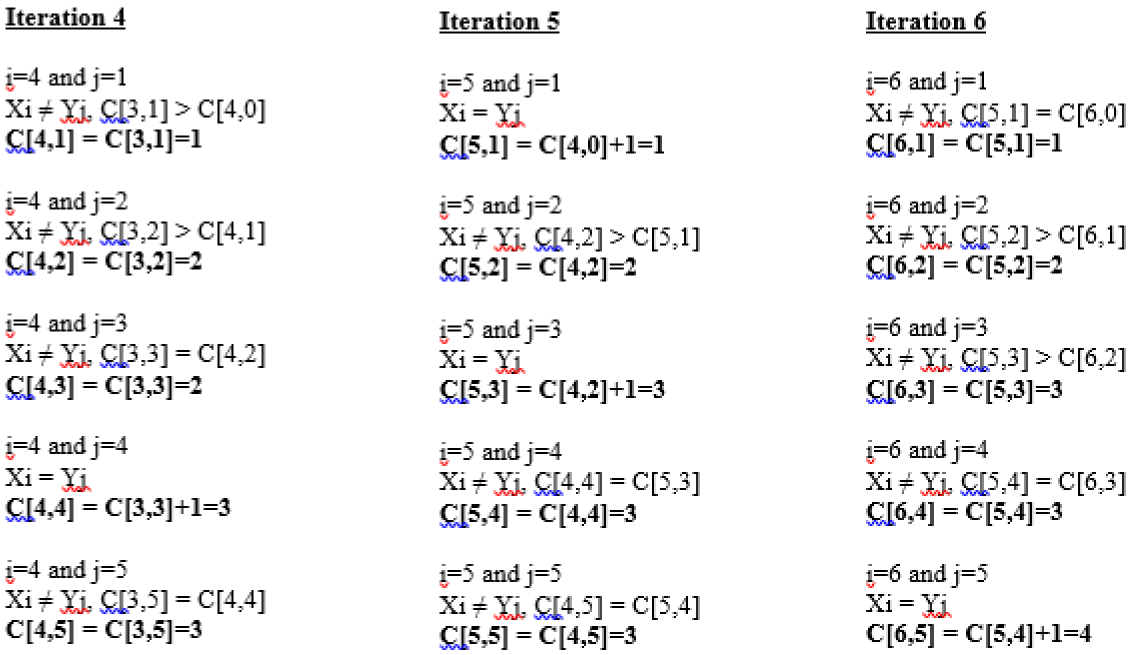
i=3 and j=1
Xi ≠ Yi, C[2,1] = C[3,0]
C[3,1] = C[2,1]=1

i=3 and j=2
Xi ≠ Yi, C[2,2] > C[3,1]
C[3,2] = C[2,2]=2

i=3 and j=3
Xi ≠ Yi, C[2,3] = C[3,2]
C[3,3] = C[2,3]=2

i=3 and j=4
Xi ≠ Yi, C[2,4] = C[3,3]
C[3,4] = C[2,4]=2

i=3 and j=5
Xi = Yi
C[3,5] = C[2,4]+1=3



Iteration 4

i=4 and j=1
Xi ≠ Yi, C[3,1] > C[4,0]
C[4,1] = C[3,1]=1

i=4 and j=2
Xi ≠ Yi, C[3,2] > C[4,1]
C[4,2] = C[3,2]=2

i=4 and j=3
Xi ≠ Yi, C[3,3] = C[4,2]
C[4,3] = C[3,3]=2

i=4 and j=4
Xi = Yi
C[4,4] = C[3,3]+1=3

i=4 and j=5
Xi ≠ Yi, C[3,5] = C[4,4]
C[4,5] = C[3,5]=3

Iteration 5

i=5 and j=1
Xi = Yi
C[5,1] = C[4,0]+1=1

i=5 and j=2
Xi ≠ Yi, C[4,2] > C[5,1]
C[5,2] = C[4,2]=2

i=5 and j=3
Xi = Yi
C[5,3] = C[4,2]+1=3

i=5 and j=4
Xi ≠ Yi, C[4,4] = C[5,3]
C[5,4] = C[4,4]=3

i=5 and j=5
Xi ≠ Yi, C[4,5] = C[5,4]
C[5,5] = C[4,5]=3

Iteration 6

i=6 and j=1
Xi ≠ Yi, C[5,1] = C[6,0]
C[6,1] = C[5,1]=1

i=6 and j=2
Xi ≠ Yi, C[5,2] > C[6,1]
C[6,2] = C[5,2]=2

i=6 and j=3
Xi ≠ Yi, C[5,3] > C[6,2]
C[6,3] = C[5,3]=3

i=6 and j=4
Xi ≠ Yi, C[5,4] = C[6,3]
C[6,4] = C[5,4]=3

i=6 and j=5
Xi = Yi
C[6,5] = C[5,4]+1=4

3) Step 3: Back Tracking

```
i=m-1
j=n-1
while(true)
{
        if((c[i][j]>c[i-1][j]) && (c[i][j]>c[i][j-1]))
        {
                PRINT Xi
                i--
                j--
        }
        else if(c[i-1][j]>=c[i][j-1])
                i--
        else
                j--
        if(c[i][j]==1 &&( (i==0) || (j==0) || (c[i-1][j]==0&&c[i][j-1]==0) ) )
        {
                PRINT Xi
                break
        }
}
```

DRY RUN:
i = m-1 = 7-1 = 6
j = n-1 = 6-1 = 5
Since C[6][5] is greater than C[5][5] and C[6][4]
PRINT X_6 = **A**
i = i-1 = 5
j = j-1 = 4
Since C[5][4] is equal to C[4][4]
i = i-1 = 4
j = 4
Since C[4][4] is greater than C[3][4] and C[4][3]
PRINT X_4 = **C**
i = i-1 = 3
j = j-1 = 3
Since C[3][3] is equal to C[2][3]
i = i-1 = 2
j = 3
Since C[2][3] is equal to C[2][2] and less than C[1][3]
i = 2
j = j-1 = 2
Since C[2][2] is greater than C[1][2] and C[2][1]
PRINT X_2 = **D**
i = i-1 = 1
j = j-1 = 1
Since C[1][1] is equal to C[0][1]
i = i-1 = 0
j = 1

NOTE: When C[i][j]=1, and either of i or j is 0(zero) we need to PRINT the element and the back tracking ends.

Moreover when C[i][j]=1, and C[i-1][j] and C[i][j-1] are both 0(zero), then it is the signal to terminate the loop after PRINTING the element, since we will not find any other common element further. So we don’t need to retrace the whole table and will apply break to terminate back tracking.

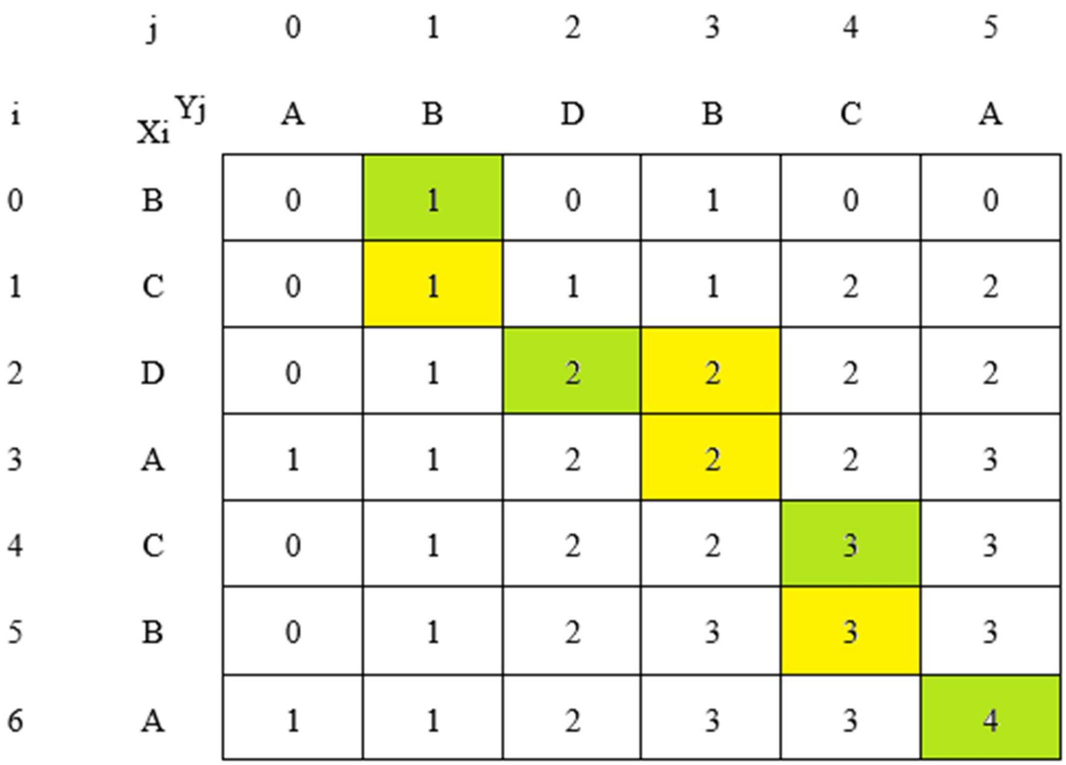
Since i=0
PRINT X_0 = **B**

Hence the final output (Longest Common Subsequence) of the given two strings is **BDCA**.

4) Length of LCS

The length of LCS of the two 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]
Here m=7 and n=6

Length of LCS = **C[m-1][n-1] = C[6][5] = 4**



j		0	1	2	3	4	5
i	Xi Yj	A	B	D	B	C	A
0	B	0	1	0	1	0	0
1	C	0	1	1	1	2	2
2	D	0	1	2	2	2	2
3	A	1	1	2	2	2	3
4	C	0	1	2	2	3	3
5	B	0	1	2	3	3	3
6	A	1	1	2	3	3	4

Fig 1: Matrix representation of the algorithm

While back tracking the matrix if we find both the upper and left value less than that of the current value we need to PRINT the respective Xi and move diagonally. If the upper value is equal to the current, we need to move upward otherwise move left.
In Figure 1, the cells with green color shows the result.

V. THE ALGORITHM

```
m = X.length
n = Y.length
let c[0..m-1, 0..n-1] be a new table
for i=0 to m-1
if(x[i]==y[0])
c[i,0]=1
else
c[i,0]=0
for j=1 to n-1
if(x[0]==y[j])
c[0,j]=1
else
c[0,j]=0
 for i=1 to m-1
for j=1 to n-1
```

```
if $x_i=y_j$
c[i,j]=c[i-1,j-1]+1
elseif c[i-1,j]>= c[i,j-1]
c[i,j]=c[i-1,j]
else c[i,j]=c[i,j-1]
return c
```

VI. RESULT AND ANALYSIS

The longest common subsequence (LCS) problem is an attempt of seeking the longest subsequence which is common to all sequence in a set of strings. Original sequences may not have continuous positions in subsequence to acquire. In information processing comparisons of data programs like molecular biology are involved in getting the idea of longest common subsequence problem. The Table 1. below demonstrates the Running time of the Revised algorithm and compares it with the well-known exiting algorithm.

Length of Sequences	Running Time in Well-known Algorithm (in Sec)	Running Time in Revised Algorithm (in Sec)
10	0.000204801	0.000139947
20	0.000280275	0.000192427
30	0.000301973	0.000225280
40	0.000390293	0.000284587
50	0.000417280	0.000371201
60	0.000458987	0.000403216
70	0.000642135	0.000431361
80	0.000707147	0.000573914

Table 1: Comparison with existing algorithm

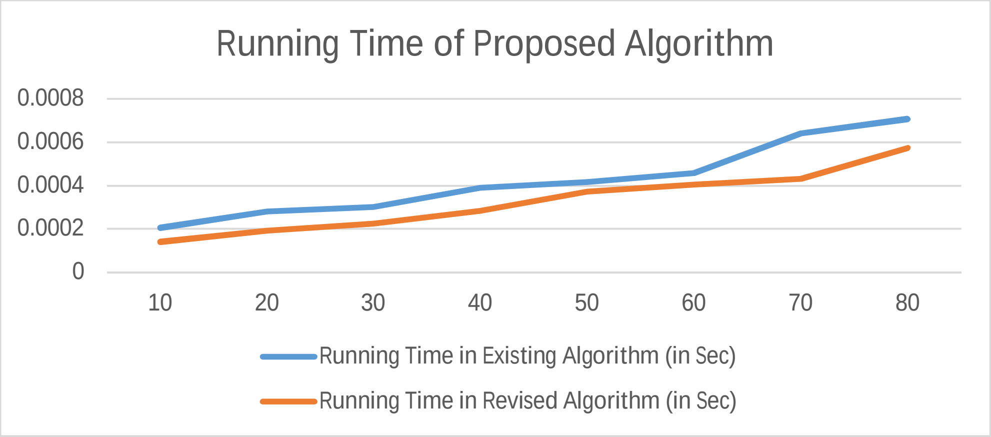


Fig 2: Graphical Representation

VII. ACCOMPLISHMENT AND FUTURE WORK

In this paper we have proposed a revised approach in order to find LCS for String Matching from protein, DNA/RNA sequences etc. After examining this algorithm, we analyse that this algorithm can efficiently be used for improving time and space complexity. Although I still feel that we have more span to improve the performance of the algorithm to find LCS using Dynamic Programming.

REFERENCES

[1] A. Apostolico, General pattern matchings, in: M.J. Atallah (Ed.), Handbook of Algorithms and Theory of Computation, CRC, Boca Raton, FL, 1998, Chapter 13.

[2] D.S. Hirschberg, Serial computations of Levenshtein distances, in: A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Oxford University Press, Oxford, 1997, pp. 123–141.

[3] T. H. Corman, C. E. Leiserson, R. L. Rivest, C. Stein, Cambridge, Introduction to Algorithm**,** Third Edition, “Dynamic Programming”, Ch. 15, sec. 15.4, pp.390-397 MA: MIT Press, 2010.

[4] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1974) 168–173.

[5] L. Bergroth, H. Hakonen, T. Raita, A Survey of Longest Common Subsequence Algorithms (IEEE Computer Society Washington, DC, USA ©2000)