1.Generic Longest common subsequence: -

-Pseudocode:

```
Input: Two strings X and Y of length m and n respectively.
Output: Length of the longest common subsequence of these two strings.
Algorithm: function LCS_Generic (X [1 to m], Y [1 to n])
    C = array (0 to m, 0 to n)
    for i: = 0 to m
       C[i,0] = 0
    for j: = 0 to n
       C [0, j] = 0
    for i: = 1 to m
        for j: = 1 to n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

- Theoretical complexity

When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming. We solve the subproblem using memoization using O (m n) extra space for a matrix to to keep the length/count of the subsequence.

Firstly, we try to make the zeroth row as well as the zeroth column as zero to help us solve the base case.

Then, we increase the count by one (from the previous diagonal matrix count) whenever we find a matching or common string literal in the iterations. If no matching or common literal found, then we just populate the corresponding matrix cell with the maximum value of the previous row column adjacent cell and the previous column row adjacent cell. By doing this until the end of the longer string we get our resultant output in the rightmost bottom cell of the matrix.

Since we iterate through each row and column element the time Complexity of the above implementation is O (m n) where ‘m’ and ‘n’ are the string lengths of strings ‘X’ and ‘Y’.

- Design choices (data structure)

Have used a two-dimensional integer type array of length C[m][n] for storing the length. Integer array is used because of its easy to access.

$$C[i,j] = \begin{cases} 0,\ if\ i = 0\ or\ j = 0 \\ C[i-1, j-1] + 1,\ if\ i, j > 0\ and\ X[i] = Y[j] \\ Max(C[i, j-1], C[i-1, j]),\ if\ i, j > 0\ and\ X[i]! = Y[j] \end{cases}$$

2. Alternative Algorithm for Longest Common Subsequence: -

- Pseudocode:

```
Input: Two strings X and Y of length m and n respectively.
Output: Length of the longest common subsequence of these two strings.
Algorithm: function LCS_Alternative (X [1 to m], Y [1 to n])
    C = array (0 to 1, 0 to n) //n is the length of the shorter string
    for k: = 0 to n
       C [1, k] = 0
    for i: = 1 to m
        for k: = 0 to n
              C [0, k] = C [1, k]
        for j: = 1 to n
            if X[i] = Y[j]
                C [1, j]: = C [0, j-1] + 1
            else
                C[i,j] := max(C[1,j-1], C[0,j])
    return C [1, n]
```

- Theoretical complexity:

When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming. We solve the subproblem using memoization using O (m+n) extra space for a matrix to keep the length/count of the subsequence.

Firstly, we try to make the zeroth row as zero to help us solve the base case.

Then, we increase the count by one (from the previous diagonal matrix count) whenever we find a matching or common string literal in the iterations. If no matching or common literal found, then we just populate the corresponding matrix cell with the maximum value of the zeroth-row same column adjacent cell and the previous column first-row adjacent cell. In the next iteration we try to override the zeroth-row elements by first-row. By doing this until the end of string we get our resultant output in the rightmost bottom cell of the matrix which is C[1][n].

Since we iterate through each row and column element the time Complexity of the above implementation is O (m n) where ‘m’ and ‘n’ are the string lengths of strings ‘X’ and ‘Y’.

- Design choices (data structure):

Have used a two-dimensional integer type array of length C[2][n] for storing the length. Integer array is used because of its easy to access. In every iteration since we just use the current row and the previous row elements to compute the current cell value we don’t require to store any other overhead rows. Hence, we try to improve the space complexity of the discussed generic algorithm to reduce the space to O(m+n) space. This is done by just keeping a 2*n length array where n is the length of the shorter string and we initialize the zeroth-row elements with zero value and start populating the first-row elements as in generic. In the next iteration we try copy the previously computed first-row element to zeroth-row and reuse the first row for the next string literals comparisons. i.e.

$$C[i,j] = \begin{cases} C[0,j-1]+1, & if\ i,j > 0\ and\ X[i] = Y[j] \\ Max(C[1,j-1], C[0,j]), & if\ i,j > 0\ and\ X[i]! = Y[j] \end{cases}$$

Experimental Results: -

- Measuring elapsed time:

To calculate the time elapsed I have used the timer library and functions given by timer.java as a support. I have captured the user start time just before calling the LCS function that returns the length of the longest common subsequence and have captured the user end time as soon as we hit a return from this select function. i.e. double startUserTimeNano = getUserTime() and double taskUserTimeNano = getUserTime( ) - startUserTimeNano; and (taskUserTimeNano/1000000000) gives us the number of seconds.

- Comparative results of 2 algorithms:

Theoretically the time complexity of both the algorithms discussed is the same and it differs by the space complexities only.

We can infer from the graphs of space vs size that the alternative(Hershberge)algorithm takes lesser or linear time when compared to generic LCS algorithm, the graph of run time vs size that the alternative algorithm suggested for LCS take almost same or better time than that of the generic LCS algorithm.

Table 1. Shows the average values observed experimentally for each type of input run 20 times.

Length of the String	Avg. run time (in sec) for LCS_Alternative algorithm	Avg. run time (in sec) for LCS_Generic algorithm	Space for LCS_Alternative algorithm in Kbytes	Space for LCS_Generic algorithm in Kbytes
1	1.06E-04	8.45E-05	27660	27688
10	1.18E-04	1.15E-04	27724	27720
100	0.001120972	0.001247791	28016	27976
1000	0.020721131	0.015735761	28796	32604
10000	0.618601915	0.74215423	28984	486276
100000	32.61871856	-	31064	1044220

The corresponding graph plot is as shown in the fig1.

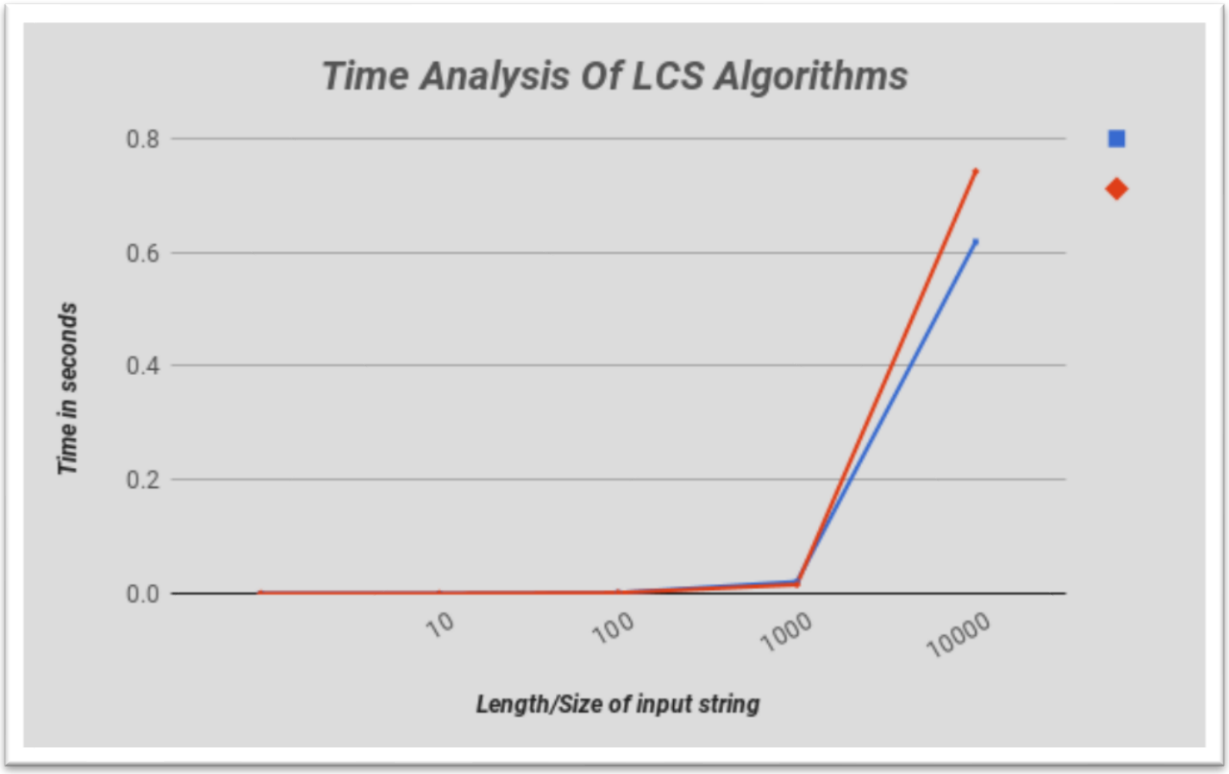

Fig 1. Plot for Run time analysis of the two LCS algorithms

(blue: LCS-alternative and red: LCS-generic)

We can observe the run time analysis of both the algorithms that goes close to each other for different string lengths. We can differentiate that the alternative space optimized algorithm runs slightly faster than the generic.

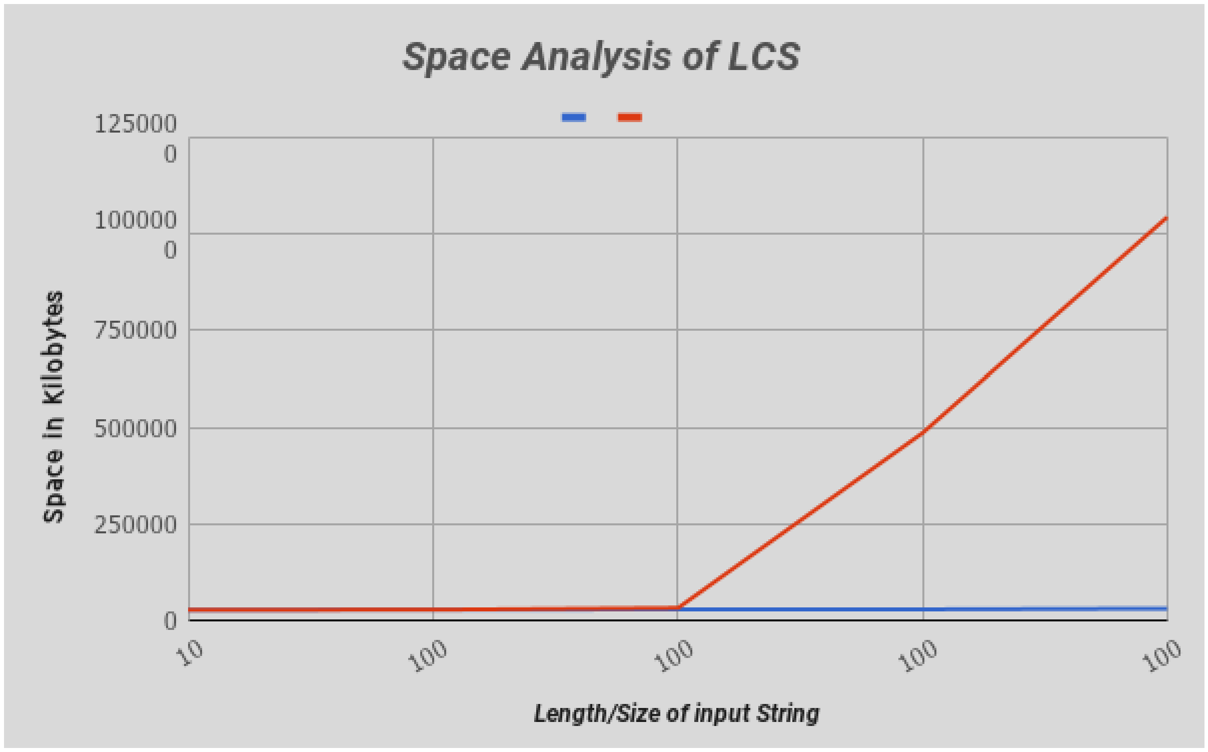

Fig 2. Plot for Runt time space analysis of the two LCS algorithms

(blue: LCS-alternative and red: LCS-generic)

From fig 2. We can infer from both the graph and the table along with the theoretical analysis of space consumed by the two algorithms. We see that the alternative algorithm takes a linear space O(m+n) whereas the plot to space for LCS algorithm takes O(n^2) space.