<u>COMPSCI 260P PROJECT #2 — LONGEST COMMON SUBSEQUENCES</u>

LONGEST COMMON SUBSEQUENCE (LCS) AND ITS COMPLEXITY

LCS is a common text processing problem used to test the similarity between two text strings. It has lots of applications in several fields, some of it include –

- ✓ Version control tool (diff)
- ✓ Genetics application
- ✓ Web crawler
- ✓ Changes in 2 versions of source code.

The time complexity of this algorithm using dynamic programming approach is **O(mn)** and space complexity is **O(mn**) for 2 string of lengths m and n.

Pseudocode

Algorithm LCS(X, Y):

Input: Strings X and Y with n and m elements, respectively

Output: For i = 0, . . . , n − 1, j = 0, . . . , m − 1, the length T[i, j] of a longest common subsequence of X[0..i] and Y [0..j]

```
for i ← 0 to n do
    for j ← 0 to m do
        if i == 0 or j == 0 then
            T[i,j] ← 0
        else if X[i] = Y [j] then
            T[i, j] ← T[i − 1, j − 1] + 1
        else
            T[i, j] ← max{T[i − 1, j] , T[i, j − 1]}
return array T
```

Design choices (data structure)

- The implementation for this algorithm is done in C++. The input string lengths - S1 and S2 is in the form of a character vector.
- It is filled with alphabet (capital letters) using in-built rand() and generate() method dynamically.
- A 2D character vector T is used in order to access the 2D matrix used for LCS. Each matrix cell is filled based on whether there is a character match or not at that instance.
- The last element of the matrix T[S1][S2] gives the resulting Longest Common Subsequence between the 2 strings.

Theoretical analysis of time complexity

- The running time for above LCS algorithm is easy to analyze, for it is dominated by two nested for-loops, with the outer one iterating n times and the inner one iterating m times.

- Since the if-statement and assignment inside the loop each requires O(1) primitive operations, this algorithm runs in **O(nm) time**, corresponding to the 2 for loops running n and m times.

Theoretical analysis of space complexity

- The space occupied by the above LCS algorithm is determined from the 2D matrix we used to visualize the matches in input strings.
- The matrix size is n*m where n is number of rows and m is number of columns and 2 for loops runs for n and m in the algorithm.
- Hence the space complexity for this algorithm is **O(nm).**

HIRSCHBERG'S LCS ALGORITHM AND ITS COMPLEXITY

In case where only the length of the LCS is required, the n*m matrix can be reduced to a 2*(min(m,n)) matrix with ease, as the dynamic programming approach explained above only needs the current and previous rows of the matrix. Hirschberg's LCS algorithm allows the construction of the optimal sequence itself in the same quadratic time **O(nm)** and linear space bounds **O(min(n,m))** where m and n are string lengths of 2 input strings.

Pseudocode

```
Algorithm LCS(X, Y ):
Input: Strings X and Y with n and m elements, respectively
Output: For XIndex = 0, 1; j = 0, . . . , m − 1, the length T[XIndex, j] of a longest
common subsequence of X[0..i] and Y [0..j]
bool XIndex
for i ← 0 to n do
    XIndex ← i&1;
    for j ← 0 to m do
        if i == 0 or j == 0 then
            T[XIndex,j] ← 0
        else if X[i] = Y [j] then
            T[XIndex, j] ← T[1-XIndex, j − 1] + 1
        else
            T[XIndex, j] ← max{T[1-XIndex, j] , T[XIndex, j − 1]}
return array T
```

Design choices (data structure)

- The implementation for this algorithm is done in C++. The input string lengths - S1 and S2 is in the form of a character vector.
- It vector is filled with alphabet (capital letters) using in-built rand() and generate() method dynamically.
- A 2D character vector T of size 2*(min(m,n)) is used in order to access the 2D matrix used for LCS. Each matrix cell is filled based on whether there is a character match or not at that instance.

- The last element of the matrix T[2][min(S1,S2)] gives the resulting Longest Common Subsequence between the 2 strings.

Theoretical analysis of time complexity

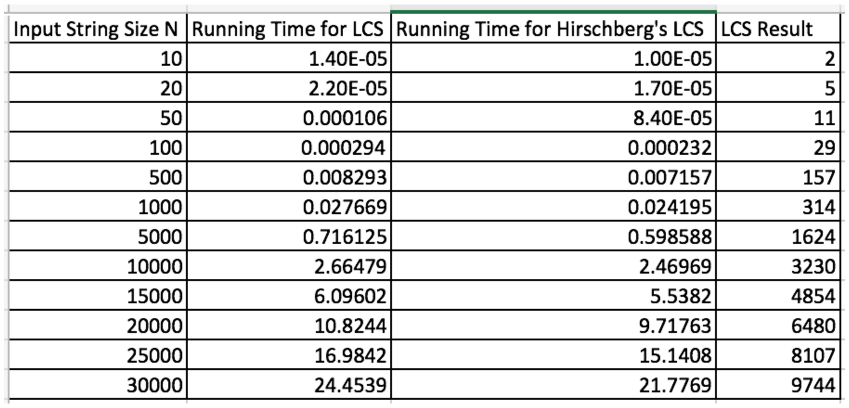
- The running time for above Hirschberg's LCS algorithm is same as original LCS algorithm, for it is dominated by two nested for-loops, with the outer one iterating n times and the inner one iterating m times.
- Since the if-statement and assignment inside the loop each requires O(1) primitive operations, this algorithm runs in **O(nm) time**, corresponding to the 2 for loops running n and m times.

Theoretical analysis of space complexity

- The space occupied by Hirschberg's LCS algorithm is determined from the 2D matrix we used to visualize the matches in input strings.
- The matrix size is 2*(min(m,n)) where n is number of rows and m is number of columns, it uses only current and previous rows for all (min(m,n)) columns, hence row size is 2 .
- Hence the space complexity for this algorithm is O(2*min(m,n)) i.e **O(m)** linear space where m≤n.

EXPERIMENTAL RESULTS

Experiments are performed on LCS and Hirschberg's LCS algorithms with different input string lengths N varying from 10 to 30,000 having 26 alphabets (capital letters) randomly.



Input String Size N	Running Time for LCS	Running Time for Hirschberg's LCS	LCS Result
10	1.40E-05	1.00E-05	2
20	2.20E-05	1.70E-05	5
50	0.000106	8.40E-05	11
100	0.000294	0.000232	29
500	0.008293	0.007157	157
1000	0.027669	0.024195	314
5000	0.716125	0.598588	1624
10000	2.66479	2.46969	3230
15000	6.09602	5.5382	4854
20000	10.8244	9.71763	6480
25000	16.9842	15.1408	8107
30000	24.4539	21.7769	9744

Measuring elapsed time in milli-seconds

- The elapsed time is measured by using suggested Timer.h in C++.
- The timer is started using t.start() before calling the LCS/ Hirschberg's LCS algorithm and elapsed time t.elapsedUserTime(a) is calculated soon after the algorithm execution is done.
- The double variable 'eUtime' has the running time of that algorithm.
- The running time for 2 algorithms are determined for different input sizes and plotted in the graph as shown below.

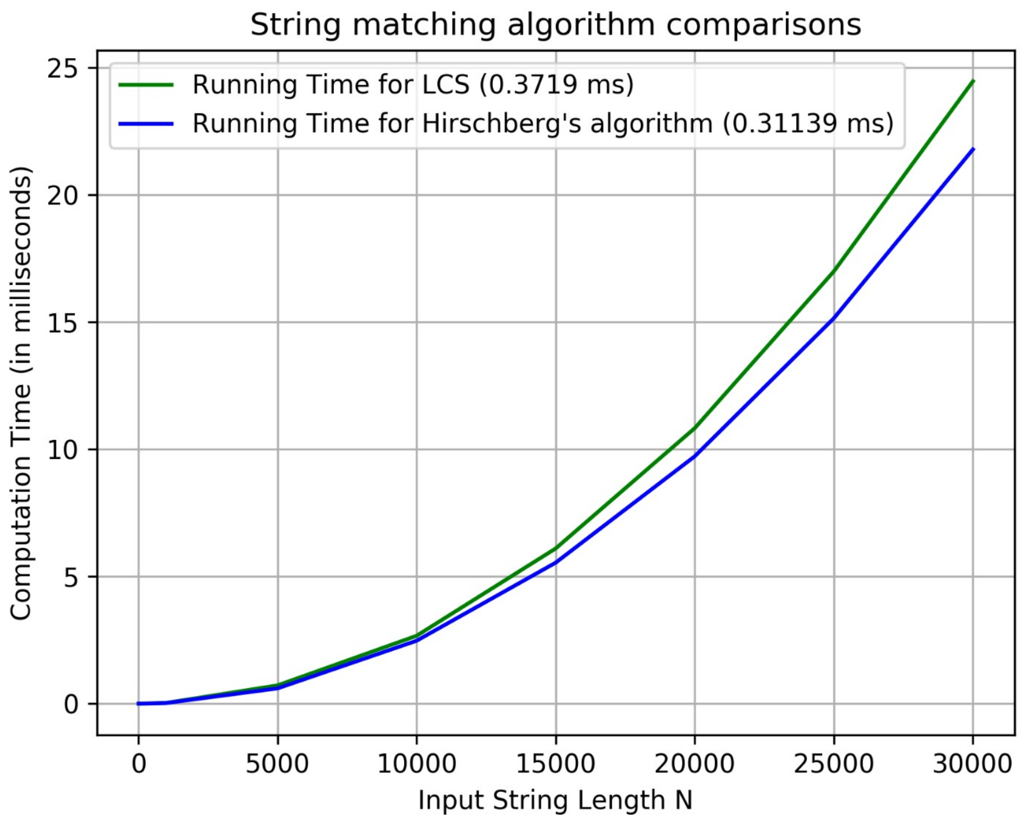
Measuring Memory Used in MB (Space Complexity Analysis)

- Experiment was performed to find the memory in MB for LCS and Hirschberg's LCS algorithm using in-built C++ library.
- The memory usage results for input strings of length 10,000 is as follows –

LCS -	Memory used: **117.849 MB**	Running Time: **2.66 ms**
Hirschberg's LCS -	Memory used: **3.302 MB**	Running Time: **2.46 ms**

- This proves that Hirschberg's LCS algorithm has reduced the space requirements considerably.
- This also corresponds well to the space complexities we theoretically determined, LCS - **O(mn)** and linear space Hirschberg's LCS – **O(m)** and the problem with input string lengths 10,000 becomes solvable in main memory itself as there are **no memory breaks** observed.

Comparative results of 2 algorithms



- When tested with different input values, it is seen that Hirschberg's LCS consumes lesser time to find the length of LCS as it runs in linear time bound and has lesser CPU overhead.
- Though the running time of both the algorithm are same (O(mn)), **Hirschberg's LCS algorithm runs faster.**
- This matches with our theoretical analysis that Hirschberg's LCS allows the construction of the optimal sequence itself in linear space bounds, which in turn reduces the memory usage/ CPU overhead and makes the algorithm perform better.

PSEUDOCODE SUMMARY & WHY HIRSCHBERG'S LCS ALGORITHM

- The original LCS algorithm works by creating a matrix of size m*n where m and n string lengths of 2 input strings.
- The first row and column is initialized with 0 and subsequent values are filled based on whether the characters match (T[i, j] ← T[i − 1, j − 1] + 1) or not (T[i, j] ← max{T[i − 1, j] , T[i, j − 1]}).
- If only the has to be found, in each iteration of outer loop we only, need **values from all columns of previous row**, so there is no need of storing all rows in our DP matrix, we can just store two rows at a time and use them.
- In that way used space will reduce from **T[m+1][n+1] to T[2][n+1],** this is Hirschberg's LCS algorithm.
- Hirschberg's LCS algorithm finds LCS in same quadratic time as original LCS but in linear space. Other implementation details for this is similar to original LCS algorithm.
- For a large string length of say 10,000, the memory overhead in LCS algorithm is **117MB**, which increases the CPU overhead and makes the running time slower – 2.66ms.
- For same input, using Hirschberg's LCS algorithm, the memory usage is **3.3MB**, way lesser than 117MB and thus there is lesser CPU overhead and the algorithm **runs faster** – 2.46ms.
- As the memory usage is less, **no memory faults** are observed in Hirschberg's LCS for string lengths more than 20,000.
- As a way to optimize space complexity (reduction from O(mn) to O(m)), Hirschberg’s LCS algorithm is preferred to find the length of LCS.