CS 260P | Algorithms | Project 2
[Longest Common Subsequences]

Contents:

1) **LCS Problem Statement:** Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ”acefg”... etc are subsequences of “abcdefg”. So,a string of length n has 2^n different possible subsequences.

Examples:
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

2) Dynamic Programming algorithm (using Memoization):

Pseudocode:

```
function LCSLength(X[1..m], Y[1..n])
   C = array(0..m, 0..n)
   for i := 0..m
      C[i,0] = 0
   for j := 0..n
      C[0,j] = 0
   for i := 1..m
      for j := 1..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]
```

Time and Space complexity:
Running time complexity for this algorithm is O(mn). One disadvantage of this dynamic programming method, is that it uses a lot of space: O(mn) for the array C.

3) ALTERNATIVE Algorithm: Linear Space approach (Prof Hirchberg's Algorithm):

Reference: http://www.ics.uci.edu/~goodrich/teach/cs260P/hw/lin-hirschberg.pdf

Pseudocode:

```
int lcs_length(char * A, char * B)
{
        allocate storage for one-dimensional arrays X and Y
        for (i = m; i >= 0; i--)
        {
            for (j = n; j >= 0; j--)
            {
                    if (A[i] == '\0' || B[j] == '\0') X[j] = 0;
                    else if (A[i] == B[j]) X[j] = 1 + Y[j+1];
                    else X[j] = max(Y[j], X[j+1]);
            }
            Y = X;
        }
        return X[0];
}
```

Time and Space complexity:

This is a recursive algorithm, with a time recurrence. We can think of this as sort of like quicksort -- we're breaking both strings into parts. But unlike quicksort it doesn't matter that the second string can be broken unequally.

$T(m,n) = O(mn) + T(m/2,k) + T(m/2,n-k)$
No matter what k is, the total size of these two subarrays is roughly mn/2. So instead we can write a simplified recurrence
$T(mn) = O(mn) + T(mn/2)$
which solves to **O(mn)** time total.

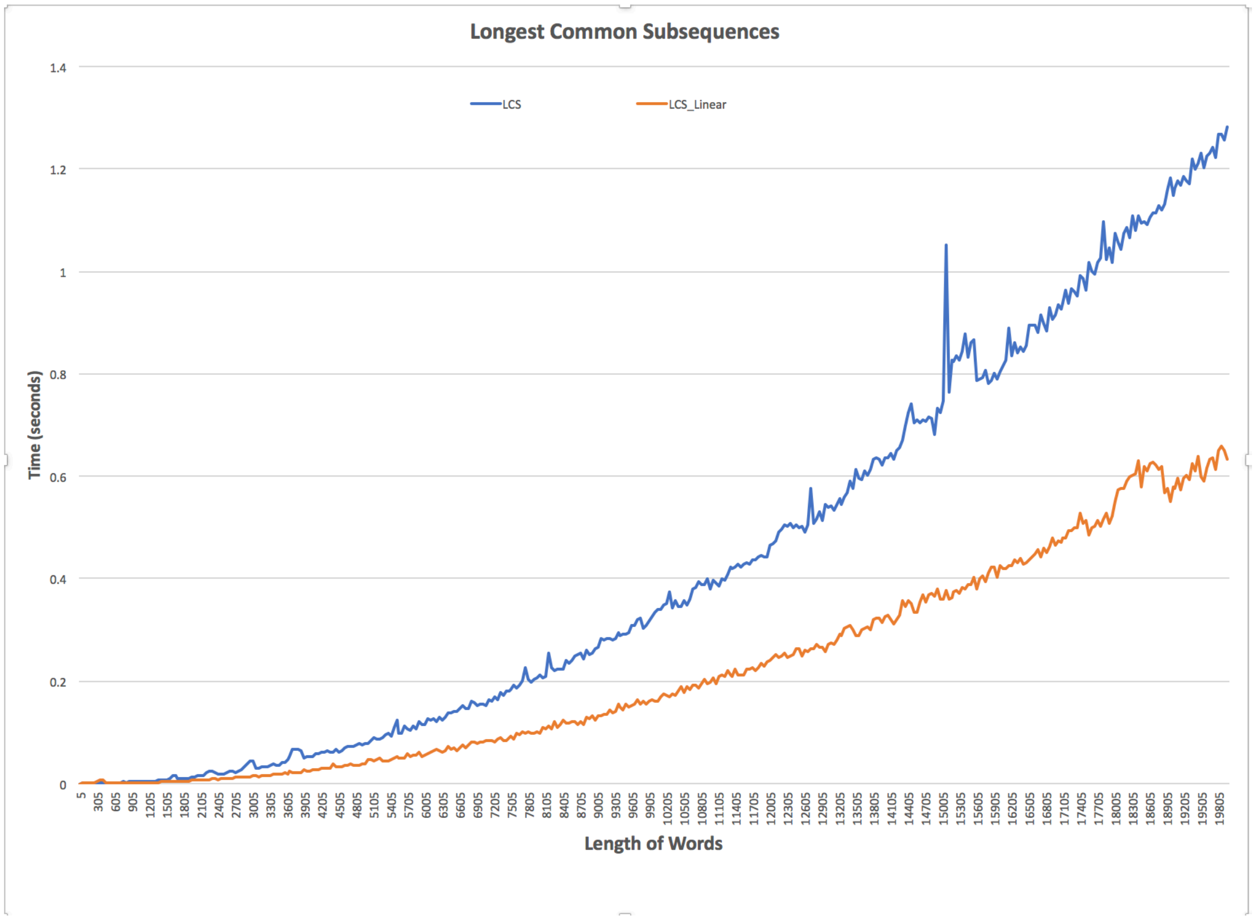
This algorithm takes roughly the same amount of **time** as before, **O(mn).** It uses a little more time to copy X into Y but this only increases the time by a constant (and can be avoided with some more care). The **space** complexity is either **O(m) or O(n),** whichever is smaller (switch the two strings if necessary so there are more rows than columns. Hence, space efficiency is much better than the previous regular Dynamic programming algorithm for LCS.

4) Experimental Analysis:

(LCS Dynamic programming vs LCS Linear Space-Dynamic Programming)

Algorithm	Memory used (MB)
LCS	155.789
LCS – Linear space	58.8828

(Memory calculated using GeeksforGeeks IDE)



Above graph shows length of words vs time(sec) taken by the two algorithms on JVM. The two algorithms have been run on words of lengths ranging from 5 to 20000,with an interval of 50.

Since the second (alternative) algorithm(LCS_Linear) takes lesser time, it can be inferred that, due to the smaller size of array taken the second case, lesser memory overhead occurs during initialization and access. Modern CPUs use a fast cache to reduce the average time taken to access main memory. Our linear space LCS code achieves maximum cache efficiency when it traverses monotonically increasing memory locations (2xN). This leads to lower running time of the second algorithm as compared to the first one.

5) Dataset:

Length of Words	LCS_Regular (Time in sec)	LCS_LinearApproach (Time in sec)
1005	0.004736	0.001471
2005	0.012392	0.007482
3005	0.044415	0.016799
4005	0.052495	0.024895
5005	0.079523	0.045984
6005	0.116442	0.056223
7005	0.154989	0.081552
8005	0.210921	0.099472
9005	0.264756	0.132977
10005	0.33324	0.160101
11005	0.396201	0.207326
12005	0.465423	0.240213
13005	0.538644	0.272126
14005	0.63645	0.326246
15005	0.746846	0.360056
16005	0.804835	0.426014
17005	0.935342	0.47398
18005	1.075276	0.550067
19005	1.148842	0.577949
19905	1.25711	0.649506