Longest Common Subsequence

Longest common subsequence (LCS) determination is an important problem in bioinformatics for establishing the relative similarity between pairs of strings. The advantage of this LCS's other comparison methods and its relevance to bioinformatics is that LCS solutions can provide a qualitative measure between biological sequences (DNA, RNA, protein). Often biological regions with close LCS sequence parody maintain relatively recent evolutionary ancestry, either from natural evolutionary drift, viral insertion, or random duplication events, LCS solutions, alignment penalties withstanding, can help demonstrate a common ancestry and streamline research within the field.

Subsequences, distinct from substrings, do not have to occupy consecutive positions to be counted as such and are concatenated irrespective of distance. Consider the following string *S*:

$$S = "command"$$

The following may be considered all valid subsets of *S*:

$$\{c, o, a, co, cm, ca, cn, cd, om, oa, on, od, mm, cnd, comm, coand, mmnd, comad, ect.\}$$

Given a second string *T:*

$$T = "combed"$$

With its own set of subsets:

{c,o,m,b,e,d,co,cm,cb,ce,cd,com,cob,coe,cod,cobed,cmbd, ect.}

The maximal number of subsequences for any given string is 2^n where n is the length of the string. However, the number of distinct possible sequences will remain less than that value because some subsequences, although derived from alternative locations within the string, may produce an identical outcome. For example the subsequences *mad* and *mad* taken as subsequences from string *S* are both valid and different subsequences since the origin of the "m" character can be obtained from its 3rd or 4th position. Furthermore, although they produce the same subsequence they are not distinct from one another.

However even with distinctness withstanding, comparisons between all possible subsequences to find a common LCS for two strings is a computationally difficult issue.

The Naïve Algorithm

The first and most obvious solution to finding the LCS between two strings would most likely be to compare all possible LCS sequences between two strings, compare the results in decreasing length order, and keep the longest match between the two. This method would take our example strings (*S* and T) and generate the following list:

String S Sub_{max}: command

String S Sub_{max-1}: ommand, cmmand, comand, comand, commnd, commad, comman

String *S* Sub_{max-2}: mmand, omand, omand, ommnd, ommad, omman, cmand, cmand …

ect…

This method enumerates across all possible substrings (Sub_{max-n}) for any given string and would generate the maximal number of possible number of substrings, 2^n, where n is the length of a given string. The enumerations between string *S* and *T* would then be compared starting from the longest subsequences found.

The problem with this method is that of operational time. Since we need to enumerate all possible subsequences associated with both strings, each containing 2^n possible answers, and with further comparisons associated with finding the correct answer the operational time becomes an NP-hard problem, having a run time of $O(2^n)$. This algorithm's runtime, given a sufficiently large enough string pair, would very quickly outpace any practical use case.

A Recursive Approach

Since the LCS problem exhibits an optimal substructure we can reformulate our algorithm to incorporate a recursive solution. However, as we shall see it won't improve on our brute force method. The recursive approach can be decomposed to three questions and will be notated using our previous examples:

1. Is the length *S* or *T* equal to 0?
    - This functions as the base case
2. Are the characters at the next to end of *S* and *T* identical?
    - Continue recursive call to next to last character with both strings
3. If not identical:
    - Continue recursive call with next to last character of the longer of the two

That's basically it. Its implementation in java would look as follows:

```
public static String recursive_LCS(String a, String b){
    int a_len = a.length();
    int b_len = b.length();

    if (a_len == 0 || b_len == 0){                          //Question 1: Base Case
       return "";
    } else if (a.charAt(a_len - 1) == b.charAt(b_len - 1))  //Question 2
    {
       return recursive_LCS(a.substring(0, a_len - 1), b.substring(0,
            b_len - 1))
            + a.charAt(a_len - 1);
    } else                                                  //Question 3
    {
```

```
            String x = recursive_LCS(a, b.substring(0, b_len - 1));
            String y = recursive_LCS(a.substring(0, a_len - 1), b);
            if (x.length() > y.length())
            {
               return x;
            } else
            {
               return y;
            }
         }
      }
```

If we examine the runtime of recursive_LCS we'll find its recurrence relations for the questions above are the following:

- Question 1: *O(1)*
- Question 2: $T(S,T) = T(S-1,T-1) + O(1)$
- Question 3: $T(S,T) = T(S,T-1) + T(S-1,T) + O(1)$

If we dig a little deeper and consider what's actually happening across each string we'll find that this algorithm will have a runtime similar to that of the brute force method previously discussed. Since the base case (Question 1) is dependent on one of the strings decrementing to a final length of 0 and with each non matching character resulting in a decremented string of either *S* or *T* depending on length (Question 3), we have a runtime which effectively enumerates all possible states of the two strings and compares them within one method. Furthermore, since the length of the brute force method is linked to the length of the string and this algorithm decrements until both strings are nearly complete of possible comparisons the runtime is: $2^{\max(S,T)}$. Where max(S,T) is the length of the two strings which is larger. This reduces to an operational time of $O(2^n)$ which is the same as the brute force method.

$$\begin{aligned} \mathrm{T(S,T)} &= O\left(2^{\max(S,T)}\right) \\ &= O(2^n) \end{aligned}$$

Dynamic Programming

To get around the NP-hard nature of the brute force algorithm it becomes necessary to turn to a dynamic programming implementation of our problem. Dynamic programming is a technique, which can be used to reduce a problems complexity load via quantifying a problem into specific subproblems. Since we have already established an optimal substructure in the recursive algorithm and given that many of our recursive calls redundantly compute similar subproblems we can establish that the recursive LCS algorithm does have overlapping subproblems, we can approach the LCS using dynamic programming.

The easiest way to demonstrate the comparisons involved with an optimal LCS solution would be to eliminate overlapping subproblems via tabular memoization within a two dimensional array. Memoization would function to lower the runtime cost of an algorithm by eliminating redundant calculations associated with overlapping subproblems. This would lower the cost of our recursive

enumeration calculations from $O(2^n)$ to $O(S*T)$ where S and T are the length of the strings in question producing an algorithm of $O(n)$ complexity. The tabularized strings S and T would look like the following:

Scoring Matrix

String S

String T		**c**	**o**	**m**	**m**	**a**	**n**	**d**
	c	1	1	1	1	1	1	1
	o	1	2	2	2	2	2	2
	m	1	2	3	3	3	3	3
	b	1	2	3	3	3	3	3
	e	1	2	3	3	3	3	3
	d	1	2	3	3	3	3	4

In the example above the data contained within the matrix represents comparisons between two letters of strings S and T. The array is generated by looping through each possible letter comparison combination starting from the first row forward and runs through the same optimal substructure and memoizes their answers to the matrix above. Following from the top left over each row the algorithm searches for matches along each character of each string and does the following:

1. In the event of a match
    a. Increments the value up and to the left of it within the array
2. In the event of a mismatch
    a. Only copies the value up and to the left of it within the array

This results in a value obtained in the bottom right of the matrix which is the number of subsequences shared in common between the two strings. However, in addition to the length of the longest common subsequence we require the sequence and that requires a method of backtracking along the matrix.

To design a backtracking algorithm it became necessary to identify those positions within the table where the actual matches are and help direct that back tracing path to provide a correct one. The answer was to generate another matrix in parallel, which the scoring matrix, which used a specific code, set to direct the back tracking algorithm based off the scoring matrix. A matrix entry of 1 would result in a back tracking left along the matrix, 2 would direct up, and 3 would be diagonal. The numbers 1 and 2 would function as directors towards 3's within the table with positions containing 3's presumed to be matches along the two strings. The matrix above, converted following this convention would look as follows:

Backtracking Matrix

String S

String T		c	o	m	m	a	n	d
	c	**3**	1	1	1	1	1	1
	o	2	**3**	1	1	1	1	1
	m	2	2	**3**	1	1	1	1
	b	2	2	2	2	2	2	2
	e	2	2	2	2	2	2	2
	d	2	2	2	2	2	2	**3**

The table above shows the first match occurring at d, traverses diagonally following a trace up along the 2's, left at the 1's and diagonally again producing a series of matches producing the following string: "dmoc", which reversed is "comd", the correct solution to our LCS problem.

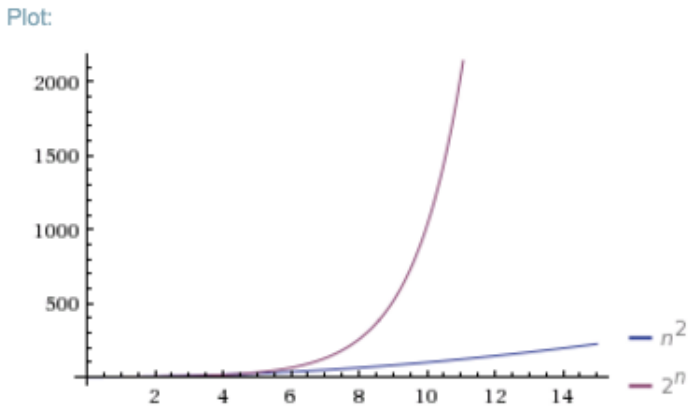
Implemented in java the code looks as the following:

```
public static String LCSDynamicProgramming(String seq1, String seq2)
  {
    //tracing variables
    int mismatch = 0;
    int traceLeft = 1;
    int traceRight = 2;
    int match = 3;

    //string lengths
    int seq1_len = seq1.length();
    int seq2_len = seq2.length();
    int x, y;

    //dynamic programming matrices
    int[][] score = new int[seq1_len + 1][seq2_len + 1];
    int[][] backTrack = new int[seq1_len + 1][seq2_len + 1];
    StringBuilder subsequence = new StringBuilder();

    // Initializing the scoring and backTracking arrays
    for (x = 0; x <= seq1_len; ++x){
       score[x][0] = 0;
       backTrack[x][0] = traceLeft;
    }
    for (y = 0; y <= seq2_len; ++y){
       score[0][y] = 0;
       backTrack[0][y] = traceRight;
    }
```

```
// This is a dynamic programming doubly nested loop which computes both
// the scoring and backtracking matrices.
for (x = 1; x <= seq1_len; ++x){
    for (y = 1; y <= seq2_len; ++y) {
        //Replaced recursive calls with memoized double nested for loop
        if (seq1.charAt(x - 1) == seq2.charAt(y - 1)){
            score[x][y] = score[x - 1][y - 1] + 1;
            backTrack[x][y] = match;
        } else{
            score[x][y] = score[x - 1][y - 1] + 0;
            backTrack[x][y] = mismatch;
        }
        if (score[x - 1][y] >= score[x][y]){    //If above is >=
            score[x][y] = score[x - 1][y];     //make = to above
            backTrack[x][y] = traceLeft;
        }
        if (score[x][y - 1] >= score[x][y]){    //If left is >=
            score[x][y] = score[x][y - 1];     //make = to left
            backTrack[x][y] = traceRight;
        }
    }
}

// The backTracking while loop
x = seq1_len;
y = seq2_len;
while (x > 0 || y > 0){
   if (backTrack[x][y] == match){
      x--;
      y--;
      subsequence.append(seq1.charAt(x));
   } else if (backTrack[x][y] == traceLeft) //gapUp = 2 up on graph
   {
      x--;
   } else if (backTrack[x][y] == traceRight) //gapLeft = 1 go left
   {
      y--;
   }
}
```

Dynamic Programming Runtime

The bluk of the runtime associated with this algorithm lies in the assignment of the two matrices (the scoring and the traceback); see highlighted section above. Since each of their for loops are doubly nested running the length of each respective string and generate two matrices their operational runtime is *O(2n*2m)* which reduces to a total overall runtime of *O(nm)*. This is significantly better than the NP-hard recursive LCS solution as within bioinformatics there are frequently queries, which require comparisons between strings numbering in the hundreds of bases. Consider the graph below comparing the predicted runtimes between two strings of equal length (an *O(n*n) and O(2^n)*):



The recursive, brute force, algorithm approach becomes immediately impractical at a string length of 10 which is barely good enough to design primers let along search entire genes for homology.