Julie Garcia
Sequence Alignment Analysis
Programming Lab #3
ALG605 420 FA16

Introduction
This lab was used to demonstrate the efficiency of a dynamic programming algorithm for the problem of finding the Longest Common Subsequence (LCS) between two strings. An input file was read in with a list of strings and each pair of strings was compared to find the longest common subsequence. I created two solutions to the problem, one that uses the dynamic programming algorithm, and one that uses a naïve algorithm, to demonstrate the vast difference in running times between the two.

What I did as an enhancement
As an enhancement to the project, I wrote a naïve algorithm that uses recursion to work through every possible combination of the two strings, comparing them to find the longest common subsequence. I wrote the naïve algorithm to only find the length of the LCS, and not the actual string. I did this because it is obvious that the running time grows exponentially when only finding the length, and didn't see any need to construct the string, because my point was demonstrated.

In the output, I compared the times of the naïve algorithm with the dynamic programming algorithm and found that the dynamic algorithm, did run a lot faster. In fact, I was only able to run the naïve algorithm on very short strings (15 base pairs or less) because the running time quickly became too long as the strings grew. Below you will find an in-depth discussion of efficiency of each algorithm, with respect to time and space. As you will see it was obvious what the better choice was.

I also downloaded real DNA sequence data.[4] I downloaded two different strains of Hepatitis D DNA and compared them. I also downloaded parts of two different human DNA sequences. The entire human DNA sequence was too long to load into one text file so I only downloaded about 800 base pairs worth of data for each.

Description of Data Structures & Justification
I used an ArrayList for storing the array of strings that are read in from the input file and compared. I considered using an array, but wanted something that I could dynamically add strings to, since I do not know the size of the list of strings until the file read is finished. Array indexing and ArrayList.get() both use constant time, but ArrayList can be slightly slower. I considered this a small tradeoff, since I would only be using the get() function minimally and not inside the Longest Common Subsequence algorithm.

The strings are passed to the class LongestCommonSubsequence in the constructor. The LongestCommonSubsequence class initializes and stores the strings as two character arrays. I convert the strings to character arrays inside of this class so that I could index each character in the strings easily and quickly. This class implements the two different LCS algorithms, described in more detail below.

For the dynamic programming algorithm, I constructed a 2-Dimensional array to count the Longest Common Subsequence. This is constructed using a bottom up approach as described below. In this case I wanted to keep it simple, with little overhead. Since I knew the size of the 2-D array would be m + 1 x n + 1 (m being the length of the first string and n being the length of the second string), I could allocate the appropriate amount of space ahead of time.

Description and justification of design decisions
I created the class LongestCommonSubsequence to store the two strings and to implement the two algorithms. I did this to encapsulate the algorithm code and keep it separate so that I could reuse it later. The class solves the problem using both dynamic programming and a naïve algorithm. The naïve algorithm is used for comparison only, to demonstrate the speed gains when using dynamic programming.

The naïve algorithm uses a recursive top-down approach, comparing the last character in each string with each call to the function. The recursive function takes four arguments: the two strings and m and n, which represent the current length of the prefix being examined. This algorithm uses comparison of the ***ith*** prefix of each string. If X, with length m, is denoted as the characters $x_1x_2x_3\ldots x_m$, then the ***ith*** prefix is $x_1x_2x_3\ldots x_i$, where $i < m$. With each successive call to the recursive function characters are dropped from the end of the strings. If there is a match in the last character of each string, both characters are dropped and 1 is added to the next recursive call. If there is not a match, then the max is found between a call to the recursive function dropping one character from the end of the first string, and the other call dropping a character from the end of the second string. The recursion tree below (see efficiency discussion) shows an example of this, and demonstrates that this problem has overlapping subproblems. With each call to the recursive function we solve one or two subproblems, depending on if we find a match or not. This makes this algorithm very inefficient because we keep calling the recursive function for the same set of substrings. The dynamic programming algorithm solves this problem.

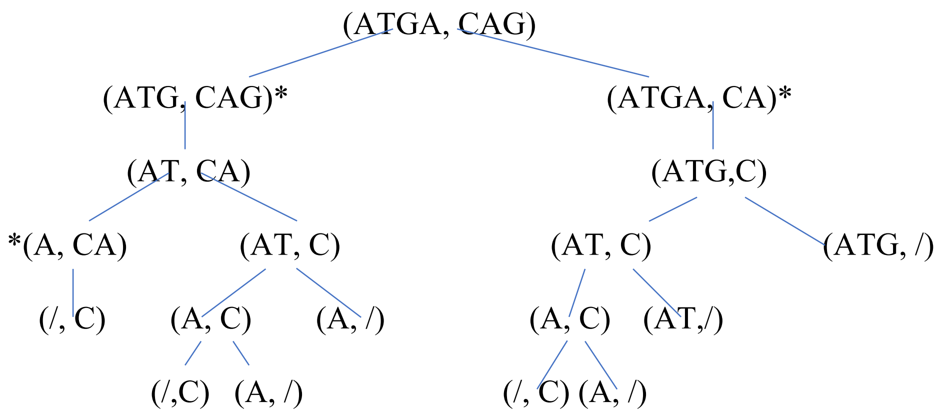
The second algorithm uses dynamic programming in a bottom-up approach. This problem lends itself well to dynamic programming because it exhibits optimal substructure. We can break the problem down into subproblems and solve them independently, and then recombine them to get the final solution. As with any dynamic programming algorithm, we eliminate all redundant comparisons. Each subproblem is solved only once. I chose the bottom-up approach because we know the size of each problem and its subproblems. I thought the bottom-up approach would be slightly faster because it lacks the overhead of recursive calls when using a top-down approach with memoization.

The first part of the dynamic programming algorithm constructs a table that counts the number of common characters. We start at the beginning of the string, comparing each character in each string only once and saving the results to a matrix. Once we get to the bigger problems, the smaller subproblems have already been completed and saved in the table. Consider the two strings X and Y, of length m and n, respectively. And the table LCS[m,n]. When we are finished LCS[m, n] will contain the length of the LCS. We iterate through the characters of each string started at (i, j) = (0,0) and filling in LCS with each value of i <=n, and j <=m. We determine the LCS[i, j] by comparing the characters at each spot. If X[i] and Y[j] are equal, then we set LCS[i, j] = LCS[i-1, j-1] + 1. If they are not equal, we take the max between LCS[i, j-1] and LCS[i-1,j] and put it into LCS[imp]. In doing this, we are counting the LCS and creating a path through the matrix that can be retraced to find the actual string in the second step.

In the second part of the dynamic programming algorithm, we use the LCS table constructed in step one. We start at LCS[m,n] and trace backwards from the end of the LCS string to the beginning. We construct the string by following these guidelines. If the two characters match at the current position, the character is added to the LCS string starting with the last position and working backwards. If the characters do not match, we move in the direction of the greater number in the table. If the numbers are the same, we can move in either direction, but always the same direction. We may come up with a different LCS string depending on which direction we use. Some strings have more than one LCS of the same length.

The FileIO class was reused from the second programming project and was designed to encapsulate the reading from and writing to the input and output files.

Efficiency with respect to both time and space
In analyzing the naïve algorithm, we can see that the running time will grow quickly as the strings to compare grow larger. Below is a recursion tree showing each call to the function. The stars(*) show where a match was found. If you trace the tree down, you can see that there are redundancies in comparisons in the tree. This is only for two very small strings, so you can imagine the amount of repeated subproblems there are for very long strings.



The naïve algorithm in the worst case (where there are no matches) makes two calls to the recursive function with each new level, so it grows in exponential time or $O(2^n)$, when $n > m$. Each call makes only simple comparisons, so they run in O(1) or constant time. This algorithm does not require any additional space.

There are two parts to the dynamic programming algorithm, the construction of the table that compares the strings characters and increments accordingly, and retracing the path back through the table to construct the LCS. The first part contains two loops, one through m, the length of the first string, and the other through n, the length of the second string. The two loops run O(mn) times. Inside the loops simple comparisons are made, in addition to a Math.max() function. These all run in constant time O(1) This first part of the algorithm costs O(mn) in running time. The second part that constructs the Longest Common Subsequence, consists of a while loop that runs until we reach the beginning of the table. This runs at most $O(m + n)$ times. <elaborate on this> Inside the while loop simple comparisons and arithmetic is done, and runs in constant, O(1), time. The entire algorithm runs in $O(mn) + O(m + n)$ time, simplified it runs in O(mn) in the worst case.

The dynamic programming algorithm requires and extra $O((m+1)(n+1))$ of space to construct the table that is need to compute the Longest Common Subsequence.

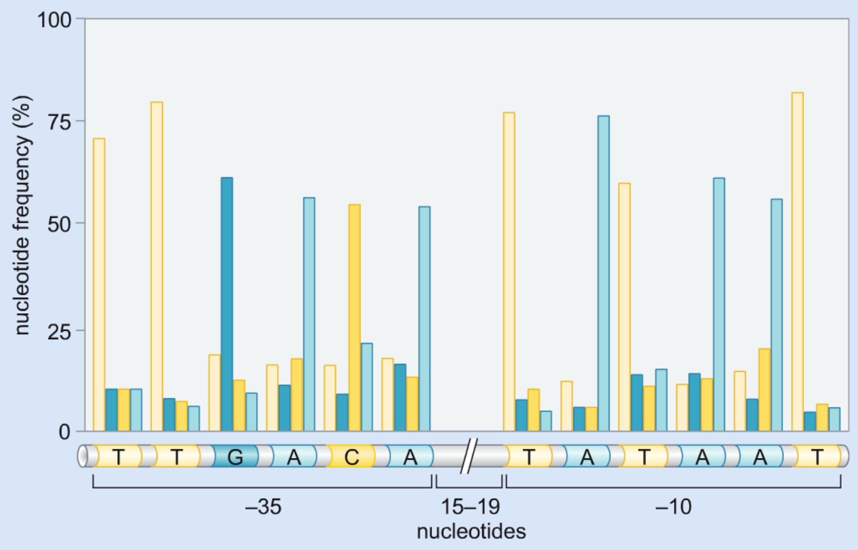
I expected to see the naïve algorithm run much slower than the dynamic programming algorithm. For small strings (15 characters or less) there was not much of a difference in running time, with both algorithms taking less than a second to run. But as the strings got larger, the dynamic programming algorithm continued to run in under a second, while the naïve algorithm took minutes (20 or more characters) to hours (30 or more characters). Obviously, the dynamic programming algorithm is much faster than the naïve algorithm. The naïve algorithm runs in exponential time, $O(2^n)$, when $n > m$. The dynamic programming algorithm runs in polynomial, $O(mn)$, time. There is a time-memory trade-off. The dynamic programming algorithm requires a bit more space than the naïve algorithm, however the difference in the time savings justifies this as the ideal choice between the two.

What I learned and what I would do differently
Before implementing these two solutions, I understood the difference between exponential running time algorithms and polynomial time algorithms, but watching them run first hand (and waiting) drove home the point. Even on small strings, there was an obvious pause each time the naïve algorithm was called. This exercise also helped me to learn how to pick out the subproblems when using dynamic programming and proving that a problem has overlapping substructure.

Next time I might try enhancing this by using a B-tree to store an entire human genome and pull into memory the pieces of the DNA one at a time. I also would like to try the top-down memoization approach to compare the two solutions and to see if the overhead of the recursive calls would cause any significant difference in time.

Application to Bioinformatics
Comparing two strings is a common theme in bioinformatics. The comparison of two strings of DNA base pairs, which consist of the set of characters $\sum = \{A, G, C, T\}$ that correspond to the bases adenine, guanine, cytosine and thymine, are often compared between organisms to find similarities. Comparing two strings of the DNA of two humans can tell you whether one human is related to another.

This discussion ties in nicely with the other course I am taking right now, which is Molecular Biology. Recently we learned about something called a consensus sequence. DNA has sequences near genes called promoters that can be recognized by enzymes that initiate transcription of the gene. The promoter contains a DNA sequence that is generally the same between each type of promoter but can vary. A consensus sequence describes the sequence that is most common across promoters of the same type. A program like this one that finds the longest common subsequence, could be used to find the consensus sequence across promoters. The program would need to be expanded to find the LCS between more than one string of DNA. Below is chart demonstrating the concept of a consensus sequence.[3]



BOX 13-1 FIGURE 1 Promoter consensus sequence and spacing consensus. (Redrawn, with permission, from Alberts B. et al. 2002. *Molecular biology of the cell*, 4th ed., p. 308, Fig. 6.12. © Garland Science/ Taylor & Francis Books LLC.)

References

1. Thomas H. Cormen. 2013. *Algorithms Unlocked*. The MIT Press.
2. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.
3. Watson, James D, Tania A. Baker, Stephen P. Bell, Alexander Gann, Michael Levine, and Richard M. Losick. *Molecular Biology of the Gene*., 2004.
4. Index of genome sequence data: ftp://ftp.ncbi.nih.gov/genomes/