## 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 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.

**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 did not know the size of the list of strings until the file read is finished. <More on ArrayList vs array>

The string are passed as character arrays to the Longest Common Subsequence algorithm. I did this so that it I could index each character in the strings easily and quickly.

For the dynamic algorithm, I constructed a 2-Dimensional array to hold the path to 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**

<TODO>

**What I did as an enhancement**

As an enhancement, 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 algorithm to only find the length of the longest common subsequence, and not the actual string. 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 took so long. 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 download some real DNA sequence data from <find site>. 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.

**Efficiency with respect to both time and space**

<Efficiency of naive algorithm – draw recursion tree>

(ATGA, CAG)

(ATG, CAG)\* (ATGA, CA)\*

(AT, CA) (ATG,C)

\*(A, CA) (AT, C) (AT, CA) (ATG, C)

(/, C) (A, C) (A, /) \*(A, CA) (AT,C) (AT, C) (ATG, /)

(/,/) (/,/) (/,/) (/, C) (A, C) (A,C) (A,/)

(/,/) (/,/) (/,) (/,/)

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 the tracing back through the table to construct the final sequence. 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 in order to construct the table that is need to compute the Longest Common Subsequence.

Obviously, the dynamic programming algorithm is much faster than the naïve algorithm. The naïve algorithm runs in exponential time, O(2mn), <CHECK THIS>, while the dynamic programming algorithm runs in polynomial, O(mn), time. There is a trade-off with regards to space, though. 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**

<describe my experience>

Next time I might try enhancing this by using a B-tree to store an entire human genome and pull in pieces of the DNA one at a time.

**Application to Bioinformatics**

<TODO>

**References**

<TODO – add site where I got sequence data from>