Morgan Baccus
CptS 350
Homework #7

Question #1

```
LongestCommonSubsequence (x, y)
lenx = length[x]
leny = length[y]

For u <- 0 lenx do
        for v <- 0 to leny do
           if (u=0 or v-0)
              lengthLCS[i,j] <- 0
           else if x[i-1] = y[j-1]
              lengthLCS[i, j] <- legthLCS[i-j-1] +1
           else lengthLCS[i, j] <- max(lengthLCS[i-1, j], lengthLCS[i, j-1]

Pointer <- lengthLCS[lenx, leny]
i <- lenx
j <- leny
while i>0 and j>0 do
        if x[i-1] = y[j-1]
           Seq[pointer-1] = x[i-1]
           i <- i-1
           j <- j-1
           pointer <- pointer -1
        else if LengthLCS[i-1, j] > lengthLCS[I, j-1]
           i <- i-1
        else j <- j-1

Answer <-  ‘ ‘
For u <-0 to length[Seq]-3 do
        if Seq[u]=a and seq[u+1]=b
           u-> u+2
        else
           answer = answer+Seq[u]
return answer
```

Question #2

Given two sequences, we must find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order in both sequences, but doesn't need to be continuous.

We will solve this using two methods.

<u>Method 1: Brute Force Method</u>

For this method, we will find all the sub-sequences of any one sequence and then compare each sub-sequence with the other sequences to see if it is common or not. Then, we will find the max length sub-sequence.

1. First, we need to find all length sub-sequences. If we have a string of length n then the sub-sequences can be of length 1 ,2 ,3, …, n. Using permutations and combination:

Total sub-seuqences = ${}^nC_1 + {}^nC_2 + \ldots {}^nC_n$

Therefore, a string of length n has 2^n-1 different possible subsequences since we do not consider subsequences with length 0.

2. Next, we will check if a sub-sequence is common to both the strings. This will take O(n) time.

The total time needed to check all the sub-sequences = $n*2^n-1$

The time complexity using the brute force method is =$O(n*2^n-1)$.

<u>Method 2: Dynamic Programming</u>

Using dynamic programming, we store the results in a table so if any same sub-sequence occurs again we can simply refer to the table and get the results. This reduces the time of computing results for same sub-sequence again and again.

Algorithm using DP

Let us have two sequences X and Y where X [0, m-1] and Y is [0, n-1].

LCS(m, n)

1. Check if length of any string is 0 or not.
(m==0 || n==0) return 0 .

2. Check if the last character of both strings matches or not.
if (X[m]== Y[n])
return (1+ LCS(m-1, n-1)

This return statement means that the last character is a match so now we will recursively call the algorithm to check the common characters in length of m-1 and n-1

3. Else if none of steps satisfy find c=max(LCS(m-1,n) , LCS(m,n-1))
return c

Question #3

Using the Jaccard index, we can implement a locality sensitve hash for strings.

First, we need to define a method of determining whether a string is a duplicate of another. The Jaccard index performs fairly well for this problem. The Jaccard index is an intersection over a union. We count the amount of common elements from two sets, and divide by the number of elements that belong to either the first set, the second set, or both.

<u>Example:</u>
Given two strings a=’abcd’ and b=’aceg’, we can calculate the Jaccard Index.

The strings both contain a and c so the size of the intersection is 2. The size of the union is 2 + 2 + 2= 8. Thus, the Jaccard Index is equal to 2/8 = 0.25. The more common words between the two strings, the higher the Jaccard Index.