Conner Mattingly
11282717
Cpts 350 HW5

1.
First we must understand how the LCS algorithm works in order to create another algorithm that utilizes the LCS or adapts it in some way. For this problem the LCS algorithm breaks the string parameters down into smaller sequences that we can call prefixes. A prefix being an original sequence that had the end of itself cut off. For example a prefix of the sequence ABCD would be ABC since the letter D was removed from the end of it.

Using the idea of a prefix we can continue with the solution. For this problem LCS runs on two parameters (a) and (B). Each call of LCS will do one of two things.

1. If a and B end with the same element then the end element of each sequence is removed, but stored (saved for later), and the LCS of the shortened sequences is sought. The removed element is then appended to the result of the second LCS operation.

2. If a and B do not end with the same element then the LCS of a and B is the longer of the two sequences returned by LCS(a,B-1) and LCS(a-1, B). The -1 indicates the last element being chopped off of the signified sequence.

This process is followed for all subsequent calls of LCS. (recursive algorithm)

Following this process we can filter the subs-equences that are returned by each LCS operation. Specifically for this problem we want to filter out any sub-sequence that contains "abb". To do this we can just check the result of each LCS operation right before it returns with the following checks:

If the last element is 'b'
  then check the second to last for another 'b'
  if it is another 'b' then check if the third to last element is an 'a'
    if it is then return the subset collected so far minus the last element, that being the first 'b' checked for.

This check and removal will ensure that any final substring that LCS discovers will never have "abb" contained within it. At most it can contain an arbitrary amount of "ab"'s.

Now for analysis.
As we saw above the LCS runs on two strings a and B. Lets say their respective lengths are n and m. The run time for LCS on its own is known to be O(n*m) This makes sense since comparisons are made between all of the elements of each string. However the question now is what the filtering addition adds to the time complexity.

Lets consider worst case. In the worst case the comparison must check three elements and perform a removal this is a constant time step. Therefore we must add 3 (or 4) to each LCS operation. This would cause the time complexity to go from O(n*m) to O(3(n*m)) or O(4(n*m)) if we count the removal.

2.
The easiest way that I can think to calculate the length would be to iterate over the subsequence of the LCS(a,B). However, I think this may be a naive approach so I would instead use the ideas I put forth in the solution to problem 1 in order to create an algorithm for d(a,B).

d(a,B) is essentially a call to LCS in which the inner steps of LCS are altered in the following way.
This again means using the two cases that LCS runs into. The first being that the last element of each string match. In this case the next call to LCS would be 1 + LCS(a-1, B-1) again the -1 indicates that the last element has been removed. The plus one is an increment on the size of the subsequence that will be returned. For the second case, that being the end elements do not match, I would just return the max of LCS(a-1,B)and LCS(a,B-1). After seeing how only the case in which the elements match will increment the return value of d(a,B), one can understand how recursively this algorithm would end up returning the correct size of the LCS.

3.
Initial thoughts lead me to think that the first thing that must be designed is a way to compare the two given circles. I think placing the circles on a x-y coordinate plane would be the easiest way to compare the points. Once these circles, and more importantly their points, are placed on the x-y plan. The points can be compared through use of the distance formula and straight equivalence comparisons for the color values.

There are many ways to do this since there is an x, y, and c value in each element. Each of these ways would also lead me to a different design for my algorithm.

One way would be to sort each array based on the color value c. That way the arrays could be iterated and checked for matching c values and lengths. If the c values do not all match, or the array lengths differ. Then the algorithm would return false since the points defy the requirements for similar circles. Next the arrays would have to be sorted again according to each of the (x,y) pairs distance from the origin. This way the arrays could be iterated again to make sure that each (x,y) pair in C1's array matches the pairs in C2's array. However, I just realized that this leads to a problem of the color values not matching for matching points requireing further comparisons. So here is another attempt at the algorithm.

First sort the arrays on the (x,y) pair's distance from the origin then compare each array element against each other to ensure similarity (similarity being that all 3 values match). I believe this to be more efficient than my previous approach.

Both of these algorithms assume that the first three requirements have been followed. Some points on each circle have been erased and C2 has already been rotated. Also, at some point, k must be validated for C1 and C2. This can be checked before or after sorting the arrays. To check for k just compare length of each array to k and make sure they are equivalent. If the lengths of the arrays must be calculated I would design that calculation to be included in my sorting of the arrays.