Wyatt Kormick
Assignment 4 CSCI4041
DUE: Sunday, November 6, 10:00pm (+1hr55min grace period) Fall 2016

IF YOUR ASSIGNMENT IS NOT ACCEPTED BY TURNITIN BECAUSE IT DOES NOT CONTAIN 120+ WORDS OF DIGITAL TEXT, 3 POINTS WILL BE DEDUCTED FROM YOUR HOMEWORK.

1. Constructing Solutions After Calculating Optimal Cost (15.4-2)

Give pseudocode to reconstruct an LCS from the completed c table and original sequences X = <x_1, x_2, …, x_m> and Y = < y_1, y_2, …, y_n> in O(m+n) time, without using the b table.

```
REPRINT-LCS(X,Y,c-table,a,b)
    if a = 0 or b = 0
        return
    if X[a] = Y[b]
        REPRINT-LCS(X,Y,c-table,a-1,b-1)    //Move to Upper-Left
        print X[a]
    else if c-table[a-1, b] >= c-table[a,b-1]
        REPRINT-LCS(X,Y,c-table,a-1,b)      //Move Up
    else
        REPRINT-LCS(X,Y,c-table,a,b-1)      //Move Left
REPRINT-LCS(X,Y,c,m,n)
```

O(m+n) because at worst it moves all the way up, m, then all the way left, n (no diagonal movements through the c-table)

2. Editing Distance (version of 15-5)

Write a polynomial time algorithm to calculate optimal cost (minimal editing distance) between 2 strings, as described in problem 15-5. However, use only the operators Delete, Insert, and Twiddle. This can be iterative or recursive. State the runtime with a brief justification.

```
EDIT-DISTANCE(X, Z, a, b)
    new array s-table[a][b]
    for i = 0 to a
        for j = 0 to b
            if i = 0                    //X is empty
                s-table[i][j] = j       //Insert all of Z into X
            else if j = 0               //Z is empty
                s-table[i][j] = i       //Delete all of X
            else if X[i] = Z[j]         //Last chars are the same
                s-table[i][j] = s-table[i-1][j-1]
            else
                s-table[i][j] = 1+ min(
                    s-table[i][j-1],    //Insert
                    s-table[i-1][j],    //Delete
                    s-table[i-2][j-2])  //Twiddle
    return s-table[a][b]
EDIT-DISTANCE(X, Z, m, n)
```

O(mn)

It iteratively fills in a table with dimensions equal to the lengths (m,n) of the two strings. It is an O(1) operation to get the value once the table has been filled in.

3. Modified Activity Selection

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. Give a polynomial-time algorithm for this problem. Justify the runtime.

```
//Assume Sorted in increasing order by finish time
//Format of Activity Array:
//{{start-time1,finish-time1,value1}, {start-time2, finish-time2, value2}, etc}
WEIGHT-ACT-SELECT( activity[][])
    n = length(activity)
    new array s-table[n]
    for i = 1 to n
        val = activity[i][3]
        prev = FIND-PREV-ACTIVITY(activity, i)
        if prev != -1
            val = val + s-table[prev]
        //If a previous activity was found
        //Add its value to the value of our current
        s-table[i] = max(val, s-table[i-1])
        //Determine if its better to include or exclude
    return table[n]

FIND-PREV-ACTIVITY(activity[][], i)
    for j = i-1 down to 0
        if activity[j][2] < activity[i][1]
            return j
    return -1
    //search for an activity with a finish time
    //less than or equal to the start of our current activity
    //Return its index if found, Return -1 if one isn't found
```

This algorithm has $O(n^2)$ running time. It iterates through a new table, filling in the values of all of the subproblems. Inside the loop another loop occurs, iterating through the list of activities as a search for the previous activity. It is then O(1) to get the value of our problem out of the generated s-table. This could be reduced to O(n log n) by changing the search algorithm into a binary search.

4. Optimal Substructure

Consider the problem of given a set $\{ x_1, x_2, \ldots, x_n \}$ of points on the real line, determine the smallest set of unit-length closed intervals that contains all of the given points. Prove that this problem exhibits optimal substructure.

Y = sort(X) = {y1, y2, … , yn}

The first (left-most) interval is [y1, y1 + 1]

The optimal solution S is the union of the left-most interval [y1, y1 + 1] and the optimal solution of all of the intervals to the right of y1 + 1

Proof:
Claim: There is an optimal solution S that contains [y1, y1 + 1]
Suppose: There is an optimal solution S' that contains [x', x' + 1] which covers y1, and x' < 1
Since y1 is the left-most point of our set, we can replace [x', x'+1] with [y1, y1+1] to get another optimal solution.
Therefore: This problem exhibits optimal substructure.

5. Greedy Choice Property

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of **Π** { i over 1 to n } a_i^b_i. Prove that the greedy choice property holds for this algorithm.

Both a and b should be sorted in monotonically increasing order.

Some global solution contains our greedy choice a_g paired to b_g

Given another global solution without our greedy choice, swap in our greedy choice

$$\frac{new\ value\ with\ greedy\ choice}{old\ value} = \frac{a_1^{b_g} * a_g^{b_1}}{a_1^{b_1} * a_g^{b_g}} = \left(\frac{a_1}{a_g}\right)^{b_g - b_1} \geq 1$$

Our solution with the greedy choice swapped in is as good as our greedy solution