Jonathan Koning
koningj@oregonstate.edu

HW 7

2. (1 point) Give an *O(mn)* algorithm for finding the longest common substring of two input strings of length *m* and *n*. For example if the two inputs are 'Philanthropic" and "Misanthropist," the output should be "anthropi."

```
def longsub(str1, str2):

    longstr = ""

    i = 0

    k = 0

    while i <len(str1):

        idxs = [idx for idx, value in enumerate(str2) if value==str1[i]]

        for idx in idxs:

            k=0

            tempstr = ""

            while i+k < len(str1) and k+idx < len(str2):

                if(str1[i+k] == str2[k+idx]):

                    tempstr+= str1[i+k]

                    k+=1

                else:

                    break

            if(len(tempstr) > len(longstr)):

                longstr = tempstr

        i+=k
```

This is O(nm) because str1 is only being iterated through once which accounts for the n. We then iterate through str2 for every instance of the checked string. This makes the overall complexity

3. (1 point) BigBucks wants to open a set of coffee shops in the I-5 corridor. The possible locations are at miles d1,...,dn in a straight line to the south of their Headquarters. The potential profits are given by p1...pn. The only constraint is that the distance between any two shops must be at least k (a positive integer).

- Construct a counterexample to show that a greedy algorithm that chooses in the order of profits could miss the optimal (most profitable) solution.

Given an array, A, of the form (p,d):

A: [(10, 1), (1, 2), (9, 3), (4, 4), (2, 5), (7, 6), (5, 7), (9, 8), (1, 9), (2, 10), (9, 11), (9, 12), (6, 13), (3, 14), (7, 15)]

And k=2. A greedy algorithm will find the following solution:

Solution: [(10, 1), (9, 3), (7, 6), (9, 8), (2, 10), (9, 12), (7, 15)]

Value = 53

The correct solution is as follows:

Solution: (10, 1), (9, 3), (7, 6), (9, 8), (9, 11), (6, 13), (7, 15)]

Value: 57

- Give an efficient dynamic programming based algorithm to maximize the profit.
    - Set all d to corresponding p value
    - Perform a breadth first search for each starting node (there will be k of them)
    - Update the value of each node if the value is less than the parent node + current node
    - Dist(d) = p
    - Dist(d) = Max(di-1, di)Dist(d) + Dist(p)

```
For all (d, p):
        Dist[d] = value
        Prev[d] = -1

#The first k values are the root nodes and so will have no parent
H = []
```

```
For i in range(k):
    h.append(i)
#Breadth first search
While len(h) >0:
    Parent = h.pop(0)
    for i in range(k):
        Child = Parent + k + i
        if(Child < len(p)):
            #If the base value of the child node + the greatest value to the Parent
            #node is greater than the current greatest value to the child node,
            #update the node and add it to the heap.
            if( (p[Child][0] + dist[p[Parent][1]) > dist[p[Child][1]]):
                prev[p[Child][1]] = p[Parent][1]
                dist[p[Child][1]] = p[Child][0] + dist[p[Parent][1]]
                h.append(Child)
```

4. (1 point) In a rope cutting problem, cutting a rope of length *n* into two pieces costs *n* time units, regardless of the location of the cut. You are given m desired locations of the cuts, X1, ..., Xm. Give a dynamic programming-based algorithm to find the optimal sequence of cuts to cut the rope into m+1 pieces to minimize the total cost.

```
#Loop through each rope, find the cut that will cut the rope in half as much as possible
#Add the cut to the sequence
#Add the current length of the rope to the cost
#Make new ropes
#Repeat until the sequence has all of the cuts

Sequence = []
Cost = 0
Ropes = [[0,n]]
While len(sequence) < len(X):
    #iterate through the ropes
    For rope in ropes:
        Cut = -1
        Dist = inf
        #find the cut that will split the current rope as close to in half as possible
        mid = rope[0] + ((rope[1]-rope[0])/2)
        For all x in X if x> rope[0] and x<rope[1]:
            if(abs(mid - x) < dist):
                Cut = x
                Dist =  abs(mid - x)
        #add cut to the sequence
        sequence.append(Cut)
        #Add to the total cost
        cost+=(rope[1]-rope[0])
```

```
#Create a new rope
Newrope = [rope[0], Cut]
#resize the cut rope
Rope[0] = Cut
#add new rope to the list of ropes
Ropes.append(newrope)
```