Problem 1

1. Suppose that you have a graph $G = (V, E)$ where each node is labeled with a digit. Design an efficient algorithm that, given two nodes $s, t \in V$, decides whether there is a path from s to t in which the node labels follow the pattern 37437437... .

   Idea: Perform a depth-first search where the routes we take are the routes that follow the 3743743743... pattern. We start at a node called "source" and end when either we don't find a path to our destination or do find a path that follows the rule. The algorithm is as follows:

```
# Just a class definition to make things
# more clear
class Vertex:
    def __init__(self, dest, label):
        self.label = label
# Determines if the next node follows
# the pattern rule
def IsValidTransition(a, b):
    if a.label == 3 and b.label == 7:
        return True
    if a.label == 7 and b.label == 4:
        return True
    if a.label == 4 and b.label == 3:
        return True
    return False

# Find if a path from source to dest
# exists that follows the 374 pattern
# assumption: We use adjacenty list
# data structure
def FindPath(V, E, source, dest):
    stack = []
    visited = []
    stack.append(source)
    # continue while we have a full
    # stack
    while len(stack) > 0:
        # pop off an element and
        # get it's neighbors
        curr = stack.pop()
        visited.append(curr)
        neighbors = E[curr]

        # If we have found a path return true
        if (dest in neighbors) and (IsValidTransition(curr, dest)):
            return True
        # OTherwise continue our depth-first search
        else:
            for n in neighbors:
                if (not n in visited) and (IsValidTransition(curr, n)):
                    stack.append(n)
    # return False if we haven't found a path
    return False
```

2. Suppose now that the labels are on edges instead of vertices. Design an algorithm that reduces this problem to the previous part, i.e., calls the previous part as a subroutine.

   Instead of transforming the problem to fit with that of the previous problem, I will modify the previous algorithm to work on edge labels (and hopefully that is worth some partial credit).

   The algorithm works in a straightforward way, we first assume we have a function called edgeLabel(a, b) which computes the label of the edge between nodes a and b. The IsValidTransisition() is essentially the same as the previous problem. The algorithm works in practically the same way as with vertex labels, but this time in our stack we keep track of the edge label we "picked up" to get to that node, and when moving forward we just check the edge label between our current node and it's neighbors. The algorithm is as follows:

```
# Find a path that satisfies the pattern
# with edge labels
def FindPathEdge(V, E, start, dest):
    # Prepare to do a stack-based
    # depth first search
    visited = []
    stack = []
    # Append our initial values
    for n in E[start]:
        # The stack saves the node and the edge label
        # that was "picked up" on our way to get to that
        # node
        stack.append((edgeLabel(start, n), n))
    while len(stack) > 0:
        # Pop off our current node
        curr = stack.pop
        visited.append(curr)
        neighbors = E[curr[1]]

        # Check if we have found a valid path
        if ((dest in neighbors) and
            IsValidTransition(curr[0], edgeLabel(curr[1], dest))):
            return True

        # Otherwise keep searching
        for n in neighbors:
            el = edgeLabel(curr[1], n)
            goodPath = IsValidTransition(curr[0], el)
            if ((not (el, n) in visited) and goodPath):
                stack.append(el, n)
    # If we've searched everything with no success,
    # return no path
    return False
```

3. Going back to vertex labels, design an algorithm to see if there is a path from s to t where the sequence of vertex labels matches a regular expression r (over the alphabet $\{0, ..., 9\}$).

   Let us use the DFA representation of our regular expression, and suppose we are given starting state *dfa_start*, the array of accepting states *dfa_accepting*, and the delta function *delta*(*current_state*, *transition*). Using this, we can modify our algorithm from part A to not only find a path through our graph from start to finish but also check if the path traversed matches our regular expression. The algorithm is as follows:

```
# Checks if a path through the graph
# can be found from source to dest
# that satisfies a regular expression
def FindPathRegex(V, E, source, dest, dfa_start, dfa_accepting, delta):
    stack = []
    visited = []
    stack.append((dfa_start, source))
    # continue while we have a full
    # stack
    while len(stack) > 0:
        # pop off an element and
        # get it's neighbors
        curr = stack.pop()
        visited.append(curr)
        neighbors = E[curr[1]]

        # Check if we've found a path
        if curr[1] == dest and curr[0] in dfa_accepting:
            return True

        # otherwise keep searching
        for n in neighbors:
            p = (delta(curr[0], neighbor.label), neighbor)
            if not p in visited:
                stack.append(p)

    # return False if we haven't found a path
    return False
```

   Another way to think about the algorithm is that as we search the graph from source to destination we are also keeping track of where we are in a DFA. If we arrive at our destination and we are in an accepting state of our DFA, we know we have arrived at a solution and can return true.

Problem 2

1. Describe an algorithm to determine whether a given parallel assignment can be serialized without additional temporary variables

   Let us represent the parallel assignment as a directed graph. The nodes / vertices of the graph are the variables being assigned, i.e. for every left hand element we have a node. The edges represent what each node is dependent on, and point from the variable to the variable / node it depends on besides itself. For example, if we have the expression:

   $$x, y = x + y, x - y$$

   Then x points to y and y points to x, forming a small cycle.

   We need to introduce a temporary variable whenever there is a cycle present in this graph. In other words, whenever this graph has no cycles, we can perform the action without introducing any new or temporary variables. The algorithm is as follows, given a graph in adjacency list form representing the previously described format:

```
# Check if we need to introduce a temporary
# variable
# True = no temp vars needed
def NoTempVarsNeeded(E, V):
    # begin by ensuring all vertices
    # are marked as new
    for vertice in V:
        vertice.status = 'new'
    # Check each individual vertice for
    # a cycle
    for vertice in V:
        if vertice.status == 'new':
            # If the vertice has a cycle
            # return False
            if IsAcyclicDFS(vertice) == False:
                return False

    return True

# Check if a given vertice has is included
# in any cycle
def IsAcyclicDFS(E, V, vertice):
    # Set our vertice status to active
    vertice.status = 'active'
    for neighbor in E[vertice]:
        # if the neighbor is currently active
        # or is not acyclic itself, then
        # we return false
        if neighbor.status == 'active':
            return False
        else if neighbor.status == 'new':
            if IsAcyclicDFS(neighbor) == False:
                return False
    vertice.status = 'finished'
    return True
```

Runtime Analysis

Analysis for this above algorithm is quite simple, because for each node we check each of it's edges. So we check every single node, and we check every single edge. Therefore, our runtime is $O(V + E)$. Whenever there is no cycle, we will end up visiting every node and every edge to check to ensure. Supposing then we have n variables to work with, the number of edges we can form is then bounded by n^2, so with n nodes and a bound of n^2 edges, our runtime is then $o(n^2)$ (little-o notation).

2. Describe an algorithm to determine whether a given parallel assignment can be serialized with exactly one additional temporary variable.

   Notice that we only need 2 or more temporary variables whenever we have more than 1 cycle. Therefore, we can modify our algorithm from the previous part to detect if we'll need more than one temporary variable by detecting if there is more than 1 cycle in the graph.

   Note: If there are no cycles in the graph we can still use the temporary variable by copying the contents of one variable into it, therefore if we need 1 or less temporary variables then we can serialize the operation with exactly one additional temporary variable. The only time we cannot serialize the operation with exactly one temp variable is whenever we need 2 or more temporary variables.

```
# Check if we need to introduce more than 1 temporary variables
# True = no more than 1 temp var needed
def NoTempVarsNeeded(E, V):
    # begin by ensuring all vertices
    # are marked as new
    for vertice in V:
        vertice.status = 'new'
    # Check each individual vertice for
    # a cycle
    for vertice in V:
        if vertice.status == 'new':
            # If the vertice has a cycle
            # return False
            if IsAcyclicDFS(vertice) == False:
                return False
    return True

# Check if a given vertice has is included
# in any cycle
def IsAcyclicDFS(E, V, vertice):
    num_cycles = 0
    # Set our vertice status to active
    vertice.status = 'active'
    for neighbor in E[vertice]:
        # if the neighbor is currently active
        # or is not acyclic itself, then
        # we return false
        if neighbor.status == 'active':
            return False
        else if neighbor.status == 'new':
            if IsAcyclicDFS(neighbor) == False:
                # We have detected a cycle, increase
                # our count
                num_cycles += 1
                # If we have more than 2 cycles, return
                # False
                if num_cycles >= 2:
                    return False
    vertice.status = 'finished'
    return True
```

Runtime Analysis

This algorithm is nearly identical to that of the last one, except this time we have a count of the number of cycles we find, leaving all else the same. Therefore, our runtime again is then $o(n^2)$ for n variables. See the last part for an explanation.

Problem 3

1. Describe and analyze an algorithm to compute the length of the longest monotonically increasing path with vertices in S.

   This algorithm is a bit more complex than the previous problems. The first step is to convert the points into a directed acyclic graph. This is simple, for each point, draw a directed line to every point who's x and y position are greater than that of itself.

   Now that we have generated our graph, we topologically sort it. This gives us an efficient order to compute the longest path. For each element in the topological sorting, we can recursively compute the longest path possible from said element using the following recursion (*notation note: LP(v) = "longest path" from vertex v)*:

$$LP(v) = \begin{cases} 0 & \text{v is a sink} \\ max\{length(v \to w) + LP(w) | v \to w \in E\} & \text{otherwise} \end{cases}$$

   Once we have performed LP(v) for all v in the graph, then we simply select the node with the longest LP value. The algorithm is as follows:

```
# Vertex class definition, may be helpful for
# reading the code
class Vertex:
    def __init__(self, index, LP=-1, status='new'):
        self.index = index
        self.LP = LP
        self.status = status

# Return the longest path from
# vertex d
def LongestPathTopological(E, V, d):
    # if d is a sink then return 0
    if len(E[d.index]) == 0:
        d.LP = 0
        return 0
    d.LP = neg_inf
    # Otherwise return the longest path
    # recursively looking at the vertices
    # it is connected to
    for edge in E[d.index]:
        LongestPathTopological(E, V, edge)
        l = Length(X[d.index], Y[d.index],
                   X[edge.index], Y[edge.index])
        l += edge.LP
        d.LP = max([l, d.LP])
    return d.LP

# Topological sort helper function, essentially
# a python implementation of the algorithm
# given by Jeff Erickson in 'Algorithms'
def TopSortDFS(E, v, S, clock):
    v.status = 'active'
    for n in E[v.index]:
        if n.status == 'new':
```

```
                clock = TopSortDFS(E, n, S, clock)
        v.status = 'finished'
        S[clock] = v
        clock -= 1
        return clock

    # Topological sort - by topologically sorting the
    # vertices it will give us an efficient way to
    # compute the longest path
    def TopSort(E, V):
        for vertice in V:
            vertice.status = 'new'
        clock = len(V) - 1
        S = [None] * len(V)
        for vertice in V:
            if vertice.status == 'new':
                clock = TopSortDFS(E, vertice, S, clock)
        return S

    # compute the longest monotonically increasing
    # path among a given set of points X and Y
    def LongestPath(X, Y):
        # initialize points and generate a graph
        # of all the points on the plane
        V = [ Vertex(i) for i in range(len(X))]
        E = [ [] for i in range(len(X))]
        for source in V:
            for dest in V:
                if source == dest:
                    continue
                elif (X[dest.index] > X[source.index]
                    and Y[dest.index] > Y[source.index]):
                    E[source.index].append(dest)
        # topologically sort elements
        S = TopSort(E, V)
        # Compute longest path possible for each
        # vertice
        for d in S:
            if d.LP == -1:
                LongestPathTopological(E, V, d)
        # return the longest path among all the
        # vertices
        return max(v.LP for v in S)
```

Runtime Analysis

(a) Graph Generation

Suppose we have n points on a plane. For every point, we compare it with every other point to see if we need to draw an edge to it. So we have n vertices, and n comparisons for each vertex, so therefore we have $\Theta(n^2)$ runtime to generate the graph.

(b) Topological Sort

Topological sort runs in $O(V + E)$ time. Since we have n vertices, and we know that the number of edges cannot be larger than n^2, we have a bound on our runtime of $O(V + E) \in o(n^2 + n)$.

(c) Longest Path Calculation

Despite the convoluted, recursive nature of the longest path calculation the longest path calculation runs in linear time. Computing the longest path for a vertex takes linear time, as it is simply checking if it is a sink or taking the max of the length plus the longest path of all the vertices it is connected to. Therefore, with n nodes, we can compute the longest path in $\Theta(n)$ time.

Graph Generation	$\Theta(n^2)$
Topological Sort	$o(n + n^2)$
LP Calculation	$\Theta(n)$

Therefore, our total runtime is $\Theta(n^2) + o(n + n^2) + \Theta(n)$, so our total runtime is that for n nodes we can compute the longest monotonically increasing path among them in $O(n^2)$ time.

Total Runtime: $O(n^2)$