6.00 Quiz 3 ----------

Name

1. _____/15
2. _____/10
3. _____/10
4. _____/15
5. _____/10
6. _____/10
7. _____/15
8. _____/15

Total _____ /100

This quiz is open book and open notes, but do not use a computer (or cell phone!). You have 120 minutes.

Please **write your name on the top of each page**, and your user name and the hour of the recitation you attend on the first page. Answer all questions in the boxes provided.

1) Are each of the following True or False? (15 points)

False	1.1. In Python, classes **cannot** be used as actual parameters.
True	1.2. If B is a subclass of A, the implementations of methods in A should **not** call methods defined only in B.
False	1.3. Monte Carlo simulations are **no**t useful for solving problems that don't involve randomness.
True	1.4. When run on the same input, agglomerative hierarchical clustering always yields the same result.
False	1.5. The 0/1 knapsack problem can be solved in O(n log n) time.

2) What does the following code print? (10 points)

```
import pylab

xVals = pylab.arange(0, 10)
yVals = 2*pylab.arange(0, 10)
a, b = pylab.polyfit(xVals, yVals, 1)
print round(a)
print round(b)
```

3) Bill flipped three fair coins. What is the probability that **exactly** two of them came up heads? (10 points)

4) Next to each item in the left column write the letter labeling the item in the right column that best matches the item in the left column. No item in the right column should be used more than once. (15 points)

☐	polymorphism	a) subclassing
☐	standard deviation	b) overlapping sub-problems
☐	clustering	c) confidence interval
☐	dynamic programming	d) linkage
☐	merge sort	e) GIGO
		f) O(n**3)
		g) O(n log n)
		h) Exceptions

The following questions all refer to the code you were asked to study in preparation for this exam. (For your convenience, a copy of the posted code is at the end of this quiz. Feel free to separate it from the rest of the quiz.)

5) Consider the call `bigTest(30, 100, 1, 1).` Is it possible to predict the value returned by `minWeightPath` based upon the value returned by `shortestPath`? Why or why not? (10 points)

6) If `g.addEdge(Edge(nodes[0],nodes[1], 1))` were replaced by `g.addEdge(Edge(nodes[0],nodes[1], -1))` in `test`, would it change the result of running `test`? If so, how? If not, why not? (10 Points)

7) Assume that `G` is a `Digraph` in which each node represents a city (i.e., `n.getName()` returns the name of a city), and the weight of each edge the cost of flying nonstop between pairs of cities. Write a function that meets the specification below. (15 points)

```
def cheapestTrip(city1, city2, G):
    """Assumes that city1 and city2 are strs.
       Returns the cost of the least expensive way to fly round trip
        between city1 to city2.
       If no such path exists (e.g., because there
       is no node corresponding to city2), it returns None."""
```

8) The function below is similar to `minWeightPath`. The difference is that it has and uses a parameter, `foo`, that `minWeightPath` does not. Describe, in English, what problem this code solves. (15 points)

```
def minWeightPath2(graph, start, end, foo = None, path = None, edge = None):
    if not (graph.hasNode(start) and graph.hasNode(end)):
        raise ValueError('Start or end not in graph.')
    if foo == None:
        foo = graph.numNodes()
    if path == None:
        path = Path(start)
    else:
        if path.getLength() >= foo:
            return None
        path = path + edge
    if start == end:
        return path
    shortest = None
    for edge in graph.edgesOf(start):
        if not path.contains(edge.getDestination()):
            newPath = minWeightPath2(graph, edge.getDestination(),
                                     end, foo, path, edge)
            if newPath != None:
                if shortest == None or\
                   newPath.getWeight() < shortest.getWeight():
                     shortest = newPath
    return shortest
```

```
import random

class Node(object):
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src, dest, weight = 1):
        self.src = src
        self.dest = dest
        self.weight = weight
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def getWeight(self):
        return self.weight
    def __str__(self):
        return str(self.src) + '->' + str(self.dest)\
               + '(' + str(self.weight) + ')'
```

```
class Digraph(object):
    def __init__(self):
        self.nodes = set([])
        self.names = set([])
        self.edges = {}
    def addNode(self, node):
        if node.getName() in self.nodes:
            raise ValueError('Duplicate node')
        else:
            self.nodes.add(node)
            self.names.add(node.getName())
            self.edges[node] = []
    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self.nodes and dest in self.nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(edge)
    def numNodes(self):
        return len(self.nodes)
    def childrenOf(self, node):
        result = []
        for e in self.edges[node]:
            if not e.getDestination() in result:
                result.append(e.getDestination())
        return result
    def edgesOf(self, node):
        result = []
        for e in self.edges[node]:
            result.append(e)
        return result
    def hasNode(self, node):
        return node.getName() in self.names
    def __str__(self):
        res = ''
        for k in self.edges:
            for e in self.edges[k]:
                res = res + str(e) + '\n'
        return res[:-1]
```

```
class Path(object):
    def __init__(self, start):
        assert type(start) == Node
        self.val = [(start, 0.0)]
    def addStep(self, edge):
        if self.val[-1][0] != edge.getSource():
            raise ValueError('Not a continuation of path')
        self.val.append((edge.getDestination(), edge.getWeight()))
    def getStart(self):
        return self.val[0][0]
    def getWeight(self):
        result = 0.0
        for step in self.val:
            result += step[1]
        return result
    def getLength(self):
        return len(self.val) - 1
    def __add__(self, edge):
        result = Path(self.getStart())
        for elem in self.val[1:]:
            result.val.append(elem)
        result.val.append((edge.getDestination(), edge.getWeight()))
        return result
    def contains(self, node):
        for step in self.val:
            if step[0] == node:
                return True
        return False
    def __str__(self):
        result = ''
        for step in self.val:
            result = result + '->' + str(step[0])
        return result[2:]

def minWeightPath(graph, start, end, path = None, edge = None):
    if not (graph.hasNode(start) and graph.hasNode(end)):
        raise ValueError('Start or end not in graph.')
    if path == None:
        path = Path(start)
    else:
        path = path + edge
    if start == end:
        return path
    shortest = None
    for edge in graph.edgesOf(start):
        if not path.contains(edge.getDestination()):
            newPath = minWeightPath(graph, edge.getDestination(),
                                    end, path, edge)
            if newPath != None:
                if shortest == None or\
                   newPath.getWeight() < shortest.getWeight():
                    shortest = newPath
    return shortest
```

```
def shortestPath(graph, start, end, visited = [], memo = {}):
    if not (graph.hasNode(start) and graph.hasNode(end)):
        raise ValueError('Start or end not in graph.')
    path = [str(start)]
    if start == end:
        return path
    shortest = None
    for node in graph.childrenOf(start):
        if (str(node) not in visited):
            visited = visited + [str(node)]
            try:
                newPath = memo[node, end]
            except:
                newPath = shortestPath(graph, node, end, visited, memo)
            if newPath == None:
                continue
            if (shortest == None or len(newPath) < len(shortest)):
                shortest = newPath
                memo[node, end] = newPath
    if shortest != None:
        path = path + shortest
    else:
        path = None
    return path
```

```
def test():
    nodes = []
    for name in range(10):
        nodes.append(Node(str(name)))
    g = Digraph()
    for n in nodes:
        g.addNode(n)
    g.addEdge(Edge(nodes[0],nodes[1], 1))
    g.addEdge(Edge(nodes[1],nodes[2], 2))
    g.addEdge(Edge(nodes[2],nodes[3], 3))
    g.addEdge(Edge(nodes[3],nodes[4], 4))
    g.addEdge(Edge(nodes[3],nodes[5], 5))
    g.addEdge(Edge(nodes[0],nodes[2], 6))
    g.addEdge(Edge(nodes[1],nodes[1], 7))
    g.addEdge(Edge(nodes[1],nodes[0], 8))
    g.addEdge(Edge(nodes[4],nodes[0], 9))
    shortest = minWeightPath(g, nodes[0], nodes[4])
    print 'The minWeight path is', shortest
    print 'The weight is', shortest.getWeight()
    shortest = shortestPath(g, nodes[0], nodes[4])
    print 'The shortest path is', shortest

def bigTest(numNodes = 30, numEdges = 100, minWeight = 1, maxWeight = 100):
    nodes = []
    for name in range(numNodes):
        nodes.append(Node(str(name)))
    g = Digraph()
    for n in nodes:
        g.addNode(n)
    for e in range(numEdges):
        src = nodes[random.choice(range(0, len(nodes)))]
        dest = nodes[random.choice(range(0, len(nodes)))]
        weight = random.choice(range(minWeight, maxWeight+1))
        g.addEdge(Edge(src, dest, weight))
    sp = shortestPath(g, nodes[0], nodes[4])
    print 'Shortest path =', sp
    sp = minWeightPath(g, nodes[0], nodes[4])
    if sp != None:
        print 'The minimum weight path =', sp
        print 'Weight of path =', sp.getWeight()
```

6.00SC Introduction to Computer Science and Programming
Spring 2011