<table>
<thead>
<tr>
<th>Question 1. (5 pts)</th>
<th>..........______________</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 2. (20 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 3. (15 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 4. (15 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 5. (10 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 6 (10 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 7 (15 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 8 (5 pts)</td>
<td>..........______________</td>
</tr>
<tr>
<td>Question 9 (5 pts)</td>
<td>..........______________</td>
</tr>
</tbody>
</table>

Make sure to write your initial in each page.
Question 1. (5 pts)

Are each of the following True or False?

- [T] Agglomerative hierarchical clustering (Bottom to top) involves $O(n)$ agglomerations.
- [F] The greedy solution to the Knapsack problem always finds the optimum.
- [T] K-means clustering computes new centroids locations at each iteration.
- [T] The greedy solutions to the knapsack problem is $O(n \log n)$.
- [F] Correlation always implies causation.
Question 2. (20 pts)

a) Write a function that returns the minimum of a list of numbers. (6 pts)
Do not use the built-in min function.

def min(L):
    '''assumes L is a list of numbers and returns the minimum'''
    minimum=L[0]
    for item in L:
        if item<minimum:
            minimum=item
    return minimum
b) Write a function that takes a list of numbers as input and returns the sum of the maximum of its negative numbers and the minimum of its non-negative numbers (including zero). Remember the maximum negative number is the negative number with the smallest absolute value.
For example, given the list [5, 2, 1, -2, -3] it should return |-2|+1=3.

- For this question you can use the min or max functions which take a list and return their minimum or maximum value respectively (but you don’t have to).
- Your function should raise a ValueError if the function contains only non-negative or only negative numbers or if the list is empty.

```python
def sumMinMax(L):
    '''assumes L is a list of numbers
    returns the sum of the absolute values of the minimum positive
    and maximum negative values'''
    if len(L)==0:
        raise ValueError('list empty')

    min_val=min(L)
    max_val=max(L)

    if min_val==0:
        raise ValueError('all non negative')
    if min_val*max_val>0:
        raise ValueError('same sign')

    max_negative=min_val
    min_positive=max_val

    for item in L:
        if item<0 and item>max_negative:
            max_negative=item
        if item>0 and item<min_positive:
            min_positive=item

    return abs(max_negative)+min_positive
```


Question 3. (15 pts)

Based on the Social Network code we provided you (and which we reproduce at the end of this exam), use a Monte-Carlo approach to compute the average degree of separation between any two people in the network. For example, if A is only friends with B and B is only friends with C, the degree of separation between A and C is 2. You can use any of the functions provided as part of the social network code.

def averageSeparation(numtrials):
    '''Return the average degree of separation between any two people in
    the network using numtrials trials of Monte-Carlo simulation.'''
    total = 0
    counter = 0
    for i in range(numtrials):
        s = sn.getRandPerson()
        d = sn.getRandPerson()
        track = BFS(s, d)
        if track != None:
            counter +=1
            degSep = track.dist(d)
            total+=degSep
    avg = total/float(counter)
    return avg

sn = SocialNetwork('finalGraph.in')
print averageSeparation(10)

Note: It’s possible for BFS to return None Type.
Question 4. (15 pts)

This question is also based on the social network code. Recall the shortest path code:

```python
def generalShortestPath(origin, destination, initialCollection, track):
    '''Return the set of nodes reachable from the origin'''
    pending = initialCollection  # collection of pending nodes
    visited = set({})           # set of visited nodes
    pending.add(origin)
    visited.add(origin)
    track.setOrigin(origin)
    while not pending.empty():
        nxt = pending.pop()
        for friend in nxt.getFriends():
            track.track(nxt, friend)
            if friend.getNode() == destination:
                return track
            if friend.getNode() not in visited:
                visited.add(friend.getNode())
                pending.add(friend.getNode())
    return None  # the destination is unreachable from the origin.
```

Change the code to find the shortest path while excluding friends whose name is ‘Kevin Bacon’

```python
def shortestPathExcludingOnePerson(origin, destination, initialCollection, track, excludeName='Kevin Bacon'):
    pending = initialCollection  # collection of pending nodes
    visited = set({})           # set of visited nodes
    pending.add(origin)
    visited.add(origin)
    track.setOrigin(origin)
    while not pending.empty():
        nxt = pending.pop()
        for friend in nxt.getFriends():
            track.track(nxt, friend)
            if friend.getNode() == destination:
                return track
            if friend.getNode() not in visited:
                visited.add(friend.getNode())
                pending.add(friend.getNode())
                if friend.getNode().getName() == excludeName:
                    visited.add(friend.getNode())
                else:
                    visited.add(friend.getNode())
        return None
```

Question 5. (10 pts)
Recall the code for the Priority class:

```python
class Priority(Collection):
    '''Priority Queue. When you use this you will get weighted shortest paths. the class needs access to the tracker in order to know the latest estimate of the shortest distance for a given node'''
    def __init__(self, track):
        assert isinstance(track, PathTrack)
        self.lst = []
        self.track = track
    def add(self, x):
        self.lst.append(x)
    def empty(self):
        return len(self.lst)==0
    def pop(self):
        ''' Walk over the pending nodes and look for the one with the shortest distance estimate to pop next.'''
        lst = self.lst
        midx = 0
        track = self.track
        mv = track.dist(lst[0])
        for i in xrange(len(lst)):
            if track.dist(lst[i]) < mv:
                midx = i
                mv = track.dist(lst[i])
        mnode = lst[midx]
        del lst[midx]
        return mnode
```

What happens if we try to run the following code:
```python
myPriority=Priority(3.14)
```

We will get an assertionError since 3.14 is a float, not an instance of PathTrack.

What is the complexity of Priority.pop?  O(N) where N is the length of the list (You can assume that lookup in a dictionary takes constant time)
Question 6. (10 pts)

In a groundbreaking scientific study, researchers at the University of Candyland tested the correlation between jellybeans of 20 different colors and acne. They divide their human subjects into 20 groups of 100 people each, where each group eats only one color of jelly beans. One month later, they record the percentage of subjects who have acne in each group and they find that while the average is 31%, the group of subjects who ate green jelly beans has a much higher percentage with acne, 42%.

Below we’ve written part of a Monte-Carlo simulation to determine the probability that this occurred by chance. Your goal is to complete the code.

```python
import random

PROB_ACNE = 0.31

def monteCarloTest(N):
    '''Compute the probability of getting the surprising result'''
    good = 0.0
    for x in xrange(0, N):
        if surprisingResult():
            good += 1.0
    return good / N

def surprisingResult():
    '''Simulate the experiment assuming that a person has a uniform probability PROB_ACNE of getting acne. Return true if the experiment supports the published result; i.e. if more than 42 people in some group have acne. Otherwise return false'''
    #Your code here

    #check for each group
    for i in xrange(num_groups):
        acne_count=0
        #perform trial for all subjects in the group
        for j in xrange(subjects_per_group):
            if random.random()<=PROB_ACNE:
                acne_count+=1
        #check if at least 1 group is above 42%
        if float(acne_count)/subjects_per_group > .42:
            return True
    #no group is above 42%
    return False

print 'Probability of getting the surprising result = ' + str(monteCarloTest(1000))
```
Question 7. (15 pts)

The syracuse series can be implemented as:

```python
def syracuse(n):
    if n==1:
        return 1
    if n%2==0:
        return syracuse(n/2)
    else:
        return syracuse(3*n+1)
```

Write a faster version called fastSyracuse by adding memoization.

```python
def fast_syracuse(n, memo={}):
    print n
    print memo
    if n==1:
        return 1
    if n%2==0:
        try:
            return memo[n/2]
        except KeyError:
            result=fast_syracuse(n/2,memo)
            memo[n/2]=result
            return result
    else:
        try:
            return memo[3*n+1]
        except KeyError:
            result=fast_syracuse(3*n+1,memo)
            memo[3*n+1]=result
            return result
```
Question 8. (5 pts)

```python
import pylab
import random as rr
xVals=[]
yVals=[]
for i in xrange(1000):
    x=rr.random()
    y=x*x+2
    xVals.append(x)
    yVals.append(y)
xVals=pylab.array(xVals)
yVals=pylab.array(yVals)
a, b, c, d=pylab.polyfit(xVals, yVals, 3)
print round(a), round(b), round(c), round(d)
```

What does this code print?

```
-0.0 1.0 -0.0 2.0
```
Question 9. (5 pts)

Link each concept on the left to the one on the right that is most related.
Page left intentionally blank in case you need more space.
Page left intentionally blank in case you need more space.
import random as rr

class DNPair:
    ''' Pair of node, distance'''
    def __init__(self, node, dist):
        self.dist = dist
        self.node = node
    def getDist(self):
        return self.dist
    def getNode(self):
        return self.node
    def __str__(self):
        return str(self.dist) + ":" + str(self.node)

class Person:
    '''Person is a node in the social graph. Each node has a
    list of edges corresponding to the person's friends. 
    Each edge is labeled with a distance representing the 
    closeness of the relationship. A small distance means 
    the relationship is very close.'''
    def __init__(self, name):
        self.name = name
        self.friends = []
    def getName(self):
        return self.name
    def addFriend(self, other, dist):
        assert isinstance(other, Person)
        self.friends.append(DNPair(other, dist))
    def getFriends(self):
        return self.friends
    def __str__(self):
        return self.name
class SocialNetwork:

def addPerson(self, name):
    if name not in self.personByName:
        p = Person(name)
        self.personByName[name] = p
        self.personList.append(p)

def __init__(self, filename):
    f = open(filename)
    self.personByName = {}
    self.personList = []
    personByName = self.personByName
    for line in f:
        x = line.split(',
        self.addPerson(x[0])
        self.addPerson(x[1])
        personByName[x[0]].addFriend(personByName[x[1]], float(x[2]))
        personByName[x[1]].addFriend(personByName[x[0]], float(x[2]))

def getRandPerson(self):
    return self.personList[rr.randint(0, len(self.personList)-1)]
class Collection:
    def add(self, x):
        pass
    def empty(self):
        pass
    def pop(self):
        pass

class Fifo(Collection):
    '''First-In-First-Out Queue. When you use this you will get Breath-first-search'''
    def __init__(self):
        self.lst = []
    def add(self, x):
        self.lst.append(x)
    def empty(self):
        return len(self.lst)==0
    def pop(self):
        return self.lst.pop(0)
class Priority(Collection):
    '''Priority Queue. When you use this you will get weighted shortest paths. The class needs access to the tracker in order to know the latest estimate of the shortest distance to a given node'''
    def __init__(self, track):
        assert isinstance(track, PathTrack)
        self.lst = []
        self.track = track
    def add(self, x):
        self.lst.append(x)
    def empty(self):
        return len(self.lst)==0
    def pop(self):
        ''' Walk over the pending nodes and look for the one with the shortest distance estimate to pop next.'''
        lst = self.lst
        midx = 0
        track = self.track
        mv = track.dist(lst[0])
        for i in xrange(len(lst)):
            if lst[i] < mv:
                midx = i
                mv = track.dist(lst[i])
        mnode = lst[midx]
        del lst[midx]
        return mnode
class PathTrack:
    ''' Class to keep track of the path to a given node and the distance from the source to the reachable nodes.'''
def __init__(self):
    self.D = {}  # Maps each visited node to a DNPair (pred, dist), where
                # pred is the predecessor of the node and dist is the
                # current estimate of the shortest distance to the node.
def setOrigin(self, o):
    self.D[o] = DNPair(o, 0.0)
def track(self, current, nxt):
    '''Record that from node current we are visiting node nxt.getNode(), which is a distance nxt.getDist() away.
    ...'''
    assert isinstance(nxt, DNPair)
    D = self.D
    if (nxt.getNode() not in D):
        dist = D[current].getDist() + 1.0
        D[nxt.getNode()] = DNPair(current, dist)
        return dist
    else:
        return D[nxt.getNode()].getDist()

def dist(self, node):
    return self.D[node].getDist()

def getPath(self, node):
    nn = node
    out = []
    while True:
        c = self.D[nn].getNode()
        out.append(nn)
        if c == nn:
            return out
        nn = c
class WPathTrack(PathTrack):
    ''' Special case of the PathTrack class that takes into account
    the weights for the path '''
    def track(self, current, nxt):
        '''Record that from node current we are visiting
        node nxt.getNode(), which is a distance nxt.getDist() away.
        '''
        assert isinstance(nxt, DNPair)
        D = self.D
        dist = nxt.getDist()
        # For Dijkstra's algorithm you may need to update the shortest distance
        # estimate and the predecessor as you discover new ways to reach a
        # node.
        if (nxt.getNode() not in D or \
           D[nxt.getNode()].getDist() > D[current].getDist() + dist):
            dist = D[current].getDist() + dist
            D[nxt.getNode()] = DNPair(current, dist)
            return dist
        else:
            return D[nxt.getNode()].getDist()

def BFS(origin, destination):
    return generalShortestPath(origin, destination, Fifo(), PathTrack())

def shortestPath(origin, destination):
    t = WPathTrack()
    q = Priority(t)
    return generalShortestPath(origin, destination, q, t)
def generalShortestPath(origin, destination, initialCollection, track):
    '''Return the set of nodes reachable from the origin'''
    pending = initialCollection  # collection of pending nodes
    visited = set({})  # set of visited nodes
    pending.add(origin)
    visited.add(origin)
    track.setOrigin(origin)
    while not pending.empty():
        nxt = pending.pop()
        for friend in nxt.getFriends():
            track.track(nxt, friend)
            if friend.getNode() == destination:
                return track
            if friend.getNode() not in visited:
                visited.add(friend.getNode())
                pending.add(friend.getNode())
    return None  # the destination is unreachable from the origin.

def avgSeparation(sn, N, pathSearch):
    '''Monte-carlo simulation to find the average degree of separation between people in the graph'''
    m = 0.0
    d = 0.0
    for i in xrange(N):
        src = sn.getRandPerson()
        dst = sn.getRandPerson()
        t = pathSearch(src, dst)
        if t != None:
            m += 1.0
            d += t.dist(dst)
    return d / m

def findAndPrintPath(src, dst):
    '''Find the shortest path between source and destination'''
    t = shortestPath(src, dst)
    if t != None:
        print 'Distance = ' + str(t.dist(dst))
        for e in t.getPath(dst):
            print e