Problem Set 4

Collaborated with: Steven Brotz, David Kawashima, Jessica Choi.
Used one (1) late token.

Problem 1

a. Calculating the stationary distribution using a system of equations and Wolfram, we get:

$$\pi = (\frac{7}{22}, \frac{7}{22}, \frac{4}{11})$$

If we take P^n as $n \to \infty$, we see that it approaches:

$$\begin{bmatrix} \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \\ \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \\ \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \end{bmatrix}$$

Thus, because each row in P^n for large n is the same, and we know that $\sum \pi(i) = 1$, and thus $\sum \pi_0(i) = 1$, for any starting π_0, we have that:

$$\lim_{n\to\infty} \pi_0 P^n = \pi = (\frac{7}{22}, \frac{7}{22}, \frac{4}{11})$$

Thus, π does not depend on π_0, meaning that our stationary distribution does not depend on our starting distribution at all. In other words, the stationary distribution represents the steady state/ equilibrium.

b. Calculating the stationary distribution using a system of equations and Wolfram, we get:

$$\pi = (\frac{7}{16}, \frac{1}{8}, \frac{1}{16}, \frac{3}{8})$$

If we take P^n as $n \to \infty$, we see that it approaches:

$$\begin{bmatrix} 0 & 0.25 & 0 & 0.75 \\ 0.875 & 0 & 0.125 & 0 \\ 0 & 0.25 & 0 & 0.75 \\ 0.875 & 0 & 0.125 & 0 \end{bmatrix}$$

From this, we can see that because not all of the rows are equivalent, that $\lim_{n\to\infty} \pi_0 P^n$ does not converge.

Problem 4

a.

```
# CommonFriends

function main(){
        # Define the input
        listOfFriendships = list of (Person, [List of Friends])

        # Run MapReduce on the list listOfFriendship
        listOfOutputs = MapReduce(listOfFriendship)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(listOfFriendships){

        # Map step:
        For each(key, value) pair in listOfFriendship, run Map(key, value)

        # Collector step:
        Collect all outputs from Map() in the form of (key, value)
                pairs in a new list called listOfMapOutputs
        Make a list of distinct keys in listOfMapOutputs
        For each unique key, concatenate the corresponding value lists

        For each such (key, listOfValues), run Reduce() on them.
        Concatenate the results from all calls to Reduce() into a
                list and return it.

}

function Map(person, listOfFriends){
        # key = person, values = listOfFriends for that person

        For each friend in listOfFriends{
                # Sort so that pairs of people get emitted in
                # lexicographical order
                EmitInteremediate(sorted(person, friend),
                        lstOfFriends)
                }
        }
}
```

```
function Reduce(pair, listValues){

        # For input ((A,B), [lst1, lst2]),
        # To get mutual friends, we simply
        # take the intersection of lst1
        # and lst2 to get mutual friends of
        # A and B

        mutual_friends = []
        lst1 = listValues[0]
        lst2 = listValues[1]
        for friend in lst1{
                if friend in lst2 {
                        add friend to mutual_friends
                }
        }
        return (pair, mutual_friends)
}
```

b.

```
# High school days

function main(){
        # Define the input
        listOfFilenames = list of filenames containing test scores
                of all students
        listKeyValues[] = list of pairs (filename, fileScores)
                where fileScores is a string containing the scores,
                separated by spaces, of the file with name 'filename'


        # Run MapReduce on the list listOfFilenames
        listOfOutputs = MapReduce(listKeyValues)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(listKeyValues){

        # Map step:
        For each(key, value) pair in listKeyValues, run Map(key, value)

        # Collector step:
```

```
        Collect all outputs from Map() in the form of (key, value)
        pairs in a new list called listOfMapOutputs.
        Make a list of distinct keys in listOfMapOutputs.
        For each unique key, create a list of the values
        to create (key, listOfValues) pairs.

        # Reduce step:
        For each such (key, listOfValues), run Reduce() on them.

        # Return the results
        Concatenate the results of all calls to Reduce() into a list.
        Sort this list in reverse (decreasing) order.
        Output this list.
}

function Map(filename, fileScores){
        # key = filename, values = scores in that filename

        # Assuming scores are separated by spaces
        score[] = Split fileScores by " ".

        for each score in scores {
                EmitIntermediate(score, 1)
        }
}

function Reduce(score, listValues){
        # key = word and value = list of 1's
        # no. of ones for each score is the
        # no. of occurrences of the word
        # in all files.
        return ([score * len(listValues)])

        # Return: list consisting of that score repeated
        # the number of times as the no. of occurrences
        # in all the files
}
```

c. The logic is as follows. Notice that if we take the unit circle centered at (0,0), it has an area equal to π. If we look at the square with corners at (-1,-1), (-1, 1), (1, -1), and (1, 1), such a square has side length = 2, so this square will have an area of 4. If we generate two numbers from our random generator (which generates numbers from [-1, +1]) and treat those two numbers as (x, y) coordinates, we are guaranteed that this point will lie in the square described above. Now to calculate π, we look at the proportion of points that not only lie in the square, but also lie in the circle. Then, if we divide the number of points that lie in the circle by the total number of points generated, it should come out to around $\frac{\pi}{4}$. Thus, if we multiply this proportion of points that lie in the circle by 4, we should be able to approximate π.

```
# Good old pi

function main(){
        # Define the input
        # Generate n (in this case we just say 1000)pairs of points
        let n = 1000
        lstOfPoints = []
        for i from 1 to n{
                let x = rand()
                let y = rand()
                Add (x, y) to lstOfPoints
        }

        # Run MapReduce on this list of points
        listOfOutputs = MapReduce(lstOfPoints)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(lstOfPoints){

        # Map step:
        For each point (x, y) in lstOfPoints, run Map((x,y))

        # Collector step:
        Collect all outputs from Map() in the form of (value)
        and concatenate all values into a new list called
        listOfMapOutputs.

        # Taking the sum of listOfMapOutputs gives us the
        # number of points lying in the circle. Dividing
        # this by n gives us proportion of all points that
        # lie within the circle

        let sum = listOfMapOutputs
        let proportion = sum / n

        # Multiply by 4 because as explained before,
```

```
        # the total square area is 4.

        let pi_approx = proportion * 4

        return pi_approx
}

function Map(point){
        # input = a single (x,y) point where
        # x and y are both between -1 and 1

        # Test if in circle. If in circle, we
        # return 1, else return 0

        If x^2 + y^2 <= 1 {
                EmitIntermediate(1)
        }
        else
        {
                EmitIntermediate(0)
        }

}
```

d.

```
# GaugeTheDistance

function main(){
        G[] = Adjacency list of the graph.
        # G[i] is a list of (neighbors, distance) tuple of node i.
        n = length(G)
        dist[] = list that will contain the distances
        from node 1 eventually. Initialize it arbitrarily.
        distUpdated[] = list that will contain distances from
        node 1 after each run of MapReduce in the
        following loop. Initialize it with $(0, \infty, \infty, \ldots, \inft

        while( NOT stoppingCriterion(dist, distUpdated)){
                dist = distUpdated
                inputs = ((i, dist[i]), G[i]) for all i
                distUpdated = MapReduce(inputs).
        }
        print the list dist[].
}
function stoppingCriterion(dist1, dist2){
```

```
        return dist1 == dist2
}

function Map((i, dist[i]), G[i]){
        # Input is a node and its distance from node 1
        # and its adjacency list. We output a tuple
        # containing (node, distance to i) based on its
        # connection/distance to the input node

        EmitIntermediate(i, dist[i])
        if (dist[i] != infty)
        {
                for each (neighbor, distance) in dist[i]
                {
                        EmitIntermediate(neighbor, distance + dist[i])
                }
        }
}

function Reduce(i, distancesFromi){
        # each node will have a list of distances,
        # take the minimum one
        return (i, min(distancesFromi))
}
```