Nolan Gutierrez
1001033225
CSE 6369 - Reinforcement Learning
Homework 1- Spring 2020
Due Date: Feb. 11 2020

1a)

Let the U_k (S_e,S_c) be the value of taking placing the elevator on floor F with the call floor being S_c and the exit floor being S_e. F represents the floor that the elevator is at for iteration k. Since the elevator has to pick the person up, take the person to his/her floor, and wait for that person to get in and out of the elevator, then U_k is equal to -7 * (|F-S_c| + |S_c - S_e| + 1). Seven is multiplied by the sum because it takes 7 seconds for the elevator to go up or down a floor, and the time it takes to get in and out of an elevator is equal to the time it takes for the elevator to move one floor. If we want to find the expected utility of taking some action F, then we can calculate that as

$$Q_k = \sum_{S_c} \sum_{S_e} U_k(S_c, S_e) \cdot P(S_c, S_e)$$

This applies because the expected value of a function of two random variables is equal to the double summation of the products of the function and the joint distribution of the random variables which are being summed over.

1b)

This n-armed bandit problem has a state, an action, a reward function, an environment, and a transition function for the states. The environment in this situation provides the new floor for which the agent is in to the agent. For this problem the states are simply the floor which the elevator is on. This means that the set of all states S = [NumberFloors] which means that the states are {1,2, … Number of Floors}. The state space can be simplified to this because whether or not a person is calling the elevator or exiting on a certain floor is irrelevant for the learning agent. The elevator's chosen floor is fixed before the start of each iteration, and the agent can not change its environment in between states.

The set of possible actions or levers which the bandit can take is similar. The set of Actions A = [Number of Floors]. The agent can choose its state at the beginning of each iteration. What happens in between iterations is out of the agent's control.

For n-armed bandit problems the probability of the agent transitioning to the chosen floor given the lever pulled (action) is one. At each iteration a person calls the elevator from floor S_c according to a probability distribution P(S_c). A person who calls from S_c wants to exit on a floor S_e according to a probability distribution P(S_e|S_c). The probability of a person calling from floor S_c while wanting to exit on S_e is P(S_c,S_e).

For this problem the learning agent uses a simple incremental averaging solution. The increment of the expected value Q_{k+1} of choosing a floor is as follows:

$Q_{k+1} = Q_k + (1/k) * (R_k - Q_k)$, where R_k is the kth reward given as a consequence of placing the elevator on a certain floor. The reward for this problem is given in 1 and 2a)..
In order to explore the action space sufficiently, epsilon-greedy action selection is applied where a small percentage of the time a random action is selected, and the best action is selected otherwise. The epsilon used for this problem was set at a high 0.5 since the learning agent seemed to not explore the action space sufficiently with the normal of 0.1.

1c)

i)

S_c	$P(S_c)$
1	1
2	0
3	0
4	0
5	0
6	0

S_e	$P(S_e \mid S_c = 1)$
1	0
2	1/5
3	1/5
4	1/5
5	1/5
6	1/5

let a = S_c, b = S_e

Q(a) = $\sum_b -7(|F - 1| + |1 - b| + 1)P(a, b)$

F = 1	$-7(\|F\ -\ 1\|\ +\ \|\ 1-b\ \|\ +\ 1)\ P(a\ =\ 1,b)$
S_e	$Q(S_e)$
2	-7(0 + 2) * 1/5
3	-7(0 + 3)* 1/5
4	-7(0 + 4)* 1/5
5	-7(0 + 5)* 1/5
6	-7(0 + 6)* 1/5
Total	-28

Q(F) = -28

Each Q value decreases by -7 leading to the following table:

Floor	Q(Floor) = $\sum_b -7(\|F\ -\ 1\|\ +\ \|\ 1-b\ \|\ +\ 1)P(a\ =\ 1,b)$
1	-28
2	-35
3	-42
4	-49
5	-56
6	-63

For this problem the floor with the highest expected utility is floor 2 at an expected utility of -28.

ii)

S_c	$P(S_c)$	$P(S_e = 1 \mid S_c)$
1	0	1
2	1/5	1
3	1/5	1

4	1/5	1
5	1/5	1
6	1/5	1

let a = S_c, b = S_e

Q(a) = $\sum_{a} -7(|F - a| + |a - 1| + 1)P(a, b = 1)$

a = 1	
S_c	$-7(\|F - a\| + \|a - 1\| + 1)\ P(a, b = 1)$
2	$-7(\|1 - 2\| + \|2 - 1\| + 1)\ 1/5$
3	$-7(\|1 - 3\| + \|3 - 1\| + 1)\ 1/5$
4	$-7(\|1 - 4\| + \|4 - 1\| + 1)\ 1/5$
5	$-7(\|1 - 5\| + \|5 - 1\| + 1)\ 1/5$
6	$-7(\|1 - 6\| + \|6 - 1\| + 1)\ 1/5$
Total	-67.2

a = 2	
S_c	$-7(\|F - a\| + \|a - 1\| + 1)\ P(a, b = 1)$
2	$-7(\|2 - 2\| + \|2 - 1\| + 1)\ 1/5$
3	$-7(\|2 - 3\| + \|3 - 1\| + 1)\ 1/5$
4	$-7(\|2 - 2\| + \|2 - 1\| + 1)\ 1/5$
5	$-7(\|2 - 2\| + \|2 - 1\| + 1)\ 1/5$
6	$-7(\|2 - 2\| + \|2 - 1\| + 1)\ 1/5$
Total	-42

a = 3	
S_c	$-7(\lvert F - a\rvert + \lvert a - 1\rvert + 1)\, P(a, b = 1)$
2	$-7(\lvert 3 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
3	$-7(\lvert 3 - 3\rvert + \lvert 3 - 1\rvert + 1)\ 1/5$
4	$-7(\lvert 3 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
5	$-7(\lvert 3 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
6	$-7(\lvert 3 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
Total	-37.8

a = 4	
S_c	$-7(\lvert F - a\rvert + \lvert a - 1\rvert + 1)\, P(a, b = 1)$
2	$-7(\lvert 4 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
3	$-7(\lvert 4 - 3\rvert + \lvert 3 - 1\rvert + 1)\ 1/5$
4	$-7(\lvert 4 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
5	$-7(\lvert 4 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
6	$-7(\lvert 4 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
Total	-42

a = 5	
S_c	$-7(\lvert F - a\rvert + \lvert a - 1\rvert + 1)\, P(a, b = 1)$
2	$-7(\lvert 5 - 2\rvert + \lvert 2 - 1\rvert + 1)\ 1/5$
3	$-7(\lvert 5 - 3\rvert + \lvert 3 - 1\rvert + 1)\ 1/5$

4	$-7(\|5-2\|+\|2-1\|+1)\ 1/5$
5	$-7(\|5-2\|+\|2-1\|+1)\ 1/5$
6	$-7(\|5-2\|+\|2-1\|+1)\ 1/5$
Total	-37.8

a = 6	
S_c	$-7(\|F-a\|+\|a-1\|+1)\ P(a,b=1)$
2	$-7(\|6-2\|+\|2-1\|+1)\ 1/5$
3	$-7(\|6-3\|+\|3-1\|+1)\ 1/5$
4	$-7(\|6-2\|+\|2-1\|+1)\ 1/5$
5	$-7(\|6-2\|+\|2-1\|+1)\ 1/5$
6	$-7(\|6-2\|+\|2-1\|+1)\ 1/5$
Total	-42

Floor = F	Q(F) = $\sum_a(\|F-a\|+\|a-1\|+1)P(a,b=1)$
1	-67.2
2	-42
3	-37.8
4	-42
5	-37.8
6	-42

For this problem the floors with the highest expected utility are floors 3 and 5. They each have an expected utility of -37.8.

iii)

S_c	$P(S_c$
1	0
2	.6
3	.1
4	.1
5	.1
6	.1

S_c	$P(S_e \mid S_c)$					
	S_e					
	1	2	3	4	5	6
1	0	0	0	0	0	0
2	6/10	0	1/10	1/10	1/10	1/10
3	1	0	0	0	0	0
4	1	0	0	0	0	0
5	1	0	0	0	0	0
6	1	0	0	0	0	0

S_c	$P(S_e , S_c)$					
	S_e					
	1	2	3	4	5	6
1	0	0	0	0	0	0
2	36/100	0	1/10	1/10	1/10	1/10
3	1	0	0	0	0	0
4	1	0	0	0	0	0
5	1	0	0	0	0	0

6	1	0	0	0	0	0

$\lvert S_e - S_c \rvert$						
	S_e					
		1	2	3	4	5	6
S_c	1	0	0	0	0	0	0
	2	1	0	1	2	3	4
	3	2	0	0	0	0	0
	4	3	0	0	0	0	0
	5	4	0	0	0	0	0
	6	5	0	0	0	0	0

let a = S_c, b = S_e

$-7 * (\sum_a \sum_b |a-b|P(a,b) + 1)$ = -8.7

This value stays constant for each action. It can be added back in after finding

$-7 \sum_a \sum_b (|F - a|P(a,b)$

Floor = F	$-7 \sum_a \sum_b (\lvert F - a \rvert P(a,b)$
1	-7(36/100 + 24/100 + 14/100)= -14
2	-7 * (10 * 1/10) = -7
3	-7 * (24/100 + 36/100 + 6/10) = -8.4
4	-7 * (72/100 + 48/100 + 4/10) -11.2
5	-7 * (108/100 + 72/100 + 4/10) = -15.4
6	-7 * (96/100 + 36/100 + 60/100) = -13.44

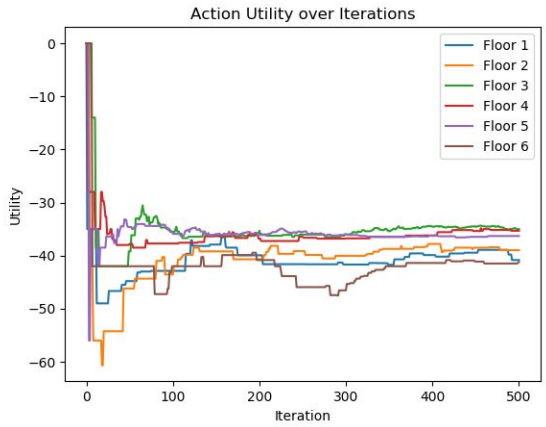
Now -8.7 is added to each entry of the above table to get the final expected utilities of placing the elevator on each floor.

Floor = F	$-7\sum_{a}\sum_{b}(\lvert F - a\rvert + \lvert a - b\rvert + 1)P(a,b)$
1	-22.7
2	-15.7
3	-17.1
4	-18.9
5	-24.1
6	-22.14

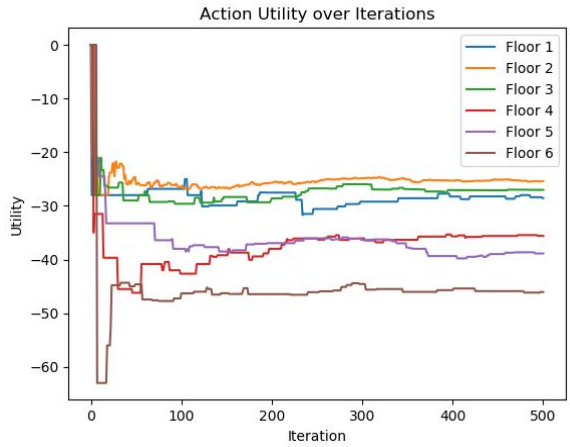
From this chart, using the distribution described in iii), the optimal floor is floor 2 with an expected utility of -15.7.

1d)

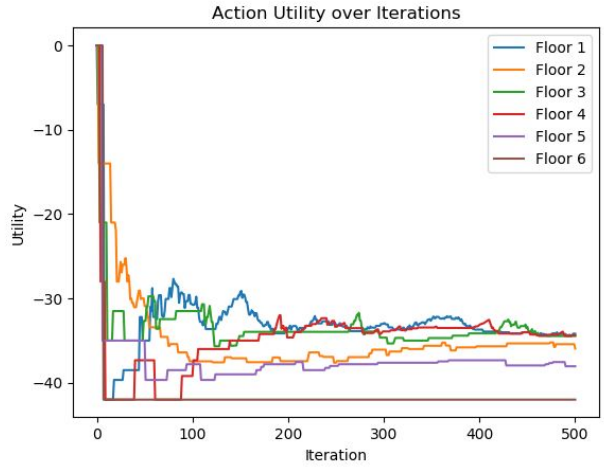
The following graph shows the utility over time using epsilon greedy action selection. Floors 3 and 4 usually have around the same utility value for this problem. With much higher iterations of around 10000 floor 3 usually comes slightly on top.



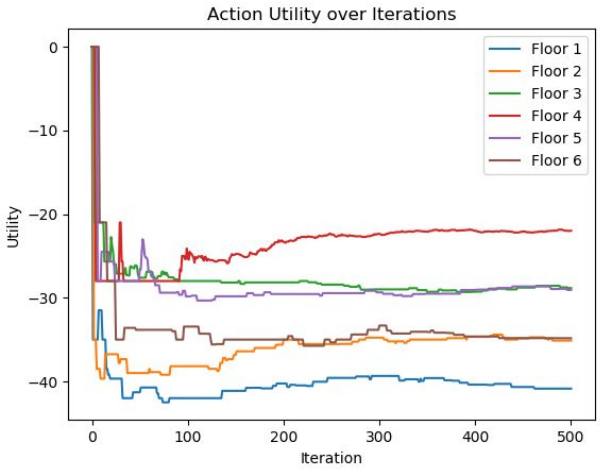
1e) Call-floors follow Poisson distribution with mean = 2. The exitFloors also follow a Poisson distribution with mean = 2.



The next graph shows the agent learning the distribution where call-floors follow a Gaussian with a mean 3 and a standard deviation of 3. The values are clipped and converted to integers to obtain the action. The exit-floors for this distribution are set to 1. For this problem more iterations were needed to find the correct floor. This represents the situation where the majority of people call from the middle floors, and they all exit on the first floor.



The next experiment will have the call-floors set to the fourth floor, but the exit floors follow a Poisson distribution where the bulk of exits are on the second floor. As expected the agent learns to park at the fourth floor.



2a)

Let Q_k be the expected utility of some floor F at iteration k . If the penalty is quadratically related to the reward, then one possible reward function would be

$$Q_k = \sum_{S_c}\sum_{S_e} U_{k,q}(S_c, S_e) \cdot P(S_c, S_e)$$

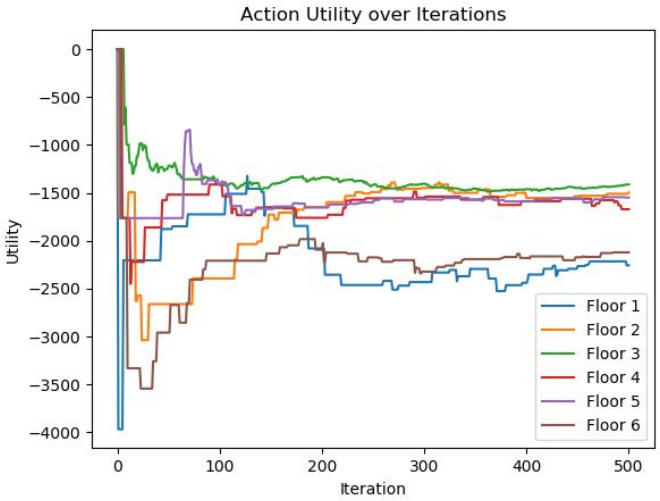
For this equation, $U_{k,q}(S_c, S_e)$ is the utility which has a penalty quadratically related to the waiting time. I chose $U_{k,q}(S_c, S_e)$ to be related to $U_k(S_c, S_e)$ by the following equation: $U_{k,q}(S_c, S_e) = -U_k(S_c, S_e)^2$.

This reward function will optionally be used in the implementation for this assignment.

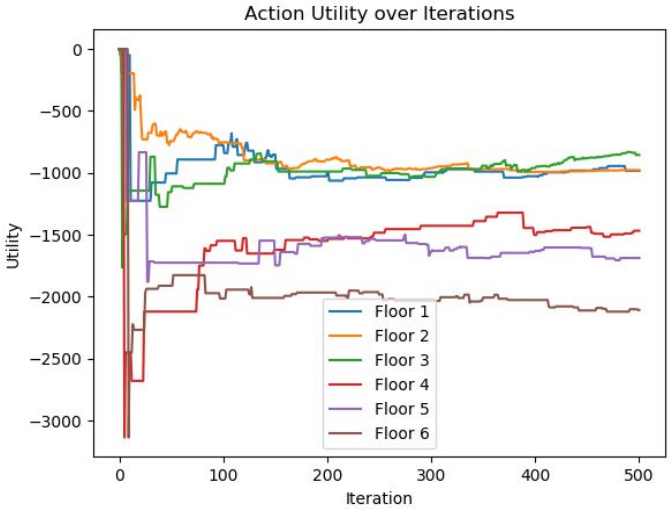
2b)

As described in the assignment each of the experiments for the distributions seen in 1d-e) are repeated below with the reward function from 2a).

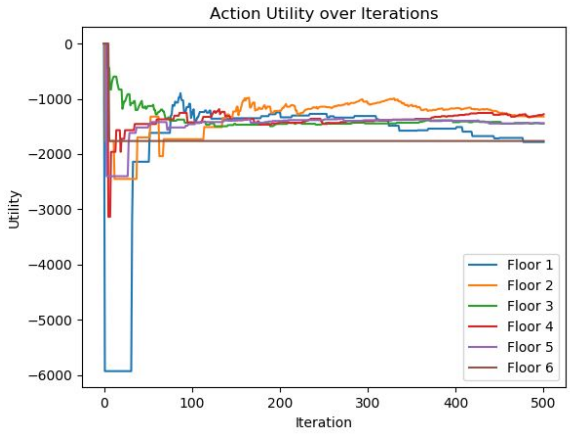
The following shows the results from the first experiment where the call-floors follow a uniform distribution and the exit floors follow the distribution as described in the problem. This experiment shows that the quadratic relation makes it much worse to travel longer distances. The utility of each of the floors in the experiment from 1d) were relatively similar. In this instance we can see the agent prefers much more to park on a floor in the middle of the building. For both situations floor 3 remains to be one of the optimal actions after 500 iterations.



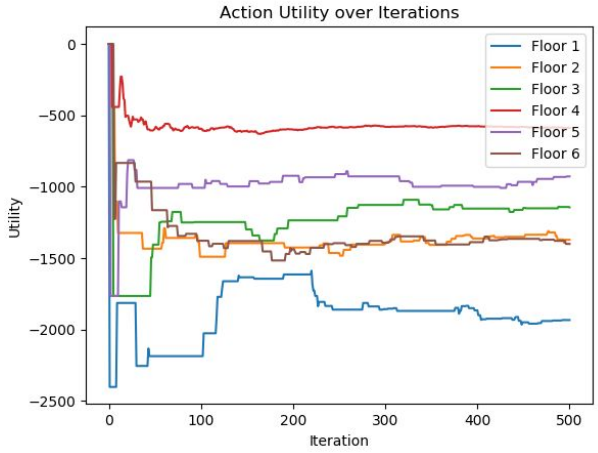
The next experiment has both the call-floors and the exit-floors following a Poisson distribution where both the exits and calls are situated near the lower levels. As with the first experiment with the penalty from 2a, this agent learns that it is much worse to park on floors 4,5, and 6. As before the optimal floor remains to be either floor 2 or 3. In this situation there probably needs to be more iterations to confirm what the agent decides the optimal floor to be.



The next graph shows the results from an experiment where the penalty is quadratically related to the wait time. For this experiment the call-floors follow a gaussian with mean 3 and standard deviation 3. People always exit on the first floor. The results compared to the previous experiment are extremely similar. The expected utility of each of the middle actions is too similar to draw a conclusion on which is the best action. This might show that with the gaussian's mean of 3, each of the floors appears to be equally bad except floor 6. There also shows to be little separation as with the first two experiments.



The final experiment sets the callFloor to 4 while having the exit-floors follow a Poisson distribution, where the mean of the Poisson distribution is 2. This experiment uses a squared penalty. As expected the agent still found that it is more advantageous to park at the fourth floor for maximum expected utility. The experiment shows that each of the floors maintain approximately the same ordering in regards to the expected utility of parking the elevator at each floor. As shown in the graph we can see that the agent learns it is much worse to park at floors farther away from floor 4 since the amount of penalty received is increasingly worse with the waiting time associated with more travel.



Implementation:

Debugger.py

```
class Debugger:
    def __init__(self, debugBool = True):
        self.debug = debugBool
    def gIS(self):
        return 'g =  d.getDebugString'
    def getDebugString(self, expression):
```

```
        #takes a string to be evaluated by exec function
        return 'if d.gDS() == True: print(\'' + expression + ':\','+  expression+')'
    def gDS(self):
        return self.debug
    def sDS(self, db):
        if db is not None:
            self.debug = db
        else : self.debug = True if not self.debug else False

#d,e = Debugger(), exec
#g = d.getDebugString
#exec('if 0 == 0: print ("hi")')
#e(g('1+2'))
```

Elevator.py

```
from Environment import Environment as env
from numpy import random
from Counter import Counter
class ElevatorAgent:
    def __init__(self, floor = 1, epsilon = 0.1, greedy = False):
        self.floor = floor
        self.actionValues = Counter()
        self.actionNumRewards = {}
        self.epsilon = epsilon
        self.greedy = greedy
    def setFloor(self,floor):
        self.floor = floor
    def getFloor(self):
        return self.floor
    def moveUP(self):
        self.floor += 1
    def moveDown(self):
        self.floor -= 1
    def getQValues(self):
        return self.actionValues.getDict().copy()
    def initQ(self, actions):
        for i in actions:
            self.actionValues.keyValue(i,0)
        for i in actions:
            self.actionNumRewards[i] = 0
    def pAF(self):
        print('actionValues: ', self.actionValues.getDict())
        print('actionNumRewards: ', self.actionNumRewards)
```

```
    def getBestAction(self):
        #This function returns the best action based on
        #current values
        return self.actionValues.argmax()
    def getAction(self):
        if self.greedy:
            bestAction,_ = self.getBestAction()
            return bestAction
        else:
            p = random.uniform(0,1)
            if p > self.epsilon:
                bestAction,_ = self.getBestAction()
                return bestAction
            else: return random.choice(list(self.actionValues.getKeys()))
    def takeAction(self):
        self.setFloor(self.getAction())
    def updateActionValue(self,
            a, #action
            r, #reward
            step = None, #stepsize
            ):
        Q = self.actionValues
        # increment reward number
        self.actionNumRewards[a] +=1
        k = self.actionNumRewards[a]
        l= 1/k if not step else step
        #incremental difference formula
        Q.keyValue(a, Q.at(a) + l * (r - Q.at(a)))
```

Environment.py

```
import math
from Debugger import Debugger
from Probability import ElevatorProb as EP
e,d = exec, Debugger()
e(d.gIS())
class Environment:
    def __init__(self,
            elevatorAgent,
            ep,
            numFloors = 6,
            episodeLength = 100,
            timeStep = 7,
            step = None,
```

```
        squaredPenalty = False,

        ):
    # Takes a probability distribution for the call floors
    # and exit floors.
    self.step = step
    self.squaredPenalty = squaredPenalty
    self.a = elevatorAgent
    self.numFloors = numFloors
    self.callBool = False
    self.exitBool = False
    self.callFloor = None
    self.exitFloor = None
    self.justFinished = False
    self.Time = 0
    self.T = episodeLength
    self.timeStep = 7
    self.startOfNextEp = episodeLength
    self.episodeNumber = 1
    self.a.initQ(self.getArms())
    self.ep = ep
def pF(self):
    print('callFloor: ', self.callFloor)
    print('exitFloor: ', self.exitFloor)
    print('callBool: ', self.callBool)
    print('exitBOol: ', self.exitBool)
    print('justFinished: ', self.justFinished)
    print('agentFloor: ', self.a.getFloor())
    print('')
def pT(self):
    print('Time: ' , self.Time)
    print('timeStep: ', self.timeStep)
    print('startOfNextEp: ', self.startOfNextEp)
    print('episodeNumber: ', self.episodeNumber)
    print('')
def pQV(self):
    print('actionValues: ', self.a.getQValues())
    print('bestAction: ', self.a.getBestAction())
    print('')

def setFloorInfo(self,
        callFloor,
        exitFloor,
```

```
        callBool,
        exitBool,
        justFinished,
        ):
    self.callFloor = callFloor
    self.exitFloor = exitFloor
    self.callBool = callBool
    self.exitBool = exitBool
    self.justFinished = justFinished
def getTimeToFinish(self, a, c,e):
    # takes a floor, calling floor and an exit floor
    return -7 * (abs(a-c) + abs(c - e) + 1)
def getUtility(self, a,c,e):
    T = self.getTimeToFinish(a,c,e)
    if self.squaredPenalty:
        return -(T ** 2)
    else: return T

def getArms(self):
    #returns a list of actions
    return [i for i in range(1,self.numFloors + 1)]
def moveElevator(self):
    # This function moves the elevator up if the calling
    # elevator is above and down otherwise. It acts similarly
    # for the exitFloor.

    a = self.a
    if self.callBool:
        if self.callFloor > a.getFloor(): a.moveUP()
        elif self.callFloor < a.getFloor(): a.moveDown()
        if a.getFloor() == self.callFloor:
            self.callBool = False
            self.exitBool = True
    elif self.exitBool:
        if self.exitFloor > a.getFloor():
            a.moveUP()
        elif self.exitFloor < a.getFloor():
            a.moveDown()
        if a.getFloor() == self.exitFloor:
            self.exitBool = False
            self.justFinished = True
def updateTime(self):
    if self.Time >= self.startOfNextEp:
```

```
            self.justFinished = True
        if self.justFinished:
            self.Time = self.startOfNextEp
            self.startOfNextEp += self.T
            self.episodeNumber += 1
            self.justFinished = False
        else: self.Time += self.timeStep

    def getPerson(self):
        # This function updates the action value based on the
        # reward obtained from picking up a person on callFloor
        # and dropping that person off at exit floor
        # This function should only be called when callfloor
        # and exit floor have been set.

        #get reward
        reward = self.getUtility(self.a.getFloor(),self.callFloor,self.exitFloor)
        #update time by length of episode
        self.Time += self.timeStep
        #update episode num
        self.episodeNumber +=1
        #update q values
        self.a.updateActionValue(self.a.getFloor(), reward, self.step)
        #sets floor for next round
        self.a.takeAction()
    def update( self):
        self.callFloor = self.ep.sampleCall()
        self.exitFloor = self.ep.sampleExit(self.callFloor)
        self.getPerson()
```

Plotter.py

```
import matplotlib.pyplot as plt
class Plot:
    def __init__(self,numFloors = 6):
        self.numFloors = numFloors
        self.floors = [ i for i in range(1,numFloors + 1)]
    def getFloors(self):
        # returns list of floor strings to be used for plotting
```

```
        myFloors = []
        for i in range(1,self.numFloors + 1):
            myFloors.append('Floor ' + str(i))
        return myFloors
    def getListsForPlot(self,myList):
        floorsValues = {}
        for i in range(1,self.numFloors + 1):
            # {1:[], 2:[], ... 6:[]}
            floorsValues[i] = []
        for i in range(len(myList)):
            for j in range(1,self.numFloors + 1):
                floorsValues[j].append(myList[i][j])
        return floorsValues
        # Floorvalues should be 6 lists of values. One for each floor.
        # Each of these will be plotted for against the number of iterations
        # which in this case will be 500
    def plotList(self,myList):
        totalTime = len(myList) # 500 default
        floorValues = self.getListsForPlot(myList)
        floorNames = self.getFloors()
        lineHs = {}
        for i in self.floors:
            lineHs[i] = plt.plot(floorValues[i], label = floorNames[i -1])
        plt.xlabel('Iteration')
        plt.ylabel('Utility')
        plt.title('Action Utility over Iterations')
        plt.legend()
        plt.show()
```

Probability.py

```
from numpy import random as r
from Counter import Counter
import numpy as np
class ElevatorProb:
    def __init__(self,
            numFloors = 6,
            callUniform = True,
            exitUniform = False,
            exitGaussian = False,
            callGaussian = False,
            exitPoisson = False,
            callPoisson = False,
```

```
        callFloor = None,
        exitFloor = None
        ):
    self.numFloors = numFloors
    self.callUniform = callUniform
    self.callGaussian = callGaussian
    self.exitUniform = exitUniform
    self.exitGaussian = exitGaussian
    self.callPoisson = callPoisson
    self.exitPoisson = exitPoisson
    self.callFloor = callFloor
    self.exitFloor = exitFloor
    self.arms = [i for i in range(1,numFloors + 1)]
def sampleCall(self, weights = None):
    # This function samples from a prior distribution
    # with a uniform distribution or the weights provided
    # by the user.
    if self.callUniform: return r.choice(self.arms)
    elif self.callGaussian: return self.getGaussianAction()
    elif self.callPoisson: return self.getPoissonAction()
    if self.callFloor is not None: return self.callFloor
    else: return r.choice(self.arms,p = weights)
def getProporCounter(self,callFloor):
    #This function allocates probabilities to each exit
    # floor conditioned on the callFloor based on what was
    # specified in 1d
    counter = Counter()
    counter.initKeys(self.arms)
    value = 0
    key = callFloor
    for i in range(1,callFloor +1):
        counter.keyValue(key,value)
        key -=1
        value +=1
    for i in range(callFloor + 1, self.numFloors + 1):
        counter.keyValue(i,5)
    return counter
def getGaussianAction(self, mean = 3, sigma = 3):
    actionFloat = r.normal(mean,sigma,(1,))
    actionClip = np.clip(actionFloat,1,self.numFloors)
    actionInt = int(actionClip)
    return actionInt
def getPoissonAction( self, mean = 2):
```

```
        action = r.poisson(mean)
        actionClip = np.clip(action,1,self.numFloors)
        return action

    def sampleExit(self,callFloor):
        #This function uses the probability distribution
        #provided by 1d for sampling from an exit given
        # what the call floor is. This function can calculate
        # the probability of an exit floor given what the call
        # floor is, and then sample accordingly.

        #The function works as follows: if a person calls from
        # floor 3 then the following table represents the
        #probability of exiting from each floor:
        # {1:2, 2: 1, 3: 0, 4: 5,5: 5, 6:5}
        # We can normalize to find the probability of
        if self.exitUniform: return r.choice(self.arms)
        if self.exitGaussian: return self.getGaussianAction()
        if self.exitPoisson: return self.getPoissonAction()
        if self.exitFloor is not None: return self.exitFloor
        exitCounter = self.getProporCounter(callFloor)
        exitCounter = exitCounter.normalize(copy = False)
        actions, probabilities = exitCounter.getItemsAsLists()
        return r.choice(actions,p =probabilities)
```

Simulator.py

```
from Elevator import ElevatorAgent
from Environment import Environment
from Plotter import Plot
class ElevatorSim:
    def __init__(self,
            agent,
            ep,
            env,
            step = None,
            numFloors = 6 ):
        self.step = step
        self.agent = agent
        self.env = env
        self.QHis = [self.agent.getQValues()]
        #saves preliminary history of 0
    def getQHis(self):
```

```
        return self.QHis
    def runForIterations(self, numIterations, plot = True):
        for i in range(numIterations):
            self.env.update()
            self.QHis.append(self.agent.getQValues())
        if plot:
             plt = Plot()
             plt.plotList(self.QHis)
```

test.py

```
from Environment import Environment as E
from Elevator import ElevatorAgent as A
from Simulator import ElevatorSim
from Probability import ElevatorProb
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--epsilon', type = float,default = 0.1)
parser.add_argument('--greedy', type = bool, default = False)
parser.add_argument('--squaredPenalty', type = bool, default = False)
parser.add_argument('--plot', type = bool, default = True)
parser.add_argument('--iterations', type = int, default = 500)

parser.add_argument('--exitGaussian', type = bool, default = False)

parser.add_argument('--callGaussian', type = bool, default = False)
parser.add_argument('--callUniform', type = bool, default = False)
parser.add_argument('--exitUniform', type = bool, default = False)
parser.add_argument('--exitPoisson', type = bool, default = False)
parser.add_argument('--callPoisson', type = bool, default = False)
parser.add_argument('--callFloor', type = int, default = None)
parser.add_argument('--exitFloor', type = int, default = None)
args = parser.parse_args()
def main():
    agent = A(epsilon = args.epsilon, greedy = args.greedy)
    ep = ElevatorProb(callUniform = args.callUniform,
            exitUniform =args.exitUniform,
            exitGaussian = args.exitGaussian,
            callGaussian = args.callGaussian,
            exitPoisson = args.exitPoisson,
            callPoisson = args.callPoisson,
            callFloor = args.callFloor,
            exitFloor = args.exitFloor)
```

```
    env = E(agent,ep,squaredPenalty = args.squaredPenalty)
    sim = ElevatorSim(agent, ep,env)
    sim.runForIterations(args.iterations, plot = args.plot)
if __name__ == "__main__":
    main()
```