山东大学

崇新学堂

2022－2023 学年第一学期

实 验 报 告

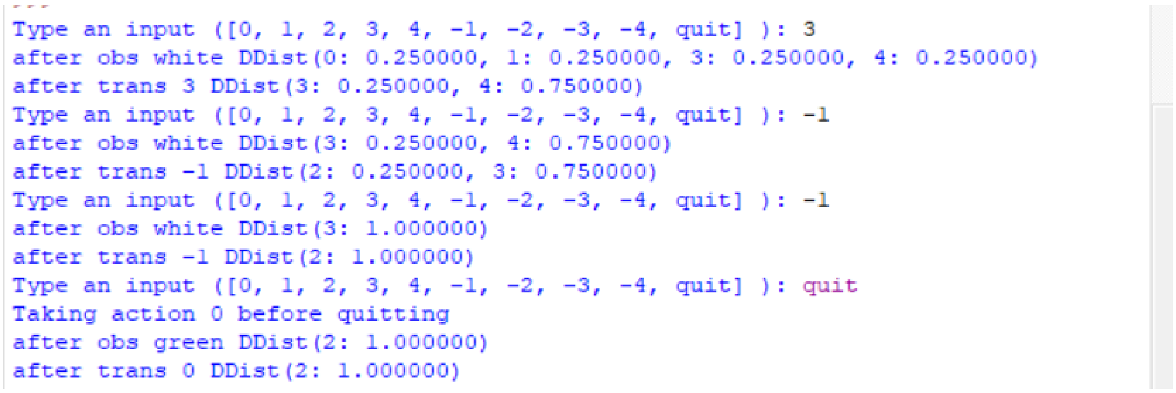
课程名称：	EECS DesignLab
实验名称：	DesignLab11

专 业 班 级	崇新 21
学 生 姓 名	施政 刘浩 张原 池弋
实 验 时 间	2022.12.1

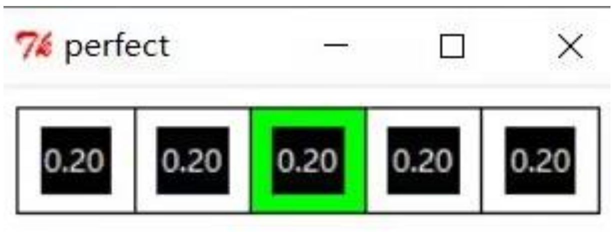
Step1

Check yourselves1

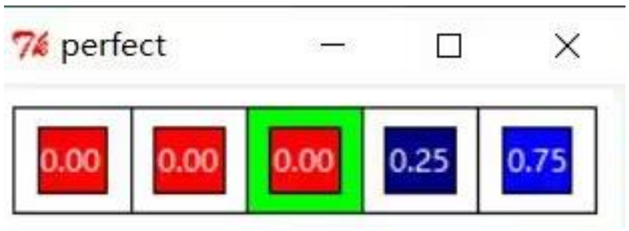
In this step, we tested the **Perfect**. The results are as follows：



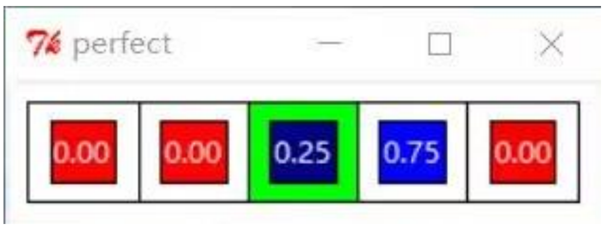
```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 3
after obs white DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
after trans 3 DDist(3: 0.250000, 4: 0.750000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -1
after obs white DDist(3: 0.250000, 4: 0.750000)
after trans -1 DDist(2: 0.250000, 3: 0.750000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -1
after obs white DDist(3: 1.000000)
after trans -1 DDist(2: 1.000000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): quit
Taking action 0 before quitting
after obs green DDist(2: 1.000000)
after trans 0 DDist(2: 1.000000)
```



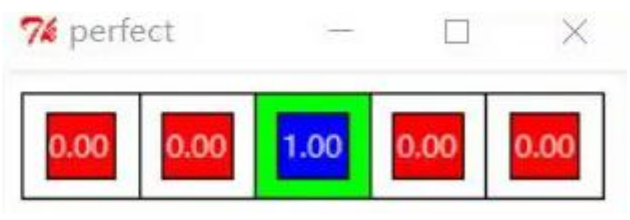
Start State



Input 3



Input -1



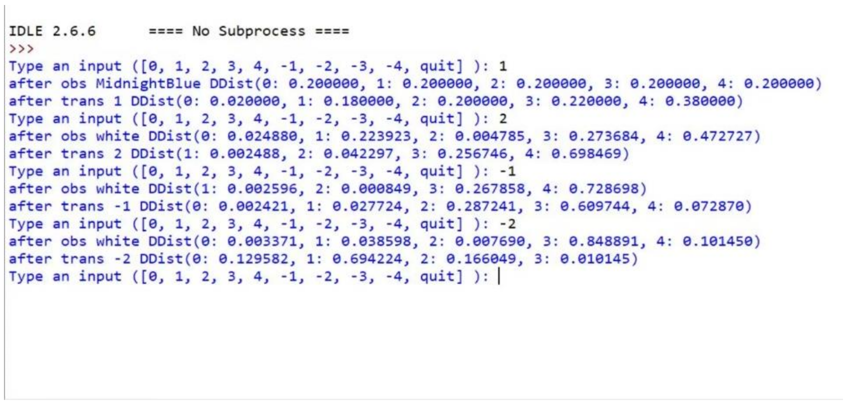
Input -1

It can be found that after inputting a number, an observation is generated first, and then the original state is modified by this observation, and the robot moves after modification, Finally, it returns a probability value in Perfect。For example, originally, the probability of all rooms is 0.2， after input -3， The robot first observes the color of the room and obtains white, under this condition, the probability of all rooms except the green room becomes 0.25. After moving three rooms to the right, the probability of the rightmost room is 0.75, the probability of the room on its left is 0.25, and the rest are 0. All subsequent judgments are the same as this one.

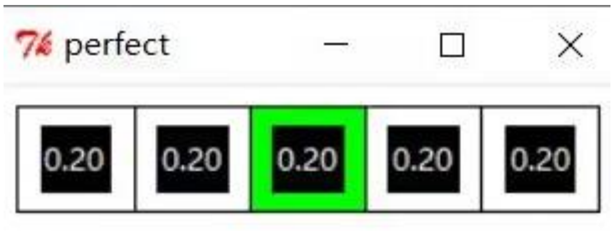
Step2

Check yourselves2

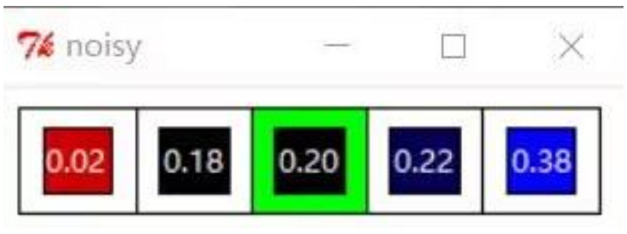
In this step, we tested the **Noisy**. The results are as follows：



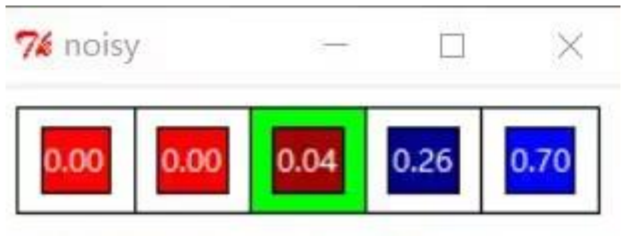
```
IDLE 2.6.6      ==== No Subprocess ====
>>>
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs MidnightBlue DDist(0: 0.200000, 1: 0.200000, 2: 0.200000, 3: 0.200000, 4: 0.200000)
after trans 1 DDist(0: 0.020000, 1: 0.180000, 2: 0.200000, 3: 0.220000, 4: 0.380000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 2
after obs white DDist(0: 0.024880, 1: 0.223923, 2: 0.004785, 3: 0.273684, 4: 0.472727)
after trans 2 DDist(1: 0.002488, 2: 0.042297, 3: 0.256746, 4: 0.698469)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -1
after obs white DDist(1: 0.002596, 2: 0.000849, 3: 0.267858, 4: 0.728698)
after trans -1 DDist(0: 0.002421, 1: 0.027724, 2: 0.287241, 3: 0.609744, 4: 0.072870)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -2
after obs white DDist(0: 0.003371, 1: 0.038598, 2: 0.007690, 3: 0.848891, 4: 0.101450)
after trans -2 DDist(0: 0.129582, 1: 0.694224, 2: 0.166049, 3: 0.010145)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 
```



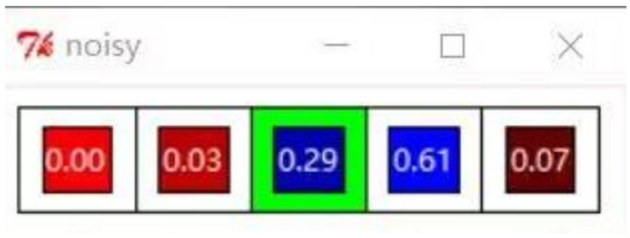
Start State



Input 1



Input 2

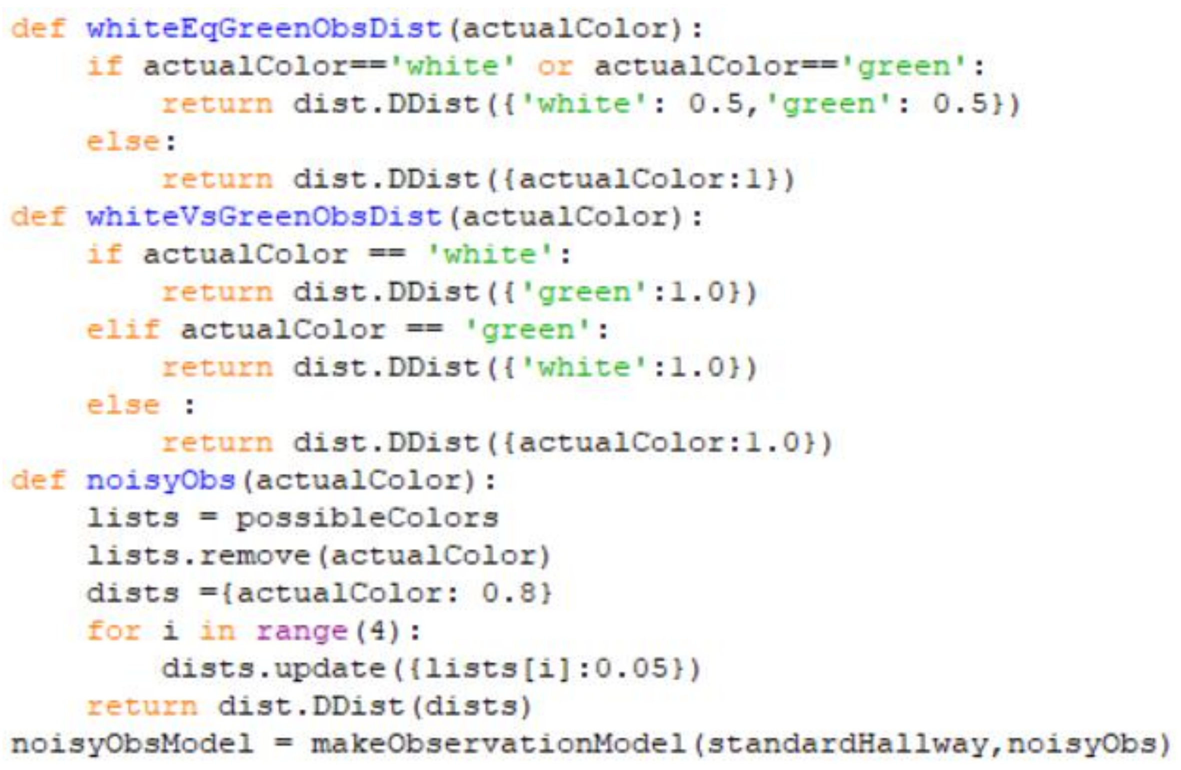


Input -1

We found that the program works the same way as Perfect, except that it adds errors, and when we want the robot to move a room to the right, it may not move one block to the right, but two blocks to either move to the left or not to move. As a result, we can never pinpoint the robot's position precisely.

Step3

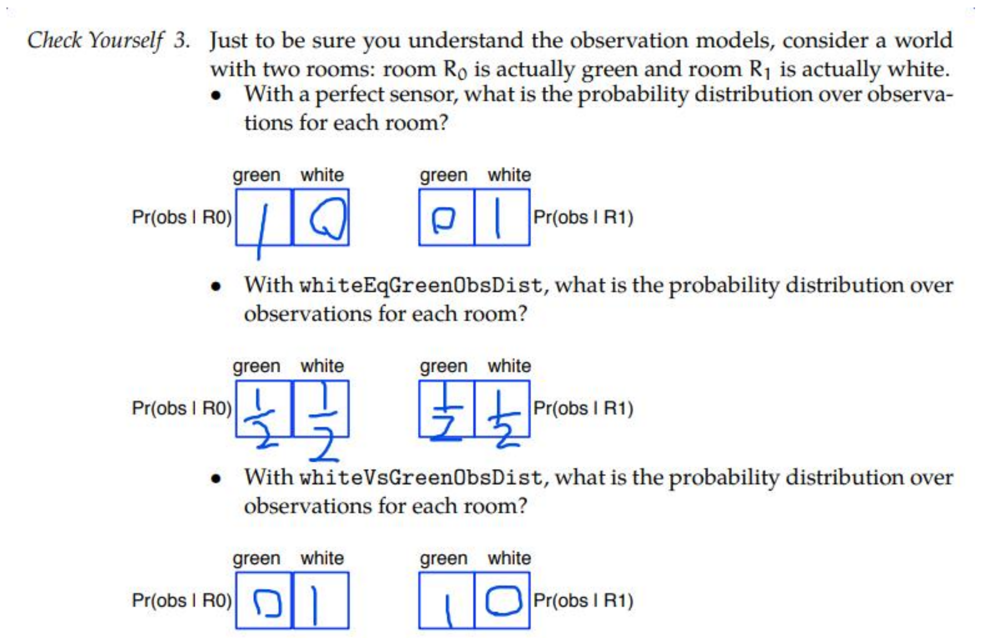
Below is our program code：



```
def whiteEqGreenObsDist(actualColor):
    if actualColor=='white' or actualColor=='green':
        return dist.DDist({'white': 0.5,'green': 0.5})
    else:
        return dist.DDist({actualColor:1})
def whiteVsGreenObsDist(actualColor):
    if actualColor == 'white':
        return dist.DDist({'green':1.0})
    elif actualColor == 'green':
        return dist.DDist({'white':1.0})
    else :
        return dist.DDist({actualColor:1.0})
def noisyObs(actualColor):
    lists = possibleColors
    lists.remove(actualColor)
    dists ={actualColor: 0.8}
    for i in range(4):
        dists.update({lists[i]:0.05})
    return dist.DDist(dists)
noisyObsModel = makeObservationModel(standardHallway,noisyObs)
```

Check yourselves3：

The results are as follows:



Check Yourself 3. Just to be sure you understand the observation models, consider a world with two rooms: room R_0 is actually green and room R_1 is actually white.

- With a perfect sensor, what is the probability distribution over observations for each room?

	green	white
Pr(obs I R0)	1	0
Pr(obs I R1)	0	1

- With whiteEqGreenObsDist, what is the probability distribution over observations for each room?

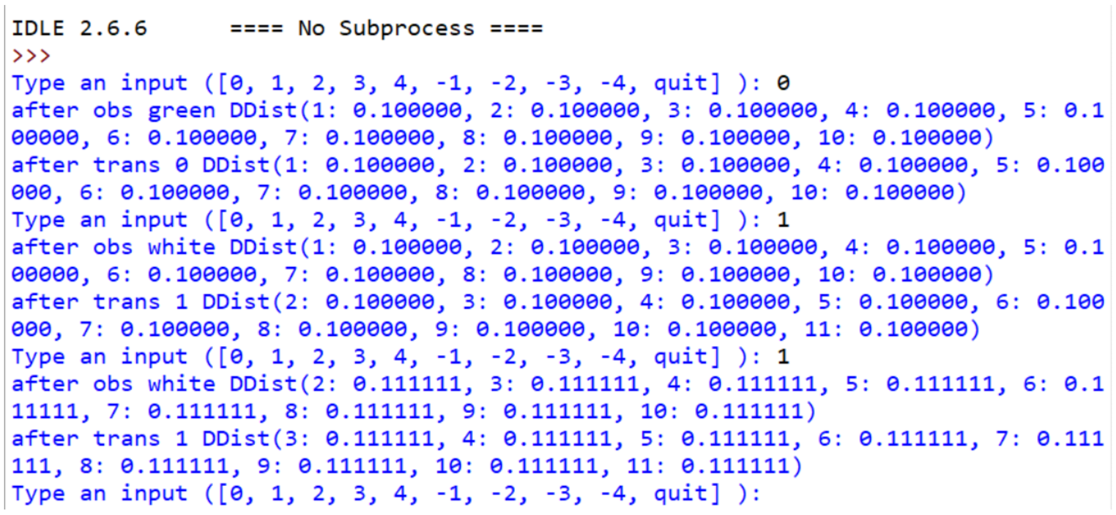
	green	white
Pr(obs I R0)	$\frac{1}{2}$	$\frac{1}{2}$
Pr(obs I R1)	$\frac{1}{2}$	$\frac{1}{2}$

- With whiteVsGreenObsDist, what is the probability distribution over observations for each room?

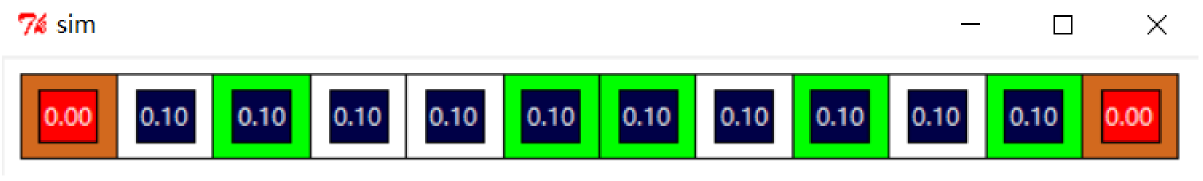
	green	white
Pr(obs I R0)	0	1
Pr(obs I R1)	1	0

Step4:

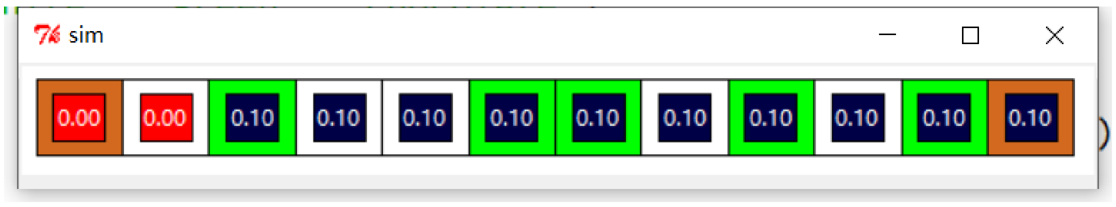
The results are as follows:



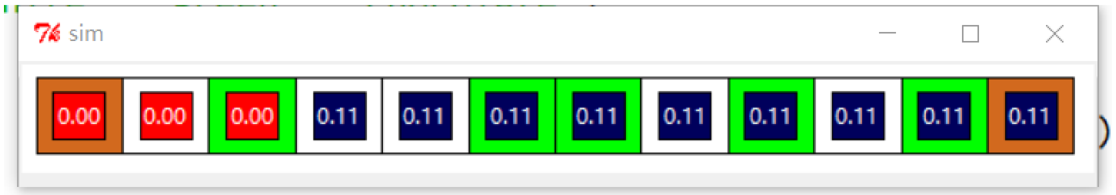
```
IDLE 2.6.6      ==== No Subprocess ====
>>> 
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs green DDist(1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.1
00000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000)
after trans 0 DDist(1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100
000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.1
00000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000)
after trans 1 DDist(2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100000, 6: 0.100
000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000, 11: 0.100000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(2: 0.111111, 3: 0.111111, 4: 0.111111, 5: 0.111111, 6: 0.1
11111, 7: 0.111111, 8: 0.111111, 9: 0.111111, 10: 0.111111)
after trans 1 DDist(3: 0.111111, 4: 0.111111, 5: 0.111111, 6: 0.111111, 7: 0.111
111, 8: 0.111111, 9: 0.111111, 10: 0.111111, 11: 0.111111)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ):
```



Input 0

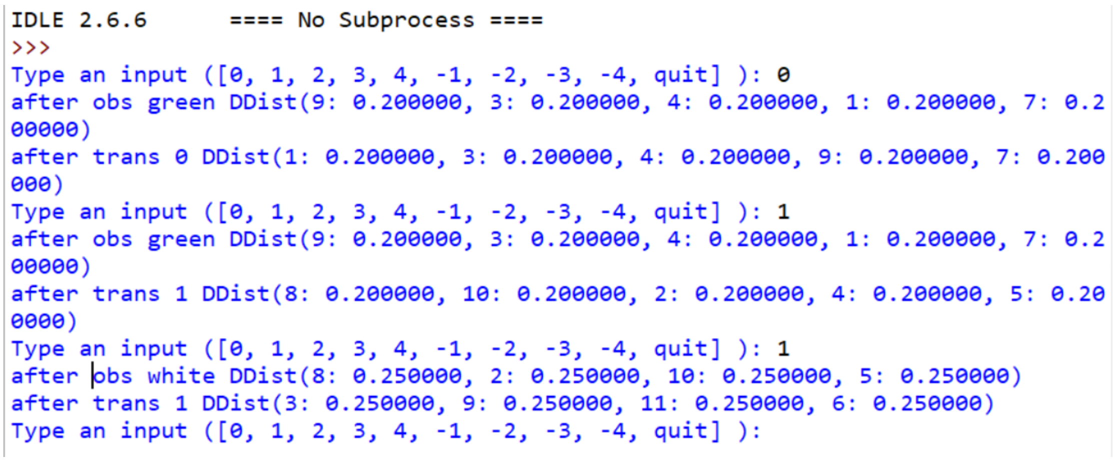


Input 1

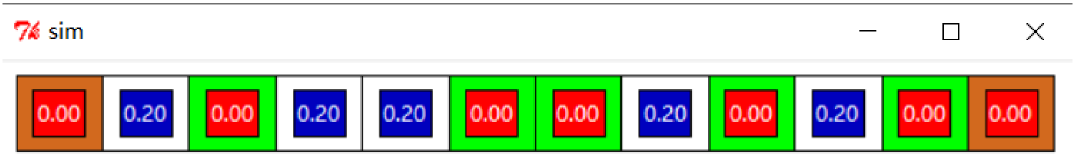


Input 1

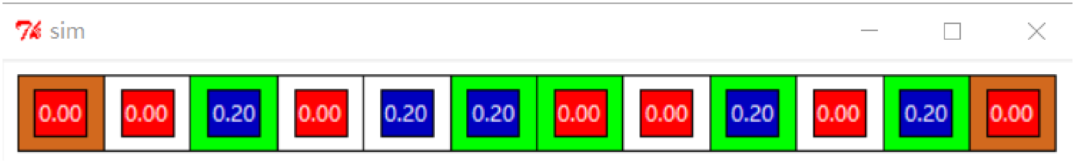
The above are the results of running **WhiteEqGreenObsDist**. The basic logic of running this program is the same as in Step 1, but the observation results are different. For example, after we input 0, We can get an observation that in this case is 'green', and then the 'green' is probably distributed, each green room is 0.2, but since the 'green' and 'white' probabilities are the same, all white and green rooms have a probability of 0.1. The probability of moving the trolley is displayed in the same way as in Step 1.



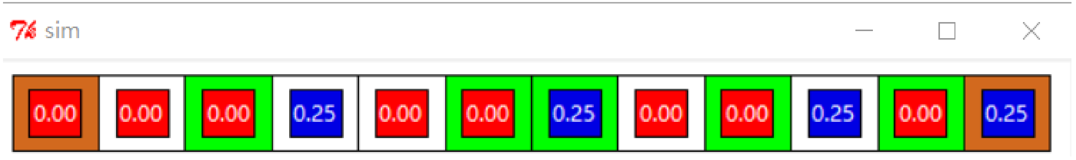
```
IDLE 2.6.6      ==== No Subprocess ====
>>>
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs green DDist(9: 0.200000, 3: 0.200000, 4: 0.200000, 1: 0.200000, 7: 0.2
00000)
after trans 0 DDist(1: 0.200000, 3: 0.200000, 4: 0.200000, 9: 0.200000, 7: 0.200
000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs green DDist(9: 0.200000, 3: 0.200000, 4: 0.200000, 1: 0.200000, 7: 0.2
00000)
after trans 1 DDist(8: 0.200000, 10: 0.200000, 2: 0.200000, 4: 0.200000, 5: 0.20
0000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(8: 0.250000, 2: 0.250000, 10: 0.250000, 5: 0.250000)
after trans 1 DDist(3: 0.250000, 9: 0.250000, 11: 0.250000, 6: 0.250000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ):
```



Input 0



Input 1



Input 1

The above are the results of running **WhiteVsGreenObsDist**. The basic logic of running this program is the same as in Step 1, but the observation results are different. If green is observed, it returns white, and if white is observed, it returns green. The rest runs in the same way as Step 1.

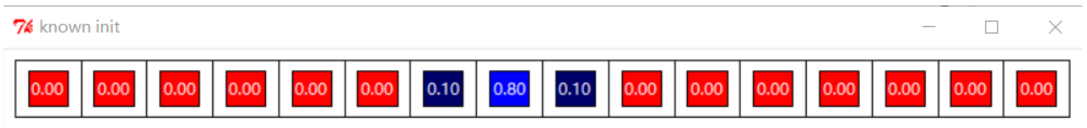
Step5

Below is our program code：

```
# Problem 11.1.2
############################################
def ringDynamics(loc,act,hallwayLength):
    new_loc = loc + act%hallwayLength
    if new_loc > hallwayLength - 1
        return(new_loc - hallwayLength)
    elif new_loc < 0
        return(new_loc + hallwayLength)
    else:
        return(new_loc)

def leftSlipTrans(nominalLoc,hallwayLength):
    if nominalLoc == 0:
        return dist.DDist({nominalLoc : 1.0})
    else :
        return dist.DDist({nominalLoc : 0.9,nominalLoc - 1 : 0.1})

def noisyTrans(nominalLoc,hallwayLength):
    if nominalLoc == 0:
        return dist.DDist({nominalLoc : 0.9,nominalLoc + 1: 0.1})
    elif nominalLoc == hallwayLength - 1:
        return dist.DDist({nominalLoc : 0.9,nominalLoc - 1: 0.1})
    else :
        return dist.DDist({nominalLoc: 0.8,nominalLoc + 1: 0.1,nominalLoc - 1: 0.1})
def black(actualColor):
    return dist.DDist({'black': 1.0})
noisyTransModel = makeTransitionModel(standardDynamics,noisyTrans)
```

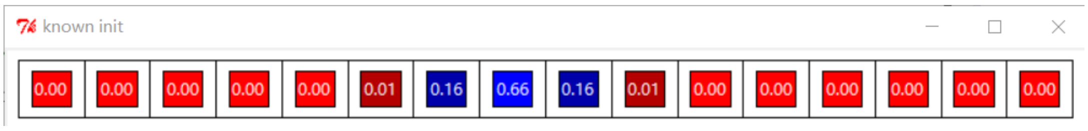
Step6：

The results are as follows：



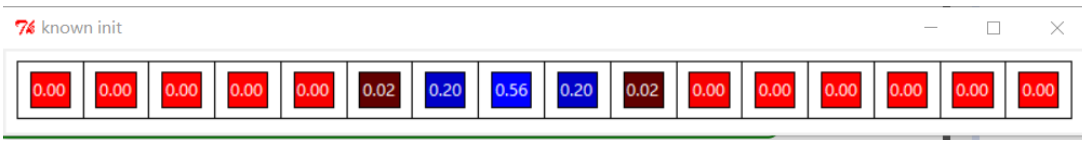
```
0.00 0.00 0.00 0.00 0.00 0.00 0.10 0.80 0.10 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

Input 0



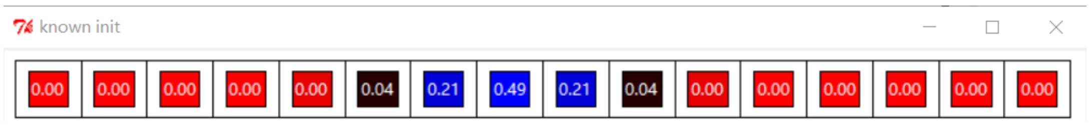
```
0.00 0.00 0.00 0.00 0.00 0.01 0.16 0.66 0.16 0.01 0.00 0.00 0.00 0.00 0.00 0.00
```

Second 0



```
0.00 0.00 0.00 0.00 0.00 0.02 0.20 0.56 0.20 0.02 0.00 0.00 0.00 0.00 0.00 0.00
```

Third 0

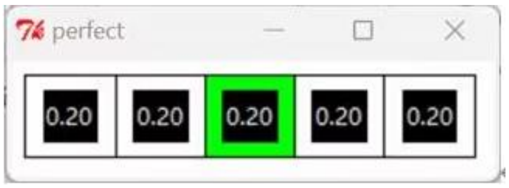


```
0.00 0.00 0.00 0.00 0.00 0.04 0.21 0.49 0.21 0.04 0.00 0.00 0.00 0.00 0.00 0.00
```

Fourth 0

We found that for each room, each observation followed the rules **{nominalLoc: 0.8, nominalLoc + 1: 0.1, nominalLoc - 1: 0.1}**, As a result, the results seen are constantly shifted to the sides, and this uncertainty increases as the number of observations increases.

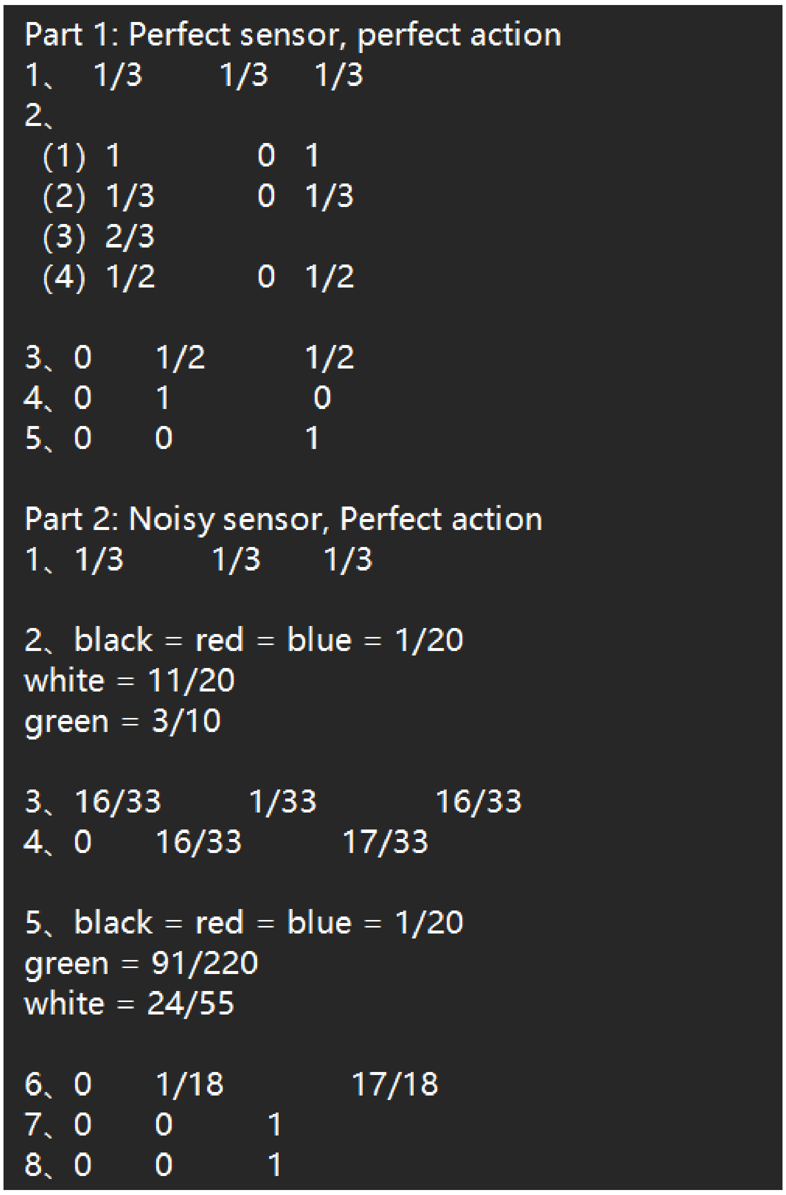
Checkoff 1:



In the initial state, the robot does not know the current position and does not observe the color of the room. All room probabilities are equal, and it is reflected in black. The robot is randomly placed in a room. After we give the first command, the first thing the robot does is observe the color of the room. In this model, there is no interference, so the robot gives a state of belief after observing the color of the room it is in. However, this belief state is based on the previous belief state and calculates the new belief state after the instruction is executed.

Step7:

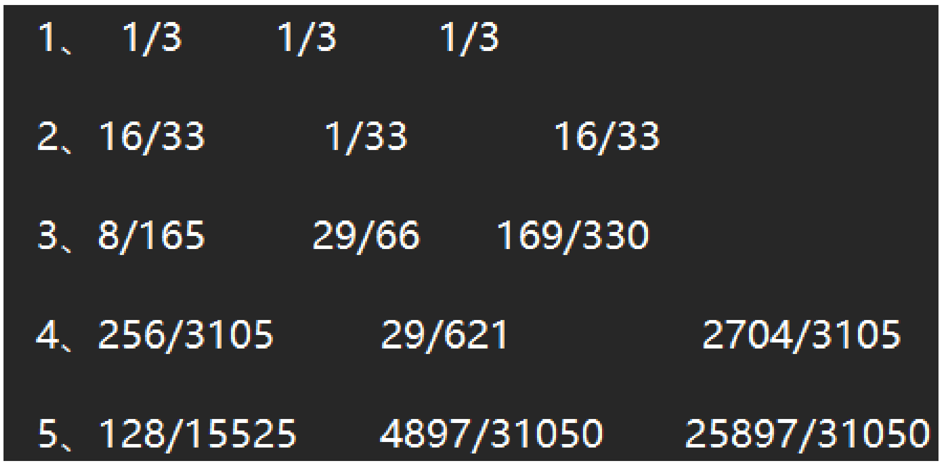
Here are our answers:



```
Part 1: Perfect sensor, perfect action
1、  1/3      1/3    1/3
2、
  (1) 1          0  1
  (2) 1/3        0  1/3
  (3) 2/3
  (4) 1/2        0  1/2

3、0     1/2        1/2
4、0     1           0
5、0     0          1

Part 2: Noisy sensor, Perfect action
1、1/3       1/3      1/3

2、black = red = blue = 1/20
white = 11/20
green = 3/10

3、16/33       1/33          16/33
4、0     16/33         17/33

5、black = red = blue = 1/20
green = 91/220
white = 24/55

6、0     1/18           17/18
7、0     0        1
8、0     0        1
```

Step8:

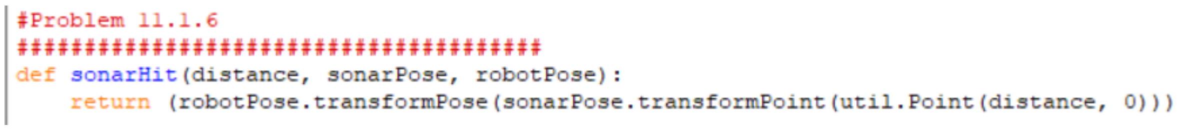
Here are our answers:



```
1、  1/3      1/3      1/3

2、16/33      1/33        16/33

3、8/165      29/66     169/330

4、256/3105      29/621          2704/3105

5、128/15525     4897/31050     25897/31050
```

Step9：

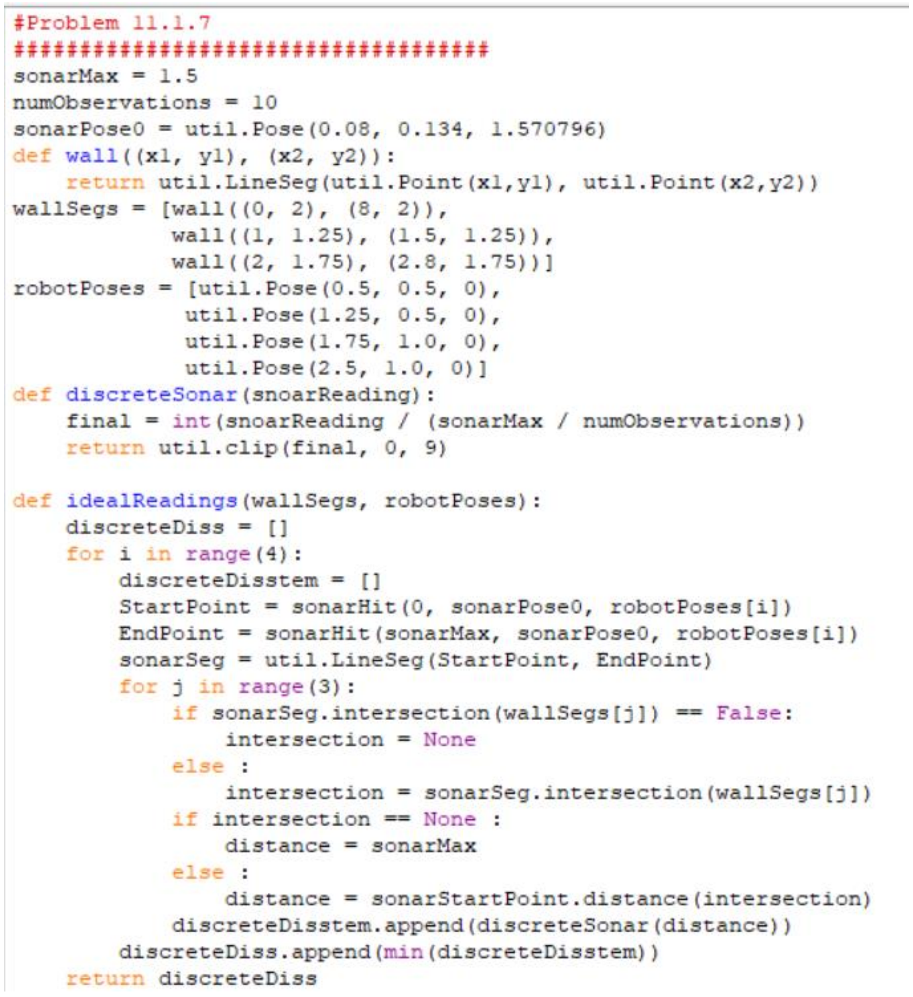
Below is our program code：



```
#Problem 11.1.6
#########################################
def sonarHit(distance, sonarPose, robotPose):
    return (robotPose.transformPose(sonarPose.transformPoint(util.Point(distance, 0))))
```

Step10：

Below is our program code：



```
#Problem 11.1.7
#####################################
sonarMax = 1.5
numObservations = 10
sonarPose0 = util.Pose(0.08, 0.134, 1.570796)
def wall((x1, y1), (x2, y2)):
    return util.LineSeg(util.Point(x1,y1), util.Point(x2,y2))
wallSegs = [wall((0, 2), (8, 2)),
            wall((1, 1.25), (1.5, 1.25)),
            wall((2, 1.75), (2.8, 1.75))]
robotPoses = [util.Pose(0.5, 0.5, 0),
             util.Pose(1.25, 0.5, 0),
             util.Pose(1.75, 1.0, 0),
             util.Pose(2.5, 1.0, 0)]
def discreteSonar(snoarReading):
    final = int(snoarReading / (sonarMax / numObservations))
    return util.clip(final, 0, 9)

def idealReadings(wallSegs, robotPoses):
    discreteDiss = []
    for i in range(4):
        discreteDisstem = []
        StartPoint = sonarHit(0, sonarPose0, robotPoses[i])
        EndPoint = sonarHit(sonarMax, sonarPose0, robotPoses[i])
        sonarSeg = util.LineSeg(StartPoint, EndPoint)
        for j in range(3):
            if sonarSeg.intersection(wallSegs[j]) == False:
                intersection = None
            else :
                intersection = sonarSeg.intersection(wallSegs[j])
            if intersection == None :
                distance = sonarMax
            else :
                distance = sonarStartPoint.distance(intersection)
            discreteDisstem.append(discreteSonar(distance))
        discreteDiss.append(min(discreteDisstem))
    return discreteDiss
```

Summary

There are three main difficulties in this experiment：

1. Principle：

The first two steps of this experiment guide the subsequent experimental steps, and when running **Perfect** and **Noisy**, our group had many discussions and finally came up with the running logic of the two programs. In the subsequent Step3 and Step5 programs, the observation method obtained in Step 1 was also used many times to finally get the answer. After completing Step 1 well, the subsequent experimental steps become relatively simple.

2. Calculate：

 In this Lab, there are several more complex calculations, most of which require Bayesian formulas to calculate conditional probabilities, and most of these calculations are more complex. Our group did not calculate the results well at the beginning, but after reading the handouts many times, combining their own derivations and many calculations, they finally came up with the answer.

3. Programming：

 The main difficulty of the programming part of this **Lab** is to combine probability theory with **Python** and to have a good grasp of the logic of the program. At the same time, this program also requires us to call classes and functions that have not been used in the past, testing our ability to learn independently that has been applied. Our group finally completed this task with the communication and cooperation of the members of the group, and realized the function of the program.