CS 4100 Homework 05: Reinforcement Learning

Due Monday 4/17 at midnight (1 minute after 11:59 pm) in Gradescope (with a grace period of 6 hours)

You may submit the homework up to 24 hours late (with the same grace period) for a penalty of 10%.

You must submit the homework in Gradescope as a zip file containing **two files:**

- The `.ipynb` file (be sure to `Kernel -> Restart and Run All` before you submit); and
- A `.pdf` file of the notebook.

For best results obtaining a clean PDF file on the Mac, select `File -> Print Review` from the Jupyter window, then choose `File-> Print` in your browser and then `Save as PDF`. Something similar should be possible on a Windows machine.

All homeworks will be scored with a maximum of 100 points; if point values are not given for individual problems, then all problems will be counted equally.

In [1]:
```
# Imports

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import random, randint, choice, normal,rand,seed
from scipy.stats import multivariate_normal
from collections import defaultdict
import sys
from tqdm import tqdm

```

Problems One -- Five: Iterated Prisoner's Dilemma (50 points total)

In the first half of the homework, we will develop an experimental framework for investigating the Iterated Prisoner's Dilemma. Please watch the lecture video for details of the IPD.

We will test two different frameworks, one where each agent in the population only plays against the agents in the environment, and so the agents are simply searching for the best strategy in that environment. In the second set of experiments, we will have the population play against the environment, and also each other; in this way, the population can learn as a whole how to (perhaps) cooperate with each other to succeed in that environment.

An agent is a list of 6 numbers, the first an integer recording the cumulative rewards, and the rest floats giving the probability P(C) of cooperating in the next round of a PD game, given the history of what happened last time in the game with this player (e.g., CD means "I cooperated last time, you defected last time" etc.)

```
 [rewards-so-far,  P(C) first time,  P(C) if CC last time, P(C) if CD, P(C) if DC, P(C) i
f DD  ]
```

Two examples are given at the end of the next cell, which you should read carefully. But do not change anything unless you check with Prof Snyder.

In [2]:
```
1 # Code for making environments↔
```

```
Mixed_Env:
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 1.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 0.5, 1.0, 0.0, 0.0, 1.0]
[0.0, 0.5, 0.5, 0.5, 0.5, 0.5]
[0.0, 1.0, 1.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 1.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 1.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0, 0.0]


Pavlov: [0, 0.5, 1.0, 0.0, 0.0, 1.0]

First move: 0.5

      C       D
  ----------------
  |       |       |
C |  1.0  |  0.0  |
  |       |       |
  ----------------
  |       |       |
D |  0.0  |  1.0  |
  |       |       |
  ----------------

Reward: 0.0

TFT: [0, 1.0, 1.0, 0.0, 1.0, 0.0]

First move: 1.0

      C       D
  ----------------
  |       |       |
C |  1.0  |  0.0  |
  |       |       |
  ----------------
  |       |       |
D |  1.0  |  0.0  |
  |       |       |
  ----------------

Reward: 0.0
```

Problem One (10 pts)

The first task is to create offspring by mutation and crossover. Complete the following template. Sample test results can be viewed on the PDF version of the homework.

Use `randint` to select the index to mutate. Use `normal(0,scale=mutate_std)` to choose a normally-distributed offset with mean 0 and standard deviation `mutate_std` . Be sure to rectify the mutated probability so that it is in the range 0..1.

Hint: You can perform an action with probability p as follows:

```
if random() < p:
    # do something with probability p
else:
    # do something else with probability 1-p
```

In [3]:

```
# Reproduction

# force probability to be in range [0..1]

def rectify(x):
    if x>1:
        return 1
    elif x<0:
        return 0
    else:
        return x

# change randomly selected probability by normally-distributed offset
# Must rectify to make sure is in range 0..1

def mutate(A,mutate_std):
    B = A.copy()                    # always make a copy so don't have two references to s

    pass         # your code here

    return B

# crossover each probability of the strategies A and B by creating new agent C,
# and for each index 1..5, copy over from A with probability crossover_p and keep value
# from B with probability 1 - crossover_p.

def crossover(A,B,crossover_p):
    C = B.copy()

    pass         # your code here

    return C

# crossover and then mutate to create child, which is returned

def make_child(A,B,crossover_p=0.5,mutate_std=0.2):

    pass

# test

seed(0)

print( mutate([0, 0,1.0,0.5,0.5,0.5], 0.1) )
print( crossover([0,1,1,1,1,1],[0,0,0,0,0,0],0.5) )
print( make_child([0,0.5,0.5,0.5,0.5,0.5],[0,0,0,0,0,0],0.5,0.2 ) )
```

```
[0, 0, 1.0, 0.5, 0.5, 0.6122794918829129]
[0, 0, 0, 0, 1, 1]
[0, 0.5, 0.5605610439106163, 0.5, 0, 0.5]
```

Problem Two (10 pts)

The next task is to write code to play a game of `num_rounds` rounds between two players. The rewards should be calculated from the beginning of the game, but you should NOT change the cumulative rewards at index 0 in the agents (these are the cumulative rewards).

Hint: When you want to determine what agent `A` should do the first time, `MoveIndex[('First','First')]` will return `1`, which is the index where the probability of `C` the for the first move is stored in `A`. If this probability is `prob_c`, then

```
choice( ['C','D'], p=[prob_c, 1-prob_c] )
```

will return `C` with probability `prob-c` and `D` with probability `1-prob_c`. You must store what each agent does, and then use it to look up the moves in the next round. You only need to save the previous round. You must add together the payoffs for all rounds to determine the rewards to return.

In [4]:
```
# Set up Environment and Population

# Payoffs

payoffs = { ('C','C'):300, ('C','D'):-100, ('D','C'):500, ('D','D'):-10 }

# look up what index should be consulted for the first round, or for what happened
# last time after the first round.

MoveIndex = { ('First','First'):1, ('C','C'):2, ('C','D'):3,\
                                     ('D','C'):4, ('D','D'):5    }

# play IPD between A and B and return reward for each of A and B at end of num_rounds rounds
def play_game(A,B,num_rounds):
    reward_A = reward_B = 0

    pass         # your code here

    return (reward_A,reward_B)

# test

A = Angel
B = Devil
print(A)
print(B)
(ra,rb) = play_game(A,B,10)
print(ra,rb)
print()

# play IPD with every member of Env and return cumulative reward
def get_reward(A,Env,num_rounds):

    pass         # your code here


# test

print(get_reward(Angel,All_Devils_Env,10))
print(get_reward(TFT,All_Angels_Env,10))
```

```
[0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0, 0.0, 0.0, 0.0, 0.0, 0.0]
-1000 5000

-10000
30000
```

This cell is used to display the results for analysis. Do not change anything without consultation.

In [5]:
```
# Keep track of winner in each round↔
```

Problem Three (15 pts)

The last task in creating the framework for experimenting with the IPD is to complete the following template, and verify that it works as expected. Follow the pseudocode and test as indicated (results may be found in the accompanying PDF).

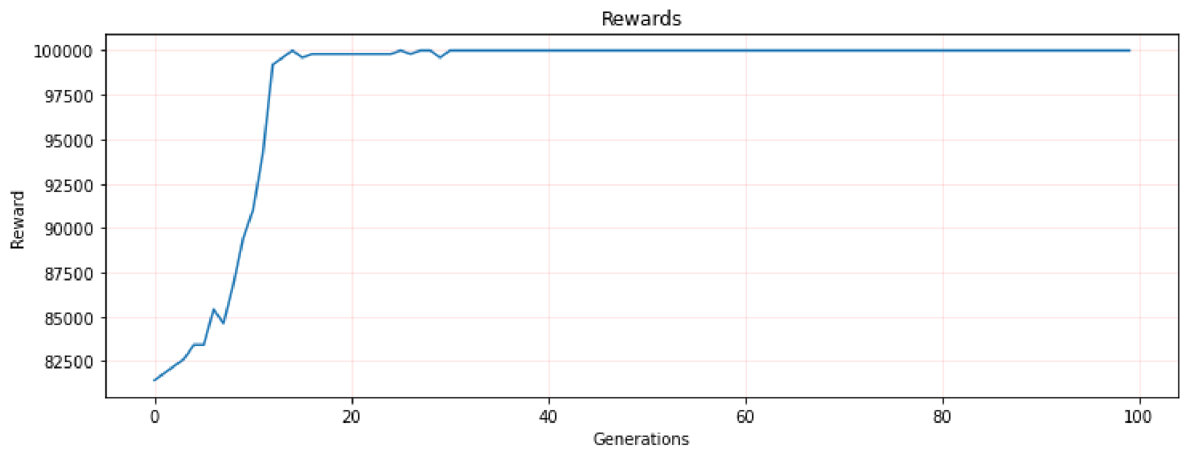
Hints: You can sort a list of lists in descending order on the first elements as shown here:

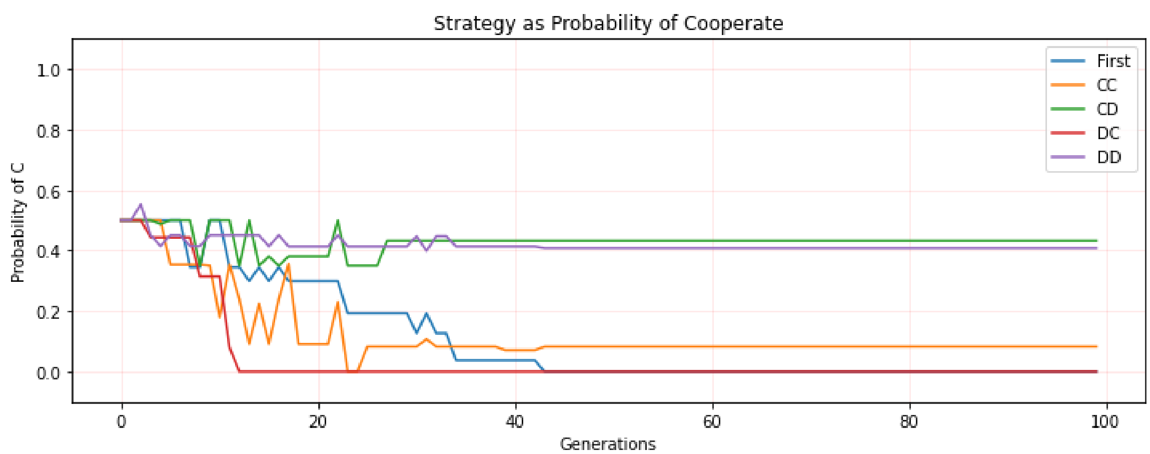
```
lst_of_lsts.sort(reverse=True, key=(lambda x: x[0]))
```

In [6]:

```python
# Run an experiment


def run_experiment(environment,population_size,num_children,
                   num_generations,crossover_prob=0.5, mutate_std=0.1,
                   num_rounds=100, play_each_other=False,
                   print_pop=False, display_evol=True):

    # make the population of random agents, which start with P(C) = 0.5 for all actions
    # use make_random_agent, which always provides a new copy of the array



    # keep track of parameters for best agent in each generation

    #             Reward First  CC  CD  DC  DD
    parameters = [    [],   [], [], [], [], [] ]



    for k in range(num_generations):

        # play each agent against the environment and insert the reward into A[0]
        # if play_each_other is True, then play against environment + population,
        # else just play against evironment



        # sort the population in descending order of rewards from this generation



        # record best agent
        record_parameters(population[0],parameters)

        # generate children: delete the last num_children agents in population (those
        # with the worst rewards in this generation), use the remaining population
        # as parents to create num_children new children to add to the population.
        # Select parents randomly from the remaining population.
        # Use choice(...., replace=False) so you don't choose the same parent twice in one '



    # Display evolution of best agents
    if print_pop:
        print_population(population)
    if display_evol:
        display_evolution(parameters)

# test
seed(0)
print("Environment: All_Angels_Env")
run_experiment(environment=Environments[0],
               population_size=10,
               num_children=5,
               num_generations=100,
               crossover_prob=0.5,
               mutate_std=0.1,
               num_rounds=20,
               play_each_other=False,
               print_pop=True,
               display_evol=True)

```

```
Environment: All_Angels_Env
[100000.0, 0.0, 0.082, 0.432, 0.0, 0.408]
[100000.0, 0.0, 0.195, 0.432, 0.0, 0.414]
[100000.0, 0.0, 0.071, 0.432, 0.0, 0.259]
[100000.0, 0.0, 0.0, 0.432, 0.0, 0.259]
[100000.0, 0.0, 0.053, 0.432, 0.0, 0.408]
[100000.0, 0.0, 0.0, 0.302, 0.0, 0.259]
[100000.0, 0.0, 0.071, 0.453, 0.0, 0.259]
[100000.0, 0.0, 0.082, 0.432, 0.0, 0.541]
[100000.0, 0.0, 0.053, 0.432, 0.0, 0.259]
[100000.0, 0.154, 0.082, 0.432, 0.0, 0.408]
```






```
Best agent in last generation:

First move: 0.0

      C       D
  ----------------
  |       |       |
C | 0.082 | 0.432 |
  |       |       |
  ----------------
  |       |       |
D |   0.0 | 0.408 |
  |       |       |
  ----------------

Reward: 100000.0
```

Problem Four (7.5 pts)

Now the fun begins.... For this problem, we would like you to run multiple experiments to investigate what strategies will evolve in each of the 8 environments defined in the first code cell above. Test things out with smaller numbers of generations and population size, but eventually you should run experiments with at least the following parameters:

```
run_experiment(environment= ... ,
               population_size=100,          # try with 10 to start, but best results wit
h at least 100
               num_children=50,              # this is 50% children, if change pop size a
lso change this
               num_generations=100,          # may need to change this depending on resul
ts
               crossover_prob=0.5,           # you can think of these two as similar to t
he learning rate:
               mutate_std=0.1,               #     smaller values will make children more
similar to parents
               num_rounds=20,                # don't do less than 20
               play_each_other=False,
               print_pop=False,
               display_evol=True)
```

Your goal is to determine what strategy evolves in each of the environments. It may not correspond to one of of the environment strategies, but you can examine the winning agent at the end and think about the choices it learned to make.

Feel free to change the parameters, as long as `play_each_other=False` and `num_rounds` is at least 20. In particular, you may see the strategy stabilize in fewer generations, or you may need to go above 100 to see the result. In general, you will get better results with larger populations.

Run your experiments, and for each, give a short explanation of what you see, and why you think that particular strategy evolved. It may not be possible to give a precise explanation, but give it a shot!

Also explain if you found better choices of the other parameters such as percentage of children, crossover probability, and the mutation standard deviation. I found good results with the values above; these three essentially affect the learning rate, and hence the rate at which it stabilizes.

Problem Five (7.5 pts)

Now we would like you to do the same as in the last problem, but with `play_each_other=True` and with just the following environments:

```
All_Devils_Env,    All_Pavlovs_Env,    All_TFTs_Env,      Mixed_Env
```

In general, you will need to run these for more than 100 generations to see a potential group strategy evolving. You may need to modify the other parameters as well. Again, for each of these four environments, show your results and provide analysis for each case.

Be sure to comment on how these may be different from the same environment in the previous problem.

Problems Six -- Ten: Q-Learning in Gridworld

For the rest of this homework, we will investigate Reinforcement Learning a Grid World, a simple problem in which a single agent moves around a 2D grid in search of a goal state where a reward sits waiting. These are all versions of the cliff walking example shown in lecture.

In general, it "costs" 1 unit to move, so that many cells may have an immediate reward of -1. There are also "holes" where the game terminates and the immediate reward is -100. The start state has a reward of 0 and the goal state has a reward of 100. The trial terminates at holes and in the goal state.

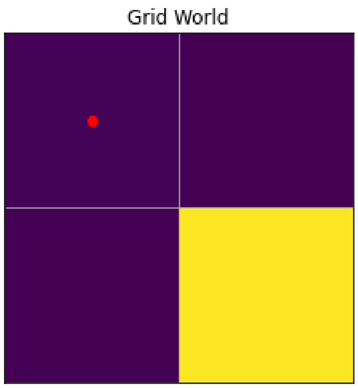
We have provided various functions to display the grid, the rewards, the Q-Table and the strategy which evolves to garner the maximum cumulative reward at the end.

The next two cells show how each world is created as an object with a grid of rewards, a start state, and terminal states (where the current trial will end).

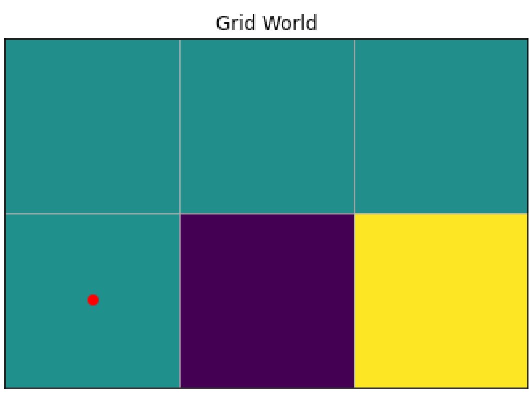
In [9]:
```
# Each grid world has a 2d matrix of rewards (which also gives the dimensions of the matrix)
```

In [10]:
```
1 # Grid worlds initialization↔
```

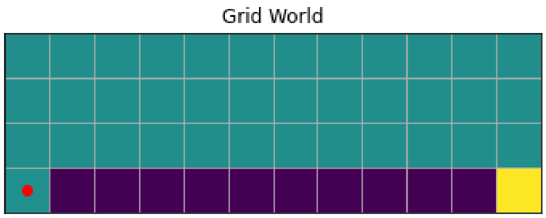
```
World 0

Rewards
---------------
|    0 |   -1 |
---------------
|   -1 |  100 |
---------------
```



```
Terminal states: [(1, 1)]


World 1

Rewards
----------------------
|   -1 |   -1 |   -1 |
----------------------
|    0 | -100 |  100 |
----------------------
```



```
Terminal states: [(1, 1), (1, 2)]


World 2

Rewards
-----------------------------------------------------------------------------------
|   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |
-----------------------------------------------------------------------------------
|   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |
-----------------------------------------------------------------------------------
|   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |   -1 |
-----------------------------------------------------------------------------------
|    0 | -100 | -100 | -100 | -100 | -100 | -100 | -100 | -100 | -100 | -100 |  100 |
-----------------------------------------------------------------------------------
```

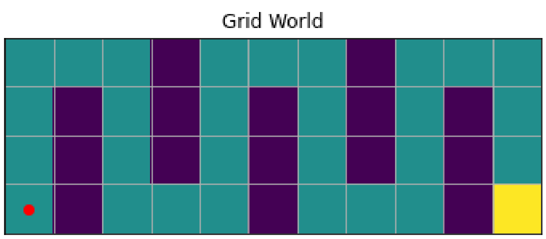


Terminal states: [(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), (3, 11)]

World 3

Rewards

-1	-1	-1	-100	-1	-1	-1	-100	-1	-1	-1
-1	-100	-1	-100	-1	-100	-1	-100	-1	-100	-1
-1	-100	-1	-100	-1	-100	-1	-100	-1	-100	-1
0	-100	-1	-1	-1	-100	-1	-1	-1	-100	100



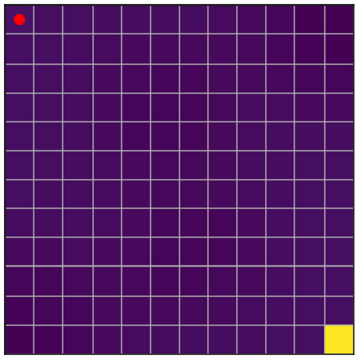
Terminal states: [(0, 3), (0, 7), (1, 1), (1, 3), (1, 5), (1, 7), (1, 9), (2, 1), (2, 3), (2, 5), (2, 7), (2, 9), (3, 1), (3, 5), (3, 9), (3, 10)]

World 4

Rewards

0	-1.04	-1.13	-1.31	-1.51	-1.64	-1.68	-1.84	-2.42	-3.45	-4.52	-4.98
-1.02	-1.09	-1.3	-1.7	-2.16	-2.4	-2.31	-2.18	-2.44	-3.22	-4.12	-4.51
-1.04	-1.16	-1.5	-2.16	-2.91	-3.28	-3.01	-2.48	-2.29	-2.62	-3.17	-3.42
-1.05	-1.19	-1.6	-2.37	-3.27	-3.7	-3.34	-2.57	-2.03	-1.98	-2.18	-2.3
-1.06	-1.18	-1.53	-2.2	-3.0	-3.41	-3.11	-2.38	-1.77	-1.52	-1.52	-1.55
-1.11	-1.18	-1.4	-1.87	-2.47	-2.87	-2.75	-2.22	-1.66	-1.33	-1.22	-1.19
-1.27	-1.3	-1.4	-1.71	-2.25	-2.77	-2.87	-2.46	-1.85	-1.37	-1.14	-1.07
-1.65	-1.63	-1.63	-1.86	-2.44	-3.14	-3.42	-3.0	-2.2	-1.52	-1.17	-1.04
-2.3	-2.21	-2.06	-2.14	-2.66	-3.39	-3.72	-3.28	-2.37	-1.6	-1.19	-1.04
-3.14	-2.97	-2.59	-2.37	-2.59	-3.08	-3.31	-2.92	-2.16	-1.5	-1.16	-1.03
-3.88	-3.63	-3.03	-2.48	-2.28	-2.39	-2.44	-2.17	-1.7	-1.3	-1.09	-1.02
-4.18	-3.89	-3.17	-2.43	-1.95	-1.76	-1.68	-1.52	-1.31	-1.13	-1.04	100.0

Grid World

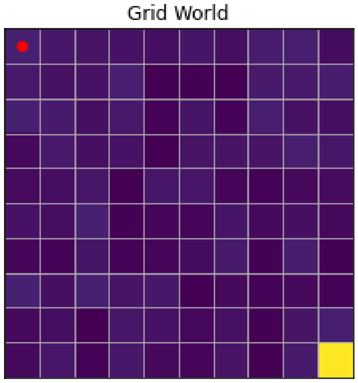


```
Terminal states: [(11, 11)]
```

```
World 5

Rewards
```

0	-2.8	-4.0	-4.6	-5.8	-3.5	-5.6	-1.1	-0.4	-6.2
-2.1	-4.7	-4.3	-0.7	-9.3	-9.1	-9.8	-1.7	-2.2	-1.3
-0.2	-2.0	-5.4	-2.2	-8.8	-3.6	-8.6	-0.6	-4.8	-5.9
-7.4	-2.3	-5.4	-4.3	-9.8	-3.8	-3.9	-3.8	-0.6	-3.2
-6.4	-5.6	-3.0	-9.4	-3.3	-3.3	-7.9	-8.7	-6.8	-6.4
-4.3	-5.6	-0.1	-9.0	-7.9	-8.4	-3.5	-7.5	-5.3	-7.6
-8.4	-8.9	-3.4	-8.6	-8.0	-6.3	-1.8	-9.0	-1.6	-9.0
-0.2	-5.3	-0.2	-4.0	-2.6	-9.6	-7.2	-8.8	-7.0	-8.8
-6.8	-5.9	-9.4	-3.1	-4.3	-7.3	-4.8	-9.1	-4.2	-0.7
-6.8	-3.3	-8.7	-2.8	-7.1	-8.2	-4.1	-9.8	-1.7	100.0

Grid World

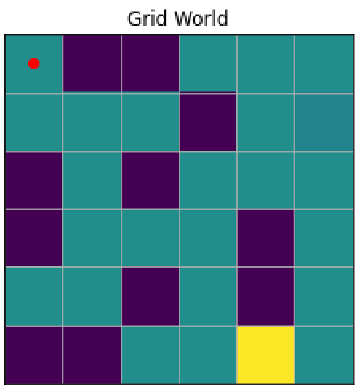


```
Terminal states: [(9, 9)]
```

World 6

Rewards

0	-100	-100	-1	-1	-1
-1	-1	-1	-100	-1	-10
-100	-1	-100	-1	-1	-1
-100	-1	-1	-1	-100	-1
-1	-1	-100	-1	-100	-1
-100	-100	-1	-1	100	-1

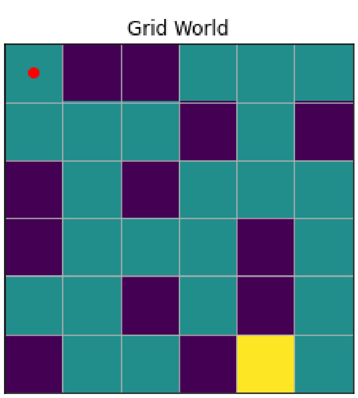



```
Terminal states: [(0, 1), (0, 2), (1, 3), (2, 0), (2, 2), (3, 0), (3, 4), (4, 2), (4, 4), (5, 0), (5, 1), (5, 4)]
```

World 7

Rewards

0	-100	-100	-1	-1	-1
-1	-1	-1	-100	-1	-100
-100	-1	-100	-1	-1	-1
-100	-1	-1	-1	-100	-1
-1	-1	-100	-1	-100	-1
-100	-1	-1	-100	100	-1

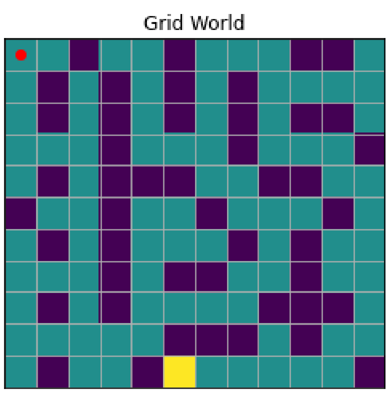


```
Terminal states: [(0, 1), (0, 2), (1, 3), (1, 5), (2, 0), (2, 2), (3, 0), (3, 4), (4, 2), (4, 4), (5, 0), (5, 3), (5, 4)]
```

```
World 8
```

Rewards

0	0	-100	-1	-1	-100	-1	-1	-1	-100	-100	-1
-1	-100	-1	-100	-1	-100	-1	-100	-1	-1	-1	-1
-1	-100	-1	-100	-1	-100	-1	-100	-1	-100	-100	-1
-1	-1	-1	-100	-1	-1	-1	-100	-1	-1	-1	-100
-1	-100	-1	-100	-100	-100	-1	-1	-100	-100	-1	-1
-100	-1	-1	-100	-1	-1	-100	-1	-1	-1	-100	-1
-1	-100	-1	-100	-1	-1	-1	-100	-1	-100	-1	-1
-1	-1	-1	-100	-1	-100	-100	-1	-1	-100	-1	-1
-1	-100	-1	-100	-1	-1	-1	-1	-100	-100	-100	-1
-1	-1	-1	-1	-1	-100	-100	-100	-1	-100	-1	-1
-1	-100	-1	-1	-100	100	-1	-1	-1	-1	-1	-100




```
Terminal states: [(0, 2), (0, 5), (0, 9), (0, 10), (1, 1), (1, 3), (1, 5), (1, 7), (2, 1), (2, 3), (2, 5), (2, 7), (2, 9), (2, 10), (3, 3), (3, 7), (3, 11), (4, 1), (4, 3), (4, 4), (4, 5), (4, 8), (4, 9), (5, 0), (5, 3), (5, 6), (5, 10), (6, 1), (6, 3), (6, 7), (6, 9), (7, 3), (7, 5), (7, 6), (7, 9), (8, 1), (8, 3), (8, 8), (8, 9), (8, 10), (9, 5), (9, 6), (9, 7), (9, 9), (10, 1), (10, 4), (10, 5), (10, 11)]
```

Problem Six (5 pts)

The first task in this set of problems is to create a dictionary which determines the allowable set of actions in each state, and a `goto` function which tells how an action moves to a new state.

A state is simply a pair `(row,col)` in the grid.

Part A

Actions are

```
Moves    = ['U','R','L','D']    =    Up, Right, Left, Down
```

Clearly, you can not move outside the allowable indices for the given grid.

Hint: Create a default dictionary which returns Moves (meaning, any move is allowed) and then add the special cases for corners and edges of the grid. You can get the dimensions of the grid using `W.num_rows` and `W.num_cols` . A default dictionary may be created as follows:

```
Dictionary = defaultdict((lambda : <default-value>))
```

In [11]:

```python
# Actions for each state
# These are lists so can use np.random.choice for exploration

Moves     = ['U','R','L','D']

def initialize_Actions(W):

    pass         # your code here


# test for several values of N
N = 1
World[N].print_rewards()
A = initialize_Actions(World[N])

print(A[(1,2)])

A
```

```
Rewards
----------------------
|   -1 |   -1 |   -1 |
----------------------
|    0 | -100 |  100 |
----------------------
['U', 'L']
```

Out[11]:

```
defaultdict(<function __main__.initialize_Actions.<locals>.<lambda>()>,
            {(0, 0): ['R', 'D'],
             (0, 2): ['L', 'D'],
             (1, 0): ['U', 'R'],
             (1, 2): ['U', 'L'],
             (0, 1): ['R', 'L', 'D'],
             (1, 1): ['U', 'R', 'L']})
```

Part B

Now you must write a function which determines the next state, given the current state and the action. An example is shown in the test. Do not bother with error checking, as this will only be called on states and moves which have been checked with a dictionary `A` created in Part A.

In [12]:

```python
# state transitions -- no error checking, will only be called on legal moves

def goto_state(s,a):

    pass         # your code here


# test

for m in Moves:
    print(goto_state((2,3),m))
```

```
(1, 3)
(2, 4)
(2, 2)
(3, 3)
```

Problem Seven (5 pts)

The next task in this set of problems is to create the Q-Table which records the current best estimate of the strategy, as discussed in lecture. As you can see from the test, the Q-Table starts with random values in the range -10 .. 0 for all legal moves (terminal states are blank).

Thus, the Q-Table is effectively a 3D array (rows, columns, and moves) but we will implement this as a dictionary with keys `(state,move) = ((row,col),move)` mapped to Q-values (floats).

Hint: Use `random()` with suitable arithmetic operations to expand and shift from the range [0..1] to the range [-10..0]. Do not worry about what states are terminal, as the Q-values will never be used!

In [13]:

```
# Q-Table is dictionary with keys (state,move) = ((row,col),move) mapped to Q-values

# Initialize with random default values in range -10..0

def initialize_Q_table(W,A):

    pass         # your code here


def print_Q_table(W,A,Q):
    print("Q-Table")
    width = 6
    precision = 4

    hrule = ('---------'*W.num_cols)+'-'

    print(hrule)
    for r in range(W.num_rows):
        for move in ['U','L','R','D']:
            for c in range(W.num_cols):
                if (r,c) in W.terminal_states:
                    print('|  '+(" "*width),end='')
                elif move in A[(r,c)]:
                    print('|'+move+':'+f"{np.around(Q[((r,c),move)],3):{width}.{precision}}'
                else:
                    print('|'+move+':'+(" "*width),end='')

            print('|')
        print(hrule)

# test -- try for several values of N

seed(0)
N = 1
W = World[N]
A = initialize_Actions(World[N])
Q = initialize_Q_table(W,A)
print_Q_table(W,A,Q)
```

```
Q-Table
----------------------------
|U:      |U:      |U:      |
|L:      |L:-4.551|L:-3.541|
|R:-4.512|R:-3.972|R:      |
|D:-2.848|D:-5.763|D:-5.624|
----------------------------
|U:-1.082|        |        |
|L:      |        |        |
|R:-0.363|        |        |
|D:      |        |        |
----------------------------
```

Problem Eight (5 pts)

Now we must create functions to determine the best allowable move, given A, Q, and the state, and write an epsilon-greedy version of the strategy `Pi` .

The best move is simply the allowable move from the current state with the maximum Q-value. Return the move as a character 'U', 'R', etc.

The epsilon-greed strategy will choose a random move from those allowable in the current state with probability `epsilon` or the best move with probability `1-epsilon` .

In [14]:
```
# Strategy code for epsilon-greedy Pi

# find move with best Q-value in state s
def best_move(A,Q,s):

    pass          # your code here


# epsilon-greedy strategy

def Pi(A,Q,s,epsilon):

    pass          # your code here


```

In [15]:
```
# Tests:   run this cell repeatedly to test for N == 1 two cells up

print(best_move(A,Q,(0,0)))    # 'D'
print(best_move(A,Q,(0,2)))    # 'L'
print()
print(Pi(A,Q,(1,0),0.0))     # 'R'
print(Pi(A,Q,(1,0),0.5))     # should give 'R' about 3x as often as 'U'
print(Pi(A,Q,(1,0),1.0))     # should give approximately same number of 'U' and 'R'
```

```
D
L

R
R
R
```

In [16]:
```
# Pretty-printing code for Strategy↔
```

Problem Nine (15 pts)

In this problem we will create the framework for running multiple trials for the agent to learn how to solve the GridWorld problem of maximizing rewards.

There are several parameters of interest, as explained in lecture:

- epsilon = for epsilon-greedy strategy, probability of exploration by random move
- lam = exponential decrease in epsilon (can not use "lambda" because that is a keyword in Python)
- num_trials = number of random trials in this experiment

(We also thought about the use of a "learning rate" parameter `alpha` and a "discount" `gamma` for future rewards, but these seemed to only be a disadvantage in this simple GridWorld scenario.)

In [1]:
```python
# Q-Algorithm code


def run_experiment(N,epsilon=0.25,lam=1.0,num_trials=1000,display=False):

    # initialize grid world N

    # use a seed to help with comparing results
    seed(0)

    # initialize A

    # initialize Q

    # Now run num_trials different trials. For each,

            # initialize the current state s to the start state

            # while s is not a terminal state

                #  determine the action a from s using the policy Pi
                #  determine the next state s1 given the action a
                #  determine the reward and the best move in s1
                #  update Q-Table with the new Q-value for current state s = sum of reward i
                #       Q-value of best move in s1;

                #  update epsilon, reduced in each successive move by lam (if lam < 1 this r
                #      you will explore less and less as the trial goes on)
                #  update current state to s1




    # calculate the cumulative reward of the path given by the strategy implied by the Q-tab
    # This will get into an infinite loop if the path has cycles! To check for cycles,
    # add all states in the path into a set, and for each new state in the path, if
    # it is already in the set, you have a cycle and a cumulative reward is meaningless.
    # set a flag  cycle = True   if a cycle is found.



    # show all the data structures if you want

    if display:
        W.display_heat_map()
        print()
        W.print_rewards()
        print()
        print_Q_table(W,A,Q)
        print()
        print_strategy(W,A,Q)
        print()

    # if there is a cycle, return 0, else return the cumulative reward

# test

#print("Reward:",run_experiment(2,num_trials=200, display=True))
```

Problem Ten (20 pts)

Now we will reap the benefit of the previous problems and find out how well the Q-Algorithm does to learn paths in these grid worlds!

For each of the 9 grid world examples, play around with the parameters

```
epsilon,    lam,       num_trials
```

to answer the following question:

```
What parameters will solve the problem in the fewest number of trials?
```

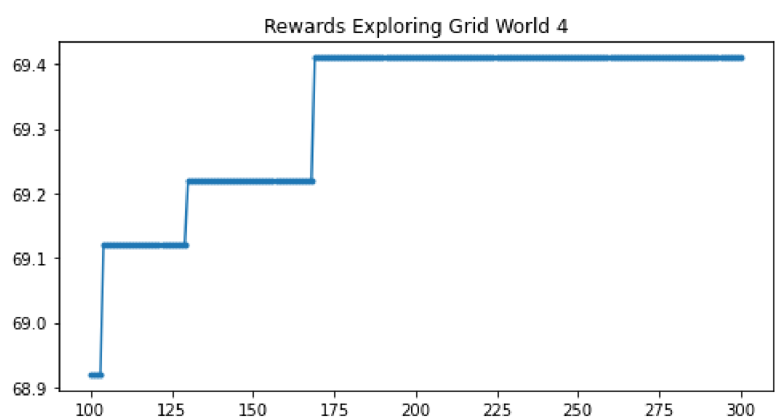
"Solving" the problem means learning a "steady-state" strategy, meaning that if you increase the number of trials, the strategy does not change. The agent may accidentally hit on an optimal strategy, but continuing to search may "unlearn" the solution. The system can only be said to have learned the optimal strategy only if it does not change as the number of trials is increased.

In order to help you with your experiments, the following function is provided, which will plot the rewards for a number of trials between two bounds. The best way to proceed is to try single experiments with various parameters as shown at the end of the previous cell, and then confirm your understanding with the `plot_rewards` function. The example shows how to determine when the problem has been solved.

Your solution to this problem is a presentation of the experiments you performed, plus your analysis. Be sure to test various values for epsilon and lam. You must do all 9 examples.

In [18]:

```python
# Plotting the rewards to find smallest number of trials which produce a steady-state maximu


def plot_rewards(N,lower_bound,upper_bound,epsilon=0.25,lam=1.0):

    X = range(lower_bound,upper_bound+1)

    Y = [run_experiment(N,epsilon=epsilon,lam=lam,num_trials=k,display=False) for k in X]

    plt.figure(figsize=(8,4))
    plt.title('Rewards Exploring Grid World '+str(N))
    plt.plot(X,Y)
    plt.scatter(X,Y,marker='.')
    plt.show()

    print("Reward:",Y[-1])

    # under assumption that steady state was reached by the upper_bound,
    # find the first time that value was found in this range

    for k in range(len(Y)-1,-1, -1):
        if Y[k] < Y[-1]:
            print("Steady state found at",X[k+1],"trials.")
            break

# example

plot_rewards(4,100,300)

```




```
Reward: 69.41
Steady state found at 169 trials.
```