Mount the Google Drive onto the Colab as the storage location.

Following the instructions returned from the below cell. You will click a web link and select the google account you want to mount, then copy the authorication code to the blank, press enter.

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Append the directory location where you upload the start code folder (In this problem, *RLalgs*) to the sys.path

E.g. dir = '/content/drive/My Drive/RL/.', start code folder is inside "RL" folder.

```
import sys
sys.path.append('/content/gdrive/My Drive/RL/RLalgs2')
#sys.path.append('</dir/to/start/code/folder/.>')
print(sys.path)
```

```
['', '/env/python', '/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/pyt
```

Your code should remain in the block marked by
############################
YOUR CODE STARTS HERE
YOUR CODE ENDS HERE
############################
Please don't edit anything outside the block.

```
% load_ext autoreload
% autoreload 2
import numpy as np
import random
import matplotlib.pyplot as plt
import gym
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1. Incremental Implementation of Average

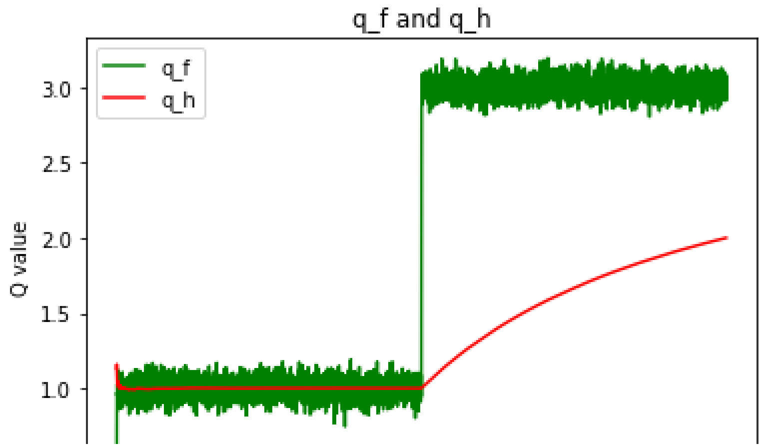
We've finished the incremental implementation of average for you. Please call the function estimate with 1/step step size and fixed step size to compare the difference between this two on a simulated Bandit problem.

```
from utils import estimate
random.seed(6885)
numTimeStep = 10000
q_h = np.zeros(numTimeStep + 1) # Q Value estimate with 1/step step size
q_f = np.zeros(numTimeStep + 1) # Q value estimate with fixed step size
FixedStepSize = 0.5 #A large number to exaggerate the difference
for step in range(1, numTimeStep + 1):
    if step < numTimeStep / 2:
        r = random.gauss(mu = 1, sigma = 0.1)
    else:
        r = random.gauss(mu = 3, sigma = 0.1)

    #TIPS: Call function estimate defined in ./RLalgs/utils.py
    ############################
    # YOUR CODE STARTS HERE
    q_h[step] = estimate(q_h[step-1],1/step, r)
    q_f[step] = estimate(q_f[step-1], FixedStepSize, r)
    # YOUR CODE ENDS HERE
    ############################

q_h = q_h[1:]
q_f = q_f[1:]
```

Plot the two Q value estimates. (Please include a title, labels on both axes, and legends)

```
############################
# YOUR CODE STARTS HERE
x = np.arange(numTimeStep)
y1 = np.array(q_f)
y2 = np.array(q_h)
plt.title("q_f and q_h")
plt.xlabel('Step')
plt.ylabel('Q value')
plt.plot(x, y1, 'g', label = 'q_f')
plt.plot(x, y2, 'r', label = 'q_h')
plt.legend()
plt.show
# YOUR CODE ENDS HERE
############################
```

```
<function matplotlib.pyplot.show>
```



2. ϵ-Greedy for Exploration

In Reinforcement Learning, we are always faced with the dilemma of exploration and exploitation. ϵ-Greedy is a trade-off between them. You are gonna implement Greedy and ϵ-Greedy. We combine these two policies in one function by treating Greedy as ϵ-Greedy where $\epsilon = 0$. Edit the function epsilon_greedy in ./RLalgs/utils.py.

```
from utils import epsilon_greedy
np.random.seed(6885) #Set the seed to cancel the randomness
q = np.random.normal(0, 1, size = 5)
###########################
# YOUR CODE STARTS HERE
greedy_action = epsilon_greedy(q,0,7225) #Use epsilon = 0 for Greedy
e_greedy_action = epsilon_greedy(q, 0.1 ,7225) #Use epsilon = 0.1
# YOUR CODE ENDS HERE
###########################
print('Values:')
print(q)
print('Greedy Choice =', greedy_action)
print('Epsilon-Greedy Choice =', e_greedy_action)
```

```
Values:
[ 0.61264537  0.27923079 -0.84600857  0.05469574 -1.09990968]
Greedy Choice = 0
Epsilon-Greedy Choice = 0
```

You should get the following results.
Values:
[0.61264537 0.27923079 -0.84600857 0.05469574 -1.09990968]
Greedy Choice = 0

3. Frozen Lake Environment

```
env = gym.make('FrozenLake-v0')
```

3.1 Derive Q value from V value

Edit function action_evaluation in ./RLalgs/utils.py.
TIPS: $q(s, a) = \sum_{s', r} p(s', r|s, a)(r + \gamma v(s'))$

```
from utils import action_evaluation
v = np.ones(16)
q = action_evaluation(env = env.env, gamma = 1, v = v)
print('Action values:')
print(q)
```

```
Action values:
[[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.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.33333333 1.33333333 1.33333333]
 [1.         1.         1.         1.        ]]
```

You should get Q values all equal to one except at State 14

Pseudo-codes of the following four algorithms can be found on Page 80, 83, 130, 131 of the Sutton's book.

3.2 Model-based RL algorithms

```
from utils import action_evaluation, action_selection, render
```

3.2.1 Policy Iteration

Edit the function policy_iteration and relevant functions in ./RLalgs/pi.py to implement the Policy

```
from pi import policy_iteration
V, policy, numIterations = policy_iteration(env = env.env, gamma = 1, max_iteration =
print('State values:')
print(V)
print('Number of iterations to converge =', numIterations)
```

```
State values:
[0.82352774 0.8235272  0.82352682 0.82352662 0.82352791 0.
 0.52941063 0.         0.82352817 0.82352851 0.76470509 0.
 0.         0.88235232 0.94117615 0.        ]
Number of iterations to converge = 7
```

You should get values close to:

State values:

[0.82352774 0.8235272 0.82352682 0.82352662 0.82352791 0.

0.52941063 0. 0.82352817 0.82352851 0.76470509 0.

0. 0.88235232 0.94117615 0.]

```
#Uncomment and run the following to evaluate your result, comment them when you gener
#Q = action_evaluation(env = env.env, gamma = 1, v = V)
#policy_estimate = action_selection(Q)
#render(env, policy_estimate)
```

3.2.2 Value Iteration

Edit the function value_iteration and relevant functions in ./RLalgs/vi.py to implement the Value Iteration Algorithm.

```
from vi import value_iteration
V, policy, numIterations = value_iteration(env = env.env, gamma = 1, max_iteration =
print('State values:')
print(V)
print('Number of iterations to converge =', numIterations)
```

```
State values:
[0.82352937 0.82352936 0.82352935 0.82352935 0.82352938 0.
 0.52941174 0.         0.82352938 0.82352939 0.76470586 0.
 0.         0.88235293 0.94117646 0.        ]
Number of iterations to converge = 500
```

You should get values close to:

State values:

[0.82352773 0.82352718 0.8235268 0.8235266 0.8235279 0.

0.52941062 0. 0.82352816 0.8235285 0.76470509 0.
0. 0.88235231 0.94117615 0.]

3.3 Model free RL algorithms

```
#Uncomment and run the following to evaluate your result, comment them when you gener
Q = action_evaluation(env = env.env, gamma = 1, v = V)
policy_estimate = action_selection(Q)
render(env, policy_estimate)
```

3.3.1 Q-Learning

Edit the function QLearning in ./RLalgs/ql.py to implement the Q-Learning Algorithm.

```
from ql import QLearning
Q = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e = 0.1)
print('Action values:')
print(Q)
```

```
Action values:
[[2.42552660e-01 1.66520496e-01 1.26361742e-01 1.15838832e-01]
 [1.11886160e-01 2.74990325e-02 3.26237730e-02 6.29792393e-02]
 [1.35774294e-01 4.39731199e-02 7.31857466e-02 3.38639129e-02]
 [7.41651258e-02 1.39998752e-02 0.00000000e+00 7.90028836e-06]
 [2.77021729e-01 4.28785406e-02 1.22460663e-01 8.02275771e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.14866962e-01 5.41309303e-02 5.17588111e-02 1.43581795e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.25357977e-01 2.91873501e-01 1.64006090e-01 2.06193487e-01]
 [2.28929962e-01 5.21484303e-01 2.35467893e-01 2.31292724e-01]
 [5.38842158e-01 2.57936543e-01 9.35360660e-02 1.03606031e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.31738099e-01 3.79939622e-01 6.44842526e-01 2.61955531e-01]
 [4.39352845e-01 8.15007934e-01 6.57678402e-01 3.75867041e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Generally, you should get non-zero action values on non-terminal states.

```
#Uncomment the following to evaluate your result, comment them when you generate the
#env = gym.make('FrozenLake-v0')
#policy_estimate = action_selection(Q)
#render(env, policy_estimate)
```

3.3.2 SARSA

Edit the function SARSA in ./RLalgs/sarsa.py to implement the SARSA Algorithm.

```
from sarsa import SARSA
Q = SARSA(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e = 0.1)
print('Action values:')
print(Q)
```

```
Action values:
[[0.03214816 0.07224132 0.04093962 0.03537322]
 [0.02371363 0.03553979 0.01779724 0.08632191]
 [0.1123205  0.05614293 0.05342828 0.03181692]
 [0.0500277  0.00300446 0.         0.00668523]
 [0.06877604 0.02585577 0.03230383 0.0125757 ]
 [0.         0.         0.         0.        ]
 [0.1143545  0.03529742 0.04573002 0.00714355]
 [0.         0.         0.         0.        ]
 [0.00294678 0.02256166 0.03258638 0.09958631]
 [0.06837357 0.05333112 0.16781575 0.06717138]
 [0.03564158 0.06946866 0.25583686 0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.06268968 0.05060381 0.36400159 0.17225095]
 [0.31443358 0.76432816 0.28629396 0.10311351]
 [0.         0.         0.         0.        ]]
```

Generally, you should get non-zero action values on non-terminal states.

```
#Uncomment the following to evaluate your result, comment them when you generate the
#env = gym.make('FrozenLake-v0')
#policy_estimate = action_selection(Q)
#render(env, policy_estimate)
```

3.4 Human

You can play this game if you are interested. See if you can get the frisbee either with or without the model.

```
from RLalgs.utils import human_play
#Uncomment and run the following to play the game, comment it when you generate the p
#env = gym.make('FrozenLake-v0')
#human_play(env)
```

4. Exploration VS. Exploitation

Try to reproduce Figure 2.2 (the upper one is enough) of the Sutton's book based on the experiment

```
# # Do the experiment and record average reward acquired in each time step
# ############################
# # YOUR CODE STARTS HERE

# # YOUR CODE ENDS HERE
# ############################
```

```
# Plot the average reward
############################
# YOUR CODE STARTS HERE
# YOUR CODE ENDS HERE
############################
```

You should get curves similar to that in the book.