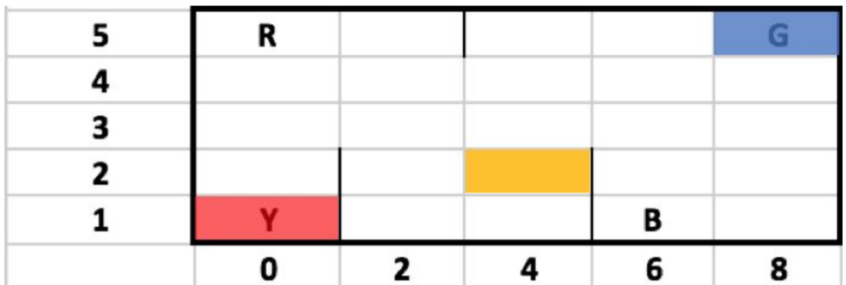
ELEN 6885 Reinforcement Learning Coding Assignment (Part 1, 2, 3)

Taxi Problem Overview

There are 4 locations (labeled by different letters) and your job is to pick up the passenger at one location and drop him off in another. You receive +20 points for a successful drop-off, and lose 1 point for every timestep it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions.



- blue: passenger
- magenta: destination
- yellow: empty taxi
- green: full taxi
- other letters: locations

Please put your code into the block marked by:

##############################
YOUR CODE STARTS HERE
YOUR CODE ENDS HERE

##############################
You should not edit anything outside of the block.

Playing with the environment

Run the cell below to get a feel for the environment by moving your agent(the taxi) by taking one of the actions at each step.

In [1]:

```
from gym.wrappers import Monitor
import gym
import random
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:

```python
"""
You can test your game now.
Input range from 0 to 5:
    0 : South (Down)
    1 : North (Up)
    2 : East (Right)
    3 : West (Left)
    4: Pick up
    5: Drop off
    6: exit_game
"""
GAME = "Taxi-v3"
env = gym.make(GAME)
env = Monitor(env, "taxi_simple", force=True)
s = env.reset()
steps = 100
for step in range(steps):
    env.render()
    action = int(input("Please type in the next action:"))
    if action==6:
        break
    s, r, done, info = env.step(action)
    print('state:',s)
    print('reward:',r)
    print('Is state terminal?:',done)
    print('info:',info)

# close environment and monitor
env.close()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

Please type in the next action:0
state: 487
reward: -1
Is state terminal?: False
info: {'prob': 1.0}
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
Please type in the next action:1
state: 387
reward: -1
Is state terminal?: False
info: {'prob': 1.0}
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
Please type in the next action:2
state: 387
reward: -1
Is state terminal?: False
info: {'prob': 1.0}
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (East)
Please type in the next action:3
state: 367
reward: -1
Is state terminal?: False
info: {'prob': 1.0}
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
Please type in the next action:4
state: 367
reward: -10
Is state terminal?: False
info: {'prob': 1.0}
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
Please type in the next action:5
state: 367
reward: -10
Is state terminal?: False
info: {'prob': 1.0}
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
```

```
|Y| : |B: |
+---------+
  (Dropoff)
Please type in the next action:6
```

1.1 Incremental implementation of average

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

In [24]:

```
def estimate(OldEstimate, StepSize, Target):
    '''An incremental implementation of average.
    OldEstimate : float
    StepSize : float
    Target : float
    '''
    NewEstimate = OldEstimate + StepSize * (Target - OldEstimate)
    return NewEstimate
```

In [18]:

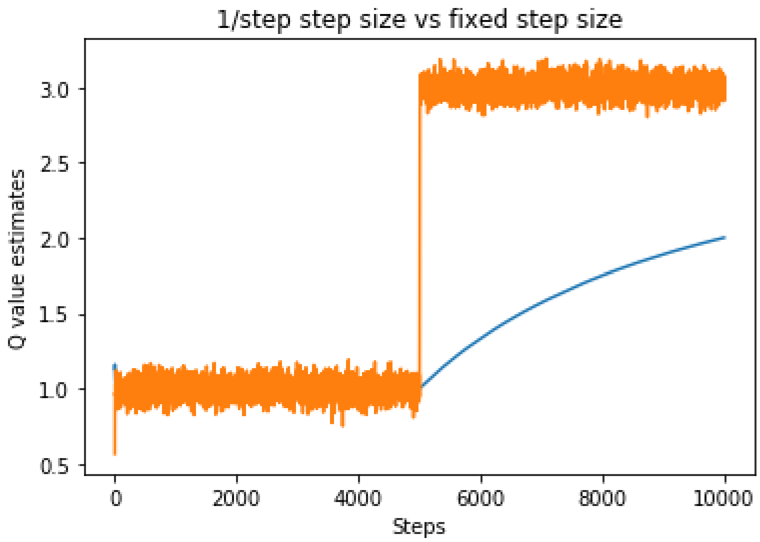
```
random.seed(6885)
numTimeStep = 10000
q_h = np.zeros(numTimeStep + 1) # Q Value estimate with 1/step step size
q_f = np.zeros(numTimeStep + 1) # Q value estimate with fixed step size
FixedStepSize = 0.5 #A large number to exaggerate the difference
for step in range(1, numTimeStep + 1):
    if step < 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:]
print(q_h,q_f)
```

```
[1.13189822 1.16039193 1.10184497 ... 2.00043755 2.00053969 2.00063503] [0.
56594911 0.87741738 0.93108421 ... 3.07677113 3.04928666 3.00159284]
```

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

In [21]:

```
import matplotlib.pyplot as plt
############################
# YOUR CODE STARTS HERE
plt.plot(q_h,label='1/step step size')
plt.plot(q_f,label='fixed step size')
plt.title('1/step step size vs fixed step size')
plt.xlabel('Steps')
plt.ylabel('Q value estimates')
plt.show()

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



1.2 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 supposed to 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 the following block.

In [2]:

```python
def epsilon_greedy(value, e, seed = None):
    '''
    Implement Epsilon-Greedy policy.

    Inputs:
    value: numpy ndarray
    A vector of values of actions to choose from
    e: float
    Epsilon
    seed: None or int
    Assign an integer value to remove the randomness

    Outputs:
    action: int
    Index of the chosen action
    '''
    assert len(value.shape) == 1
    assert 0 <= e <= 1

    if seed != None:
        np.random.seed(seed)

    ############################
    # YOUR CODE STARTS HERE
    if random.random() >= e:
        action = np.argmax(value)
    else:
        action = np.random.choice(len(value))

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

    return action
```

In [33]:

```python
np.random.seed(6885) #Set the seed forreproducability
q = np.random.normal(0, 1, size = 5)
############################
# YOUR CODE STARTS HERE
greedy_action = epsilon_greedy(q, 0, seed = 6885)
e_greedy_action = epsilon_greedy(q, 0.1, seed = 6885)
# 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
Epsilon-Greedy Choice = 0

1.3 Exploration VS. Exploitation

Try to reproduce Figure 2.2 (the upper one is enough) of the Sutton's book based on the experiment described in Chapter 2.3.

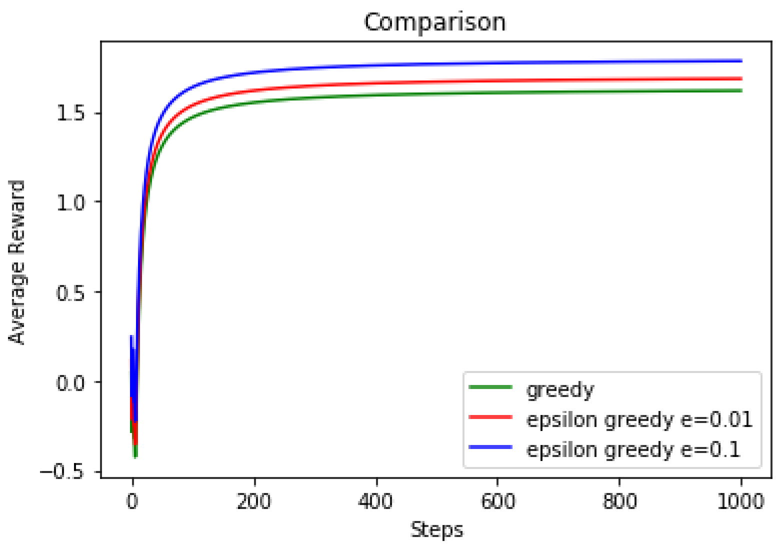
In [78]:

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

def average_reward(eps, num_of_steps, num_of_trials, num_actions):
    reward_hist = []
    for i in range(num_of_trials):

        q = np.zeros(num_actions)
        trial_reward = np.zeros(num_of_steps)
        act_count = np.zeros(num_actions)


        for step in range(num_of_steps):

            if step < num_actions:
                np.random.seed(600+10*step)
                action = step
                reward = np.random.normal(0, 1)
            else:
                action = epsilon_greedy(q, eps, seed = 6885)
                reward = q[action]
            act_count[action] += 1
            q[action] = estimate(q[action], 1/act_count[action], reward)
            trial_reward[step] = reward

        reward_hist.append(np.cumsum(trial_reward) / (np.arange(num_of_steps)
    return np.mean(reward_hist, axis=0)

num_of_steps = 1000
num_of_trials = 2000
num_actions = 10
a = average_reward(0, num_of_steps, num_of_trials, num_actions)
print(a.mean())
b = average_reward(0.01, num_of_steps, num_of_trials, num_actions)
print(b.mean())
c = average_reward(0.1, num_of_steps, num_of_trials, num_actions)
print(c.mean())

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

```
1.5400882444923836
1.606801787321578
1.7074325371044137
```

In [82]:

```
# Plot the average reward
############################
# YOUR CODE STARTS HERE
plt.plot(a ,'g', label='greedy')
plt.plot(b ,'r',label='epsilon greedy e=0.01')
plt.plot(c ,'b',label='epsilon greedy e=0.1')
plt.title('Comparison')
plt.xlabel('Steps')
plt.ylabel('Average Reward')
plt.legend()
plt.show()

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



Question 2

In this question, you will implement the value iteration and policy iteration algorithms to solve the Taxi game problem

2.1 Model-based RL: value iteration

For this part, you need to implement the helper functions action_evaluation(env, gamma, v), and extract_policy(env, v, gamma) in utils.py. Understand action_selection(q) which we have implemented.
Use these helper functions to implement the value iteration algorithm below.

In [42]:

```python
import numpy as np
from helpers import utils
def value_iteration(env, gamma, max_iteration, theta):
    """
    Implement value iteration algorithm. You should use extract_policy to for

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nextstate,
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    gamma: float
            Discount factor.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns:
    ----------
    value function: np.ndarray
    policy: np.ndarray
    """
    V = np.zeros(env.nS)

    ############################
    # YOUR CODE STARTS HERE
    for i in range(max_iteration):
        delta = 0
        for s in range(env.nS):
            v = V[s]
            Q = utils.action_evaluation(env, gamma, V)
            V[s] = np.max(Q[s])
            delta = max(delta, abs(v-V[s]))

        if delta < theta:
            break

    policy = utils.extract_policy(env, V, gamma)
    # YOUR CODE ENDS HERE
    ############################

    return V, policy
```

After implementing the above function, read and understand the functions implemented in evaluation_utils.py, which we will use to evaluate our value iteration policy

In [48]:
```
from helpers import evaluation_utils
import gym
GAME = "Taxi-v3"
env = gym.make(GAME)
V_vi, policy_vi = value_iteration(env, gamma=0.95, max_iteration=6000, theta=
# visualize how the agent performs with the policy generated from value itera
evaluation_utils.render_episode(env, policy_vi)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
|_| : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
|_: | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Episode reward: 15.000000
```

In [49]:
```
# evaluate the performance of value iteration over 100 episodes
evaluation_utils.avg_performance(env, policy_vi)
```

Out[49]:
```
8.555555555555555
```

2.2 Model-based RL: policy iteration

In this part, you are supposed to implement policy iteration to solve the Taxi game problem.

In [85]:

```
from helpers import utils
def policy_iteration(env, gamma, max_iteration, theta):
    """Implement Policy iteration algorithm.

    You should use the policy_evaluation and policy_improvement methods to
    implement this method.

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nextstate,
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    gamma: float
            Discount factor.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns:
    ----------
    value function: np.ndarray
    policy: np.ndarray
    """

    V = np.zeros(env.nS)
    policy = np.zeros(env.nS, dtype=int)
    ############################
    # YOUR CODE STARTS HERE
    for i in range(max_iteration):
        V = policy_evaluation(env, policy, gamma, theta)
        policy, policy_stable = policy_improvement(env, V, policy, gamma)
        #print(policy_stable)
        if policy_stable:
            break

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

    return V, policy


def policy_evaluation(env, policy, gamma, theta):
    """Evaluate the value function from a given policy.

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nextstate,
            env.nS: int
```

```
                    number of states
            env.nA: int
                    number of actions

    gamma: float
            Discount factor.
    policy: np.array
            The policy to evaluate. Maps states to actions.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns
    -------
    value function: np.ndarray
            The value function from the given policy.
    """
    V = np.zeros(env.nS)
    ############################
    # YOUR CODE STARTS HERE
    while True:
        delta = 0
        for s in range(env.nS):
            v = 0

            for a in range(env.nA):
                if a == policy[s]:
                    action_prob =1
                else:
                    action_prob = 0
                for  prob, next_state, reward, done in env.P[s][a]:
                    v += action_prob * prob * (reward + gamma * V[next_state]
            delta = max(delta, abs(v-V[s]))
            V[s] = v

        if delta < theta:
            break

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

    return V


def policy_improvement(env, value_from_policy, policy, gamma):
    """Given the value function from policy, improve the policy.

    Parameters
    ----------
    env: OpenAI env
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nextstate,
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
```

```
    value_from_policy: np.ndarray
            The value calculated from the policy
    policy: np.array
            The previous policy.
    gamma: float
            Discount factor.

    Returns
    -------
    new policy: np.ndarray
            An array of integers. Each integer is the optimal action to take
            in that state according to the environment dynamics and the
            given value function.
    stable policy: bool
            True if the optimal policy is found, otherwise false
    """
    ############################
    # YOUR CODE STARTS HERE
    policy_stable = True
    new_policy = utils.extract_policy(env, value_from_policy, gamma)
    for s in range(env.nS):
        old_action = policy[s]

        if old_action != new_policy[s]:
            policy_stable = False
    # YOUR CODE ENDS HERE
    ############################

    return new_policy, policy_stable
```

In [3]:

```
## Testing out policy iteration policy for one episode
GAME = "Taxi-v3"
#evaluation_utils.render_episode(env, policy_vi)
env = gym.make(GAME)
V_pi, policy_pi = policy_iteration(env, gamma=0.95, max_iteration=6000, theta
```

In [88]:

```
# visualize how the agent performs with the policy generated from policy iter
evaluation_utils.render_episode(env, policy_pi)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+


+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
|_| : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| :_: : : |
| | : | : |
|Y| : |B: |
```

```
+---------+
  (East)
+---------+
|R: | : :G|
| : | : : |
| : :_: : |
| | : | : |
|Y| : |B: |
+---------+
  (East)
+---------+
|R: | : :G|
| : | : : |
| : : :_: |
| | : | : |
|Y| : |B: |
+---------+
  (East)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : |_: |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Episode reward: 10.000000
```

In [91]:

```
# evaluate the performance of policy iteration over 100 episodes
print(evaluation_utils.avg_performance(env, policy_pi))
```

```
8.373737373737374
```

Part 3: Q-learning and SARSA

3.1 Model-free RL: Q-learning

In this part, you will implement Q-learning.

In [87]:

```
def QLearning(env, num_episodes, gamma, lr, e):
    """Implement the Q-learning algorithm following the epsilon-greedy explor
    Update Q at the end of every episode.

    Parameters
    ----------
    env: gym.core.Environment
      Environment to compute Q function
    num_episodes: int
      Number of episodes of training.
    gamma: float
      Discount factor.
    learning_rate: float
      Learning rate.
    e: float
      Epsilon value used in the epsilon-greedy method.


    Returns
    -------
    np.array
      An array of shape [env.nS x env.nA] representing state, action values
    """
    Q = np.zeros((env.nS, env.nA))
    ############################
    # YOUR CODE STARTS HERE
    avg_ep_rew = []
    total_ep_rew = []
    for ep in range(num_episodes):
        state = env.reset()
        done = False
        avg_rew, step, total_rew = 0, 0, 0
        while not done:
            action = epsilon_greedy(Q[state], e, seed = None)
            next_state, reward, done, info = env.step(action)
            Q[state][action] += lr*(reward + gamma*np.max(Q[next_state]) - Q[
            state = next_state
            step += 1
            total_rew = total_rew + reward
            avg_rew = total_rew/step

        avg_ep_rew.append(avg_rew)
        total_ep_rew.append(total_rew)

        if ep% 100 ==0 or ep == num_episodes-1:
            avg = np.mean(avg_ep_rew)
            tot = np.sum(total_ep_rew)
            print("Episode no.: {} . Average reward: {} . Total Reward: {}".f

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

    return Q, total_ep_rew
```

In [95]:

```python
Q, total_ep_rew = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr
print('Action values:')
print(Q)
```

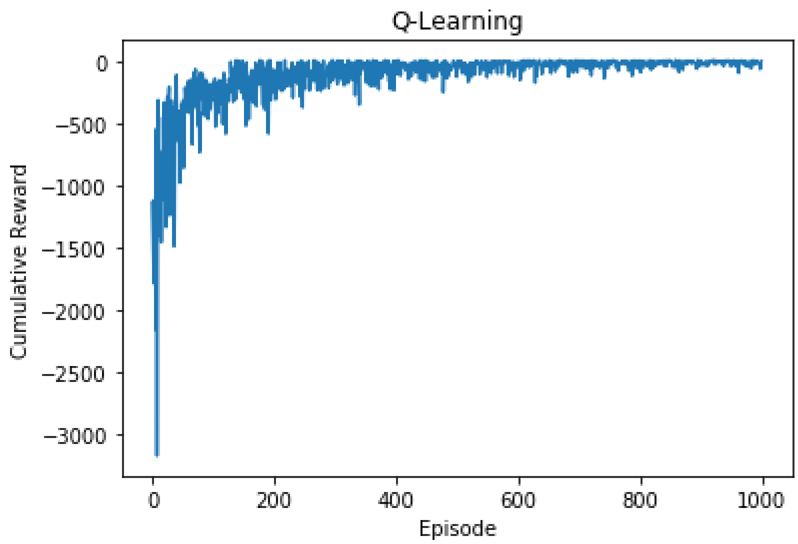
```
Episode no.: 0 . Average reward: -2.6619718309859155 . Total Reward: -1134
Episode no.: 100 . Average reward: -1.431777806330508 . Total Reward: -5905
0
Episode no.: 200 . Average reward: -1.2541495735739723 . Total Reward: -784
48
Episode no.: 300 . Average reward: -1.1488270192601 . Total Reward: -90210
Episode no.: 400 . Average reward: -1.0407502041754746 . Total Reward: -973
42
Episode no.: 500 . Average reward: -0.9548573814726444 . Total Reward: -102
132
Episode no.: 600 . Average reward: -0.8798196258776297 . Total Reward: -105
289
Episode no.: 700 . Average reward: -0.7852422183724562 . Total Reward: -107
356
Episode no.: 800 . Average reward: -0.7159295515370775 . Total Reward: -108
834
Episode no.: 900 . Average reward: -0.6472249841825299 . Total Reward: -109
838
Episode no.: 999 . Average reward: -0.5897991943358961 . Total Reward: -110
518
Action values:
[[ 0.          0.          0.          0.          0.          0.        ]
 [-3.50395995 -3.79829428 -3.90398205 -3.87008063  9.76869791 -4.78318205]
 [-0.82356986 -1.8        -1.88581086 -1.79919    14.74340144 -2.95375687]
 ...
 [-1.09365895 -0.97311181 -1.1        -1.07987579 -2.         -2.86565895]
 [-2.69911938 -2.66520158 -2.7        -0.54073504 -4.59829421 -3.82462824]
 [-0.1        -0.1        -0.1         6.82965287 -1.         -1.        ]]
```

In [ ]:

```python
#Uncomment the following to evaluate your result, comment them when you gener
# from helpers.utils import action_selection
# from helpers.evaluation_utils import render_episode
# #env = gym.make('Taxi-v3')
# policy_estimate = action_selection(Q)
# render_episode(env, policy_estimate)
render_episode_Q(env, Q)
```

In [96]:

```
plt.plot(total_ep_rew)
plt.title('Q-Learning')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
#plt.legend()
plt.show()
```



3.2 Model-free RL: SARSA

In this part, you will implement Sarsa.

In [83]:

```python
def SARSA(env, num_episodes, gamma, lr, e):
    """Implement the SARSA algorithm following epsilon-greedy exploration.
    Update Q at the end of every episode.

    Parameters
    ----------
    env: gym.core.Environment
      Environment to compute Q function
    num_episodes: int
      Number of episodes of training
    gamma: float
      Discount factor.
    learning_rate: float
      Learning rate.
    e: float
      Epsilon value used in the epsilon-greedy method.


    Returns
    -------
    np.array
      An array of shape [env.nS x env.nA] representing state-action values
    """

    Q = np.zeros((env.nS, env.nA))
    ############################
    # YOUR CODE STARTS HERE
    avg_ep_rew = []
    total_ep_rew = []
    for ep in range(num_episodes):
        state = env.reset()
        action = epsilon_greedy(Q[state], e, seed = None)
        done = False
        avg_rew, step, total_rew = 0, 0, 0

        while not done:
            next_state, reward, done, info = env.step(action)
            next_action = epsilon_greedy(Q[next_state], e, seed = None)
            Q[state][action] += lr*(reward + gamma*Q[next_state][next_action]

            state = next_state
            action = next_action
            step += 1
            total_rew = total_rew + reward
            avg_rew = total_rew/step

        avg_ep_rew.append(avg_rew)
        total_ep_rew.append(total_rew)

        if ep% 100 ==0 or ep == num_episodes-1:
            avg = np.mean(avg_ep_rew)
            tot = np.sum(total_ep_rew)
            print("Episode no.: {} . Average reward: {} . Total Reward: {}".f

    return Q, total_ep_rew
```

In [84]:

```python
import time
def render_episode_Q(env, Q):
    """Renders one episode for Q functionon environment.

      Parameters
      ----------
      env: gym.core.Environment
        Environment to play Q function on.
      Q: np.array of shape [env.nS x env.nA]
        state-action values.
    """

    episode_reward = 0
    state = env.reset()
    done = False
    while not done:
        env.render()
        time.sleep(0.5)
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        episode_reward += reward

    print ("Episode reward: %f" %episode_reward)
```

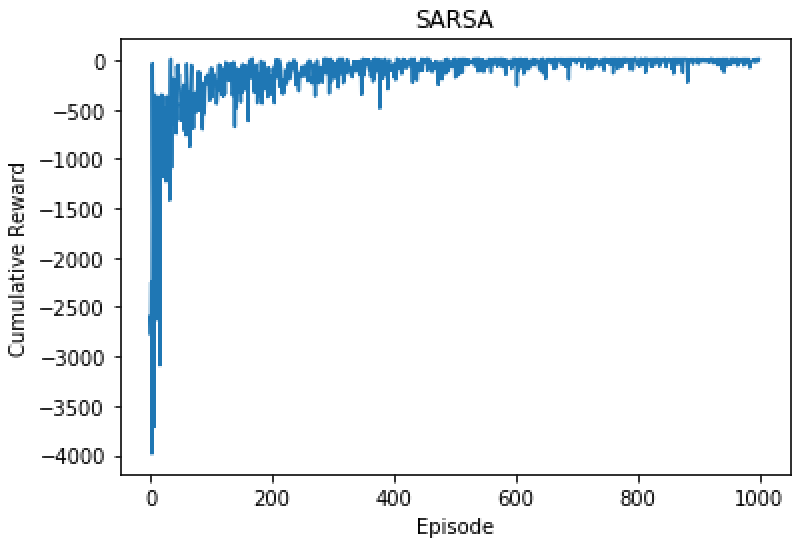
In [104]:

```python
GAME = "Taxi-v3"
env = gym.make(GAME)
Q, total_ep_rew = SARSA(env = env.env, num_episodes = 1000, gamma = 1, lr = 0
print('Action values:')
print(Q)
```

```
Episode no.: 0 . Average reward: -2.920134983127109 . Total Reward: -2596
Episode no.: 100 . Average reward: -1.4210930952781753 . Total Reward: -662
37
Episode no.: 200 . Average reward: -1.3135414698287307 . Total Reward: -865
72
Episode no.: 300 . Average reward: -1.2206595544333916 . Total Reward: -988
58
Episode no.: 400 . Average reward: -1.1074879884779836 . Total Reward: -106
981
Episode no.: 500 . Average reward: -1.0290211160250329 . Total Reward: -113
151
Episode no.: 600 . Average reward: -0.9248544111182803 . Total Reward: -116
394
Episode no.: 700 . Average reward: -0.8418434786745939 . Total Reward: -119
592
Episode no.: 800 . Average reward: -0.753543423234892 . Total Reward: -1211
71
Episode no.: 900 . Average reward: -0.6778790855558101 . Total Reward: -122
632
Episode no.: 999 . Average reward: -0.6242465907041139 . Total Reward: -123
847
Action values:
[[ 0.          0.          0.          0.          0.          0.        ]
 [-4.5220393  -4.071337   -4.34050694 -4.40562881  0.67758558 -4.45934676]
 [-1.76747381 -1.71286283 -1.70035131  0.80003852 10.7687492  -1.58941309]
 ...
 [-1.14190497 -1.01884737 -1.26971431 -1.18851604 -2.78200329 -1.91      ]
 [-3.52232126 -3.42777602 -3.44685268 -3.41276205 -4.92637215 -5.80134915]
 [-0.19       -0.2        -0.19        3.6181     -2.719      -1.91      ]]
```

In [105]:

```
plt.plot(total_ep_rew)
plt.title('SARSA')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
#plt.legend()
plt.show()
```



In [ ]:

```
# Uncomment the following to evaluate your result, comment them when you gene
# env = gym.make('Taxi-v2')
# policy_estimate = action_selection(Q)
# render(env, policy_estimate)
```