Question 1

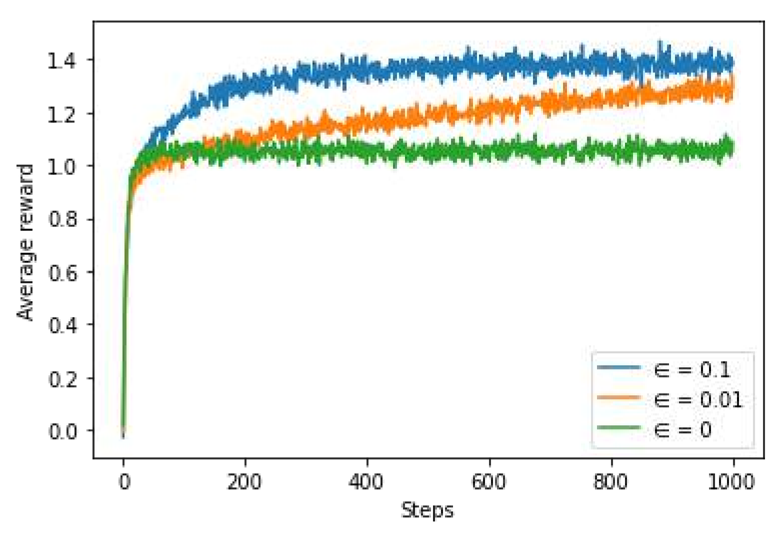
In [15]:
```python
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Image
```

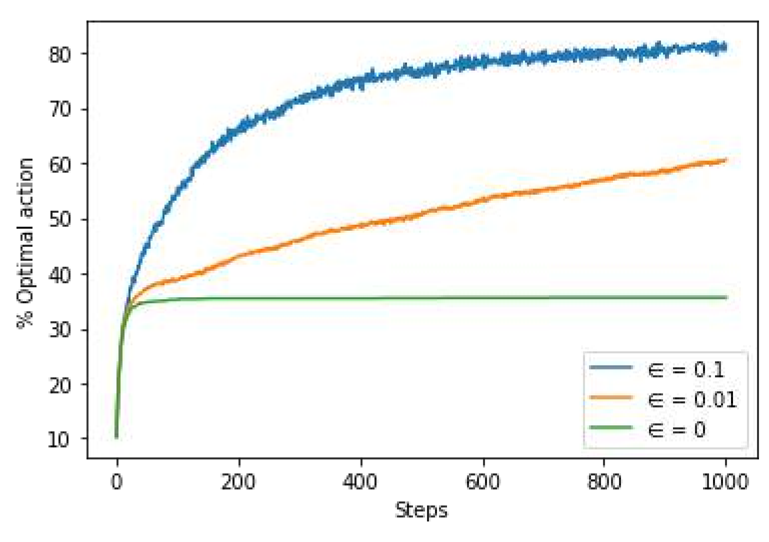
In [16]:
```python
def ten_armed_testbed(eps):
    # Initialize
    arms = 10
    epsilon = eps
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]
    abs_error = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update Qt, Nt, rewards, absolute error and optimal_arm
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1
            else:
                true_rew = np.random.normal(true_At,1)
                abs_error[iteration] += abs(Rt-true_rew)

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes
        abs_error[iteration] /= episodes

    return rewards,optimal_arm, abs_error
```

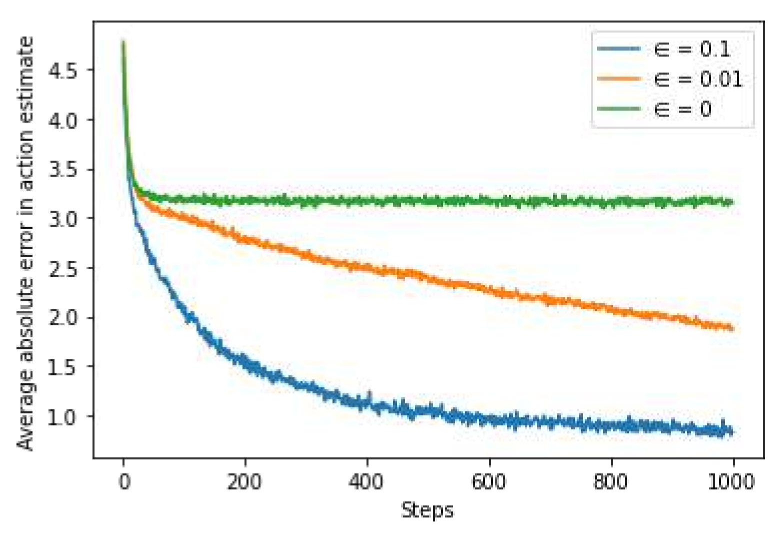
In [17]:
```python
def ten_armed_testbed_variable_epsilon():
    # Initialize
    arms = 10
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]
    abs_error = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At
            # epsilon changes as 1/0.1*(iteration+1)

            epsilon = 1/(0.1*(iteration+1))

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update Qt, Nt, rewards, abs_error and optimal_arm
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1
            else:
                true_rew = np.random.normal(true_At,1)
                abs_error[iteration] += abs(Rt-true_rew)

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes
        abs_error[iteration] /= episodes

    return rewards,optimal_arm,abs_error
```

Plots of Average Rewards, % Optimal Action, Average absolute error in action estimate for ∈ = 0, 0.01, 0.1

In [18]:
```python
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

rewards1,optimal_arm1,abs_error1 = ten_armed_testbed(0)
rewards2,optimal_arm2,abs_error2 = ten_armed_testbed(0.01)
rewards3,optimal_arm3,abs_error3 = ten_armed_testbed(0.1)

# plotting figure 1 - Average rewards
plt.figure()
plt.plot(time, rewards3, label='∈ = 0.1')
plt.plot(time, rewards2, label='∈ = 0.01')
plt.plot(time, rewards1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.legend()
plt.show()

#plotting figure 2 - % Optimal action
plt.figure()
plt.plot(time, optimal_arm3, label='∈ = 0.1')
plt.plot(time, optimal_arm2, label='∈ = 0.01')
plt.plot(time, optimal_arm1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('% Optimal action')
plt.legend()
plt.show()

#plotting figure 3 - Average absolute error in action estimate
plt.figure()
plt.plot(time, abs_error3, label='∈ = 0.1')
plt.plot(time, abs_error2, label='∈ = 0.01')
plt.plot(time, abs_error1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('Average absolute error in action estimate')
plt.legend()
plt.show()
```







Plots of Average Rewards, % Optimal Action, Average absolute error in action estimate for ∈ that changes with time (∈(t) = 1/(0.1*t) , t ≥ 1)

Justification that ∈(t) satisfies equation 2.7 :

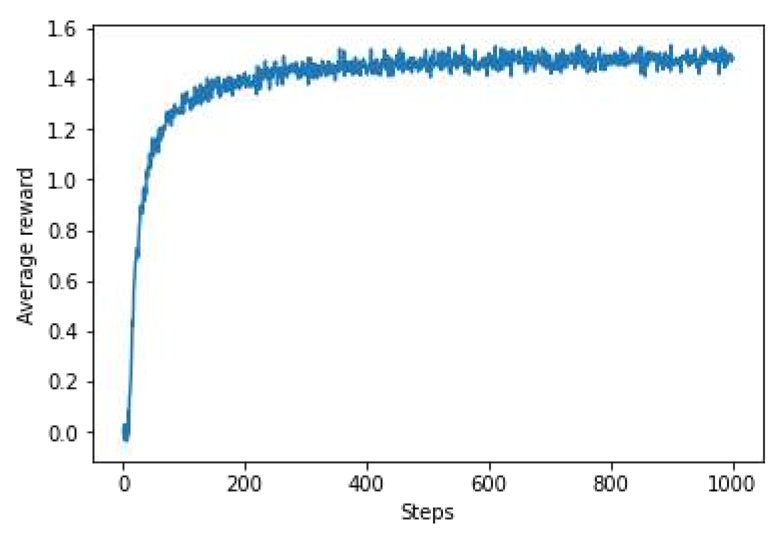
In [19]:
```python
Image(filename='justification_epsilon.png')
```

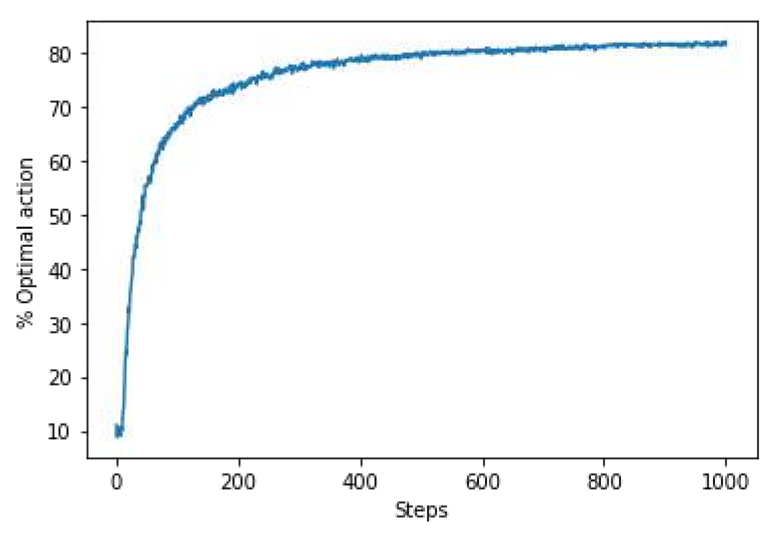
Out[19]:

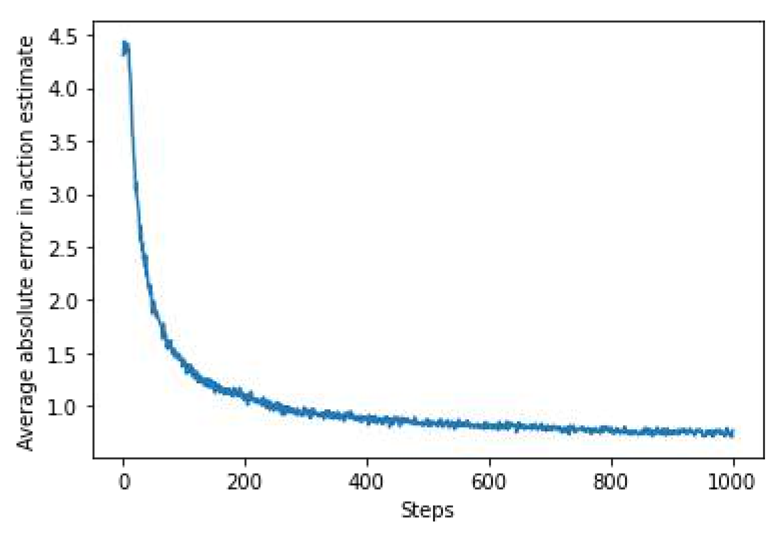
$$\sum_{t=1}^{\infty} \in(t) = 10 * \sum_{t=1}^{\infty} \frac{1}{t} = \infty$$

$$\sum_{t=1}^{\infty} \in(t)^2 = 100 * \sum_{t=1}^{\infty} \frac{1}{t^2} \approx \frac{100 * \Pi^2}{6} < \infty$$

In [20]:
```python
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

rewards,optimal_arm,abs_error = ten_armed_testbed_variable_epsilon()

# plotting figure 1 - Average rewards
plt.figure()
plt.plot(time, rewards)
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.show()

#plotting figure 2 - % Optimal action
plt.figure()
plt.plot(time, optimal_arm)
plt.xlabel('Steps')
plt.ylabel('% Optimal action')
plt.show()

#plotting figure 3 - Average absolute error in action estimate
plt.figure()
plt.plot(time, abs_error)
plt.xlabel('Steps')
plt.ylabel('Average absolute error in action estimate')
plt.show()
```







In [ ]:

Question 2

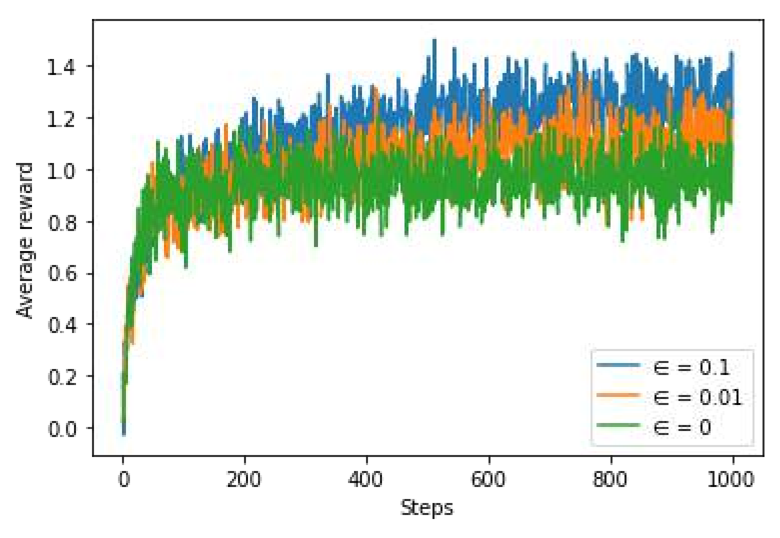
In [21]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

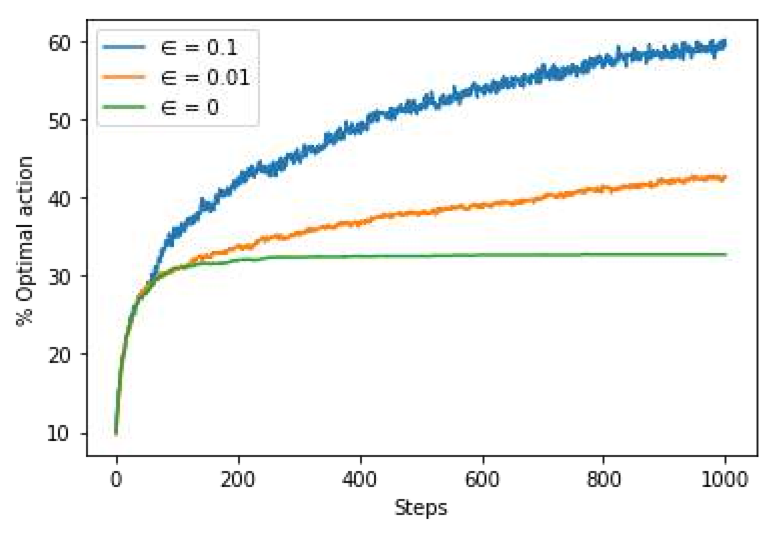
In [22]:
```python
def ten_armed_testbed(eps):
    # Initialize
    arms = 10
    epsilon = eps
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]
    abs_error = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],4)

            # Step 3 : Update Qt, Nt, rewards, absolute error and optimal_arm
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1
            else:
                true_rew = np.random.normal(true_At,4)
                abs_error[iteration] += abs(Rt-true_rew)

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes
        abs_error[iteration] /= episodes

    return rewards,optimal_arm, abs_error
```

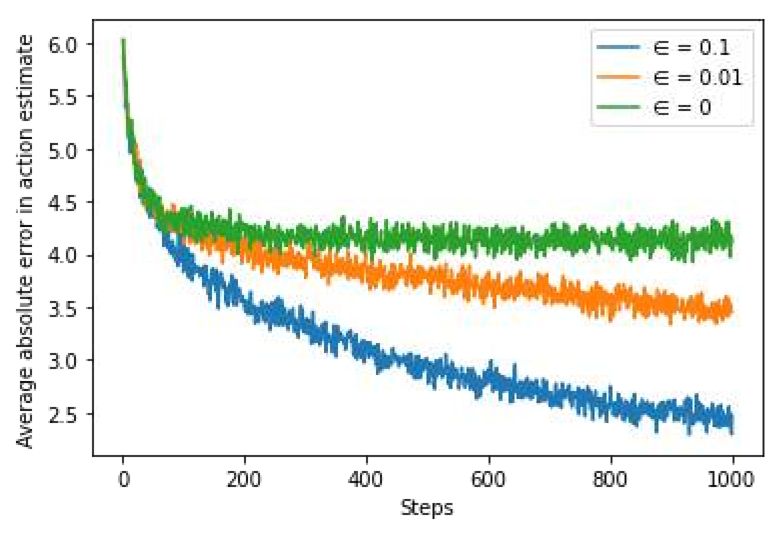
In [23]:
```python
def ten_armed_testbed_variable_epsilon():
    # Initialize
    arms = 10
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]
    abs_error = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At
            # epsilon changes as 1/0.1*(iteration+1)

            epsilon = 1/(0.1*(iteration+1))

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],4)

            # Step 3 : Update Qt, Nt, rewards, abs_error and optimal_arm
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1
            else:
                true_rew = np.random.normal(true_At,4)
                abs_error[iteration] += abs(Rt-true_rew)

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes
        abs_error[iteration] /= episodes

    return rewards,optimal_arm,abs_error
```

Plots of Average Rewards, % Optimal Action, Average absolute error in action estimate for ∈ = 0, 0.01, 0.1

In [24]:
```python
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

rewards1,optimal_arm1,abs_error1 = ten_armed_testbed(0)
rewards2,optimal_arm2,abs_error2 = ten_armed_testbed(0.01)
rewards3,optimal_arm3,abs_error3 = ten_armed_testbed(0.1)

# plotting figure 1 - Average rewards
plt.figure()
plt.plot(time, rewards3, label='∈ = 0.1')
plt.plot(time, rewards2, label='∈ = 0.01')
plt.plot(time, rewards1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.legend()
plt.show()

#plotting figure 2 - % Optimal action
plt.figure()
plt.plot(time, optimal_arm3, label='∈ = 0.1')
plt.plot(time, optimal_arm2, label='∈ = 0.01')
plt.plot(time, optimal_arm1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('% Optimal action')
plt.legend()
plt.show()

#plotting figure 3 - Average absolute error in action estimate
plt.figure()
plt.plot(time, abs_error3, label='∈ = 0.1')
plt.plot(time, abs_error2, label='∈ = 0.01')
plt.plot(time, abs_error1, label='∈ = 0')
plt.xlabel('Steps')
plt.ylabel('Average absolute error in action estimate')
plt.legend()
plt.show()
```

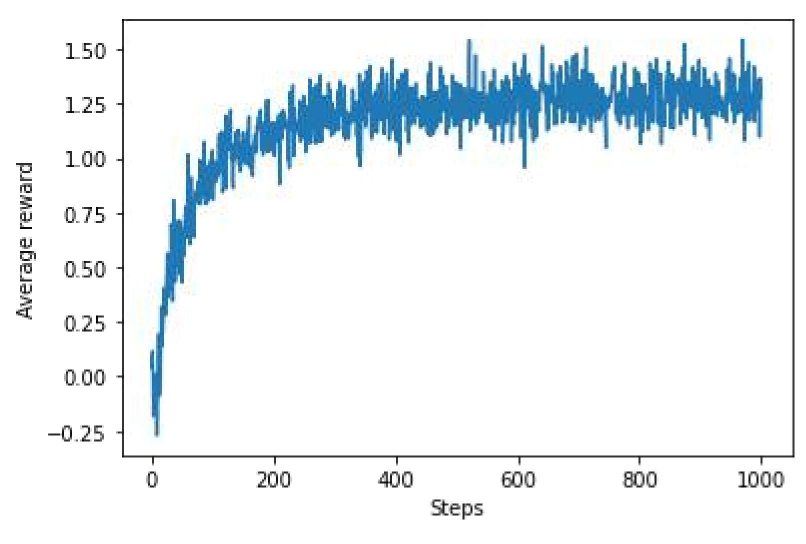


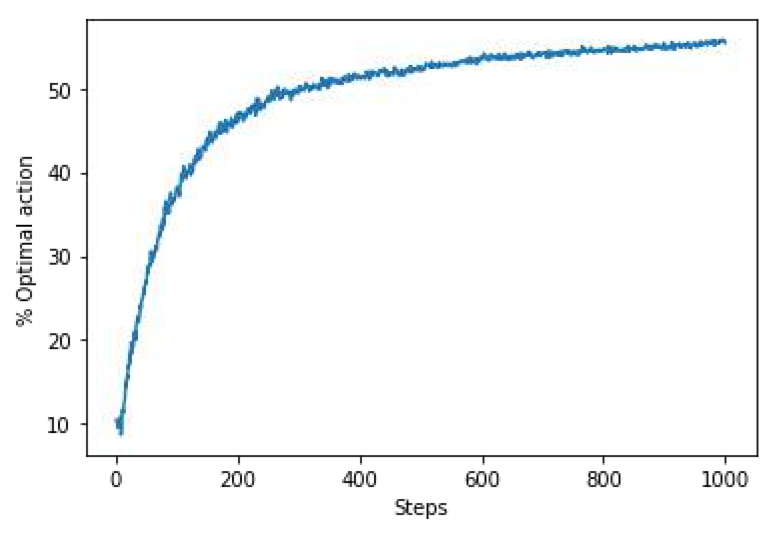


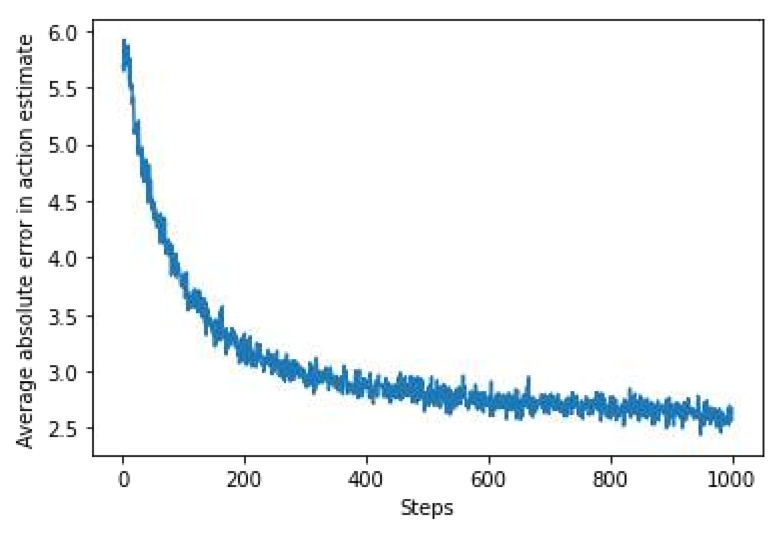


Plots of Average Rewards, % Optimal Action, Average absolute error in action estimate for ∈ that changes with time (∈(t) = 1/t , t ≥ 1)

In [25]:
```python
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

rewards,optimal_arm,abs_error = ten_armed_testbed_variable_epsilon()

# plotting figure 1 - Average rewards
plt.figure()
plt.plot(time, rewards)
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.show()

#plotting figure 2 - % Optimal action
plt.figure()
plt.plot(time, optimal_arm)
plt.xlabel('Steps')
plt.ylabel('% Optimal action')
plt.show()

#plotting figure 3 - Average absolute error in action estimate
plt.figure()
plt.plot(time, abs_error)
plt.xlabel('Steps')
plt.ylabel('Average absolute error in action estimate')
plt.show()
```







In [ ]:

<u>Question 3</u>

In terms of probability of selecting the best action, ∈ = 0.01 would perform better in the long run.
∈ = 0 would imply that the algorithm does not explore and only exploits the current estimates of Qt(a) for all arms a which might not be optimal estimates. Thus, ∈ = 0 does not perform well in the long run.
∈ = 0.1 would imply that the algorithm exploits 90% ((1-∈)*100) and explores 10% (∈*100) of the time. Thus, the **algorithm selects the optimal action 91%** (1-∈+∈/10) **of the time**.
∈ = 0.01 would imply that the algorithm exploits 99% and explores 1% of the time. Thus, the **algorithm selects the optimal action 99.1%of the time**. This is desirable as in the long run, the greedy estimates would be close to the true estimates and exploration would not be useful as we would want to exploit our current estimates to maximize expected reward. Since the probability of selecting the best action is higher for ∈ = 0.01 (by 8.1%), ∈ = 0.01 would perform better in the long run.

In terms of cumulative reward, ∈ = 0.01 would perform better in the long run. Since ∈ = 0 does not explore and only exploits current estimates of values of action, ∈ = 0 does not perform well in the long run.
Cumulative reward can be expressed as Probability of getting reward from greedy algorithm * expected reward value from greedy algorithm + probability of getting reward randomly * expected reward value from random selection of arms.
In the long run, expected reward value from greedy algorithm would be close to the true reward value (qt(a) = q*(a) as t tends to infinity). From the figure, the expected reward in the long run is approximately 1.5
Also, expected reward value from random selection of arms is 0 (as E[x] = 0 for a normal random variable)
For ∈ = 0.01, cumulative reward = (1-∈)*(1.5) + (∈)*(0) = 0.99*1.5 = 1.485
Similarly, **cumulative reward for ∈ = 0.1 is 0.9*1.5 = 1.35**
Since cumulative reward is higher for ∈ = 0.01 (by 0.135), ∈ = 0.01 would perform better in the long run.

Epsilon that changes with time, ∈(t) = 1/(0.1)*t where t>=1 is the timestep performs better than constant ∈ for both cumulative reward and probability of selecting the best action in the long run as the algorithm selects the optimal action (1-(9/10)*∈)*100 % of the time ie (1-9/(100*t))*100 % of the time. As t tends to infinity, the probability of selecting the best action tends to 100%. Also, as timestep increases, the algorithm explores less and exploits more which leads to better cumulative reward in comparison with constant epsilon

Question 4

Date ______
Page ______

Sample Mean

Let R_i denote the reward received after the i^{th} selection of arm a and let Q_n denote the estimate of its action value after it has been selected $n-1$ times

$$Q_{n+1} = \frac{R_1 + R_2 + \cdots + R_n}{n} \qquad (n \geq 1)$$

$$Q_{n+1} = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n} + \frac{R_n}{n}$$

$$= \left(\frac{n-1}{n}\right)\left(\frac{R_1 + \cdots + R_{n-1}}{n-1}\right) + \frac{R_n}{n}$$

$$= \left(\frac{n-1}{n}\right) Q_n + \frac{R_n}{n}$$

$$Q_{n+1} = \frac{R_n}{n} + \left(\frac{n-1}{n}\right) Q_n$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right) Q_n$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right)\left[\frac{R_{n-1}}{n-1} + \left(1-\frac{1}{n-1}\right) Q_{n-1}\right]$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right)\frac{R_{n-1}}{n-1} + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right) Q_{n-1}$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right)\frac{R_{n-1}}{n-1} + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\left[\frac{R_{n-2}}{n-2} + \left(1-\frac{1}{n-2}\right) Q_{n-2}\right]$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right)\frac{R_{n-1}}{n-1} + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\frac{R_{n-2}}{n-2} + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\left(1-\frac{1}{n-2}\right) Q_{n-2}$$

$$= \frac{R_n}{n} + \left(1-\frac{1}{n}\right)\frac{R_{n-1}}{n-1} + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\frac{R_{n-2}}{n-2} + \cdots + \left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\left(1-\frac{1}{n-2}\right)\cdots\left(1-\frac{1}{2}\right)\left(1-\frac{1}{1}\right) Q_1$$

$\therefore$ The coefficient of $Q_1 = 0$

$\therefore$ Q_{n+1} is independent of Q_1 $\forall\, n \geq 1$

$\therefore$ Sample Mean for an arm a is not influenced by initial choice of Q_1 $\forall\, a$.

Constant Step-size α

$$Q_{n+1} = Q_n + \alpha(R_n - Q_n)$$

$$= \alpha R_n + Q_n(1-\alpha)$$

$$= \alpha R_n + (1-\alpha)(\alpha R_{n-1} + Q_{n-1}(1-\alpha))$$

$$= \alpha R_n + \alpha(1-\alpha)R_{n-1} + (1-\alpha)^2 Q_{n-1}$$

$$= \alpha R_n + \alpha(1-\alpha)R_{n-1} + (1-\alpha)^2 Q_{n-1} + \cdots$$

$$\cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1$$

$$\therefore Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

For a smaller α, $1-\alpha$ is larger & thus dependence on Q_1 increases as $(1-\alpha)^n$ increases.

Suppose, we choose step size of $\beta_n = \frac{\alpha}{\bar{a}_n}$

where $\alpha > 0$ and $\bar{a}_n = \bar{a}_{n-1} + \alpha(1-\bar{a}_{n-1}) \ \forall n \geq 1$

with $\bar{a}_0 = 0$

$$Q_{n+1} = Q_n + \beta_n(R_n - Q_n)$$

$$= \beta_n R_n + (1-\beta_n)Q_n$$

Date
Page

$= \beta_n R_n + (1-\beta_n)\left(\beta_{n-1} R_{n-1} + (1-\beta_{n-1}) Q_{n-1}\right)$

$= \beta_n R_n + (1-\beta_n)(\beta_{n-1}) R_{n-1} + (1-\beta_n)(1-\beta_{n-1}) Q_{n-1}$

$= \beta_n R_n + (1-\beta_n)(\beta_{n-1}) R_{n-1} + (1-\beta_n)(1-\beta_{n-1}) \times \left[\beta_{n-2} R_{n-2} + (1-\beta_{n-2}) Q_{n-2}\right]$

$= \beta_n R_n + (1-\beta_n)(\beta_{n-1}) R_{n-1} + (1-\beta_n)(1-\beta_{n-1})(\beta_{n-2}) R_{n-2} + (1-\beta_n)(1-\beta_{n-1})(1-\beta_{n-2}) Q_{n-2}$

$= \beta_n R_n + (1-\beta_n)(\beta_{n-1}) R_{n-1} + (1-\beta_n)(1-\beta_{n-1})(\beta_{n-2}) R_{n-2} + \cdots + (1-\beta_n)(1-\beta_{n-1})(1-\beta_{n-2}) \cdots (1-\beta_1) Q_1$

$\beta_1 = \frac{\alpha}{\bar{a}_1}$

$\bar{a}_1 = \bar{a}_0 + \alpha(1-\bar{a}_0) = \alpha$

$\therefore \beta_1 = \frac{\alpha}{\alpha} = 1$

$\therefore (1-\beta_1) = 0 \Rightarrow$ coefficient of Q_1 is 0

$\therefore Q_{n+1}$ is independent of Q_1 $\forall a$

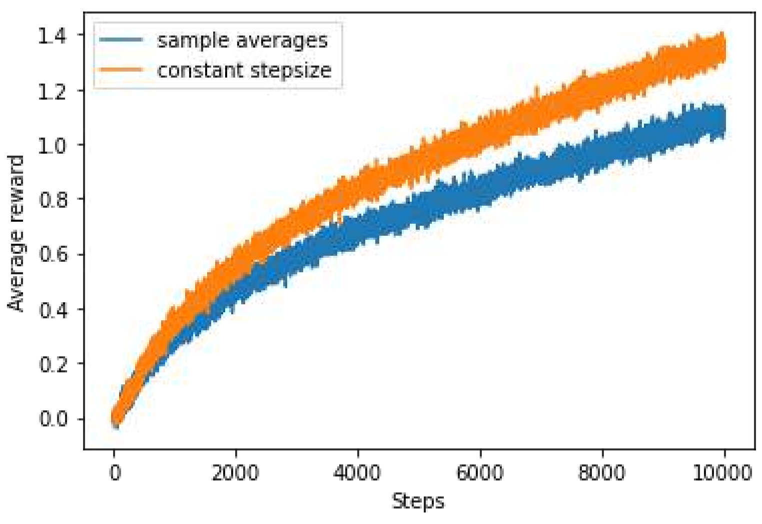
Question 5

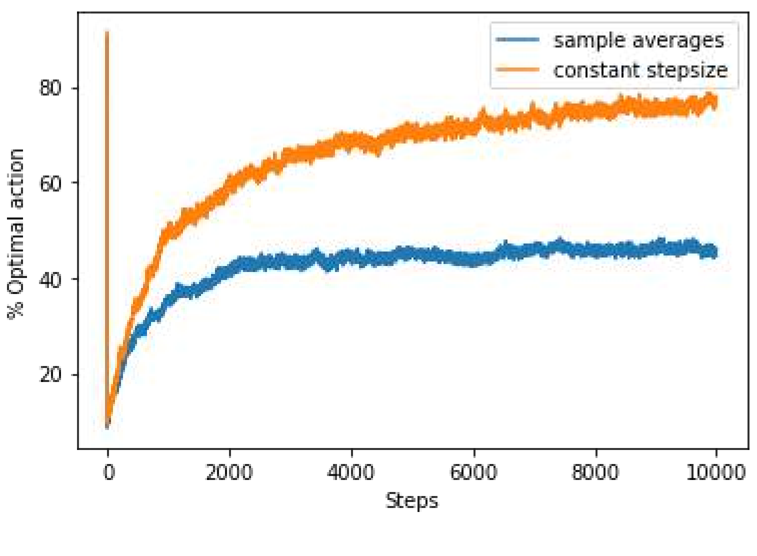
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
def ten_armed_testbed_sample_averages():
    # Initialize
    arms = 10
    epsilon = 0.1
    timestamps = 10000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = [0 for i in range(0,arms)]

        #Step 2 : simulate 10-arm bandit for 10000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update true_q
            increment_by = np.random.normal(0,0.01,arms)
            for i in range(0,arms):
                true_q[i] += increment_by[i]

            # Step 4 : Update Qt, Nt, rewards and optimal_arm_selection
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return rewards,optimal_arm
```

In [3]:
```
def ten_armed_testbed_constant_stepsize():
    # Initialize
    arms = 10
    epsilon = 0.1
    timestamps = 10000
    episodes = 2000
    stepsize = 0.1
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = [0 for i in range(0,arms)]

        #Step 2 : simulate 10-arm bandit for 10000 timestamps
        Qt = [0.0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update true_q
            increment_by = np.random.normal(0,0.01,arms)
            for i in range(0,arms):
                true_q[i] += increment_by[i]

            # Step 4 : Update Qt, rewards and optimal_arm_selection
            Qt[At] += stepsize*(Rt-Qt[At])
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return rewards,optimal_arm
```

In [4]:
```
timestamps = 10000
time = [i for i in range(1,timestamps+1)]

rewards1,optimal_arm1 = ten_armed_testbed_sample_averages()
rewards2,optimal_arm2 = ten_armed_testbed_constant_stepsize()

# plotting figure 1 - Average reward
plt.figure()
plt.plot(time, rewards1, label='sample averages')
plt.plot(time, rewards2, label='constant stepsize')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.legend()
plt.show()

# plotting figure 2 - % Optimal action
plt.figure()
plt.plot(time, optimal_arm1, label='sample averages')
plt.plot(time, optimal_arm2, label='constant stepsize')
plt.xlabel('Steps')
plt.ylabel('% Optimal action')
plt.legend()
plt.show()
```





Question 6

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Image
```

In [2]:

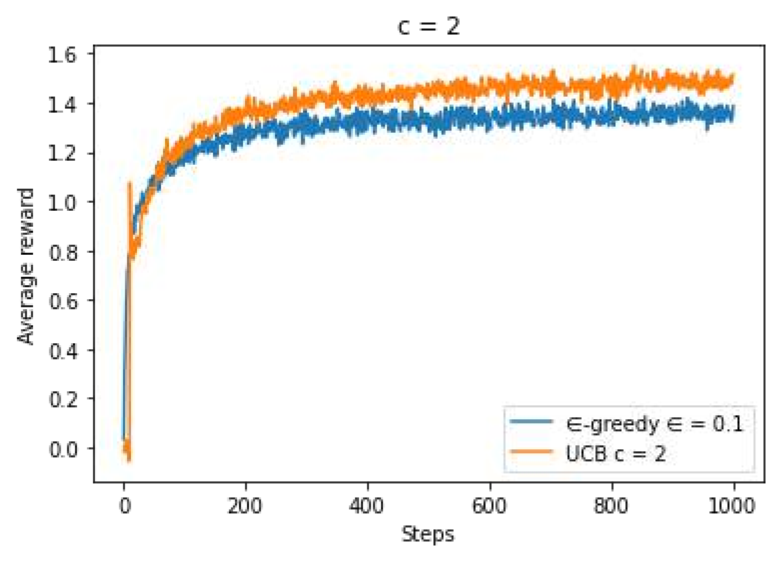
```python
def ten_armed_testbed():
    # Initialize
    arms = 10
    epsilon = 0.1
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            if np.random.uniform(0,1) < epsilon :
                # choose random arm
                At = np.random.randint(0,arms,dtype=int)
            else:
                # choose greedy arm
                At = np.argmax(Qt)

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update Qt, Nt, rewards and optimal_arm_selection
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return rewards,optimal_arm
```

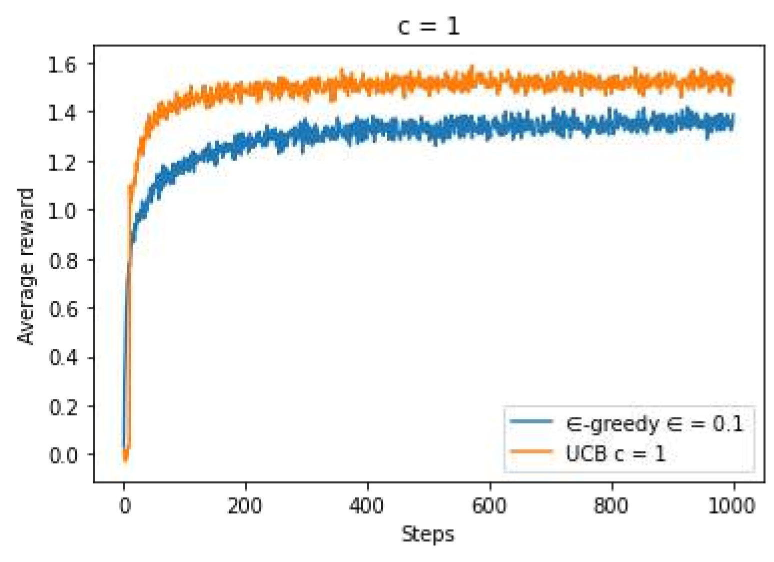
In [3]:

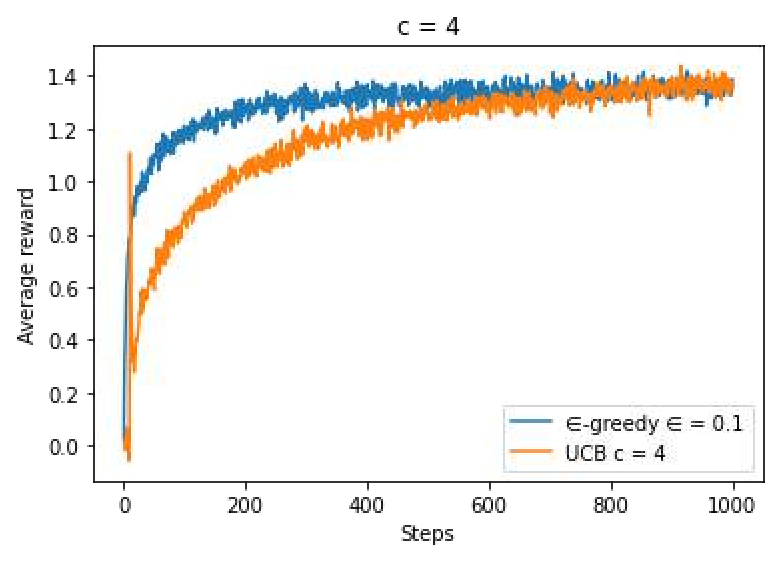
```python
def ten_armed_testbed_ucb(c1):
    # Initialize
    arms = 10
    c = c1
    timestamps = 1000
    episodes = 2000
    rewards = [0 for i in range(0,timestamps)]
    optimal_arm = [0 for i in range(0,timestamps)]

    for episode in range(0, episodes):

        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(0,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Qt = [0.0 for i in range(0,arms)]
        Nt = [0 for i in range(0,arms)]

        for iteration in range (0,timestamps):
            # arm chosen in timestamp t is At
            # corresponding reward is Rt
            # optimal arm is true_At

            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)

            # Step 1 : Choose arm
            flag = 0
            for i in range(0,arms):
                if Nt[i]==0:
                    At = i
                    flag = 1
                    break

            if flag==0:
                At = np.argmax(Qt + c * np.sqrt(np.log((iteration+1))/Nt))

            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)

            # Step 3 : Update Qt, Nt, rewards and optimal_arm_selection
            Nt[At] += 1
            Qt[At] += (Rt-Qt[At])/Nt[At]
            rewards[iteration] += Rt
            if At == true_At:
                optimal_arm[iteration] += 1

    for iteration in range(0, timestamps):
        rewards[iteration] /= episodes
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return rewards,optimal_arm
```

In [4]:

```python
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

rewards1,optimal_arm1 = ten_armed_testbed()
rewards2,optimal_arm2 = ten_armed_testbed_ucb(2)
rewards3,optimal_arm3 = ten_armed_testbed_ucb(1)
rewards4,optimal_arm4 = ten_armed_testbed_ucb(4)

# plotting figure 1 - c = 2
plt.figure()
plt.plot(time, rewards1, label='∈-greedy ∈ = 0.1')
plt.plot(time, rewards2, label='UCB c = 2')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.title('c = 2')
plt.legend()
plt.show()

# plotting figure 1 - c = 1
plt.figure()
plt.plot(time, rewards1, label='∈-greedy ∈ = 0.1')
plt.plot(time, rewards3, label='UCB c = 1')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.title('c = 1')
plt.legend()
plt.show()

# plotting figure 1 - c = 4
plt.figure()
plt.plot(time, rewards1, label='∈-greedy ∈ = 0.1')
plt.plot(time, rewards4, label='UCB c = 4')
plt.xlabel('Steps')
plt.ylabel('Average reward')
plt.title('c = 4')
plt.legend()
plt.show()
```







Justification

In [2]:

```python
Image(filename='justification_q6.png')
```

Out[2]:

In Upper-Confidence-Bound Action Selection, at timestep t action At is selected as :

$$At = argmax_a \left[Qt(a) + c\sqrt{\frac{ln(t)}{Nt(a)}} \right]$$

From steps 1 - 10 all actions are picked once (as Nt(a)=0 for all actions initially). At timestep 11, all actions have been picked once and thus the term $c\sqrt{\frac{ln(t)}{Nt(a)}}$ is the same for all arms. Depending on the rewards received from each arm during steps 1 - 10, the algorithm, on step 11, on average picks the optimal arm as the optimal arm would have resulted in a higher reward. Thus over the 2000 episodes the agent on average picks the optimal arm(with highest expected return) on step 11 which causes the spike on step 11. After picking the optimal arm, a_opt , in step 11, Nt(a_opt) becomes 2. The term $c\sqrt{\frac{ln(t)}{Nt(a)}}$ will be smaller for a_opt than for other arms which have Nt as 1.
If c = 2 then the term $2 * \sqrt{\frac{ln(t)}{Nt(a)}}$ dominates and in step 12, the optimal action is not picked as its Nt is 2. Thus, an action with less return is picked which leads to decrease in average return for step 12. Thereafter, it's Nt becomes 2 and another action with smaller return is picked.

If c = 1, then the impact of the term $\sqrt{\frac{ln(t)}{Nt(a)}}$ reduces and the impact of Qt(a_opt) increases, the probability of selecting the optimal arm again increases. Thus, over the 2000 episodes the agent on average picks the optimal arm(with highest expected return) again on step 12 which makes the spike at step 11 less prominent.

If c = 4, then the impact of the term $4 * \sqrt{\frac{ln(t)}{Nt(a)}}$ becomes substantial and the optimal action is not picked at step 12 and over the 2000 episodes the agent picks different arms at step 12 which reduces the average return.

In [ ]:

Question 7

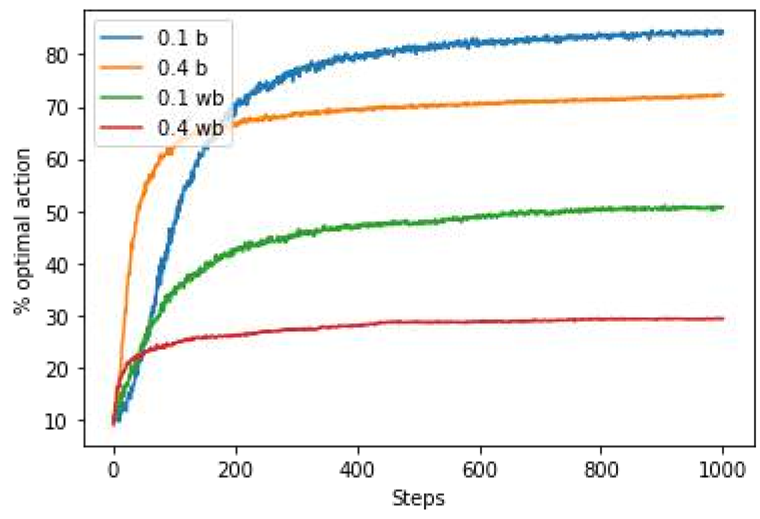
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
def calc_Preference(H):
    exp = np.exp(H)
    return exp / np.sum(exp)
```

In [3]:
```
# baseline, Rt_bar is the average of rewards including Rt
def ten_armed_testbed_gradient_with_baseline(eps):
    # Initialize
    arms = 10
    alpha = eps
    timestamps = 1000
    episodes = 2000
    optimal_arm = [0 for i in range(0,timestamps)]
    list_arms = [i for i in range(0,arms)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(4,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Ht = [0.0 for i in range(0,arms)]
        Pit = [0.0 for i in range(0,arms)]

        av_rewards = 0
        n = 0

        for iterations in range(0, timestamps):
            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)
            # Step 1 : Choose arm At
            pit = calc_Preference(Ht)
            At = np.random.choice(list_arms,p=pit)
            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)
            # Step 3 : Update n and rewards
            n += 1
            av_rewards = av_rewards + (Rt - av_rewards)/n
            #Step 4 : Update Action Preferences
            Ht[At] = Ht[At] + alpha * (Rt - av_rewards) * (1 - pit[At])
            for i in range(0,arms):
                if At!=i:
                    Ht[i] = Ht[i] - alpha * (Rt - av_rewards) * pit[i]
            # Step 5 : Update optimal_action
            if At == true_At:
                optimal_arm[iterations] += 1


    for iteration in range(0, timestamps):
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return optimal_arm
```

In [4]:
```
def ten_armed_testbed_gradient_without_baseline(eps):
    # Initialize
    arms = 10
    alpha = eps
    timestamps = 1000
    episodes = 2000
    optimal_arm = [0 for i in range(0,timestamps)]
    list_arms = [i for i in range(0,arms)]

    for episode in range(0, episodes):
        # Step 1 : assign q*(a) for all arms a
        true_q = np.random.normal(4,1,10)

        #Step 2 : simulate 10-arm bandit for 1000 timestamps
        Ht = [0.0 for i in range(0,arms)]
        Pit = [0.0 for i in range(0,arms)]

        for iterations in range(0, timestamps):
            # Step 0 : Get optimal arm
            true_At = np.argmax(true_q)
            # Step 1 : Choose arm At
            pit = calc_Preference(Ht)
            At = np.random.choice(list_arms,p=pit)
            # Step 2 : Receive reward
            Rt = np.random.normal(true_q[At],1)
            #Step 3 : Update Action Preferences
            Ht[At] = Ht[At] + alpha * (Rt) * (1 - pit[At])
            for i in range(0,arms):
                if At!=i:
                    Ht[i] = Ht[i] - alpha * (Rt) * pit[i]
            # Step 5 : Update optimal_action
            if At == true_At:
                optimal_arm[iterations] += 1


    for iteration in range(0, timestamps):
        optimal_arm[iteration] = (optimal_arm[iteration]*100)/episodes

    return optimal_arm
```

In [5]:
```
timestamps = 1000
time = [i for i in range(1,timestamps+1)]

optimal_arm1 = ten_armed_testbed_gradient_with_baseline(0.1)
optimal_arm2 = ten_armed_testbed_gradient_with_baseline(0.4)

optimal_arm3 = ten_armed_testbed_gradient_without_baseline(0.1)
optimal_arm4 = ten_armed_testbed_gradient_without_baseline(0.4)

# plotting figure 1
plt.figure()
plt.plot(time, optimal_arm1, label='0.1 b') #alpha = 0.1, with baseline
plt.plot(time, optimal_arm2, label='0.4 b') #alpha = 0.4, with baseline
plt.plot(time, optimal_arm3, label='0.1 wb') #alpha = 0.1, without baseline
plt.plot(time, optimal_arm4, label='0.4 wb') #alpha = 0.4, without baseline
plt.xlabel('Steps')
plt.ylabel('% optimal action')
plt.legend()
plt.show()
```



In [ ]: