Reinforcement Learning Assignment 1

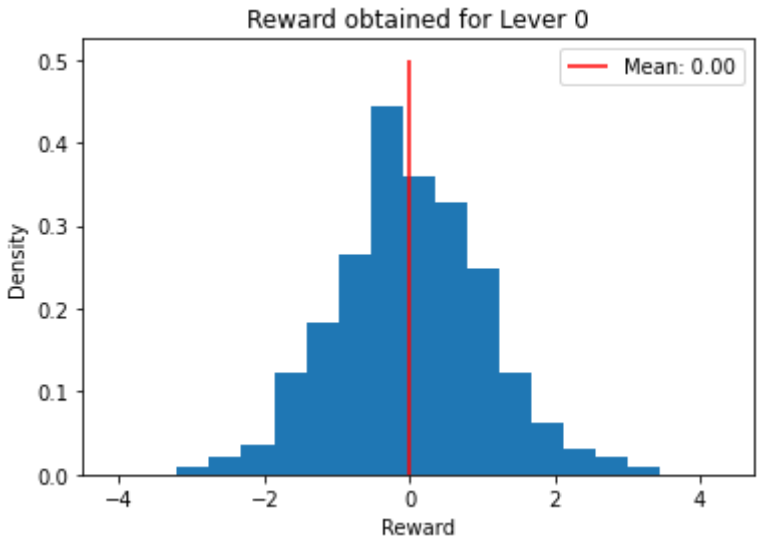
Louise Beyers 21591644

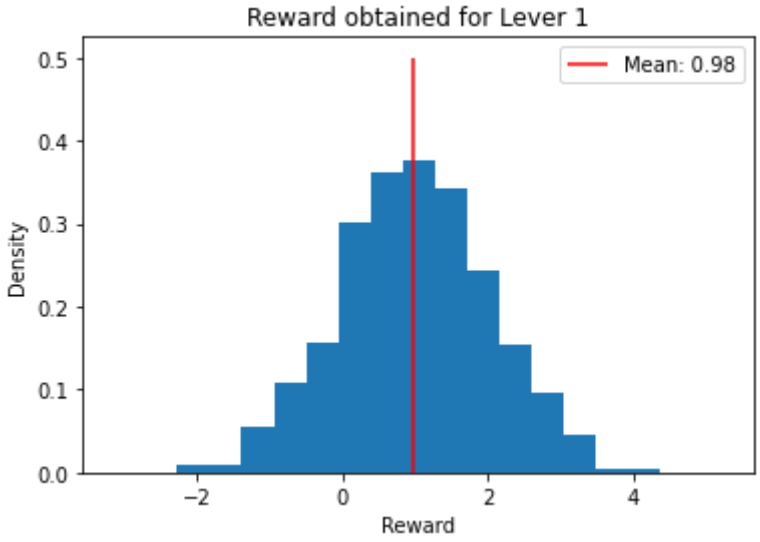
In [25]:
```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la
import seaborn as sns
import copy
```

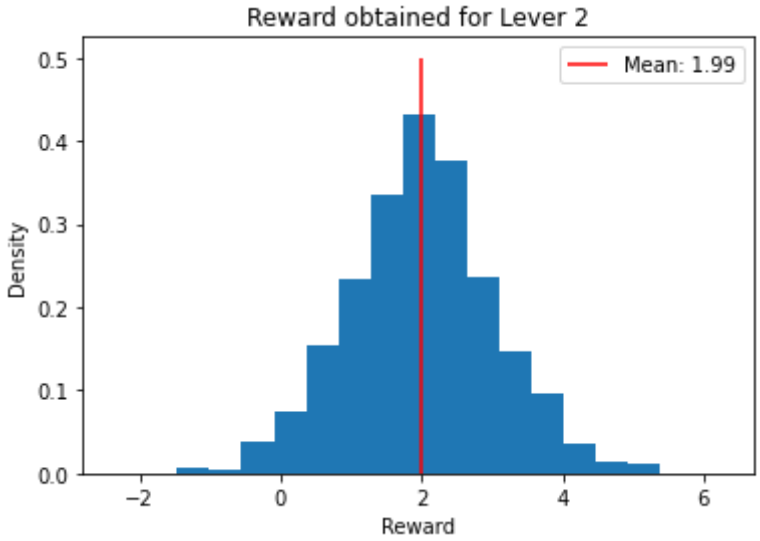
Question 1

a)

In [2]:
```
a = [0,1,2]

n_samples = 1000

for a_val in a:
    sample = np.random.normal(loc=a_val,size=n_samples)
    mean = np.mean(sample)
    bins = np.linspace(min(sample)-1,max(sample)+1,20)
    plt.title('Reward obtained for Lever '+str(a_val))
    plt.hist(sample,bins=bins,density=True)
    plt.vlines(x=mean,ymin=0,ymax=0.5,color='red',label='Mean: '+"{:.2f}".for
    plt.xlabel('Reward')
    plt.ylabel('Density')
    plt.legend()
    plt.show()
```







The sample mean rewards for each lever are all quite close to the true means, and the shapes of the histograms suggest underlying normal distributions. Note the overlap between the values of rewards obtained through different levers - Lever 0 gave some rewards above 2, Lever 1 gave rewards above 2 and below 0, and Lever 2 gave rewards below 0. Judging by these distributions, it is not possible to confidently say which lever would be responsible for a reward of, say, 2.

b)

In [3]:

```
# function to return estimated value function
def est_val_func(A,R):
    # initialise q with the large bnegatives everywhere
    q = np.zeros(3) + float('-inf')

    # convert datatype to use numpy functions
    A = np.array(A)
    R = np.array(R)

    # find the number of observed actions
    seen, counts = np.unique(A,return_counts=True)

    # set q for every seen action
    for i,a in enumerate(seen):
        # get the sum of rewards
        numerator = np.sum(R[np.where(A==a)])
```

```
        # set the value of q
        q[a] = numerator/counts[i]

    return q
```

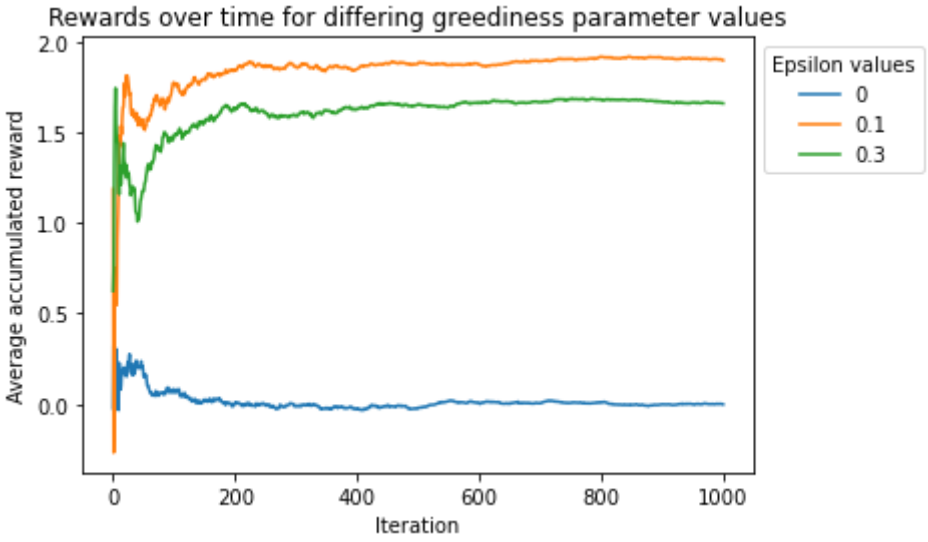
c)

In [4]:

```
np.random.seed(2)    # seed which gives nicely-interpretable results

N = 1000    # number of iterations
epses = [0,0.1,0.3]    # epsilon values to test

a_vals = [0,1,2]    # action options

# storage for the action and reward sequences
A_listlist = []
R_listlist = []

# loop over epsilon values
for eps in epses:
    # initialise action
    a1 = np.random.choice(a_vals)
    A_list = [a1]
    R_list = [np.random.normal(loc=a1)]

    for n in range(1,N):
        # estimate value function
        q = est_val_func(A_list,R_list)

        # act randomly with probability epsilon
        if np.random.rand()<eps:
            an = np.random.choice(a_vals)
        else:
            an = a_vals[np.argmax(q)]

        # get reward based on action
        rn = np.random.normal(loc=an)

        # store reward and action
        A_list.append(an)
        R_list.append(rn)

    A_listlist.append(A_list)
    R_listlist.append(R_list)
```

In [5]:

```
# plot the average accumulated rewards
for eps,R_list in zip(epses,R_listlist):
    plt.plot(range(N),np.cumsum(np.array(R_list))/np.array(range(1,N+1)),labe
plt.legend(bbox_to_anchor=(1,1),title='Epsilon values')
plt.title('Rewards over time for differing greediness parameter values')
plt.xlabel('Iteration')
plt.ylabel('Average accumulated reward')
plt.show()
```



We see here that the reward when $\epsilon = 0$ and the greedy choice is taken all the time doesn't seem to move far from 0. This is because the first action taken will always be chosen again by the greedy part of the algorithm. Since the first action must have been to pull Lever 0, the resulting reward sequence is essentially a sample of 1000 pulls of Lever 0, which is why the mean is converging to 0.

When $\epsilon = 0.3$, we see that after about 250 iterations the average reward starts to level out at about 1.5. The reason for this is that enough exploration took place that the greedy policy found the correct lever to pull (Lever 2), but because we do not anneal ϵ, random actions are taken often enough to lower the average reward to 1.5.

Similarly, when $\epsilon = 0.1$, we see that after about 250 iterations the average reward starts to level out at just below 2. Again, enough exploration took place that the greedy policy found the correct lever to pull (Lever 2), but because we do not anneal ϵ, random actions are taken often enough to lower the average reward.

d)

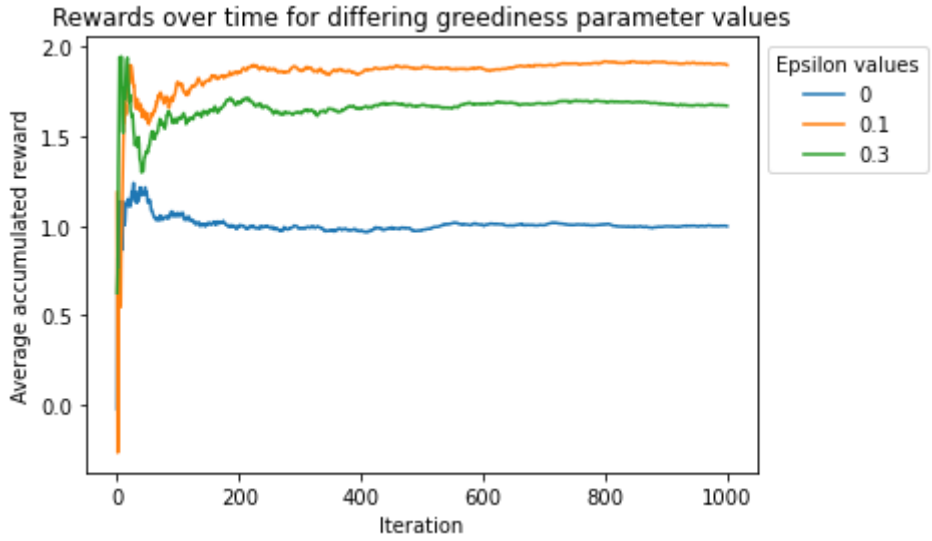
In [6]:

```
np.random.seed(2)    # seed which gives nicely-interpretable results

N = 1000    #number of iterations
epses = [0,0.1,0.3]    # epsilon values to test

a_vals = [0,1,2]    # action options

# storage for the action and reward sequences
A_listlist = []
R_listlist = []

# loop over epsilon values
for eps in epses:
    # initialise action
    a1 = np.random.choice(a_vals)
    A_list = [a1]
    R_list = [np.random.normal(loc=a1)]

    # initialise annealing factor and value function
    alpha = 1
    q = np.zeros(3)
    q[a1] = alpha*R_list[-1]
```

```
    for n in range(2,N+1):

        if np.random.rand()<eps:
            an = np.random.choice(a_vals)
        else:
            an = a_vals[np.argmax(q)]
        rn = np.random.normal(loc=an)

        q[an] = (1-alpha/n)*q[an]+alpha/n*rn

        A_list.append(an)
        R_list.append(rn)

    A_listlist.append(A_list)
    R_listlist.append(R_list)
```

In [7]:

```
for eps,R_list in zip(epses,R_listlist):
    plt.plot(range(N),np.cumsum(np.array(R_list))/np.array(range(1,N+1)),labe
plt.legend(bbox_to_anchor=(1,1),title='Epsilon values')
plt.title('Rewards over time for differing greediness parameter values')
plt.xlabel('Iteration')
plt.ylabel('Average accumulated reward')
plt.show()
```



The initialisation of Q still matters greatly here.

For $\epsilon = 0$, we see some exploration take place (the first lever pulled was Lever 0, the next Lever 1). However, since Q is initialised with zeros, the algorithm can only sometimes escape initial choices which give negative rewards and will keep pulling the lever which first gives a value function estimate above 0. This is why we see that the accumulated reward seems to converge to the mean of Lever 1.

When $\epsilon = 0.3$, we see that the average reward moves faster towards what will be its final value, because the decaying calculation of Q allows the algorithm to place less importance on earlier rewards.

Similarly, when $\epsilon = 0.1$, we see that the average reward sooner moves to the region in which it will converge. This also indicates better estimation of Q.

The final values to which the algorithm converges may be interpreted similarly to those in Question 1c).

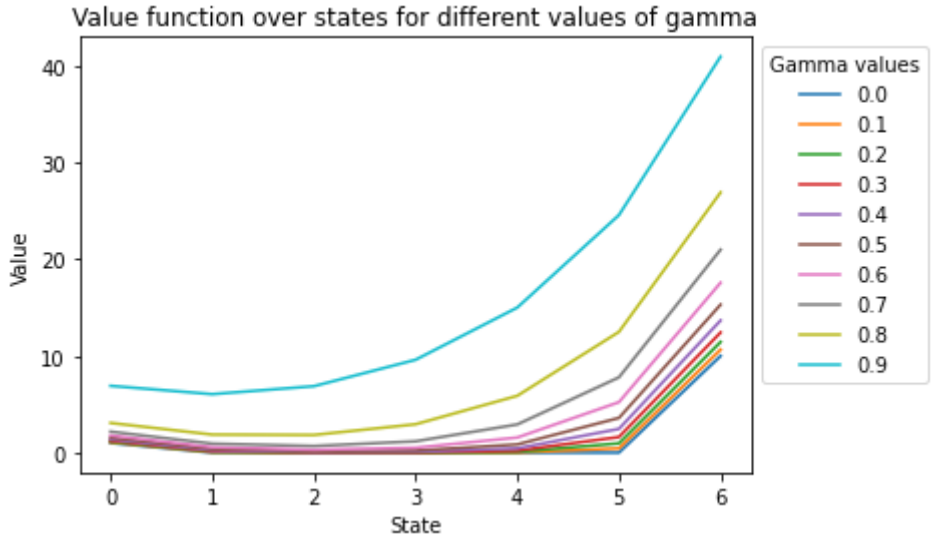
Question 2

a)

In [8]:
```
# problem formulation

# parameters and states
gammas = np.array(range(10))/10
n_states = 7
states = np.array(range(n_states)).astype(int)

# encoding the transition probabilities
P = np.zeros((n_states,n_states))
P = P+np.eye(n_states)*0.2
P[0,0] = P[-1,-1] = 0.6
for i in range(n_states-1):
    P[i,i+1] = 0.4
    P[i+1,i] = 0.4
print('P: ')
print(P)

# encoding the rewards - to be used throughout where applicable
row = np.zeros(n_states)
row[0] = 1
row[-1] = 10
print('Rewards: ')
print(row)
```

```
P:
[[0.6 0.4 0.  0.  0.  0.  0. ]
 [0.4 0.2 0.4 0.  0.  0.  0. ]
 [0.  0.4 0.2 0.4 0.  0.  0. ]
 [0.  0.  0.4 0.2 0.4 0.  0. ]
 [0.  0.  0.  0.4 0.2 0.4 0. ]
 [0.  0.  0.  0.  0.4 0.2 0.4]
 [0.  0.  0.  0.  0.  0.4 0.6]]
Rewards:
[ 1.  0.  0.  0.  0.  0. 10.]
```

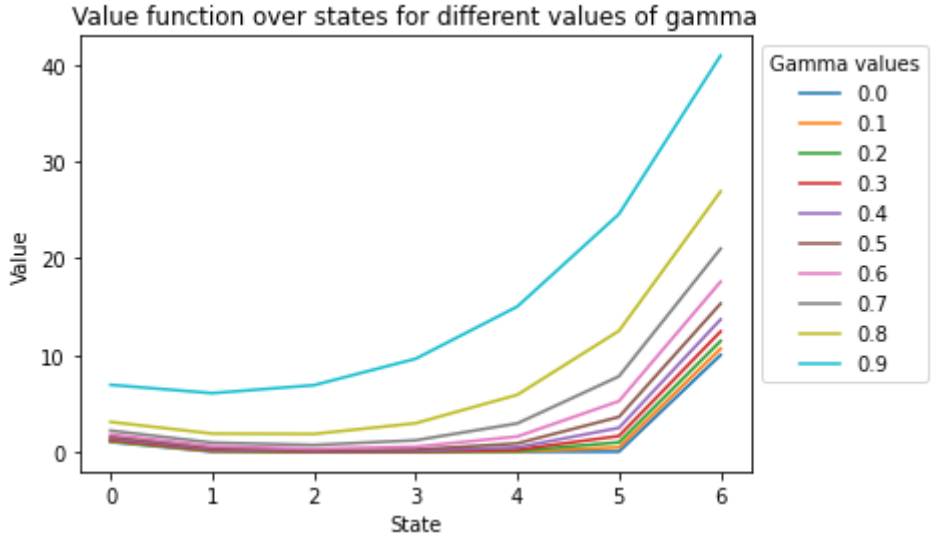
In [9]:
```
# plot solutions to the linear Bellman equation
for g in gammas:
    # direct solve
    v = la.inv(np.eye(n_states)-g*P)@row
    # plot values
    plt.plot(v,label=str(g))
plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function over states for different values of gamma')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```



b)

In [10]:

```
n_iters = 1000

for g in gammas:
    v = np.zeros(n_states)

    # iterative solve
    for n in range(n_iters):
        v = row+g*P@v

    plt.plot(v,label=str(g))

plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function over states for different values of gamma')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```



c)

In [11]:

```
def gen_traj(s0,N,states,P,row):

    traj = [s0]
```

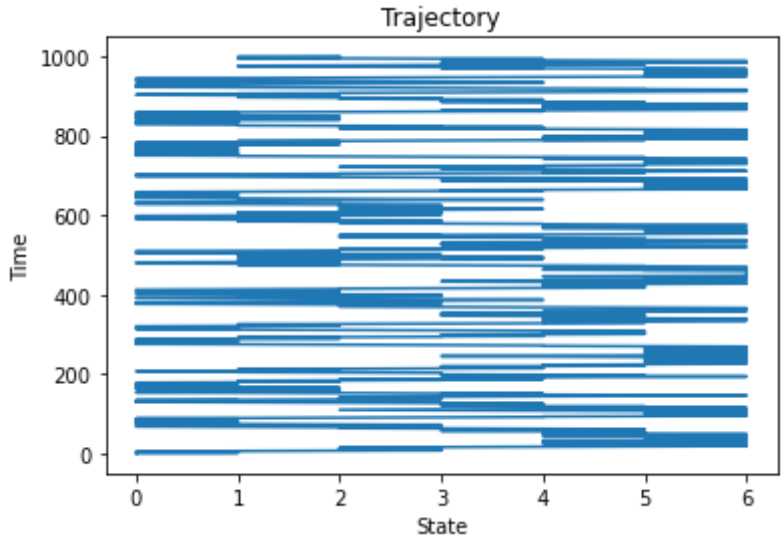
```
    R_list = [row[s0]]
    s = s0

    for n in range(1,N):
        s = np.random.choice(states,p =P[s,:])
        traj.append(s)
        R_list.append(R_list[-1]+row[traj[-2]])

    return traj, R_list
```

In [12]:

```
t, r = gen_traj(0,1000,states,P,row)
```

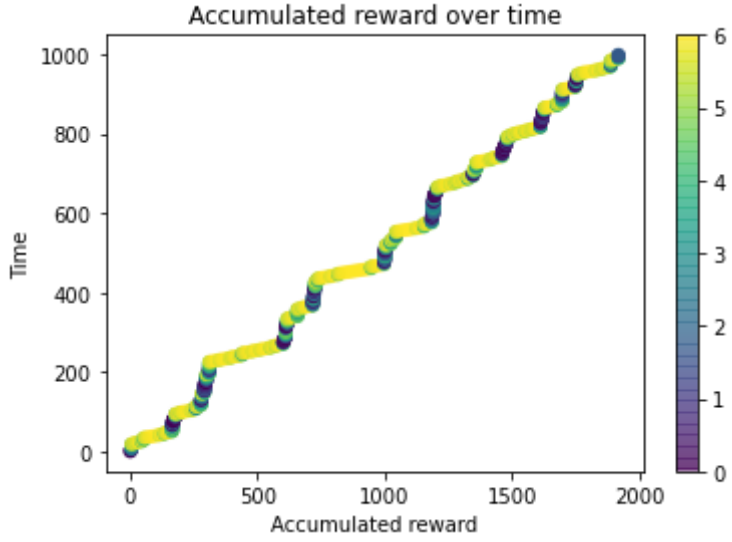
In [13]:

```
plt.title('Trajectory')
plt.xlabel('State')
plt.ylabel('Time')
plt.plot(t,range(1000))
plt.show()
```



In [14]:

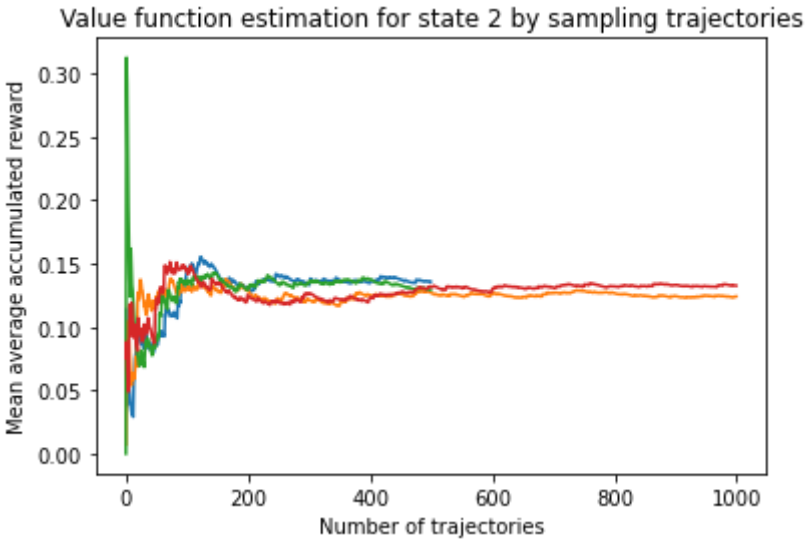
```
plt.title('Accumulated reward over time')
plt.xlabel('Accumulated reward')
plt.ylabel('Time')
plt.scatter(r,range(1000),c=t,alpha=0.5)
plt.colorbar()
plt.show()
```



d)

In [27]:
```
gamma = 0.5
s0 = 2
n_traj = [500,1000]
len_traj = [100,500]

N_traj, Len_traj = np.meshgrid(n_traj,len_traj)

V_list = []

for N_t,L_t in zip(N_traj.flatten(),Len_traj.flatten()):
    v = []
    for n_t in range(N_t):
        t, r = gen_traj(s0,L_t,states,P,row)
        pows_of_gamma = np.array(range(L_t))
        v.append(np.sum(np.multiply(gamma**pows_of_gamma,r)))

    q = np.cumsum(np.array(v))/np.array(range(1,N_t+1))
    V_list.append(q)
```

In [28]:
```
for N_t, q in zip(N_traj.flatten(),V_list):
    plt.plot(range(N_t),q)

plt.title('Value function estimation for state 2 by sampling trajectories')
plt.xlabel('Number of trajectories')
plt.ylabel('Mean average accumulated reward')
plt.show()
```



In [29]:
```
v_a = la.inv(np.eye(n_states)-gamma*P)@row
v_a = v_a[2]
for q in V_list:
    print(q[-1])
print(v_a)
```

```
0.13567103938062017
0.12396883094019112
0.12918036068052785
0.13254013898246372
0.1304331838806863
```

The estimated value of the state is much higher than the empirical value.

Question 3

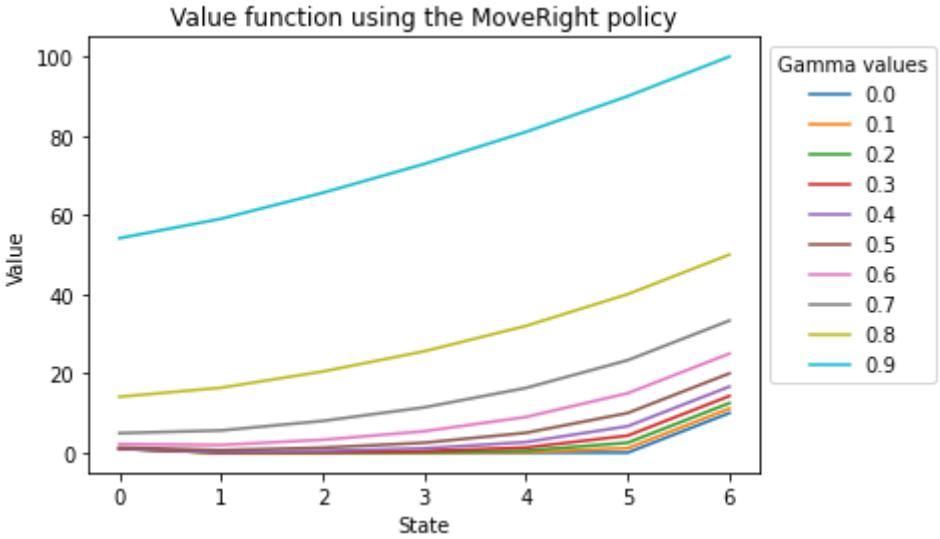
a)

In [31]:
```
n_states = 7

# for a=0
P_a0 = np.zeros((n_states,n_states))
P_a1 = np.zeros((n_states,n_states))

P_a0[-1,-1] = 1
for i in range(n_states-1):
    P_a0[i,i+1] = 1
print('MoveRight:')
print(P_a0)

P_a1[0,0] = 1
for i in range(n_states-1):
    P_a1[i+1,i] = 1
print('MoveLeft:')
print(P_a1)
```

```
MoveRight:
[[0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1.]]
MoveLeft:
[[1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]]
```

b)

In [19]:
```
# plot solutions to the linear Bellman equation
for g in gammas:
    # direct solve
    v = la.inv(np.eye(n_states)-g*P_a0)@row
    # plot values
    plt.plot(v,label=str(g))
plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function using the MoveRight policy')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```

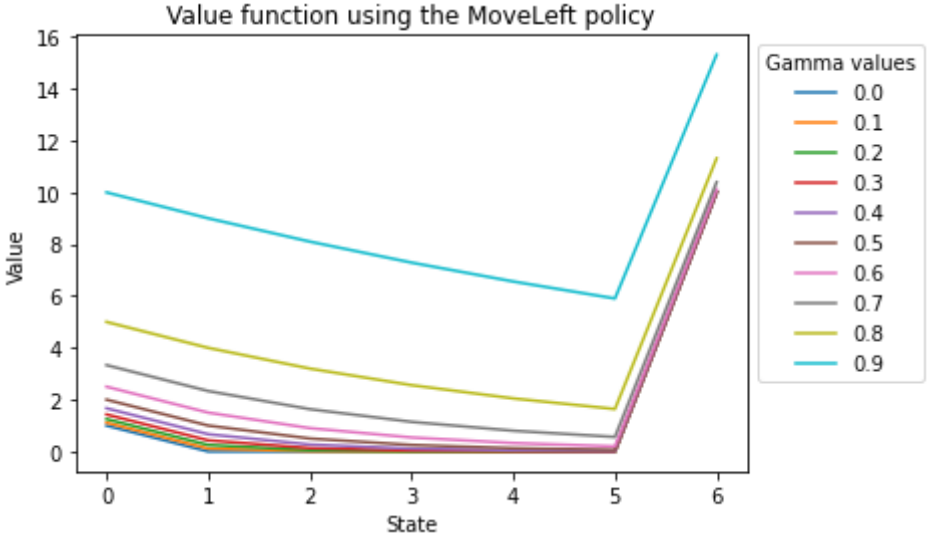


For the MoveRight policy alone, the value function value tends to increase as the state number increases.This makes sense, since the closer you start to the right, the more rewards you will obtain from staying at state 6 in the end. For small values of gamma, however, the value function is slightly higher at state 0 because the discount you eventually obtain at state 6 is discounted so heavily that the reward obtained by leaving state 0 contributes more heavily to the value function.

The value function for states to the right of 0 do not include any of the reward obtained by leaving state 0, because state 0 will never be reached using the MoveRight policy in these cases.

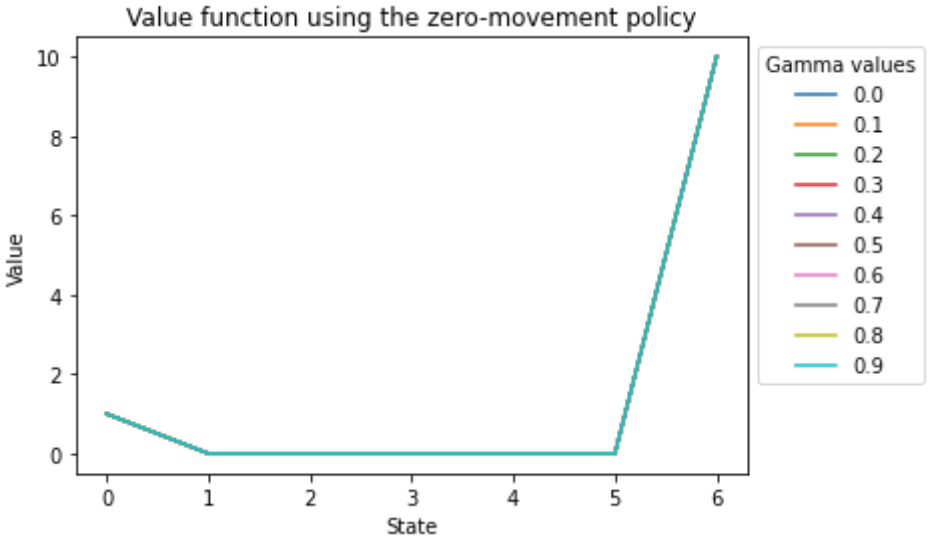
In [20]:

```
# plot solutions to the linear Bellman equation
for g in gammas:
    # direct solve
    v = la.inv(np.eye(n_states)-g*P_a1)@row
    # plot values
    plt.plot(v,label=str(g))
plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function using the MoveLeft policy')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```



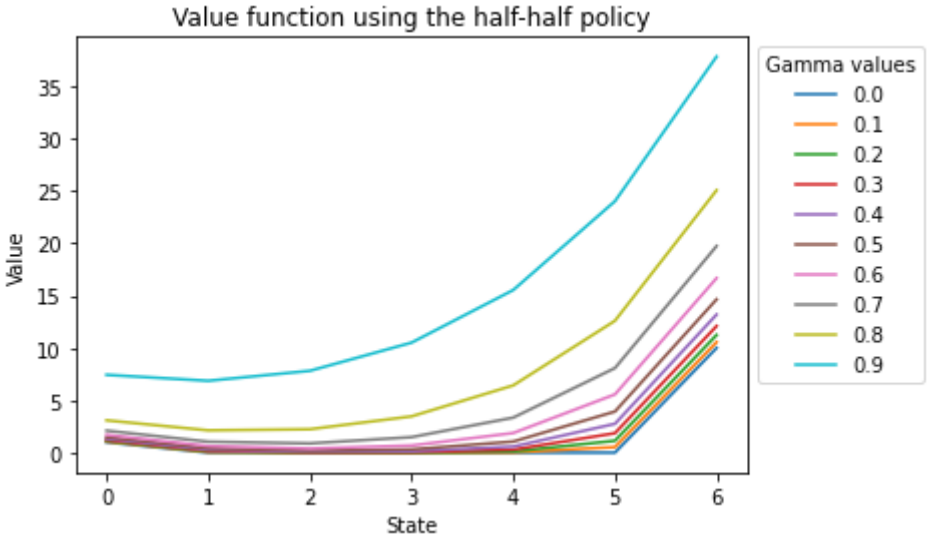
We see the opposite happen when we consider the MoveLeft policy as opposed to the MoveRight policy. The trends are stretched according to the scale of the rewards at states 0 and 6, but otherwise the exact same trends occur in reverse as for the MoveRight policy.

In [21]:
```
# plot solutions to the linear Bellman equation
for g in gammas:
    # direct solve
    v = la.inv(np.eye(n_states)-np.zeros(n_states))@row
    # plot values
    plt.plot(v,label=str(g))
plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function using the zero-movement policy')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```



The zero-movement policy means that the value function values equal the rewards given at that state. This makes sense because the value function is meant to measure expected return of the state, and the expected return of being in a state when you must stay in the state is exactly its reward.

In [22]:
```
# plot solutions to the linear Bellman equation
for g in gammas:
    # direct solve
    v = la.inv(np.eye(n_states)-g*(0.5*P_a1+0.5*P_a0))@row
    # plot values
    plt.plot(v,label=str(g))
plt.legend(bbox_to_anchor=(1,1),title='Gamma values')
plt.title('Value function using the half-half policy')
plt.xlabel('State')
plt.ylabel('Value')
plt.show()
```



For the half-half policy, we obtain some combination of the MoveRight and MoveLeft value function values. Starting from the value function which is just the list of rewards at $\gamma = 0$, we increase γ. As a result, the value function at states 0 and 6 increases by a factor influenced by the scales of the respective rewards and by the value of γ. Visually, the effects of these increases ripple down (once again by some factor determined by γ, the scales of the rewards and the distance to the reward-giving state) to the states closer to the rewarding states. Since state 6 gives a much higher reward than state 0, the approximate slope of the value function is positive even from state 2. This means that if an agent starts from any state above state 1, it will be incentivised to move towards state 6.

c)

Bellman optimality equation:

$$v_*(s) = max_a E[R_{t+1} + \Gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$
$$= max_a [\rho(s,a) + \Gamma (P_a v_*)(s)]$$

where the second term is the s-th component of the matrix-vector product. Since the reward of a state is collected when you leave a state (even if you go back to the same state with the action that instigated the leaving), the reward is *independent of the action*. The reward term may therefore be taken outside of the maximising equation. Similarly, Γ is independent of the action.

$$v_*(s) = \rho(s) + \Gamma max((P_0 v_*)(s), (P_1 v_*)(s))$$

As a vector equation, this may be written as:

$$\mathbf{v}_* = \rho + \Gamma max(P_0 \mathbf{v}_*, P_1 \mathbf{v}_*)$$

Equivalently:

Bellman optimality equation:

$$v_*(s) = max_a E[R_{t+1} + \Gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

Since the reward of a state is collected when you leave a state (even if you go back to the same state with the action that instigated the leaving), the reward is *independent of the action*. Suppose action a gives new state s'. Then the above equation simplifies to:

$$v_*(s) = max_a(\rho(s') + \Gamma v_*(s'))$$

For states 1 to 5, this becomes:

$$v_*(s) = max(\rho(s-1) + \Gamma v_*(s-1), \rho(s+1) + \Gamma v_*(s+1))$$

For states 0 and 6, it simplifies to:

$$v_*(0) = max(\rho(0) + \Gamma v_*(0), \rho(1) + \Gamma v_*(1))$$

and

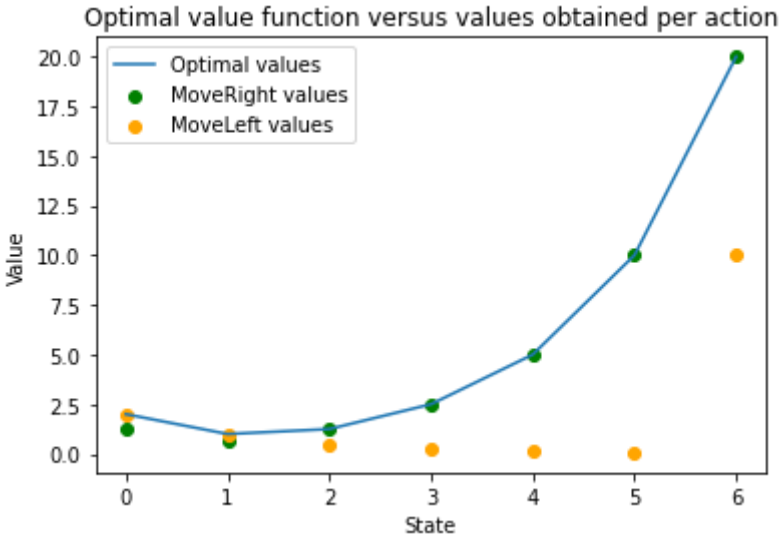
$$v_*(6) = max(\rho(5) + \Gamma v_*(5), \rho(6) + \Gamma v_*(6))$$

respectively.

d) / e)

In [87]:
```
#hyperparameters
gamma = 0.5
n_iters = 1000

# values and policy initialised as zeros
v_k = np.zeros(n_states)
pi_k = np.zeros(n_states).astype(int)

for i in range(n_iters):
    for s in range(n_states):
        # calculate the updated v-dependent terms for a0 and a1
        choices = np.array([P_a0[s,:]@v_k,P_a1[s,:]@v_k])

        # find the maximum - store the index as optimal pi
        pi_k[s] = int(np.argmax(choices))

        v_k[s] = row[s]+gamma*choices[pi_k[s]]
```

In [88]:
```
# plot MoveRight value function
v_a0 = la.inv(np.eye(n_states)-gamma*(P_a0))@row
plt.scatter(states,v_a0,color='green',label='MoveRight values')

# plot MoveRight value function
v_a1 = la.inv(np.eye(n_states)-gamma*(P_a1))@row
plt.scatter(states,v_a1,color='orange',label='MoveLeft values')

# plot v star
plt.plot(v_k, label='Optimal values')
plt.title('Optimal value function versus values obtained per action')
plt.xlabel('State')
plt.ylabel('Value')
plt.legend()
plt.show()
```



Since the optimal policy is deterministic, the optimal actions at each step are also deterministic. Because we will always take the same optimal actions based on the starting state, the optimal value function at each state is based on one specific action, and will therefore be the same as the value function for that action at that state.

This is why we see that the optimal value function at each state matches at least one of the value function values for the deterministic policies. For example, in states 0 and 1, the optimal action to choose is MoveLeft, which is why the optimal value function matches the value function for the MoveLeft policy at those states.

Another thing to note is that the optimal value function matches the highest values available for the deterministic functions - naturally, since we want our rewards to be maximised.

f)

Since the system we are working in has a constant reward system and no changing transition probabilities, staying in a state which gives no reward is never optimal. Therefore, even if we give the option of staying at the states which give no reward, the optimal policy will never choose to stay in such a state.

The optimal policy would therefore not change.

Since the future reward of a non-reward-giving state does not depend on whether you are allowed to stay in that state when a transition probability is not enforced for staying, the optimal value function would also not change.

If moves from state 3 to state 6 were added, then the optimal solution would change. The value function of 3 would increase, and 3 would have a higher return than 4 since there is a shorter path between 3 and 6 than 4 and 6.

Out of curiosity, we verify the statement above:

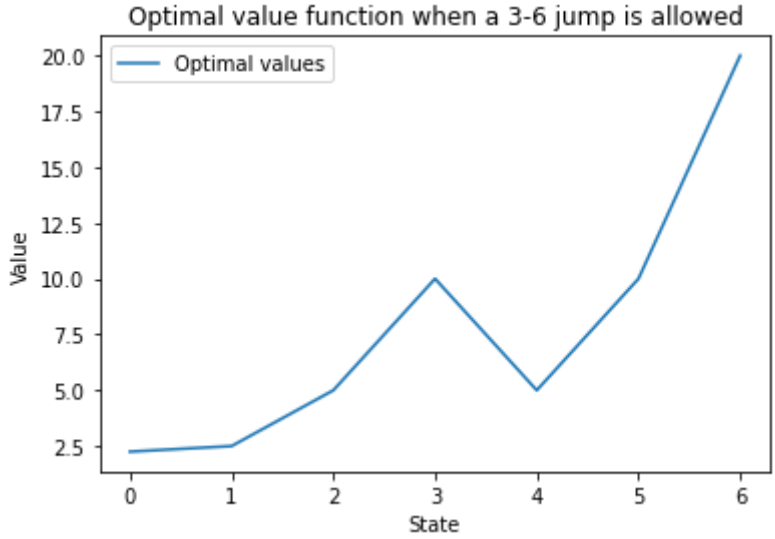
In [34]:
```
#hyperparameters
gamma = 0.5
n_iters = 1000

# values and policy initialised as zeros
v_k = np.zeros(n_states)
```

```
pi_k = np.zeros(n_states).astype(int)

P_3to6 = np.zeros((n_states,n_states))
P_3to6[3,6] = 1

for i in range(n_iters):
    for s in range(n_states):
        # calculate the updated v-dependent terms for a0 and a1
        choices = np.array([P_a0[s,:]@v_k,P_a1[s,:]@v_k,P_3to6[s,:]@v_k])

        # find the maximum - store the index as optimal pi
        pi_k[s] = int(np.argmax(choices))

        v_k[s] = row[s]+gamma*choices[pi_k[s]]
print(v_k)
```

```
[ 2.25  2.5   5.   10.    5.   10.   20.  ]
```

In [35]:

```
# plot v star
plt.plot(v_k, label='Optimal values')
plt.title('Optimal value function when a 3-6 jump is allowed')
plt.xlabel('State')
plt.ylabel('Value')
plt.legend()
plt.show()
```



Question 4

a) TD(0)

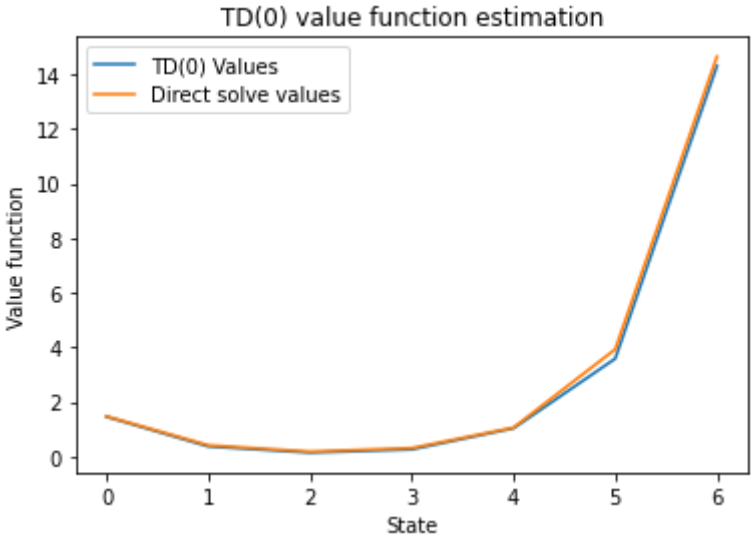
In [36]:

```
#parameters
w_a0 = 0.5
w_a1 = 0.5
gamma = 0.5
alpha = 0.1
n_epi = 1000
N = 1000

#input to be evaluated
pi = w_a0*P_a0+w_a1*P_a1

#initialise values
V = np.zeros(n_states)
```

```
#TD(0) algorithm
for i in range(n_epi):
    S = np.random.choice(n_states)
    for j in range(N):
        #randomly choose new state using row S of pi as the probabilities
        S_prime = np.random.choice(states,p=pi[S,:])

        #observe R - since reward is taken upon leaving a state
        R = row[S]

        V[S] = V[S] + alpha*(R+gamma*V[S_prime]-V[S])
        S = S_prime
```

In [37]:

```
plt.plot(V,label='TD(0) Values')

#result from Q3
v_q3 = la.inv(np.eye(n_states)-gamma*(0.5*P_a1+0.5*P_a0))@row
plt.plot(v_q3,label='Direct solve values')
plt.title('TD(0) value function estimation')
plt.legend()
plt.xlabel('State')
plt.ylabel('Value function')
plt.show()
```



b) SARSA Algorithm

In [38]:

```
def epsilon_greedy(eps,Q,S,actions):
    # decide how to act
    if np.random.rand()<eps:
        # randomly choose action
        A = np.random.choice(actions)
    else:
        # use Q
        A = np.argmax(Q[S,:])
    return A
```

In [49]:

```
# parameters
gamma = 0.5
alpha = 0.2
eps = 0.1
n_epi = 1000
```

```
N = 1000
n_actions = 2
actions = np.array(range(2))

#input to be evaluated
Q = np.random.rand(n_states,n_actions)



P = np.array([P_a0,P_a1])


#SARSA algorithm
for i in range(n_epi):
    S = np.random.choice(n_states)

    A = epsilon_greedy(eps,Q,S,actions)

    for j in range(N):
        #observe R
        R = row[S]

        #observe S_prime
        S_prime = np.argmax(P[A,S,:])

        A_prime = epsilon_greedy(eps,Q,S_prime,actions)

        Q[S,A] = Q[S,A] + alpha*(R+gamma*Q[S_prime,A_prime]-Q[S,A])

        S = S_prime
        A = A_prime
```

In [50]:

```
print("Optimal policy:")
print(np.argmax(Q,axis=1))
print('\nOptimal Q:')
print(Q)
```

```
Optimal policy:
[1 1 0 0 0 0 0]

Optimal Q:
[[ 1.48204561  1.9939137 ]
 [ 0.55532435  0.9636951 ]
 [ 1.10304397  0.43115154]
 [ 2.08955931  0.49597527]
 [ 4.32835779  0.98370621]
 [ 9.69079282  1.93217566]
 [19.71556254 14.77984169]]
```

c) Q-learning

In [84]:

```
# parameters
gamma = 0.5
alpha = 0.2
eps = 0.1
n_epi = 1000
N = 1000
n_actions = 2
actions = np.array(range(2))

#input to be evaluated
Q = np.random.rand(n_states,n_actions)
```

```
P = np.array([P_a0,P_a1])


#Q-learning algorithm
for i in range(n_epi):
    S = np.random.choice(n_states)

    for j in range(N):
        #choose A
        A = epsilon_greedy(eps,Q,S,actions)

        #observe R
        R = row[S]

        #observe S_prime
        S_prime = np.argmax(P[A,S,:])

        Q[S,A] = Q[S,A] + alpha*(R+gamma*max(Q[S_prime,:])-Q[S,A])

        S = S_prime
```

In [52]:

```
print("Optimal policy:")
print(np.argmax(Q,axis=1))
print('\nOptimal Q:')
print(Q)
```

```
Optimal policy:
[1 1 0 0 0 0 0]

Optimal Q:
[[ 1.5        2.       ]
 [ 0.625      1.       ]
 [ 1.25       0.4737459]
 [ 2.5        0.62497  ]
 [ 5.         1.25     ]
 [10.         2.5      ]
 [20.        15.       ]]
```

d)

The differences between SARSA and the Q-learning algorithm lie in how the Q matrix is updated.

In the SARSA algorithm, Q is updated according to an action chosen using, in an epsilon-greedy way, a policy derived from Q.

However, in the Q-learning algorithm, Q is updated according to the optimal action given a new state, regardless of whether that action ends up being chosen next. This allows the Q-learning algorithm to move Q in an optimal direction even while the algorithm is in its explorative stages.

Question 5

In [59]:

```
# function adapted to take two-dimensional states
def epsilon_greedy_bigger(eps,Q,S,actions):
    if np.random.rand()<eps:
        # randomly choose action
        A = np.random.choice(actions)
    else:
        # use Q
```

```
        A = np.argmax(Q[S[0],S[1],:])
    return A
```

In [60]:

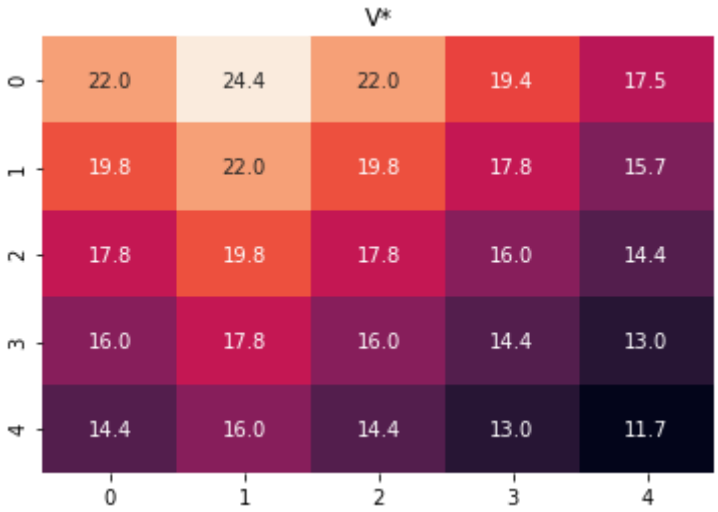
```
# system formulation: avoid large sparse adacency matrices
n_actions = 4
actions = np.array([[-1,0],[0,1],[1,0],[0,-1]])    # N, E, S, W

# reward indexed by row, column and action
grid = 5
rho = np.zeros((grid,grid,n_actions))
rho[0,:,0] = -1    #penalty for north if at top
rho[-1,:,2] = -1    #penalty for south if at bottom
rho[:,-1,1] = -1    #penalty for east if at right side
rho[:,0,3] = -1    #penalty for west if at left side
rho[0,1,:] = 10    #reward for a move from A
rho[0,3,:] = 5    #reward for a move from B
```

In [55]:

```
# parameters
gamma = 0.9
alpha = 0.2
eps = 0.1
n_epi = 1000
N = 1000

A = np.array([0,1])
B = np.array([0,3])

#input to be evaluated
Q = np.random.rand(grid,grid,n_actions)

#SARSA algorithm
for i in range(n_epi):
    S = np.random.choice(grid,size=2,replace=True)

    for j in range(N):
        #choose A
        A_idx = epsilon_greedy_bigger(eps,Q,S,n_actions)
        #observe R
        R = rho[S[0],S[1],A_idx]

        #observe S_prime
        if R == 10:
            S_prime = np.array([4,1])
        elif R == 5:
            S_prime = np.array([2,3])
        elif R == -1:
            S_prime = S
        else:
            S_prime = S+actions[A_idx]

        Q[S[0],S[1],A_idx] = Q[S[0],S[1],A_idx] + alpha*(R+gamma*max(Q[S_prim

        S = S_prime
```

In [56]:

```
v_star = np.amax(Q,axis=2)
sns.heatmap(v_star,annot=True,fmt='.1f',cbar=False)
plt.title('V*')
plt.show()
```



In [82]:
```
# process Q for some visual representation of the optimal policy
Q_2d = np.around(Q,1)    # tolerance of 1 decimal place
Q_maxes = np.amax(Q_2d,axis=2)

opt_pi = ["-_.,"]*grid**2

replaces = "-_.,"
dirs = "NESW"

for i in range(n_actions):
    q_layer = Q_2d[:,:,i]
    add_dir = np.where(q_layer.flatten()==Q_maxes.flatten())[0]
    for match in add_dir:
        opt_pi[match] = opt_pi[match].replace(replaces[i],dirs[i])
    #opt_pi[np.where(q_layer.flatten()==Q_maxes.flatten())][i] = dirs[i]
```

In [86]:
```
print("Estimated optimal policy:")
for i in range(grid):
    print(opt_pi[i*5:(i+1)*5])
```

```
Estimated optimal policy:
['-E.,', 'NESW', '-_.W', 'NES,', '-_.W']
['NE.,', 'N_.,', 'N_.W', '-_.W', 'N_.,']
['NE.,', 'N_.,', 'N_.W', 'N_.W', '-_.W']
['NE.,', 'N_.,', 'N_.W', 'N_.,', 'N_.,']
['NE.,', 'N_.,', 'N_.W', 'N_.,', '-_.W']
```

The values for v_* are very similar to those in the book - in the most recent run of the algorithm, only one value differed (in the decimal place) from any of the values in the textbook.

Similarly, the optimal policy seems to match the book well, except for the righthand side. Since we are using epsilon-greedy, it is likely that the reason why the righthand side is often missing an optimal direction or has a slightly less than optimal direction (as in the case of the upper right corner, second from the top) is that the less optimal space is underexplored, so Q is not refined in those areas.