SI140A Project : 王柯皓2021533025 徐培钧2021533041

Project: Performance Evaluation of Bandit Algorithms

- In this project, you will implement several classical bandit algorithms, evluate their performance via numerical comparison and finally gain inspiring intuition.

ϵ-greedy Algorithm ($0 \leq \epsilon \leq 1$)

Algorithm 1 ϵ-greedy Algorithm

Initialize $\widehat{\theta}(j) \leftarrow 0, \text{count}(j) \leftarrow 0, j \in \{1,2,3\}$

for $t = 1, 2 \ldots, N$ **do**

$$I(t) \leftarrow \begin{cases} \arg\max\limits_{j \in \{1,2,3\}} \ \widehat{\theta}(j) & w.p. \ 1-\epsilon \\ \text{randomly chosen from}\{1,2,3\} & w.p. \ \epsilon \end{cases}$$

  $\text{count}(I(t)) \leftarrow \text{count}(I(t)) + 1$

  $\widehat{\theta}(I(t)) \leftarrow \widehat{\theta}(I(t)) + \frac{1}{\text{count}(I(t))}\left[r_{I(t)} - \widehat{\theta}(I(t))\right]$

end for

UCB (Upper Confidence Bound) Algorithm

Algorithm 2 UCB Algorithm

for $t = 1, 2, 3$ **do**

  $I(t) \leftarrow t$

  $\text{count}(I(t)) \leftarrow 1$

  $\widehat{\theta}(I(t)) \leftarrow r_{I(t)}$

end for

for $t = 4, \ldots, N$ **do**

$$I(t) \leftarrow \arg\max_{j \in \{1,2,3\}} \left(\widehat{\theta}(j) + c \cdot \sqrt{\frac{2\log(t)}{\text{count}(j)}}\right)$$

  $\text{count}(I(t)) \leftarrow \text{count}(I(t)) + 1$

  $\widehat{\theta}(I(t)) \leftarrow \widehat{\theta}(I(t)) + \frac{1}{\text{count}(I(t))}\left[r_{I(t)} - \widehat{\theta}(I(t))\right]$

end for

Note: c is a positive constant with a default value of 1.

TS (Thompson Sampling) Algorithm

Algorithm 3 TS Algorithm

Initialize Beta parameter $(\alpha_j, \beta_j), j \in \{1,2,3\}$

1: **for** $t = 1, 2 \ldots, N$ **do**
2: # *Sample model*
3: **for** $j \in \{1,2,3\}$ **do**
4: Sample $\widehat{\theta}(j) \sim \text{Beta}(\alpha_j, \beta_j)$
5: **end for**
6: # *Select and pull the arm*

$$I(t) \leftarrow \arg\max_{j\in\{1,2,3\}} \widehat{\theta}(j)$$

7: # *Update the distribution*

$$\alpha_{I(t)} \leftarrow \alpha_{I(t)} + r_{I(t)}$$
$$\beta_{I(t)} \leftarrow \beta_{I(t)} + 1 - r_{I(t)}$$

8: **end for**

Problems

1. Now suppose we obtain the parameters of the Bernoulli distributions from an oracle, which are shown in the following table. Choose $N = 5000$ and compute the theoretically maximized expectation of aggregate rewards over N time slots. We call it the oracle value. Note that these parameters $\theta_j, j \in \{1,2,3\}$ and oracle values are unknown to all bandit algorithms.

Arm j	**1**	**2**	**3**
θ_j	0.7	0.5	0.4

Your anwser of problem 1 in Part I

We know that if we want to get the theoretically maximized expectation, we just need to the select largest $\theta_j, j \in \{1,2,3\}$ for all slots and we will get the expectation

$$E[\sum_{t=1}^{N} r_{I(t)}] = 5000 \cdot 0.7 = 3500$$

2. Implement aforemented three classical bandit algorithms with following settings:
   - $N = 5000$
   - ϵ-greedy with $\epsilon \in \{0.1, 0.5, 0.9\}$.
   - UCB with $c \in \{1, 5, 10\}$.
   - TS with
     - $\{(\alpha_1, \beta_1) = (1,1), (\alpha_2, \beta_2) = (1,1), (\alpha_3, \beta_3) = (1,1)\}$
     - $\{(\alpha_1, \beta_1) = (601,401), (\alpha_2, \beta_2) = (401,601), (\alpha_3, \beta_3) = (2,3)\}$

Your anwser of problem 2 in Part I

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import random, math, copy
### Import more packages if you need
```

In [ ]:
```
### Feel free to insert more blocks or helper functions if you need.
or_p=[0.7,0.5,0.4]
```

In [ ]:
```
### Implementation of epsilon-Greedy:
### n is the number of time slots, epsilon is the parameter of the algorithm
### return the total reward
def greedy(n, epsilon):
    #the theta we need to update
    gre_p=[0,0,0]
    #the count
    count=[0,0,0]
    #the total reward
    final_r=0
    for i in range(n):
        w=0
        p_1=np.random.uniform(0,1)
        #epsilon operation
        if(p_1<epsilon):
            a=random.choice([0,1,2])
            p_2=np.random.uniform(0,1)
            if(p_2<or_p[a]):
                final_r+=1
                w=1
            count[a]+=1
            gre_p[a]=gre_p[a]+(w-gre_p[a])/count[a]
            #1-epsilon operation
        else:
            a=gre_p.index(max(gre_p))
            p_2=np.random.uniform(0,1)
            if(p_2<or_p[a]):
                final_r+=1
                w=1
            count[a]+=1
            gre_p[a]=gre_p[a]+(w-gre_p[a])/count[a]
    return final_r
```

In [ ]:
```
### Implementation of UCB Algorithm:
### n is the number of time slots, c is the parameter of the algorithm
### return the total reward
def UCB(n, c):
    #the theta we need to update
    gre_p=[0,0,0]
    #the count
    count=[0,0,0]
    #theta + delta
    judge_p=[0,0,0]
    #the total reward
    final_r=0
    for i in range (n):
        w=0
        if(i==0 or i==1 or i==2):
            p_2=np.random.uniform(0,1)
            if(p_2<or_p[i]):
                final_r+=1
                w=1
            count[i]+=1
            gre_p[i]=w
        else:
            judge_p[0]=gre_p[0]+c*math.sqrt(2*math.log(i)/count[0])
```

```
            judge_p[1]=gre_p[1]+c*math.sqrt(2*math.log(i)/count[1])
            judge_p[2]=gre_p[2]+c*math.sqrt(2*math.log(i)/count[2])
            #get the index of the largest theta + delta
            a=judge_p.index(max(judge_p))
            p_2=np.random.uniform(0,1)
            if(p_2<or_p[a]):
                final_r+=1
                w=1
            count[a]+=1
            gre_p[a]=gre_p[a]+(w-gre_p[a])/count[a]
    return final_r
```

In [ ]:
```
### Implementation of TS Algorithm
### n is the number of time slots, a and b are the parameters of the algorithm
### return the total reward
def TS(n, a, b):
    #prior distribution
    B=[[a[0],b[0]],[a[1],b[1]],[a[2],b[2]]]
    gre_p=[0,0,0]
    final_r=0
    for i in range (n):
        w=0
        gre_p[0]=np.random.beta(B[0][0],B[0][1])
        gre_p[1]=np.random.beta(B[1][0],B[1][1])
        gre_p[2]=np.random.beta(B[2][0],B[2][1])
        #get the index based on the posterior distribution
        a=gre_p.index(max(gre_p))
        p_2=np.random.uniform(0,1)
        if(p_2<or_p[a]):
            final_r+=1
            w=1
        B[a][0]=B[a][0]+w
        B[a][1]=B[a][1]+1-w
    return final_r
```

3. Regard each of the above setting in problem 2 of Part I as an experiment (in total 8 experiments).

Run each experiment 200 independent trials (change the random seed). Plot the final result (in terms of rewards and regrets) averaged over these 200 trials.

Your anwser of problem 3 in Part I

In [ ]:
```
### Your code for problem 1.3. Feel free to insert more blocks or helper functic
result=[0,0,0,0,0,0,0,0]
for i in range (200):
    result[0]+=greedy(5000,0.01)
    result[1]+=greedy(5000,0.5)
    result[2]+=greedy(5000,0.9)
    result[3]+=UCB(5000,1)
    result[4]+=UCB(5000,5)
    result[5]+=UCB(5000,10)
    result[6]+=TS(5000,[1,1,1],[1,1,1])
    result[7]+=TS(5000,[601,401,2],[401,601,3])
result[0]=result[0]/200
result[1]=result[1]/200
result[2]=result[2]/200
```

```python
result[3]=result[3]/200
result[4]=result[4]/200
result[5]=result[5]/200
result[6]=result[6]/200
result[7]=result[7]/200
print("The rewards:")
print("greedy algorithm with 0.1", result[0])
print("greedy algorithm with 0.5", result[1])
print("greedy algorithm with 0.9", result[2])
print("ucb algorithm with 1", result[3])
print("ucb algorithm with 5", result[4])
print("ucb algorithm with 10", result[5])
print("ts algorithm with (1,1),(1,1),(1,1)", result[6])
print("ts algorithm with (601,401),(401,601),(2,3)", result[7])
print("The regrets:")
print("greedy algorithm with 0.1", 3500-result[0])
print("greedy algorithm with 0.5", 3500-result[1])
print("greedy algorithm with 0.9", 3500-result[2])
print("ucb algorithm with 1", 3500-result[3])
print("ucb algorithm with 5", 3500-result[4])
print("ucb algorithm with 10", 3500-result[5])
print("ts algorithm with (1,1),(1,1),(1,1)", 3500-result[6])
print("ts algorithm with (601,401),(401,601),(2,3)", 3500-result[7])
```

```
The rewards:
greedy algorithm with 0.1 3488.18
greedy algorithm with 0.5 3075.595
greedy algorithm with 0.9 2745.695
ucb algorithm with 1 3410.46
ucb algorithm with 5 2980.885
ucb algorithm with 10 2827.24
ts algorithm with (1,1),(1,1),(1,1) 3484.075
ts algorithm with (601,401),(401,601),(2,3) 3489.835
The regrets:
greedy algorithm with 0.1 11.820000000000164
greedy algorithm with 0.5 424.4050000000002
greedy algorithm with 0.9 754.3049999999998
ucb algorithm with 1 89.53999999999996
ucb algorithm with 5 519.1149999999998
ucb algorithm with 10 672.7600000000002
ts algorithm with (1,1),(1,1),(1,1) 15.925000000000182
ts algorithm with (601,401),(401,601),(2,3) 10.164999999999964
```

Thus, we have the answer

$$E(R)_g = \begin{cases} 3488.18 & \epsilon = 0.1 \\ 3075.56 & \epsilon = 0.5 \\ 2745.70 & \epsilon = 0.9 \end{cases}$$

$$E(R)_u = \begin{cases} 3410.46 & c = 1 \\ 2980.89 & c = 5 \\ 2827.24 & c = 10 \end{cases}$$

$$E(R)_t = \begin{cases} 3480.08 & (\alpha_1,\beta_1),(\alpha_2,\beta_2),(\alpha_3,\beta_3) = (1,1),(1,1),(1,1) \\ 3489.84 & (\alpha_1,\beta_1),(\alpha_2,\beta_2),(\alpha_3,\beta_3) = (601,401),(401,601),(2,3) \end{cases}$$

$$re_g = \begin{cases} 11.82 & \epsilon = 0.1 \\ 424.41 & \epsilon = 0.5 \\ 754.30 & \epsilon = 0.9 \end{cases}$$

$$re_u = \begin{cases} 89.54 & c = 1 \\ 519.11 & c = 5 \\ 672.76 & c = 10 \end{cases}$$

$$re_t = \begin{cases} 15.93 & (\alpha_1, \beta_1), (\alpha_2, \beta_2), (\alpha_3, \beta_3) = (1,1), (1,1), (1,1) \\ 10.16 & (\alpha_1, \beta_1), (\alpha_2, \beta_2), (\alpha_3, \beta_3) = (601,401), (401,601), (2,3) \end{cases}$$

4. Compute the gaps between the algorithm outputs (aggregated rewards over N time slots) and the oracle value. Compare the numerical results of ϵ-greedy, UCB, and TS.
   - Which one is the best?
   - Discuss the impacts of ϵ, c, and α_j, β_j, respectively.

Your anwser of problem 4 in Part I

the Gap between the algorithm outputs and the oracle value are shown below

$$Gap_g = \begin{cases} 11.82 & \epsilon = 0.1 \\ 424.41 & \epsilon = 0.5 \\ 754.30 & \epsilon = 0.9 \end{cases}$$

$$Gap_u = \begin{cases} 89.54 & c = 1 \\ 519.11 & c = 5 \\ 672.76 & c = 10 \end{cases}$$

$$Gap_t = \begin{cases} 15.93 & (\alpha_1, \beta_1), (\alpha_2, \beta_2), (\alpha_3, \beta_3) = (1,1), (1,1), (1,1) \\ 10.16 & (\alpha_1, \beta_1), (\alpha_2, \beta_2), (\alpha_3, \beta_3) = (601,401), (401,601), (2,3) \end{cases}$$

In [ ]:

```
### Your code for problem 1.4. Feel free to insert more blocks or helper functio
# helper func based on greedy for multi-choice
def mean1(x1,j,h):
    a=0
    for i in range (h):
        a+=greedy(j,x1)
    a=a/h
    return a
# helper func based on UCB for multi-choice
def mean2(x1,j,h):
    a=0
    for i in range (h):
        a+=UCB(j,x1)
    a=a/h
    return a
# helper func based on TS for multi-choice
def mean3(x1,x2,j,h):
    a=0
    for i in range (h):
        a+=TS(j,x1,x2)
    a=a/h
```

```
    return a
# helper func based on greedy for finding the theta*
def mean1_1(x1,j,h):
    a=0
    for i in range (h):
        if(j!=0):
            a+=greedy(j,x1)/j
    a=a/h
    return a
# helper func based on UCB for finding the theta*
def mean2_1(x1,j,h):
    a=0
    for i in range (h):
        if(j!=0):
            a+=UCB(j,x1)/j
    a=a/h
    return a
# helper func based on TS for finding the theta*
def mean3_1(x1,x2,j,h):
    a=0
    for i in range (h):
        if(j!=0):
            a+=TS(j,x1,x2)/j
    a=a/h
    return a
```

First we consider the numerical results of ϵ-greedy, UCB, and TS.

In order to avoid the error caused by array order, we assume the real paramters to be 、

Arm j	**1**	**2**	**3**
θ_j	0.4	0.7	0.5

so the result won't change.

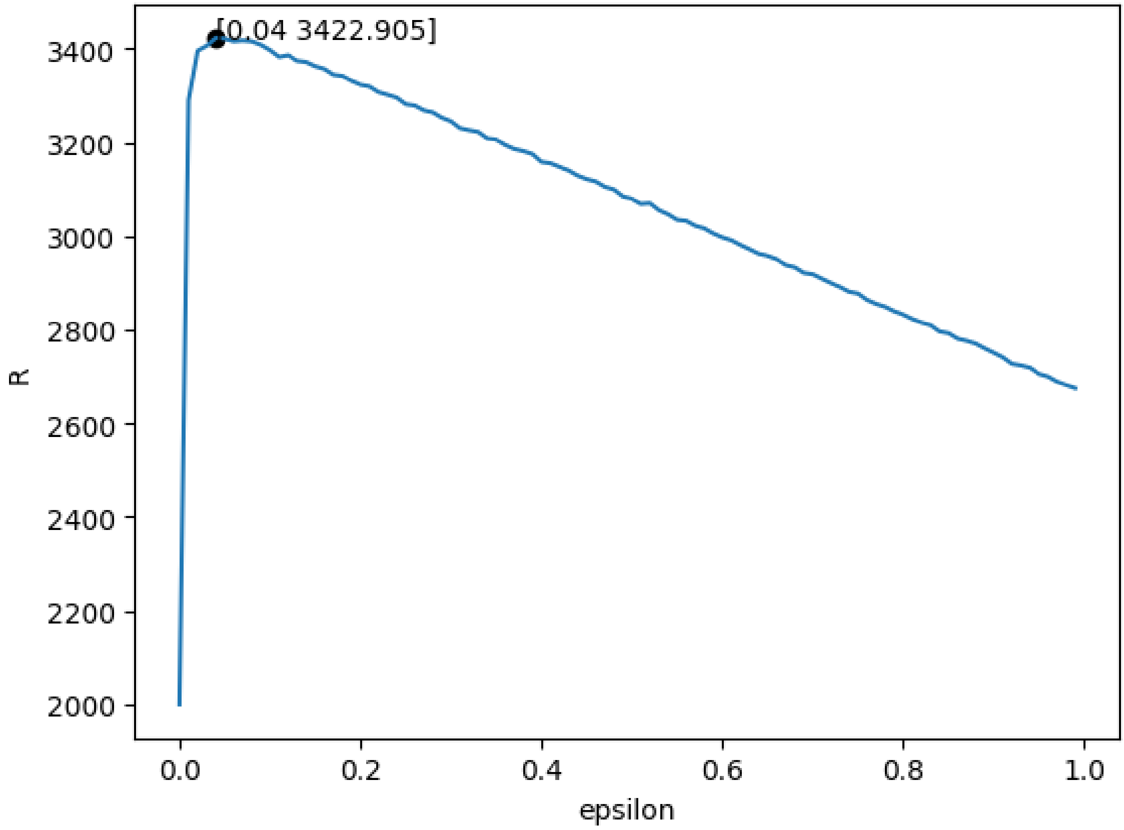
To start with, we want to know the best proper results that each algorithm can provide. Thus, we first foucs on the ϵ-greedy algorithm and we let the ϵ to be various to find the value of ϵ which could provide the best result.

Then, we will get a plot below.

（See complete code in the code file sent through email）

In [ ]:
```
or_p[0]=0.4
or_p[1]=0.7
or_p[2]=0.5
x1=[i/100 for i in range(100)]
y1=[mean1(x1[i],5000,200) for i in range(100)]
y_1_max=y1.index(max(y1))
show_max='['+str(x1[y_1_max])+' '+str(y1[y_1_max])+']'
plt.plot(x1[y_1_max],y1[y_1_max],'ko')
plt.plot(x1,y1)
plt.annotate(show_max,xy=(x1[y_1_max],y1[y_1_max]),xytext=(x1[y_1_max],y1[y_1_ma
```

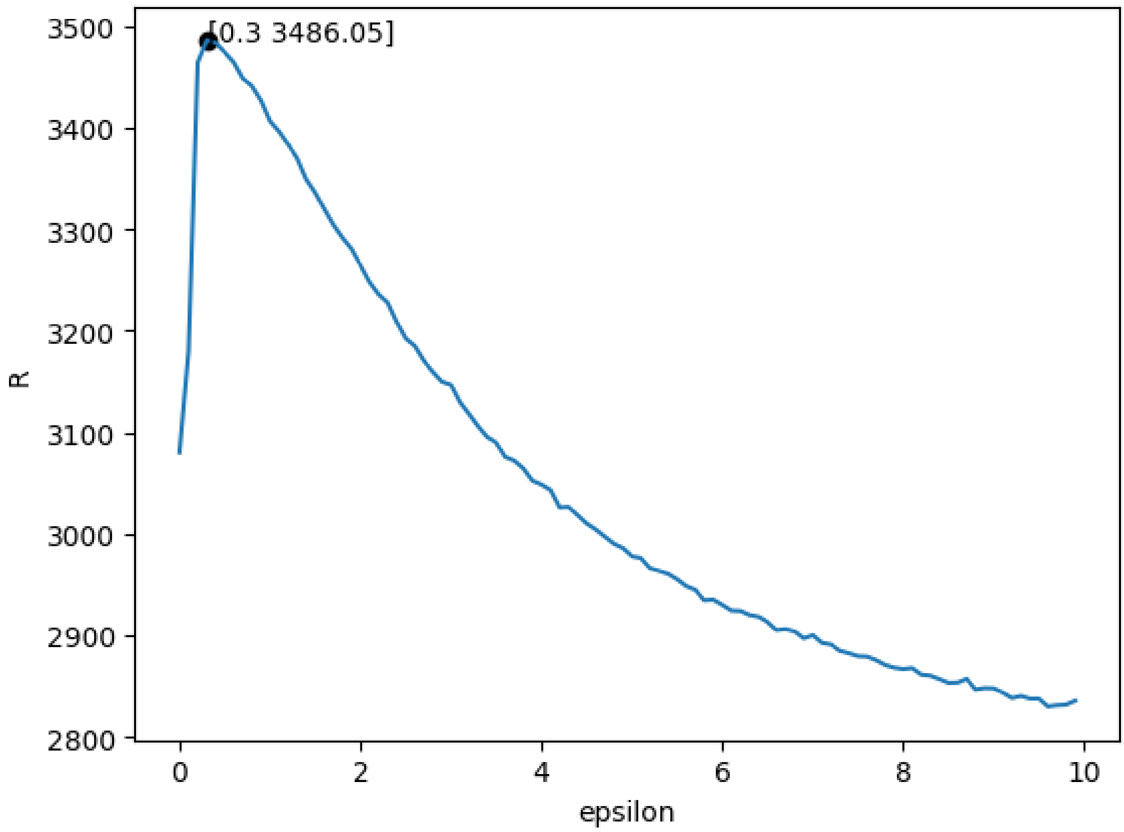
```
plt.xlabel("epsilon")
plt.ylabel("R")
plt.show()
```



From the plot, we can get the ideal ϵ to be 0.04 whose leading result is 3422.905. Then, we apply the same method to the UCB algorithm to find the proper c. We will get another plot shown below.

(See complete code in the code file sent through email)

In [ ]:
```
x2=[i/10 for i in range(100)]
y2=[mean2(x2[i],5000,200) for i in range(100)]
y_2_max=y2.index(max(y2))
show_max='['+str(x2[y_2_max])+' '+str(y2[y_2_max])+']'
plt.plot(x2[y_2_max],y2[y_2_max],'ko')
plt.plot(x2,y2)
plt.annotate(show_max,xy=(x2[y_2_max],y2[y_2_max]),xytext=(x2[y_2_max],y2[y_2_ma
plt.xlabel("epsilon")
plt.ylabel("R")
plt.show()
```



From the plot, we can get the ideal c to be 0.3 whose leading result is 3486.05.

Finally, we focus on the TS algorithm. We know that the core idea of TS algorithm is Beta-Binomial Conjugacy so the proper α β should be close to the ideal number of successes or failures in a fixed N times tries with a certain arm according to the real probability of its θ

Therefore, according to the oracle's probability. We will get one possible ideal pairs of α β to be $(\alpha_{0.7}, \beta_{0.7}) = (700, 300)$,$(\alpha_{0.5}, \beta_{0.5}) = (500, 500)$, $(\alpha_{0.4}, \beta_{0.4}) = (400, 600)$

(We know that the number of α β could represent the number of successes or failures in the previous experiments so the larger these values are means the more experiments had been done, and the more accurate our prior distribution will be. However, we cannot allow them to increase infinitely here. Therefore, we specify the total of number of tries to be 1000)

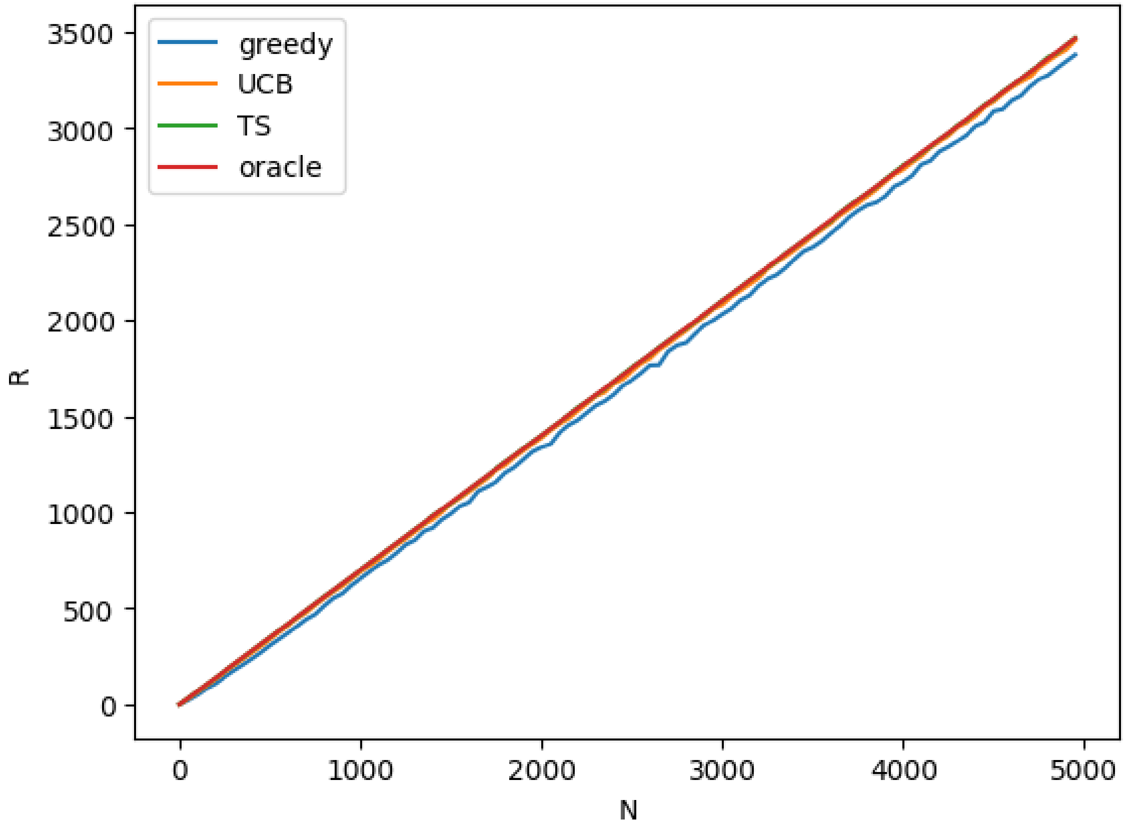
From the analysis above, we change the settings of three algorithm:

- ϵ-greedy with $\epsilon = 0.04$.
- UCB with $c = 0.3$.
- TS with
  - $\{(\alpha_1, \beta_1) = (400, 600), (\alpha_2, \beta_2) = (700, 300), (\alpha_3, \beta_3) = (500, 500)\}$

Then, we consider the change in the final results and the gaps by varying the total number of time slots i.e. N and we will get two plots below

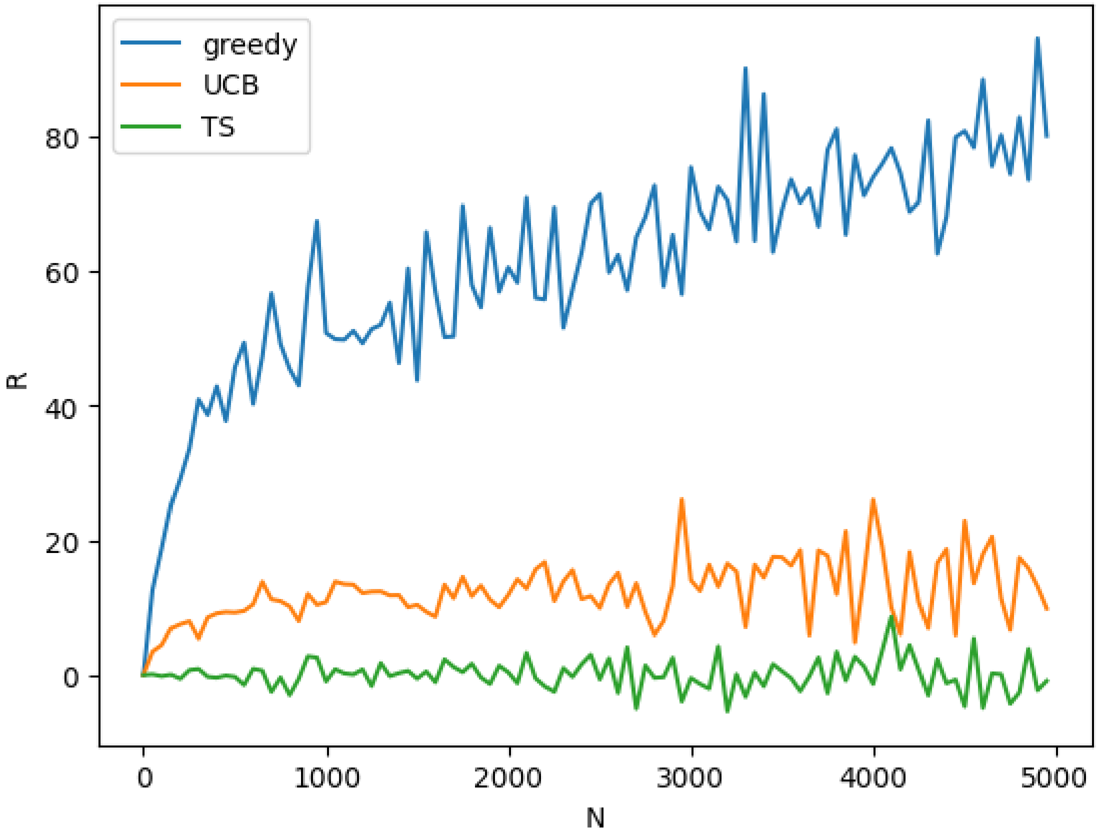
In [ ]:
```
x1=[50*i for i in range(100)]
y1=[mean1(0.04,x1[i],100) for i in range(100)]
```

```
x2=[50*i for i in range(100)]
y2=[mean2(0.3,x2[i],100) for i in range(100)]
x3=[50*i for i in range(100)]
y3=[mean3([400,700,500],[600,300,500],x3[i],100) for i in range(100)]
x4=[50*i for i in range(100)]
y4=[0.7*x4[i] for i in range(100)]
plt.plot(x1,y1,label="greedy")
plt.plot(x2,y2,label="UCB")
plt.plot(x3,y3,label="TS")
plt.plot(x4,y4,label="oracle")
plt.legend()
plt.xlabel("N")
plt.ylabel("R")
plt.show()
```



In [ ]:

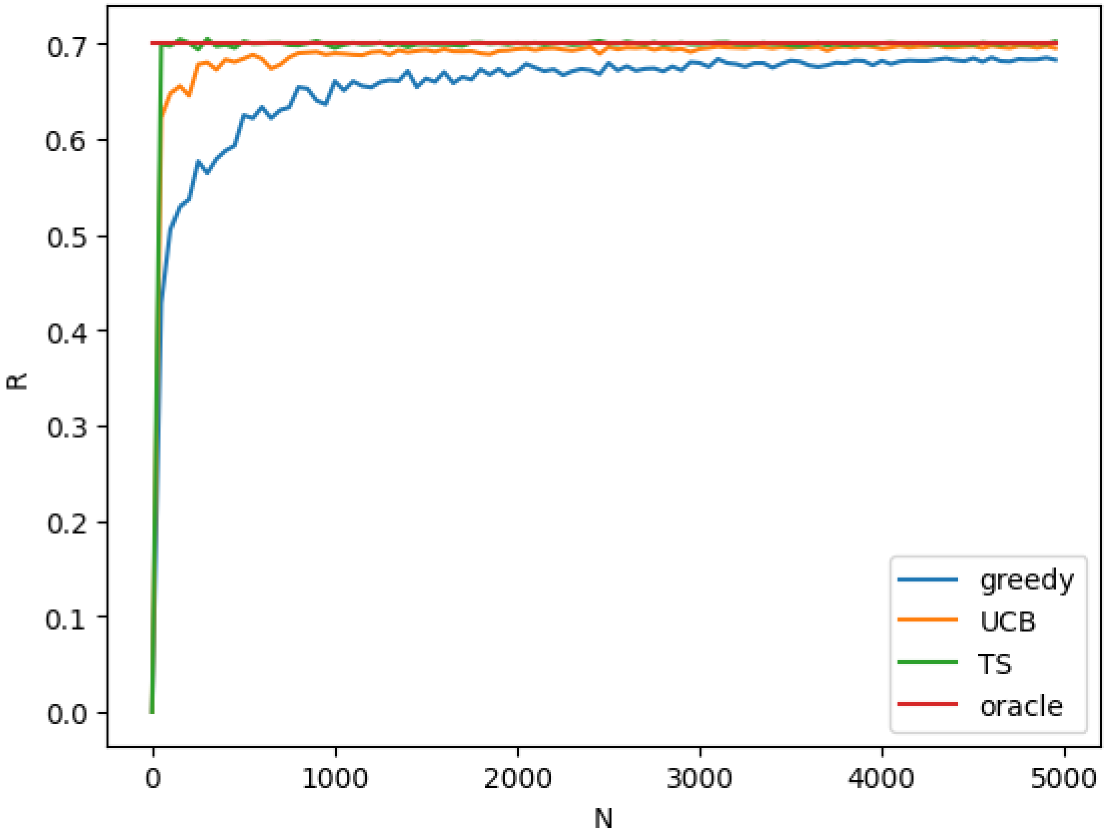
```
x1=[50*i for i in range(100)]
y1=[0.7*x1[i]-mean1(0.04,x1[i],100) for i in range(100)]
x2=[50*i for i in range(100)]
y2=[0.7*x2[i]-mean2(0.3,x2[i],100) for i in range(100)]
x3=[50*i for i in range(100)]
y3=[0.7*x3[i]-mean3([400,700,500],[600,300,500],x3[i],100) for i in range(100)]
plt.plot(x1,y1,label="greedy")
plt.plot(x2,y2,label="UCB")
plt.plot(x3,y3,label="TS")
plt.legend()
plt.xlabel("N")
plt.ylabel("R")
plt.show()
```



From the plot above, it is obvious that the greedy algorithm is worse than the other two algorithms because its gap tending towards linear growth.

Additionally, when we compare the the reference θ^* (the largest θ we got by averaging the value of the θ_j we choose each slot), we will get another plot.

In [ ]:
```
x1=[50*i for i in range(100)]
y1=[mean1_1(0.04,x1[i],50) for i in range(100)]
x2=[50*i for i in range(100)]
y2=[mean2_1(0.3,x2[i],50) for i in range(100)]
x3=[50*i for i in range(100)]
y3=[mean3_1([400,700,500],[600,300,500],x3[i],50) for i in range(100)]
x4=[50*i for i in range(100)]
y4=[0.7 for i in range(100)]
plt.plot(x1,y1,label="greedy")
plt.plot(x2,y2,label="UCB")
plt.plot(x3,y3,label="TS")
plt.plot(x4,y4,label="oracle")
plt.legend()
plt.xlabel("N")
plt.ylabel("R")
plt.show()
```



From the plot, we know that the reference θ^* gained from the TS algorithm is closer to the real one. Generally, we find that with the best parameters we found, TS performed best, the UCB performed second best while the greedy's result seemed to be the worst.

However, the results obtained from the image are similar to the theoretical analysis results but we still need to discuss the difference between them through theoretical analysis to gain a more convincing conclusion.

To better compare these three algorithms We decide to give a brief introduction to them.

1. Greedy algorithm

   - As we all know, the sample mean

     $$\bar{X} = \frac{1}{n}\sum_{j=1}^{n} X_j$$

     tends to the mean μ of the real distribution when n is large. Greedy algorithm also uses this property. We know that for every θ_j, we let

     $$\theta_j = \frac{\sum r_{\theta_j}}{n}$$

     where r_{θ_j} is the reward gained through θ_j and n is the number of slots that we select θ_j. Thus, we have when $n->\infty$, θ_j is close to the value gained from oracle. Furthermore, we specify two operations, one is that we have the probability of ϵ to explore the value of each θ_j randomly and change the θ_j from the feedback

$$\theta_j = \frac{\sum r_{\theta j} + 1/0}{n+1}$$

and the other one is that we have the probability of $1-\epsilon$ to select the largest θ_j based on the value of θ_j we have now. During the whole experiment, we just keep selecting one of the two operations and ultimately get the result.

2. UCB algorithm

   - The UCB algorithm also based on the property of the sample mean which means that we update our θ_j also by applying

$$\theta_j = \frac{\sum r_{\theta j} + 1/0}{n+1}$$

. However, it uses the Chernoff-Hoeffding Bound to consider the difference between every θ_j we updated in time slots and its value gained from oracle.

$$P(|\theta_j - \theta_\mu| \le \delta) \ge 1 - 2e^{-2n\delta^2}$$

(θ_μ is the value gained from oracle)

Thus, we just need to select a proper value of δ to let $1-2e^{-2n\delta^2}$ become small enough, and we could regard $\theta_j + \delta$ a new θ_j for our comparison to find the proper $I(t)$ in each slot and we define them as θ_j^1. Due to the $1-2e^{-2n\delta^2}$ becoming small, θ_j^1 will be extremely similar to the θ_μ. Therefore, finding the proper $I(t)$ through θ_j^1 is more reasonable. According to the caculation, we find that when $\delta = \sqrt{\frac{2lnt}{n}}$ (where t is the number of slots and n is the number of slots selecting θ_j), we have

$$P(|\theta_j - \theta_\mu| \le \sqrt{\frac{2lnt}{n}}) \ge 1 - \frac{2}{t^4}$$

the $1-\frac{2}{t^4}$ is small enough so we can define $\delta = c\sqrt{\frac{2lnt}{n}}$ and get the result by finding the max θ_j^1 in all time slots.

3. TS algorithm

   - The TS algorithm is mainly based on Beta-Binomial Conjugacy. Thus, once we gain the proper prior distribution for each θ_j, we can just regard each slot as a bernouli trail with the selected θ_j. If the reward is 1, we just gain the posterior distribution as $(\alpha+1,\beta)$, if the reward is 0, we just gain the posterior distribution as $(\alpha,\beta+1)$. Then, we let this posterior distribution to be the new prior distribution and so on.

Based on the introduction above, we know that for the greedy algorithm, there is a probability of ϵ to explore the value of each θ_j randomly, but this randomness will lead to a huge difference in the number of times each θ_j is explored, which leads to that some θ_j may only be explored several times in the entire experiment process. Too few explorations of certain θ_j will result in a large difference between itself and the real possibility, which leads to the results that do not match the oracle's values. Thus, the greedy algorithm is the worst among these three algorithms.

Then, the UCB algorithm has some improvement comparing to the greedy algorithm because for those θ_j s that have been contiuely selected throughout the initial stage, the algorithm would reduce their value of δ because the distance $|\theta_j - \theta_\mu|$ became smaller which meaned we just needed a smaller δ to fill part of this distance, so their θ_j^1 participating in the comparison was basically equivalent to the θ_j obtained from the experiment. For those θ_j that were not selected at the initial stage, their δ s continued to increase because as $1 - 2e^{-2n\delta^2}$ became smaller with larger t, the distance $|\theta_j - \theta_\mu|$ didn't change, so we needed a larger δ to fill part of the distance to satisfy the Chernoff-Hoeffding Bound. If their real value was greater than some selected θ_j s', then their δ s' increase would ultimately cause the value of their θ_j^1 used for comparison to exceed others θ_j^1 corresponding to the θ_j s that have been selected but its real value is smaller than theirs. Therefore, they could be more likely to be selected in the remaining slots.

For the TS algorithm, due to the existence of a prior distribution, we can make the difference between θ_j and the value gained by oracle tend to 0 in very few experiments, as long as we can obtain an accurate prior distribution. That is to say, if our prior distribution is accurate enough, the TS algorithm would be better than the UCB algorithm because we do not need a lot of experiments to update the value of θ_j and reduce the value of δ to gain the ideal value. However, if the prior distribution is not accurate, the TS algorithm would be worse than the UCB algorithm because an improper prior distribution may require us to correct the errors in the prior distribution through more experiments, resulting in a posterior distribution that is closer to the true distribution.

To sum up, in my opinion, the UCB algorithm is the best algorithm among the three algorithms because it is always difficult to obtain an appropriate prior distribution in practical problems, which requires a large number of experimental samples, and an improper prior distribution will cause us to spend more time to form a correct posterior distribution. As for the Greedy algorithm, the randomness problem in it might lead to a large deviation in the result. Therefore, the UCB algorithm has a greater fault tolerance rate and a more stable demand for time slots, making it the preferred choice in real applications.

Second, we focus on the impacts of ϵ, c, and α_j, β_j, respectively.

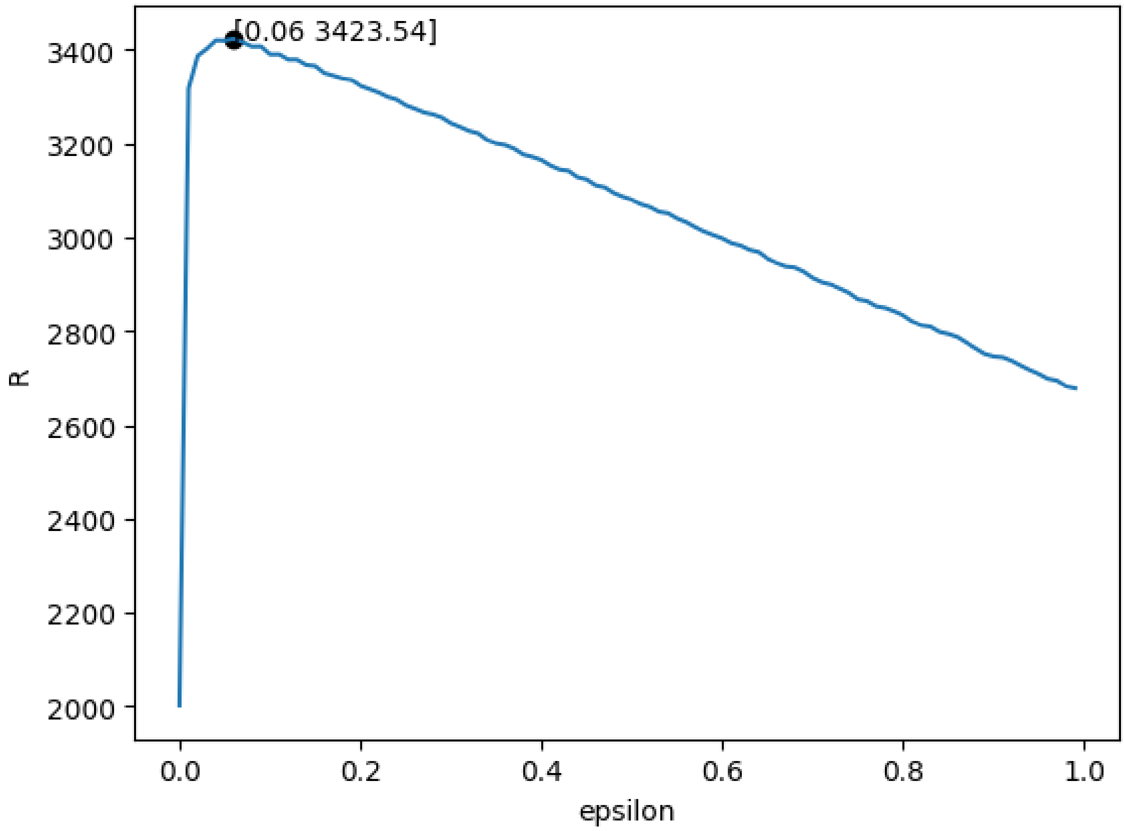
1. ϵ:
    - We know that for every slot, ϵ is the probability to randomly choose one of the θ_j s and update its value through our reward. Therefore, if ϵ is large, Our algorithm process tends to randomly choose one arm from multiple arms each slot to play the game, rather than the arm with the highest probability of winning a reward. Thus, ϵ should be as small as possible to reduce the impact of randomness. However, if the ϵ is too small, we cannot gain a proper exploration process because once we update a θ_j by random selection, we will use this θ_j

for a extreme long time because other θ_j couldn't be selected randomly for a long time to update so they remained 0. Thus, the largest θ_j will be the first fortunate θ_j we select in a rare opportunity of random selection and this will definitely lead to a wrong result.

Therefore, ϵ should not be extreme small or big and from the plot, we get the proper ϵ is 0.06.

(See complete code in the code file sent through email)

In [ ]:
```
x1=[i/100 for i in range(100)]
y1=[mean1(x1[i],5000,200) for i in range(100)]
y_1_max=y1.index(max(y1))
show_max='['+str(x1[y_1_max])+' '+str(y1[y_1_max])+']'
plt.plot(x1[y_1_max],y1[y_1_max],'ko')
plt.plot(x1,y1)
plt.annotate(show_max,xy=(x1[y_1_max],y1[y_1_max]),xytext=(x1[y_1_max],y1[y_1_ma
plt.xlabel("epsilon")
plt.ylabel("R")
plt.show()
```



2. c

- From the Chernoff-Hoeffding Bound, we know that if we want that the θ_j could use less number of time slots to tend towards θ_μ, the value gained from oracle, we need $1-2e^{-2n\delta^2}$ to be smaller. We have

$$P(|\theta_j - \theta_\mu| \le c\sqrt{\frac{2lnt}{n}}) \ge 1 - \frac{2}{t^{4c^2}}$$

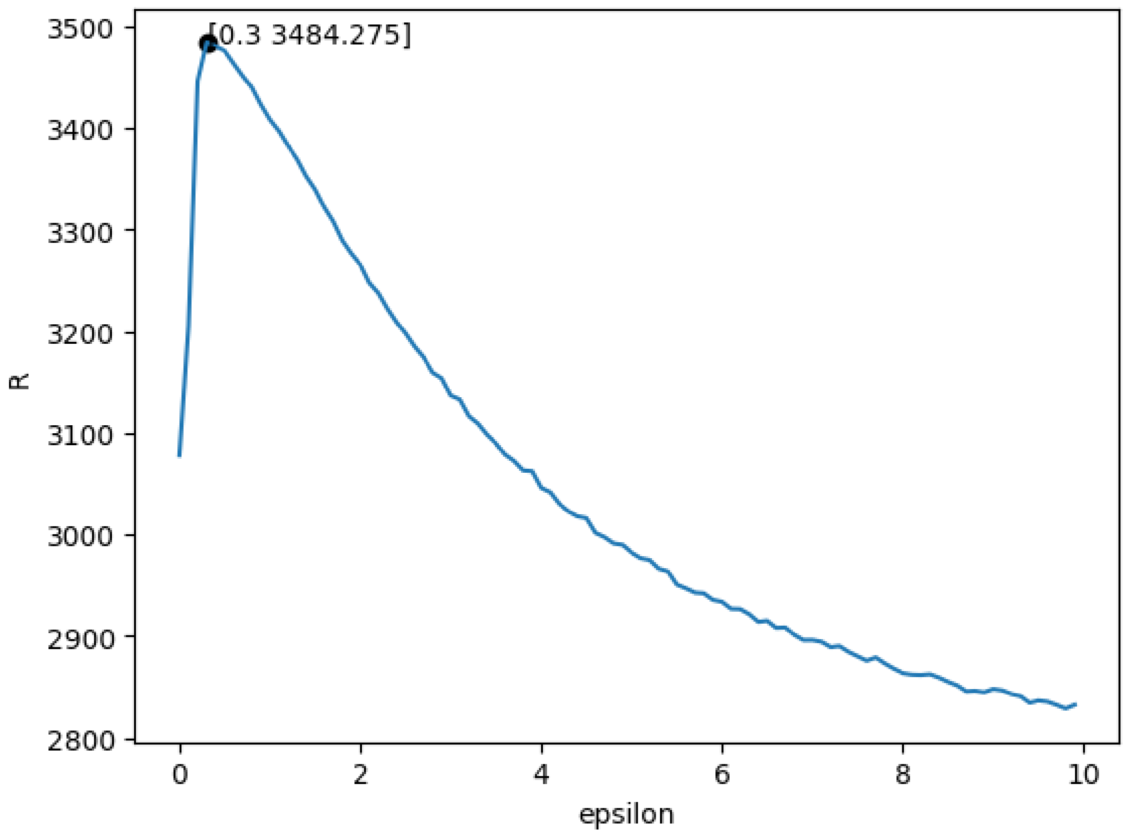
. We find that c should be as small as possible to reduce the value of $1 - 2e^{-2n\delta^2}$. However, there is one thing that we need to pay attention to which is that we still need $\theta_j + \delta$ to be the basis for us to select $I(t)$ so if $c- > 0$, we have $\delta- > 0$ and $\theta_j + \delta- > \theta_j$. Therefore, we just compare the θ_j based on their Initial value and it will also lead to a wrong result.

Thus, c should not be extreme small or big and from the plot, we get the proper c is 0.3.

(See complete code in the code file sent through email)

In [ ]:

```
x2=[i/10 for i in range(100)]
y2=[mean2(x2[i],5000,200) for i in range(100)]
y_2_max=y2.index(max(y2))
show_max='['+str(x2[y_2_max])+' '+str(y2[y_2_max])+']'
plt.plot(x2[y_2_max],y2[y_2_max],'ko')
plt.plot(x2,y2)
plt.annotate(show_max,xy=(x2[y_2_max],y2[y_2_max]),xytext=(x2[y_2_max],y2[y_2_ma
plt.xlabel("epsilon")
plt.ylabel("R")
plt.show()
```



3. α_j, β_j

- From the analysis above, we know that if α_j, β_j can form a accurate prior distribution, the results we obtained in the initial stage of the experiment will have a very small difference from the ideal value. However, if our prior distribution is improper, we need extensive number of experiments to correct our distribution. As for the example given for this problem, a proper (α_j, β_j) for a θ_j should be $(\theta_\mu M, (1-\theta_\mu)M)$ where M is large and θ_μ is the value we gain from oracle.

5. Give your understanding of the exploration-exploitation trade-off in bandit algorithms.

Your anwser of problem 5 in Part I

1. ϵ-greedy

   - The main exploration part of ϵ-greedy algorithm is the operation that we randomly choose a θ_j to update its value. The main exploitation part of it is the operation we choose the largest θ_j based on the current value of all θ_j. Therefore, for each slot, we have a probability of ϵ to perform the main exploration process and a probability of $1-\epsilon$ to perform the main exploitation process, which means the trade-off of the exploration-exploitation is based on the value of ϵ. We find that the process of randomly selection is a total exploration process but for the process of selecting the largest θ_j, it not only can be regard as a exploration process because it still need to receive the feedback to update the value of θ_j, but also can be regard as a exploitation process because in every slot, we select the arm with the largest probability to gain the reward. Therefore, it is possible for us to abandon the complete random exploration part because it is inefficient and might not be beneficial to all θ_j. Therefore, we decide to only remain the $1-\epsilon$-operation and hope to improve it.

2. UCB

   - Based on the ϵ-greedy algorithm, the UCB algorithm successfully improved the $1-\epsilon$-operation by using the Chernoff-Hoeffding Bound so we can explore and exploit at the same time, which means there is a better balance between them. It compares $\theta_j+\delta$ instead of the single θ_j and select the largest $\theta_j+\delta$ to update the corresponding θ_j .
   - This algorithm ensure the comprehensiveness of exploration with the changes in δ during the whole process as we mentioned in the answer of problem 3. First, our exploration can be regared randomly at the beginning because all θ_j s' Initial value just based on one exploration result. Second, we begin to use Chernoff-Hoeffding Bound to select the "best choice" during the process. For the θ_j s which were selected during the initial random selection stage, their $\theta_j+\delta$ became closer to θ_μ which is the value gained from oracle, and θ_j s also became closer to θ_μ so their δ became smaller. However, for the θ_j s which were not selected during the initial random selection stage, their $\theta_j+\delta$ would

also trend to θ_μ because when t became larger, $1 - \frac{2}{t^{4c^2}}$ became smaller but θ_j s didn't change. Therefore, although the value of selected θ_j seemed to become larger than the non-selected θ_j s' at the beginning, if the non-selected ones'θ_μ is larger than the selected one, the non-selected one will definitely be selected as well during the process with the large number of time slots because the distance between the $\theta_j + \delta$ of the non-selected one and the θ_μ will finally smaller than the distance between the $\theta_j + \delta$ of the non-selected one and the $\theta_j + \delta$ of the selected one.

- Additionally, the UCB algorithm also optimizes the exploitation process. The greedy algorithm uses θ_j as the basis for choosing $I(t)$, but the initial value of θ_j is far from the real value of it for a long time, so taking it as the basis will bring a certain amount of errors at the beginning. However, the USB algorithm uses $\theta_j + \delta$ instead of θ_j because through the Chernoff-Hoeffding Bound, $\theta_j + \delta$ will trend to θ_μ as t becoming larger, even the initial $\theta_j + \delta$ is much closer to θ_μ than θ_j is. Therefore, our exploitation would depend on a more accurate value and our result will also be better.

3. TS

- The TS algorithm is a little different from the two other algorithm we mentioned above. Instead of starting with the exploration, the TS algorithm entered the exploitation part in its initial stage. It have to use its prior distribution to obtain the posterior distribution and then get the result. It also mixs the exploitation part and the exploration part into one step as the UCB algorithm does. However, the exploration and the exploitation are more closely related in the TS algorithm because, for example, our q th exploitation is completely based on the posterior distribution, and the q th posterior distribution is completely based on the $q-1$ th posterior distribution (our $q-1$ th exploitation based on) and our q th exploration. Therefore, the count of slots and the count of reward do not have a direct impact on exploitation and exploration. The exploitation and exploration just interact with each other. Thus, if our initial prior distribution is extremely closer to the real distribution, we will get a perfect posterior distribution in a very small number of slots because the initial exploitation part will then lead to a ideal exploration result (a ideal reward) and the exploration result will make our posterior distribution more reasonable so the next exploitation can perform better. Under such mutual promotion, our results will be more accurate than the two algorithms above. However, if our initial prior distribution is bad, the bad exploitation result will lead to a poor result and we need a extremely long time to optimize our posterior distribution because we can only make corrections through the results of exploration.

6. We implicitly assume the reward distribution of these three arms are independent. How about the dependent case? Can you design an algorithm to exploit such information to obtain a better result?

Your anwser of problem 6 in Part I

We find that if the cases are no longer independent, it will be extremely hard to find a certain distribution for the θ_j. Thus, we decide to give up focusing on the distribution of these θ_j and we consider on how to get the highest rewards. Thus, we just need to focus on the probability of choosing each arm and we define it as $P_{t,j}$. We hope that the arm with the largest probability could also be the arm we exactly need for maximizing the ideal total reward. We find a algorithm called "Exp3" to solve this problem.

First, we define R_t as reward we get for each time slot so we have

$$P_{t,j} = P(I_t = j | I_1, I_2, A_3, \dots, I_{t-1}, R_1, R_2, \dots R_{t-1})$$

which shows that $P_{t,j}$ is based on the choices and rewards in the previous $t-1$ slots.

Second, we define a estimator

$$\bar{r}_{t,j} = 1 - \frac{I(I(t)=j)(1-r_t)}{P_{t,j}}$$

where $I(I(t)=j)$ is an indicator to indicate whether we have chosen the j th arm so we have

$$E(\bar{r}_{t,j}) = 1 - \frac{1-r_t}{P_{t,j}} E(I(I(t)=j)) = 1 - \frac{1-r_t}{P_{t,j}} P_{t,j} = r_t$$

which means that based on the previous $t-1$ slots'conditions, the expectation of the estimated value $\bar{r}_{t,j}$ is basically consistent with r_t

Third, we define $S_{t,j} = \sum_{i=1}^{t} \bar{r}_{t,j}$ so we can map $S_{t,j}$ into probability by a method called exponential weighting. Then, we will get a equation:

$$P_{t,j} = \frac{P_{t-1,j} e^{\eta S{t-1,j}}}{\sum_{i=1}^{t} P_{t-1,j} e^{\eta S{t-1,i}}}$$

where η is what we called learn rate. When η is large, we are more likely to apply exploitation strategy which means that we generally select the largest θ_j of all current value to obtain the reward, just as the $1-\epsilon$-operation in ϵ greedy algorithm. However, when η is small, we are more likely to apply exploration strategy which means that we generally select the θ_j randomly to obtain the reward, just as the ϵ-operation in ϵ greedy algorithm.

The pseudo code for the following operation is shown below.

Exp3
input η

Initialize $S_{0,j}=0, P_{1,j}=\frac{1}{3}, j\in\{1,2,3\}$

1: **for** $t=1,2,\ldots,N$ do

2: $\quad P_{t,j}=\frac{P_{t-1,j}e^{\eta S_{t-1,j}}}{\sum_{i=1}^{t}P_{t-1,j}e^{\eta S_{t-1,i}}}$

3: $\quad I(t)\leftarrow \arg\max_{j\in\{1,2,3\}} P_{t,j}$

4: $\quad$ **for** $j=1,2,3$

5: $\qquad S_{t,j}=S_{t-1,j}+1-\frac{I(A_t=j)(1-r_t)}{P_{t,j}}$

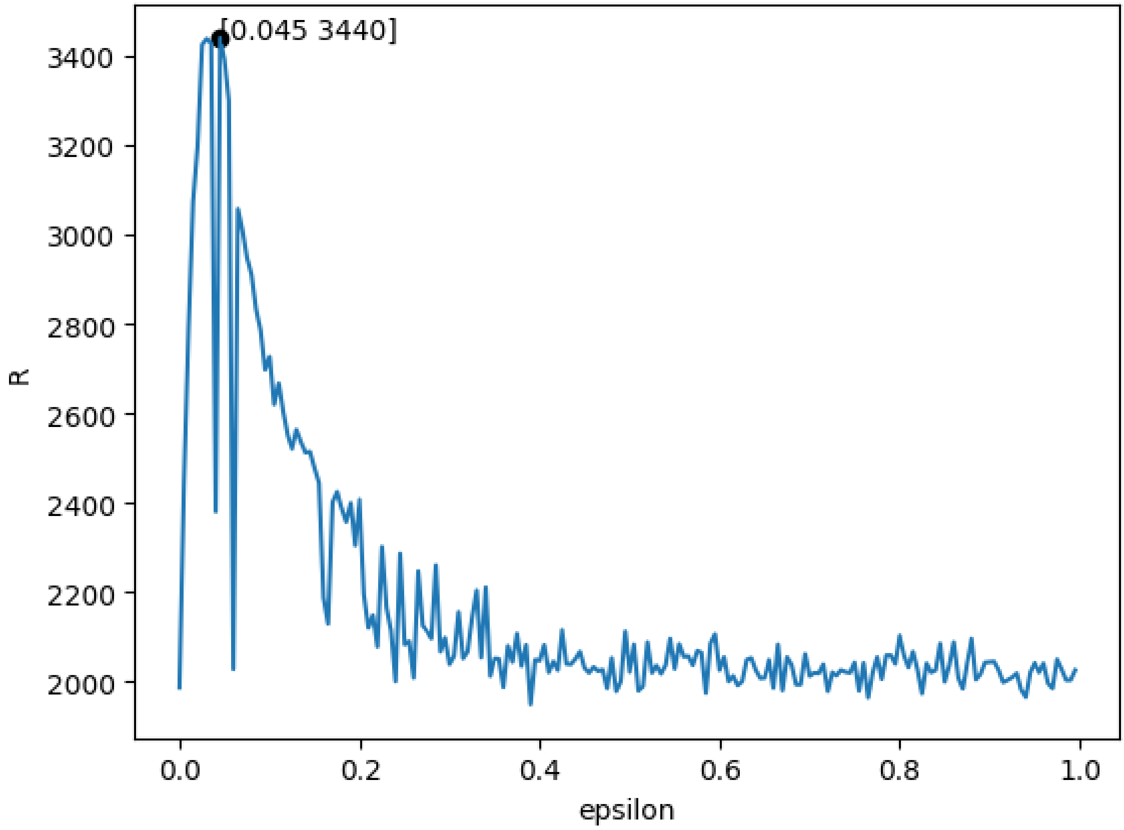
6: $\qquad$ **end for**

7:**end for**

(See complete code in the code file sent through email)

In [ ]:
```
### Your code for problem 1.6. Feel free to insert more blocks or helper functic
x5=[i/200 for i in range(200)]
y5=[0 for i in range(200)]
for j in range (200):
    p1=[1/3,1/3,1/3]
    p2=[0,0,0]
    S=[0,0,0]
    b=0
    l=x5[j]
    for i in range (5000):
        Y=[0,0,0]
        X=0
        if(i!=0):
            #update the p_t$
            p1[0]=p2[0]*math.exp(l*S[0])/p2[0]*(math.exp(l*S[0])+math.exp(l*S[1]
            p1[1]=p2[1]*math.exp(l*S[1])/p2[1]*(math.exp(l*S[0])+math.exp(l*S[1]
            p1[2]=p2[2]*math.exp(l*S[2])/p2[2]*(math.exp(l*S[0])+math.exp(l*S[1]
        a=p1.index(max(p1))
        Y[a]=1
        p_2=np.random.uniform(0,1)
        if(p_2<or_p[a]):
            X=1
            b+=1
            #update the S$
        S[0]=S[0]+1-Y[0]*(1-X)/p1[0]
        S[1]=S[1]+1-Y[1]*(1-X)/p1[1]
        S[2]=S[2]+1-Y[2]*(1-X)/p1[2]
        #update the p_t-1$
        p2[0]=p1[0]
        p2[1]=p1[1]
        p2[2]=p1[2]
    y5[j]=b
y_5_max=y5.index(max(y5))
show_max='['+str(x5[y_5_max])+' '+str(y5[y_5_max])+']'
plt.plot(x5[y_5_max],y5[y_5_max],'ko')
```

```
plt.plot(x5,y5)
plt.annotate(show_max,xy=(x5[y_5_max],y5[y_5_max]),xytext=(x5[y_5_max],y5[y_5_ma
plt.xlabel("epsilon")
plt.ylabel("R")
plt.show()
```



From the plot above, we know the ideal η should be 0.045

References:

[1]: Tor Lattimore, University of Alberta, Csaba Szepesvári, University of Alberta. Bandit Algorithms. pp. 127 - 141. The Exp3 Algorithm

Part II: Bayesian Bandit Algorithms

There are two arms which may be pulled repeatedly in any order. Each pull may result in either a success or a failure. The sequence of successes and failures which results from pulling arm i ($i \in \{1,2\}$) forms a Bernoulli process with unknown success probability θ_i. A success at the t^{th} pull yields a reward γ^{t-1} $(0 < \gamma < 1)$, while an unsuccessful pull yields a zero reward. At time zero, each θ_i has a Beta prior distribution with two parameters α_i, β_i and these distributions are independent for different arms. These prior distributions are updated to posterior distributions as arms are pulled. Since the class of Beta distributions is closed under Bernoulli sampling, posterior distributions are all Beta distributions. How should the arm to pull next in each time slot be chosen to maximize the total expected reward from an infinite sequence of pulls?

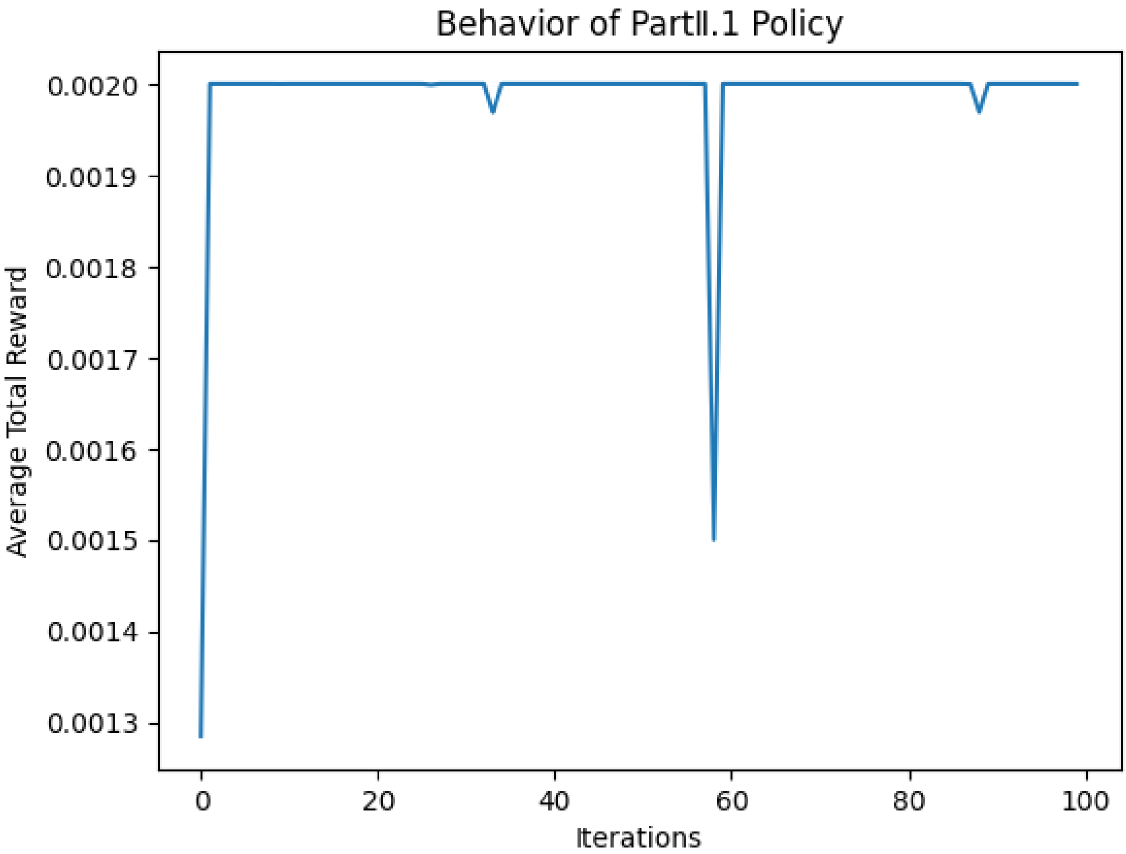
1. One intuitive policy suggests that in each time slot we should pull the arm for which the current expected value of θ_i is the largest. This policy behaves very good in most cases. Please design simulations to check the behavior of this policy.

Your anwser of problem 1 in Part II

In [ ]:
```
### Your code for problem 2.1. Feel free to insert more blocks or helper functio
import numpy as np
import matplotlib.pyplot as plt
num_arms = 2
alpha = [1, 1]  # Prior parameters for arm 1 and arm 2
beta = [1, 1]   # Prior parameters for arm 1 and arm 2
gamma = 0.5     # Reward parameter
num_iterations = 100   # Number of iterations
num_time_slots = 1000  # Number of time slots in each iteration

total_rewards = np.zeros(num_iterations)
for iteration in range(num_iterations):
    # Initialize prior distributions
    theta = np.random.beta(alpha, beta)
    # Perform pulls in each time slot
    for t in range(num_time_slots):
        # Calculate expected values
        expected_values = theta
        # Select arm with largest expected value
        selected_arm = np.argmax(expected_values)
        # Simulate pull and observe outcome
        outcome = np.random.binomial(1, theta[selected_arm])
        # Update parameters of selected arm
        if outcome == 1:
            alpha[selected_arm] += 1
        else:
            beta[selected_arm] += 1
        # Update total reward
        total_rewards[iteration] += gamma**(t) * outcome
# Calculate average total reward over iterations
average_rewards = total_rewards / num_time_slots

plt.plot(range(num_iterations), average_rewards)
plt.xlabel('Iterations')
plt.ylabel('Average Total Reward')
plt.title('Behavior of PartⅡ.1 Policy')
plt.show()
```




```
We implement the code by following the steps below:

1.Define some necessary parameters

2.Initialize the prior distributions for each arm using Beta
distribution with parameters αi and βi.

3.For each time slot:
    Calculate the expected value of each arm using the current
Beta distributions.
    Select the arm with the largest expected value.
    Pull the selected arm and observe the outcome (success or
failure).
    Update the parameters of the corresponding Beta distribution
based on the observed outcome.
    Repeat the above steps for subsequent time slots.
4.Analyze the results:
    Calculate the average total reward obtained over all
iterations.
    Plot the average total reward over iterations to observe the
performance of the policy.

Conclusion: It's easy to find that the policy behaves very good
in most cases since "Average Total Reward" always keeps in a
high level between "Iterations" [0,100].
```

2. However, such intuitive policy is unfortunately not optimal. Please provide an example to show why such policy is not optimal.

Your anwser of problem 2 in Part II

（See complete code in the code file sent through email）

In [ ]:
```
import numpy as np
theta = [0.2, 0.8]  # True success probabilities
gamma = 0.5
alpha = [1, 1]  # Prior parameters for arm 1 and arm 2
beta = [1, 1]   # Prior parameters for arm 1 and arm 2
num_time_slots = 10

for t in range(num_time_slots):
    # Calculate expected values
    expected_values = np.array([alpha[i] / (alpha[i] + beta[i]) for i in range(2
    # Select arm with largest expected value
    selected_arm = np.argmax(expected_values)
    # Simulate pull and observe outcome
    outcome = np.random.binomial(1, theta[selected_arm])
    # Update parameters of selected arm
    if outcome == 1:
        alpha[selected_arm] += 1
    else:
        beta[selected_arm] += 1
    print(f"Time Slot {t+1}: Selected Arm {selected_arm+1}, Outcome {outcome}, A
```

```
Time Slot 1: Selected Arm 1, Outcome 0, Alpha [1, 1], Beta [2, 1]
Time Slot 2: Selected Arm 2, Outcome 1, Alpha [1, 2], Beta [2, 1]
Time Slot 3: Selected Arm 2, Outcome 0, Alpha [1, 2], Beta [2, 2]
Time Slot 4: Selected Arm 2, Outcome 1, Alpha [1, 3], Beta [2, 2]
Time Slot 5: Selected Arm 2, Outcome 1, Alpha [1, 4], Beta [2, 2]
Time Slot 6: Selected Arm 2, Outcome 1, Alpha [1, 5], Beta [2, 2]
Time Slot 7: Selected Arm 2, Outcome 0, Alpha [1, 5], Beta [2, 3]
Time Slot 8: Selected Arm 2, Outcome 0, Alpha [1, 5], Beta [2, 4]
Time Slot 9: Selected Arm 2, Outcome 1, Alpha [1, 6], Beta [2, 4]
Time Slot 10: Selected Arm 2, Outcome 0, Alpha [1, 6], Beta [2, 5]
```

```
In our example, the intuitive policy consistently selects arm 2
（except the 1st time） with higher initial success probability
in each time slot. As a result, the algorithm keeps updating the
parameters of arm 2 based on the observed successes, and arm 2's
Beta distribution gradually shifts towards higher success
probabilities.

However, since arm 1 has a lower initial success probability, it
may take more time slots to observe a success from arm 1. If we
explore arm 1 more in the very beginning, we may have a chance
to discover that arm 1 has a higher potential success
probability, which will lead to higher rewards in the long run.

The example shows that the intuitive policy always selecting the
arm with the largest current expected value of θi is not optimal
in situations where the initial probabilities may not accurately
represent the long-term behavior of the arms. As we have done in
```

```
Part Ⅰ, the UCB algorithm or TS is often employed to balance
exploration and exploitation and achieve better long-term
rewards.
```

3. For the expected total reward under an optimal policy, show that the following recurrence equation holds:

$$\begin{aligned} R_1(\alpha_1,\beta_1) = & \frac{\alpha_1}{\alpha_1+\beta_1}[1+\gamma R(\alpha_1+1,\beta_1,\alpha_2,\beta_2)] \\ & + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R(\alpha_1,\beta_1+1,\alpha_2,\beta_2)]; \\ R_2(\alpha_2,\beta_2) = & \frac{\alpha_2}{\alpha_2+\beta_2}[1+\gamma R(\alpha_1,\beta_1,\alpha_2+1,\beta_2)] \\ & + \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R(\alpha_1,\beta_1,\alpha_2,\beta_2+1)]; \\ R(\alpha_1,\beta_1,\alpha_2,\beta_2) = & \max\{R_1(\alpha_1,\beta_1), R_2(\alpha_2,\beta_2)\}. \end{aligned}$$

Your anwser of problem 3 in Part II

First we need to figure out the meaning of 3 notations:

- R1(α1, β1) represents the expected total reward when selecting arm 1 with prior parameters α1 and β1.
- R2(α2, β2) represents the expected total reward when selecting arm 2 with prior parameters α2 and β2.
- R(α1, β1, α2, β2) represents the overall expected total reward under an optimal policy, where α1, β1, α2, β2 are the prior parameters for arm 1 and arm 2.

Consider the expected total reward when selecting arm 1 or arm 2 in the current time slot:

1. When selecting arm 1:
   - The expected reward for selecting arm 1 is γ multiplied by the reward obtained from the current pull, which is either 1 (success) or 0 (failure).
   - If arm 1 is successful (outcome = 1), the prior parameters for arm 1 are updated to α1 + 1 and β1.
   - If arm 1 fails (outcome = 0), the prior parameters for arm 1 are updated to α1 and β1 + 1.
   - The expected total reward for selecting arm 1 can be calculated recursively using the updated prior parameters.
2. When selecting arm 2:
   - The expected reward for selecting arm 2 is γ multiplied by the reward obtained from the current pull, which is either 1 (success) or 0 (failure).
   - If arm 2 is successful (outcome = 1), the prior parameters for arm 2 are updated to α2 + 1 and β2.
   - If arm 2 fails (outcome = 0), the prior parameters for arm 2 are updated to α2 and β2 + 1.

- The expected total reward for selecting arm 2 can be calculated recursively using the updated prior parameters.

Based on these considerations, the recurrence equation for the expected total reward can be expressed as follows:

$$\begin{aligned} R_1(\alpha_1,\beta_1) = & \frac{\alpha_1}{\alpha_1+\beta_1}[1+\gamma R(\alpha_1+1,\beta_1,\alpha_2,\beta_2)] \\ & +\frac{\beta_1}{\alpha_1+\beta_1}[\gamma R(\alpha_1,\beta_1+1,\alpha_2,\beta_2)]; \\ R_2(\alpha_2,\beta_2) = & \frac{\alpha_2}{\alpha_2+\beta_2}[1+\gamma R(\alpha_1,\beta_1,\alpha_2+1,\beta_2)] \\ & +\frac{\beta_2}{\alpha_2+\beta_2}[\gamma R(\alpha_1,\beta_1,\alpha_2,\beta_2+1)]; \\ R(\alpha_1,\beta_1,\alpha_2,\beta_2) = & \max\left\{R_1(\alpha_1,\beta_1),R_2(\alpha_2,\beta_2)\right\}. \end{aligned}$$

In these equations, we recursively calculate the expected total reward for selecting arm 1 and arm 2, considering the updated prior parameters after each pull. The overall expected total reward is the maximum between R1(α1, β1) and R2(α2, β2), representing the optimal policy of selecting the arm with the highest expected value at each time slot. The recurrence equation provides a framework for evaluating the expected total reward under an optimal policy in the Bayesian bandit problem. By solving the recurrence equation, we can compute the expected total reward for specific values of α1, β1, α2, and β2.

4. For the above equations, how to solve it exactly or approximately?

Your anwser of problem 4 in Part II

```
We think there are at least two approaches:

1. Dynamic Programming: By starting from the base cases and
iteratively calculating the values for different states, we can
build up the solutions until we reach the desired state. This
approach can be computationally expensive as the number of
states increases, but it guarantees an exact solution.

2. Monte Carlo Method: Using Monte Carlo simulation or Monte
Carlo tree search, we can sample from the state space and
estimate the expected total reward based on the sampled
outcomes. By repeating this process multiple times and averaging
the results, we can approximate the solutions to the recurrence
equations. The method can be used when the state space is large
or when an exact solution is not feasible.

The choice of method depends on the complexity of the problem,
the size of the state space, and the available computational
resources. In some cases, it may be possible to derive closed-
form solutions for specific scenarios or simplify the recurrence
equations using certain assumptions. However, exact solutions
```

```
may not be easily achievable. Thus, approximation methods are
commonly employed to estimate the expected total reward.
```

5. Find the optimal policy.

Your anwser of problem 5 in Part II

The main idea is that we apply Depth Search to the bandit while simultaneously record the selection of recursion every time to get the total expected reward based on $\alpha_1, \alpha_2, \beta_1, \beta_2$ i.e. $R(\alpha_1, \beta_1, \alpha_2, \beta_2)$. Then, from the recursive selection we have recorded, we can construct a tree structure to represent our entire recursive process so we let $R(\alpha_1, \beta_1, \alpha_2, \beta_2)$ with initial $\alpha_1, \alpha_2, \beta_1, \beta_2$ to be the root node and the last $R(\alpha_1, \beta_1, \alpha_2, \beta_2)$ which we gained in the N th slot to be the leaf node . Therefore, our selection for each slot only needs to follow the path from the root node to the leaf node and we will get the highest total reward.

The code below shows the difference between the total reward we gained from the algorithm and the ideal total reward.

（See complete code in the code file sent through email）

In [ ]:

```
### Your code for problem 2.5. Feel free to insert more blocks or helper functio
cmp=[0,0]
ct=[0,0,0,0]
r=0
r1=0
t=100
def iter(t,a1,b1,a2,b2):
    if(t!=1):
        t-=1
        R=max(a1/(a1+b1)*(1+gamma*iter(t,a1+1,b1,a2,b2))+b1/(a1+b1)*(gamma*iter(
    else:
        R=max(a1/(a1+b1),a2/(a2+b2))
    return R
for i in range (5000):
    p_2=np.random.uniform(0,1)
    if(p_2<theta[1]):
        r+=gamma**i
print("from dp",iter(15,2000,8000,8000,2000))
print("ideal",r)
```

```
from dp 1.5999511718750001
ideal 1.871081077404026
```

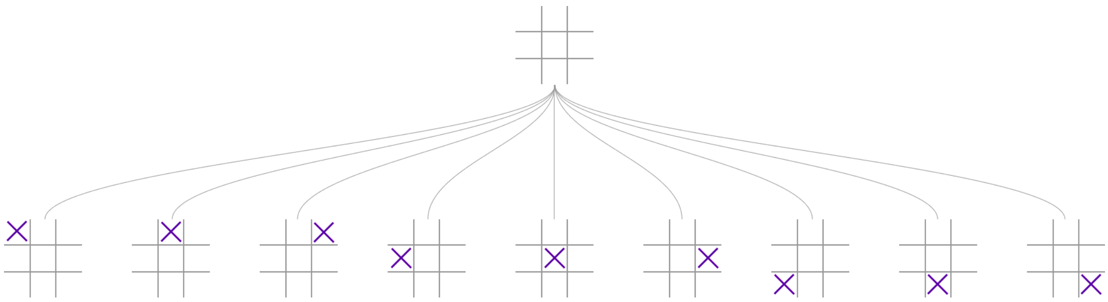
The error may be significant because when the total number of time solts is 15, the recursive process requires a lot of time to get the result

Project: Monte Carlo Tree Search Mini Problem Set

- In this project, you will learn and implement the Monte Carlo Tree Search (MCTS) algoritm on the Tic Tac Toe game.

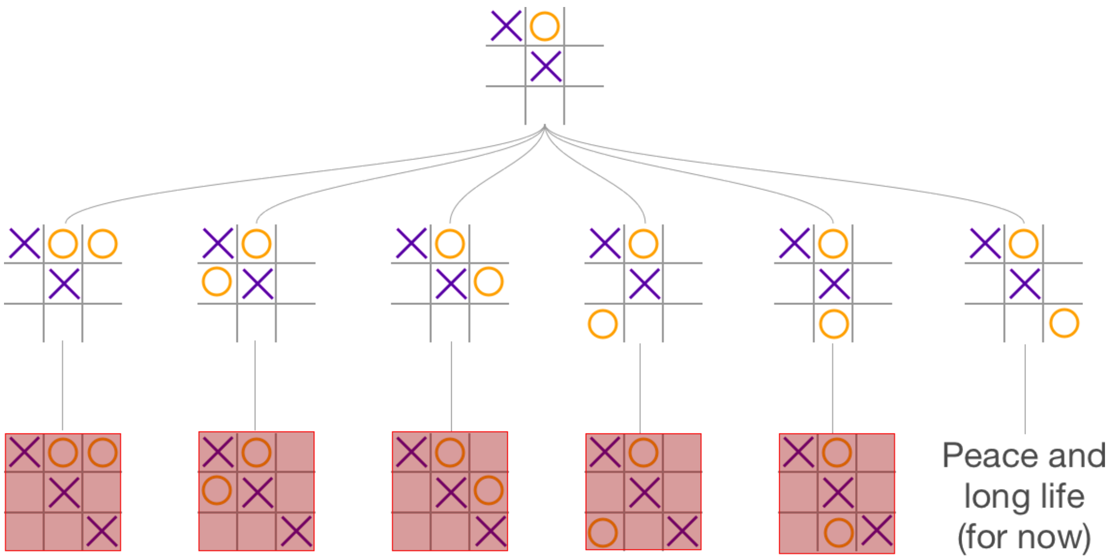
What is a tree search?

Trees are a special case of the graph problems previously seen in class. For example, consider Tic Tac Toe. From the starting blank board, each choice of where to draw an X is a possible future, followed by each choice of where to draw an O. A planner can look at this sprawl of futures and choose which action will most likely lead to victory.



Breadth First Tree Search

In a breadth first tree search the planner considers potential boards in order of depth. All boards that are one turn in the future are considered first, followed by the boards two turns away, until every potential board has been considered. The planner then chooses the best move to make based on whether the move will lead to victory or to defeat. In the following example, a breadth first search would identify that all other moves lead to a loss and instead pick the rightmost move.



Monte Carlo Tree Search

The problem with breadth first search is that it isn't at all clever. Tic Tac Toe is one of the simpler games in existence, but there are nearly three-hundred sixty thousand possible sets of moves for a BFS-based planner to consider. In a game with less constrained movement, like chess, this number exceeds the number of atoms in the known universe after looking only a couple of turns into the future. A Breadth First Search is too tied up with being logical and provably correct. Monte Carlo Tree Search leaps ahead to impulsively go where no search has gone before. In simpler terms, BFS is Spock while MCTS is Kirk.

MCTS performs its search by repeatedly imagining play-throughs of the game or scenario, traveling down the entire branch of the game tree until it terminates. Based on how this play-through went, MCTS then updates the value of each node (move) involved based on whether it won or lost the playthrough. The Monte Carlo component comes from the fact that it chooses moves at random, not based on heuristics or visit count. This nondeterminism greatly increases the potential space it can explore even though its exploration will be much less rigorous.

For example, let's look at the BFS tree above. The bad red moves have a high probability of loss because in 1/5 of the playthroughs X will instantly win. The correct, rightmost move will have a lower probability of a loss because X doesn't have this 1/5 chance of winning. MCTS will discard the red moves because of this higher loss probability, assigning them lower values whenever it selects them during a randomized play-through.

MCTS Algorithm

Check out the paper.

Problem Set API

Before you write your very own MCTS bot, you need to get sped up on the API you will be using. But even before getting into the API, please run the following to import the API and some boilerplate code:

In [ ]:
```
import algo
import sim
import random
import time
from tests import *
from game import *
```

The `Board` Class

The `Board` class is a template class that represents the state of a board at a single point in a game. We've created a `ConnectFourBoard` subclass that handles the mechanics for you. For any `Board` instance `board`, you have access to the following methods:

`board.get_legal_actions()` : Returns a python set of `Action` class instances. Each element in the set is a valid action that can be applied to the `board` to create a new `Board` instance. See the `Action` API section below.

`board.is_terminal()` : Returns `True` if `board` is an endgame board. Returns `False` otherwise.

`board.current_player_id()` : Returns an integer that represents which player is expected to play next. For example, if this method returns 0, then the player who is the first player in some simulation of a game should be the next one to play an action. This is used internally in the `Simulation` class for bookkeeping, but you will need it when you do the backup step of the MCTS algorithm.

`board.reward_vector()` : Returns a n-element tuple, where n is the number of players, that contains the rewards earned by each player at this particular `board`. For Connect Four, $n = 2$. Thus this method may return something like `(1,-1)`, meaning the player with ID 0 had a reward of 1, and the player with ID 1 has reward -1.

The `Action` Class

The `Action` class is for representing a single action that is meant to alter a `board`. We have written a `ConnectFourAction` subclass for you. Instances are hashable, so to check if two actions are the same, `hash(action1) == hash(action2)` can be used. For any `Action` instance `action`, you will only need the following method:

`action.apply(board)` : Given a `Board` instance `board`, returns a new `Board` instance that represents the board after that action has been performed. If the `action` cannot be applied, an error is thrown.

The `Node` Class

The `Node` class represents a single node in the MCTS tree that is constructed during each iteration of the algorithm. You will be interacting with this class the most. If you remember the algorithm, each node contains certain pieces of information that's associated with it. For any `Node` instance `node`, you have the following methods at you disposal:

`Node(board, action, parent)` : The constructor takes three arguments. First, a `Board` instance `board` that the node will represent. Second, an `Action` instance 'action' that represents the incoming action that created `board`. Finally, a `Node` instance `parent`. For a root node, you would pass `None` in for both `action` and `parent`.

`node.get_board()` : Returns the `Board` instance that `node` is representing.

`node.get_action()` : Returns the incoming `Action` instance.

`node.get_parent()` **:** Returns the parent `Node` instance.

`node.get_children()` **:** Returns a list of `Node` instances that represent the children that have been expanded thus far.

`node.add_child(child)` **:** Add a `Node` instance `child` to the list of expanded children under `node`.

`node.get_num_visits()` **:** Returns the number of times `node` has been visited.

`node.get_player_id()` **:** This just returns `board.current_player_id()`, where `board` is the board that was passed into the constructor.

`node.q_value()` **:** Returns the total reward that the `node` has accumulated. This reward is contained in a variable `node.q` that you can access if it needs to be changed during the algorithm.

`node.visit()` **:** Doesn't return anything, but increments the internal counter that keeps track of how many times the `node` has been visited.

`node.is_fully_expanded()` **:** Return `True` is all children that can be reached from this node have been expanded. Returns `False` otherwise.

`node.value(c)` : Returns the calculated UCT value for this node. The parameter `c` is the *exploration* constant.

The `Player` Class

The `Player` class represents, you guessed it, a player. You won't have to actually deal with this class at all in this problem set. It exists for running the simulation at the end. However, if you interested, you may look at `game.py` to see what methods are used.

The `Simulation` Class

The `Simulation` class is used for setting up a simulation for multiple players to play a game. You again don't need to worry about this class, as it is for running the simulation at the end. Refer to `game.py` if your curious about how it works.

Problem Set Code

In the following parts, we will ask you to implement the Monte Carlo Tree Search algorithm to beat the bot. You will be implementing the pseudocode starting on page 10 of the MCTS paper

Default Policy

The first step is to implement the default policy, which plays through an entire game session. It chooses actions at random, applying them to the board until the game is over.

It then returns the reward vector of the finished board.

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

Browne, et al.

Note: You can use `random.choice(my_list)` to select a random item from `my_list`

In [ ]:
```
#####################################################################
# randomly picking moves to reach the end game
# Input: BOARD, the board that we want to start randomly picking moves
# Output: the reward vector when the game terminates
#####################################################################
def default_policy(board):
    while not board.is_terminal():
        legal_actions = board.get_legal_actions()
        action = random.choice(list(legal_actions))
        board = action.apply(board)
    return board.reward_vector()
```

In [ ]:
```
test_default_policy(default_policy)
```

```
test passed
test passed
test passed
```

Tests passed!!

Tree Policy

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

Tree Policy Pseudocode (Browne, et al.)

The tree policy performs a depth-first search of the tree using `best_child` and `expand`. If it encounters an unexpanded node it will return the expanded child of that node. Otherwise, it continues its search in the best child of the current node.

The `best_child` function find the best child node if a node is fully expanded. It also takes the exploitation constant as an argument.

The `expand` function expands a node that has unexpanded children. It must get all current children of the node and all possible children of the node then add one of the possible children to the node. It should then return this newly added child.

Best Child

$$\textbf{function } \text{BESTCHILD}(v, c)$$
$$\textbf{return } \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

Browne, et al.

Note: For convenience, we've implemented a function that returns the heuristic inside the max operator. Look at the function `node.value(c)` for the `NODE` class API and save yourself the headache.

In [ ]:
```
############################################################
# get the best child from this node (using heuristic)
# Input: NODE, the node we want to find the best child of
#        C,    the exploitation constant
# Output: the child node
############################################################
def best_child(node, c):
    children = node.get_children()
    best_score = float("-inf")
    best_child = None
    for child in children:
        exploit_term = child.q_value() / child.get_num_visits()
        explore_term = c * (math.sqrt(2 * math.log(node.get_num_visits()) / chil
        score = exploit_term + explore_term
        if score > best_score:
            best_score = score
            best_child = child
    return best_child
```

In [ ]:
```
test_best_child(best_child)
```

```
test passed
```

Tests passed!!

Expand

$$\textbf{function } \text{EXPAND}(v)$$
$$\text{choose } a \in \text{untried actions from } A(s(v))$$
$$\text{add a new child } v' \text{ to } v$$
$$\text{with } s(v') = f(s(v), a)$$
$$\text{and } a(v') = a$$
$$\textbf{return } v'$$

Browne, et al.

In [ ]:
```
###########################################################
# expand a node since it is not fully expanded
# Input: NODE, a node that want to be expanded
# Output: the child node
###########################################################
def expand(node):
    board = node.get_board()
    legal_actions = board.get_legal_actions()
    expanded_children = node.get_children()
    unexpanded_actions = legal_actions - set(child.get_action() for child in exp
    chosen_action = random.choice(list(unexpanded_actions))
    new_board = chosen_action.apply(board)
    new_child_node = Node(new_board, chosen_action, node)
    node.add_child(new_child_node)
    return new_child_node
```

In [ ]:
```
test_expand(expand)
```

Tests passed!!

```
test passed
```

Tree Policy

function $\text{TREEPOLICY}(v)$
  while v is nonterminal **do**
    if v not fully expanded **then**
      return $\text{EXPAND}(v)$
    else
      $v \leftarrow \text{BESTCHILD}(v, Cp)$
  return v

Browne, et al.

In [ ]:
```
##########################################################
# heuristically search to the leaf level
# Input: NODE, a node that want to search down
#         C, the exploitation value
# Output: the leaf node that we expand till
##########################################################
def tree_policy(node, c):
    while not node.get_board().is_terminal():
        if not node.is_fully_expanded():
            return expand(node)
        else:
            node = best_child(node, c)
    return node
```

In [ ]:
```
test_tree_policy(tree_policy, expand, best_child)
```

```
test passed
```

Tests passed!!

Backup

Now its time to make a way to turn the reward from `default_policy` into the information that `tree_policy` needs. `backup` should take the terminal state and reward from `default_policy` and proceed up the tree, updating the nodes on its path based on the reward.

function $\textsc{Backup}(v, \Delta)$
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Browne, et al.

In [ ]:
```
##############################################################
# reward update for the tree after one simulation
# Input: NODE, the node that we want to backup from
#        REWARD_VECTOR, the reward vector of this exploration
# Output: nothing
##############################################################
def backup(node, reward_vector):
    while node is not None:
        player_id = node.get_player_id()
        node.q += reward_vector[player_id - 1]
        node.visit()
        node = node.get_parent()
```

In [ ]:
```
test_backup(backup)
```

```
test passed
```

Tests passed!!

Search (UCT)

Time to put everything together! Keep running `tree_policy`, `default_policy`, and `backup` until you run out of time! Finally, return the best child's associated action.

function $\textsc{UctSearch}(s_0)$
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow \textsc{TreePolicy}(v_0)$
    $\Delta \leftarrow \textsc{DefaultPolicy}(s(v_l))$
    $\textsc{Backup}(v_l, \Delta)$
  return $a(\textsc{BestChild}(v_0, 0))$

Browne, et al.

In [ ]:

```python
######################################################################
# monte carlo tree search algorithm using UCT heuristic
# Input: BOARD, the current game board
#        TIME_LIMIT, the time limit of the calculation in second
# Output: class Action represents the best action to take
####################################################################
def uct(board, time_limit):
    start_time = time.time()
    root = Node(board, None, None)
    while (time.time() - start_time) < time_limit:
        leaf = tree_policy(root, 1.0 / math.sqrt(2))
        reward_vector = default_policy(leaf.get_board())
        backup(leaf, reward_vector)
    return best_child(root, 0).get_action()
```

In [ ]:

```python
test_uct(uct) # this test takes 15-30 seconds
```

```
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
```

Tests passed!!

The Final Challenge

Time to show Stonn the power of human ingenuity! Win at least 9 out of 10 games to triumph!

In [ ]:

```python
sim.run_final_test(uct)
```

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 0 won

Player 1 won

Player 1 won

`Player 1  won`

`Player 1  won`

You win!!

Stonn sits back in shock, displaying far more emotion than any Vulcan should.

"Cadet, it looks like your thousands of years in the mud while we Vulcans explored the cosmos were not in vain. Congratulations."

The class breaks into applause! Whoops and cheers ring through the air as Captain James T. Kirk walks into the classroom to personally award you with the Kobayashi Maru Award For Excellence In Tactics.

The unwelcome interruption of your blaring alarm clock brings you back to reality, where in the year 2200 Earth's Daylight Savings Time was finally abolished by the United Federation of Planets.