CS747: Assignment 1

Aryan Bhosale
210040024
Indian Institute of Technology, Bombay

I. Task 1: Implementation of Sampling Algorithms

Problem Statement

In this first task, you will implement the sampling algorithms: (1) UCB, (2) KL-UCB, and (3) Thompson Sampling. This task is straightforward based on the class lectures. The instructions below tell you about the code you are expected to write.

In this section, the sampling algorithms Upper Confidence Bound (UCB), Kullback-Leibler Upper Confidence Bound (KL-UCB) and Thompson Sampling have been implemented in code and analysis of regret over different horizons is carried out.

A. Upper Confidence Bound (UCB) Algorithm

Formula for UCB is: $$\text{ucb}_a^t = \hat{p_a^t} + \sqrt{\frac{2\ln(t)}{u_a^t}}$$

where $\hat{p_i}$ is the empirical mean of rewards from arm a and u_a^t is the number of times a has been sampled at time t.
The algorithm balances exploration and exploitation to maximize the total reward. The UCB algorithm works by selecting the arm with the highest upper confidence bound. In this implementation, it is ensured that each arm is pulled at least once at random in the beginning after which we can easily update the upper confidence bound of each arm based on the observed rewards. The algorithm then selects the arm with the highest upper confidence bound for exploitation.

1) Plot: The exploration bonus ensures that all arms are pulled at some time eventually. Also, the more some arm is pulled, the lesser the exploration bonus for the same. This helps in achieving sub-linear regret. It follows from the plot that regret is O(log(T)), as expected from the derivation and analysis of UCB, which proved the regret bound by expanding the summations.

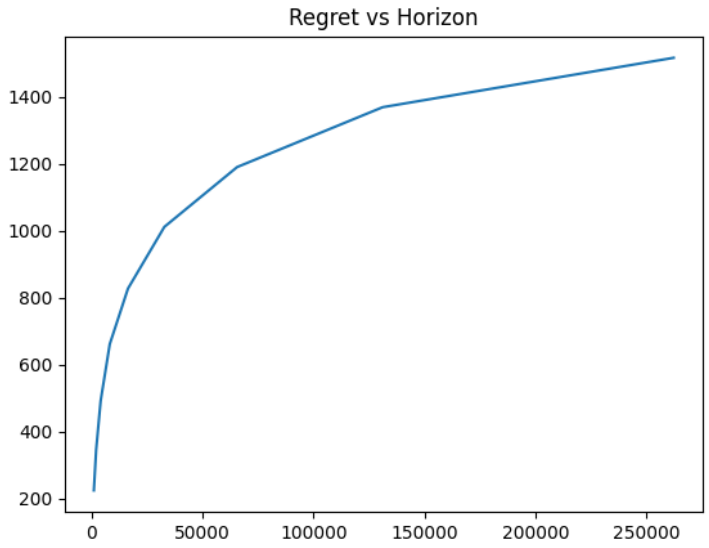

Fig. 1. UCB: Regret v. Horizon

2) Code Description: Please refer to coloured annotations for detailed description.

Variables

Listing 1. Variables and their meanings

```
self.num_arms = num_arms # no of arms in the bandits
self.curr_timestep = 0 # the current time 't' as defined in the formula
self.counts = np.zeros(num_arms) # array to count the no. of times an arm has been pulled
self.initial_pulls = np.arange(num_arms) # a shuffled array to ensure we pull all arms atleast
    once in the beginning
np.random.shuffle(self.initial_pulls)
self.values = np.full((num_arms,), 1e-5) # the emprical mean of rewards
self.ucb_arms = np.zeros(num_arms) # array to store upper confidence bound for arms at every
    iteration
```

Pull Function

Listing 2. get pull function

```
## to ensure all arms are pulled
if self.curr_timestep < self.num_arms:
            return self.initial_pulls[self.curr_timestep]
## choosing the arm with the highest upper confidence bound
for i in range(self.num_arms):
    #formula for upper confidence of an arm at time t
    self.ucb_arms[i] = self.values[i] + math.sqrt((2*math.log(self.curr_timestep))/self.counts[
        i])

return np.argmax(self.ucb_arms)
```

Reward Updation

Listing 3. get reward function

```
n = self.counts[arm_index]
if n==0:
    self.values[arm_index] = (self.values[arm_index]+reward)/2
else:
    # taking a running average to store empirical mean of rewards
    self.values[arm_index] =  ((n- 1) / n) * self.values[arm_index] + (1 / n) * reward

self.curr_timestep += 1
self.counts[arm_index] += 1
```

B. Kullback-Leibler Upper Confidence Bound (KL-UCB) Algorithm

Formula for KL-UCB is:

$$KL - UCB_a^t = \max\left\{q \in [\hat{p_a^t}, 1] : u_a^t \mathrm{KL}\left(P_i, Q_i(t)\right) \leq ln(t) + cln(ln(t))\right\} where, c \geq 3$$

where $\hat{p_i}$ is the empirical mean of rewards from arm a and u_a^t is the number of times a has been sampled at time t.

It is an extension of the UCB algorithm that uses the Kullback-Leibler (KL) divergence. The algorithm selects the arm with the highest upper confidence bound, which in this case is calculated using the KL divergence and the number of times the arm has been pulled. The KL-UCB algorithm has been shown to have a logarithmic regret bound, i.e. that the expected loss of the algorithm compared to the best arm decreases logarithmically with the number of rounds. This algorithm tends to be computationally heavy because of the load of calculating the KL-Divergence.

1) Plot: It follows from the plot that regret is O(log(T)). We know that KL-UCB provides tighter bounds which is normally due to the sub-sub-linear term. The value for regret is significantly lower when compared to UCB. It must be noted that c=0 has been used since that has been proved to be better by the authors of *The KL-UCB Algorithm for Bounded Stochastic Bandits and Beyond*.

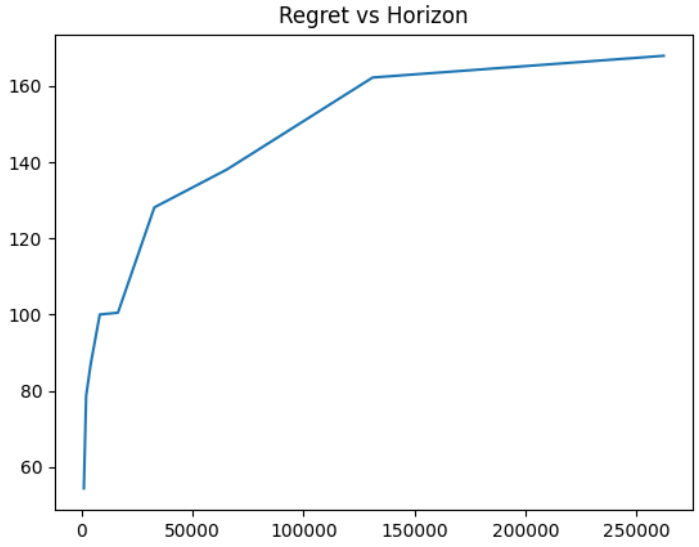

Fig. 2. KL-UCB: Regret v. Horizon

2) Code Description: Please refer to coloured annotations for detailed description.

Variables

Listing 4. Variables and their meanings (same as UCB)

```
self.num_arms = num_arms # no of arms in the bandits
self.curr_timestep = 0 # the current time 't' as defined in the formula
self.counts = np.zeros(num_arms) # array to count the no. of times an arm has been pulled
self.initial_pulls = np.arange(num_arms) # a shuffled array to ensure we pull all arms atleast
    once in the beginning
np.random.shuffle(self.initial_pulls)
self.values = np.full((num_arms,), 1e-5) # the emprical mean of rewards
self.ucb_arms = np.zeros(num_arms) # array to store upper confidence bound for arms at every
    iteration
```

Pull Function

Listing 5. get pull function

```
## to ensure all arms are pulled
if self.curr_timestep < self.num_arms:
            return self.initial_pulls[self.curr_timestep]
## choosing the arm with the highest upper confidence bound calculated using KL divergence
## Binary search has been used below to calculate KL-UCB
for i in range(self.num_arms):
    p_a_t = self.values[i]
    lower = p_a_t
    upper = 1
    tolerance = 0.01

    while  (upper - lower)>tolerance:
        mid = (lower+upper)*0.5
        kl_term = kl_div(p_a_t, mid)
        if kl_term < math.log(self.curr_timestep)/self.counts[i] :
            lower = mid
        else:
            upper = mid
    self.kl_ucb_arms[i] = (lower+upper)*0.5

return np.argmax(self.kl_ucb_arms)
```

Reward Updation

Listing 6. get reward function (same as UCB)

```
n = self.counts[arm_index]
if n==0:
    self.values[arm_index] = (self.values[arm_index]+reward)/2
else:
    # taking a running average to store empirical mean of rewards
    self.values[arm_index] =  ((n- 1) / n) * self.values[arm_index] + (1 / n) * reward

self.curr_timestep += 1
self.counts[arm_index] += 1
```

KL Divergence Function

Listing 7. kldiv

```
##function to calculate the KL divergence between a given x and y
def kl_div(x, y):
    if x == 0:
        return math.log(1/(1-y))
    elif x==1:
        return math.log(1/y)
    term1 = x*(math.log(x/y))
    term2 = (1-x)*math.log((1-x)/(1-y))
    return term1 + term2
```

C. Thompson Sampling Algorithm

According to this algorithm, for every arm we draw a sample from the corresponding beta distribution x_a^t which is defined as follows.
If at time t, an arm has succeeded s_a^t times and failed f_a^t times the beta distribution is defined as: $\beta(s_a^t + 1, f_a^t + 1)$ where α is $s_a^t + 1$ and β is $f_a^t + 1$ which implies x_a^t for every arm is: $x_a^t \sim \text{Beta}\,(s_a^t + 1, f_a^t + 1)$
We pull the arm that has the highest x_a^t.

1) Plot: We have achieved sub-linear regret. Additionally this alogrithm is very easy to implement. It is known to work well in practice. The random sampling using Beta distribution accounts for the empirical mean and associated uncertainty with it. We add one so that the coefficients are always positive irrespective of the value of successes and failures. Thompson sampling has clearly minimised regret better than both the UCB algorithms as regret is curtailed to below 120. We can also see that regret is O(log(T)).

Fig. 3. Thompson: Regret v. Horizon

2) Code Description: Please refer to coloured annotations for detailed description.

Variables

Listing 8. Variables and their meanings

```
self.num_arms = num_arms # no of arms in the bandits
self.successes = np.zeros(num_arms) #no. of times a pull for an arm was successful
self.fails = np.zeros(num_arms) #no. of times a pull for an arm was a failure
self.sampler_vals = np.zeros(num_arms) # array to store the sampled value for arms at every
    iteration
```

Pull Function

Listing 9. get pull function

```
## choosing the arm with the highest sampled value from its beta distribution
for i in range(self.num_arms):
    alpha = self.successes[i] + 1
    beta = self.fails[i] + 1
    self.sampler_vals[i] = np.random.beta(alpha, beta)
return np.argmax(self.sampler_vals)
```

Reward Updation

Listing 10. get reward function

```
#updating the no. of failures and successes
if(reward > 0):
    self.successes[arm_index] += 1
else:
    self.fails[arm_index] += 1
```

II. Task 2 A

Problem Statement

This task explores the effect of the difference between the means of arms on the regret accumulated by the UCB algorithm. For this task, take two-armed bandit instances (with means [p1, p2]) with the higher mean arm fixed (say, p1), and vary the other arm's mean (p2) from 0 to p1. For the assignment, p1 = 0.9, and p2 varies from 0 to 0.9 (both inclusive) in steps of 0.05. Do this for a horizon of 30000. Plot

the variation of regret with p2. Clearly state your observations from the plot, and explain the reason(s) behind the observations.

1) Plot: When p2 is closer to p1 (i.e.around 0.9), the regret is higher since it's more challenging to distinguish the arms. When p2 is significantly lower than p1, the regret is low because the UCB algorithm identifies the better arm more quickly and exploits it. The plot illustrates the balance between exploration (choosing arms with uncertain outcomes) and exploitation (choosing the arm with the highest estimated mean) in the UCB algorithm. The algorithm initially explores more when p2 is close to p1 but shifts towards exploitation as the difference between p1 and p2 becomes more significant.

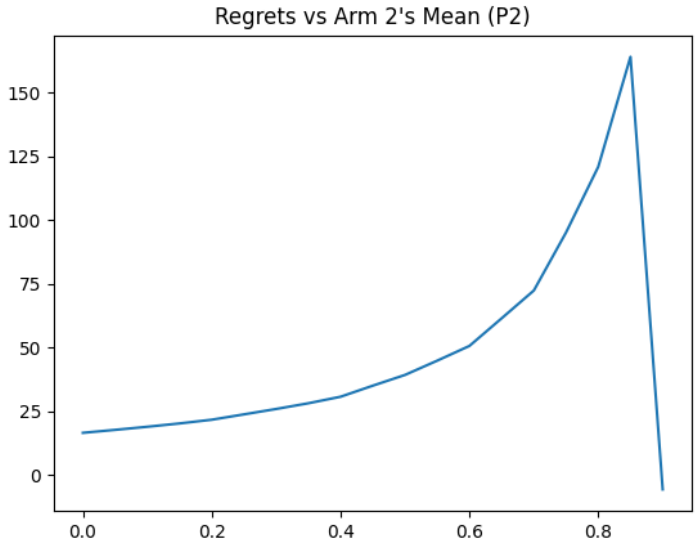

Fig. 4. Task 2A: Regret v. Mean of Arm 2 (p2)

Listing 11. get pull function

```
#creating the array of arm means as described
task2Ap2s = np.arange(0, 0.901, 0.05)
task2Ap1s = np.full((task2Ap2s.shape), 0.9)

#calling the function to execute UCB algorithm as written in the code in Task 1
regrets_ucb_2A = task2(UCB, 30000, task2Ap1s, task2Ap2s)
```

III. Task 2 B

Problem Statement

This task involves comparing the behaviour of UCB and KL-UCB algorithms on two-armed bandits. The task is to compare the effect of the value of the means on the algorithms while keeping the difference between the means fixed. For this task, take Delta = p1 - p2 = 0.1. Again, vary p2 from 0 to 0.9 (both inclusive) in steps of 0.05 for a horizon of 30000. Plot the variation of regret for both algorithms on different plots with respect to the instances (you may plot with respect to either p1 or p2, but clearly label that in your plots). State and compare your observations for the variations of regret for each algorithm and explain the reason behind your observations.

Part 1: UCB

1) Plot: The plot for UCB algorithm has been shown in the figure.

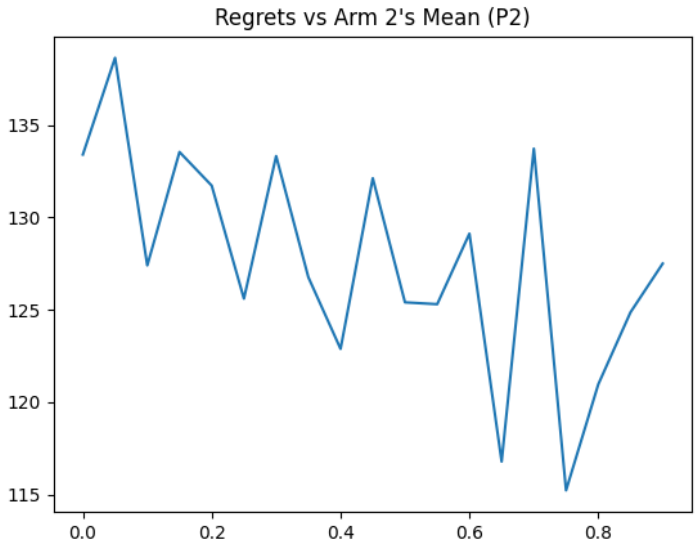

Fig. 5. Task 2B UCB: Regret v. Mean of Arm 2 (p2)

2) Code Description: Algorithm

Listing 12. alogorith calling function

```
#creating the array of arm means as described
task2Bp2s = task2Ap2s
task2Bp1s = np.add(task2Bp2s, 0.1)

#calling the function to execute UCB algorithm as written in the code in Task 1
regrets_ucb_2B = task2(UCB, 30000, task2Bp1s, task2Bp2s)
```

Plotter Function

Listing 13. plotter function

```
#x, y are the respective axes which get p2 and the regrets respectively
def plotter(x, y, what_this_is):
  plt.plot(x, y)
  title = "Regrets vs Arm 2's Mean (P2)"
  plt.title(title)
  #the plot is then saved as a .png file
  plt.savefig("task1-{}-{}.png".format(what_this_is, time.strftime("%Y%m%d-%H%M%S")))
  plt.clf()
```

Part 2: KL-UCB

A. Plot

The plot for KL-UCB algorithm has been shown in the figure.

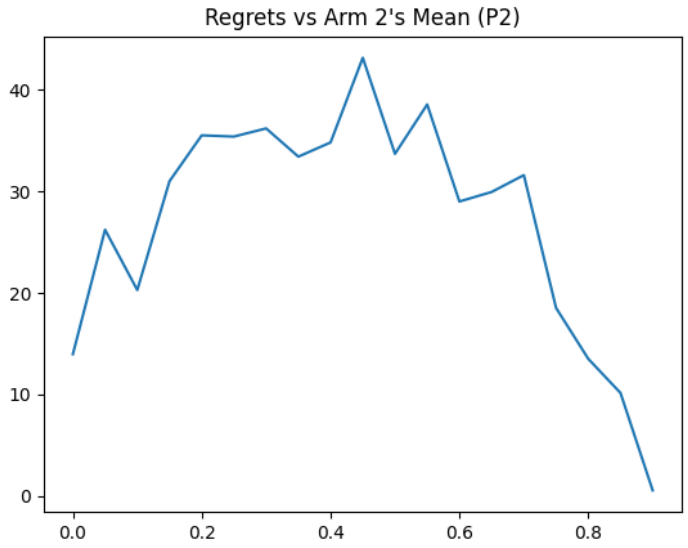

Fig. 6. Task 2B KL-UCB: Regret v. Mean of Arm 2 (p2)

1) Code Description: Algorithm

Listing 14. alogorith calling function

```
#creating the array of arm means as described
task2Bp2s = task2Ap2s
task2Bp1s = np.add(task2Bp2s, 0.1)

#calling the function to execute UCB algorithm as written in the code in Task 1
#horizon = 30,000
regrets_kl_ucb_2B = task2(KL_UCB, 30000, task2Bp1s, task2Bp2s)
```

Interpretation

Graph 1 - UCB Graph 2 - KL-UCB As expected KL-UCB is better as minimizing regret since it provides tighter bounds. Additionallym when both arms have high values of means, we see that it does not matter which arm is chosen which results in a lesser value of regret. The peak at p2 = 0.5(approximately) can be explained as follows. In this scenario, the algorithm may find it challenging to distinguish between them, leading to more exploration and higher regret.

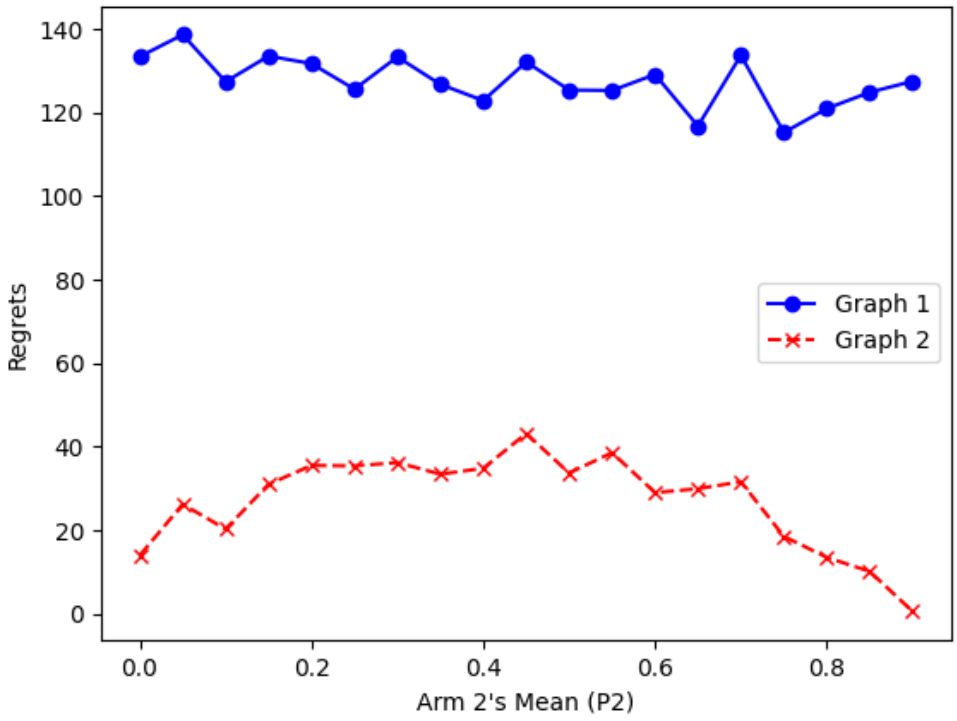

Fig. 7. Task 2B Comparative: Regret v. Mean of Arm 2 (p2)

IV. Task 3

Problem Statement

This task involves dealing with a bandit instance where your pulls are no longer guaranteed to be successful and have a probability of giving faulty outputs. When you pull an arm, it has a certain known probability of giving the correct output of the arm, otherwise it returns a 0 or 1 uniformly at random. Your task is to come up with a good algorithm to maximise the reward for this faulty bandit setting.

All Tried Solutions

The algorithm used is a modification of the UCB algorithm. The only change is in the reward updation function. Since we know the probability with which we get a faulty output, at each iteration, we will decide whether we are considering the obtained reward as the output of the bandit or a random result. If it is an actual result, we update the arm means and counts. Otherwise we just increase the no. of timesteps so that we can have higher exploration bonus. This algorithm was also tried by using KL-UCB as the bound based on which the decision is taken however that lead to higher computational load and did not improve rewards by much which is why that method was discarded.

1) Code Description: Algorithm

Listing 15. alogorith calling function

```
#deciding whether this reward is due to a fault or an actual pull of an arm
fault_or_not = np.random.choice([0,1], p=[self.fault, self.not_fault])
#0==fault, 1==not fault
if fault_or_not:
    n = self.counts[arm_index]
    if n==0:
```

```
        self.values[arm_index] = (self.values[arm_index]+reward)/2
    else:
        self.values[arm_index] =  ((n- 1) / n) * self.values[arm_index] + (1 / n) * reward
    self.good_ones += 1
    self.counts[arm_index] += 1
#updating the no. of timesteps irrespective of which case were in
self.curr_timestep += 1
```

Best Solution from All Tried Solutions

When Thompson Sampling was run without any modifications, we find that it outperforms all the above solutions. It seems to be immune to this faulty bandit problem. This is why Thompson Sampling has been used as the solution for this Task instead of the above listed methods. Thompson Sampling has been described in Task 1 Section C.

V. Task 4: Multi-multi-arm Bandit

Problem Statement

This task involves dealing with two bandit instances at once! In this task, whenever you specify an arm index to pull, one of two given bandit instances is chosen uniformly at random, and the arm corresponding to your provided index is pulled (both instances have equal number of arms). Once an arm is pulled, the environment returns the reward obtained along with which bandit instance was chosen for that pull. Your task is to come up with a good algorithm to maximise the reward for this multi-multi-armed bandit setting.

All Tried Solutions

In this we use a different modification of UCB algorithm. Since we do not know the set from which the arm at that arm, we maintain UCB values for all arms of both sets, take an average of ucb values for each arm index and then pick the arm for which the averaged UCB value is maximum. This was tried with KL-UCB but again, it took longer to run and did not improve performance by much which is why it was discarded. A variation of Thompson Sampling was also tried where I averaged the sampled values from the beta distribution for the 2 sets but that gave way worse results because averaging in that case is clearly useless.

1) Code Description: Variables

Listing 16. Variables

```
## Twice the number of variables have been defined as in Vanilla UCB since we are keeping track
     of both sets separately
self.curr_timestep = 0
#to check if all arms have been pulled atleast once
self.once_0 = 0
self.once_1 = 0
#no. of pulls for an arm
self.counts_0 = np.full((num_arms,), 1e-5)
self.counts_1 = np.full((num_arms,), 1e-5)
self.initial_pulls = np.arange(num_arms)
np.random.shuffle(self.initial_pulls)
#empirical mean of rewards
self.values_0 = np.full((num_arms,), 1e-5)
self.values_1 = np.full((num_arms,), 1e-5)
self.ucb_arms_0 = np.zeros(num_arms)
self.ucb_arms_1 = np.zeros(num_arms)
self.actual_ucbs = np.zeros(num_arms)
```

Pull Function

Listing 17. Variables

```
##ensuring that all arms are pulled at least once
if self.once_0 + self.once_1 < 2*self.num_arms:
    if self.once_0 < self.once_1:
        return self.initial_pulls[self.once_0]
    else:
        return self.initial_pulls[self.once_1]
##taking average of UCB values of both arms of same index
for i in range(self.num_arms):
    self.ucb_arms_0[i] = self.values_0[i] + math.sqrt((2*math.log(self.curr_timestep))/self.
        counts_0[i])
    self.ucb_arms_1[i] = self.values_1[i] + math.sqrt((2*math.log(self.curr_timestep))/self.
        counts_1[i])
    self.actual_ucbs[i] = (self.ucb_arms_0[i] + self.ucb_arms_1[i])*0.5
return np.argmax(self.actual_ucbs)
```

Reward Updation

Listing 18. Variables

```
##if arm from the first set is pulled
if set_pulled == 0:
    n = self.counts_0[arm_index]
    # if n==1e-5:
    #     self.values_0[arm_index] = (self.values_0[arm_index]+reward)/2
    # else:
    #     self.values_0[arm_index] =  ((n- 1) / n) * self.values_0[arm_index] + (1 / n) *
        reward
    self.values_0[arm_index] =  ((n- 1) / n) * self.values_0[arm_index] + (1 / n) * reward
    self.once_0 += 1
    self.counts_0[arm_index] += 1
##if arm from the second set is pulled
else:
    n = self.counts_1[arm_index]
    # if n==1e-5:
    #     self.values_1[arm_index] = (self.values_1[arm_index]+reward)/2
    # else:
    #     self.values_1[arm_index] =  ((n- 1) / n) * self.values_1[arm_index] + (1 / n) *
        reward
    self.values_1[arm_index] =  ((n- 1) / n) * self.values_1[arm_index] + (1 / n) * reward
    self.once_1 += 1
    self.counts_1[arm_index] += 1

self.curr_timestep += 1
```

Best Solution from All Tried Solutions

When Thompson Sampling was run without any modifications, we find that it outperforms all the above solutions. It seems to be immune to this faulty bandit problem. This is why Thompson Sampling has been used as the solution for this Task instead of the above listed methods. Thompson Sampling has been described in Task 1 Section C.