ASSIGNMENT 1

SHUBHAM LOHIYA
18D100020

18th September 2020

—

CS 747

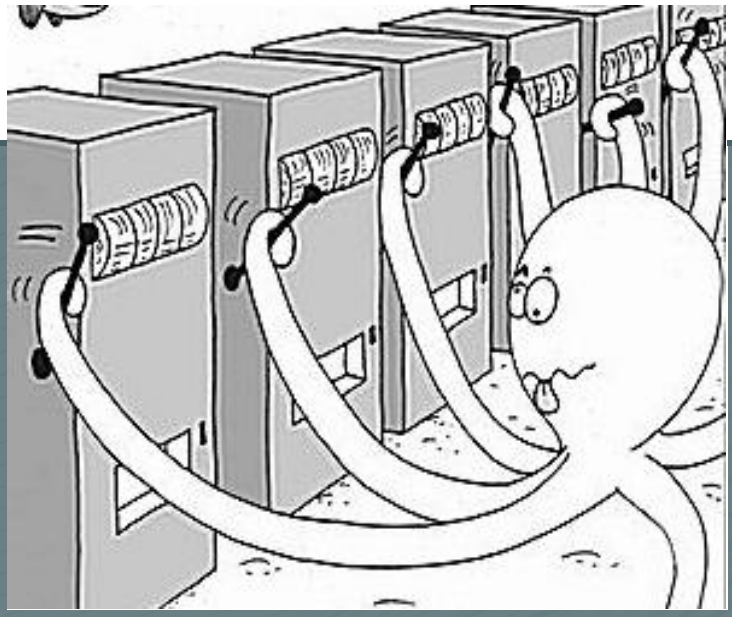
—

Shivaram Kalyanakrishnan

INTRODUCTION

This document is a report for the first assignment of the CS 747 – Fundamentals of Intelligent and Learning Agents course offered at IIT Bombay.

This assignment is based on Multi-armed Bandits and explores various algorithms to maximize the bandit's reward and compares the efficacy of each algorithm.



Task 1 – Implementing Algorithms

ε-Greedy algorithm

The algorithm maintains a belief about the mean rewards of each arm *(a)* according to the formula:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}},$$

where $\mathbb{1}_{predicate}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not.

If the denominator is zero (arm not pulled even once yet), we define $Q_t(a) \doteq 0$

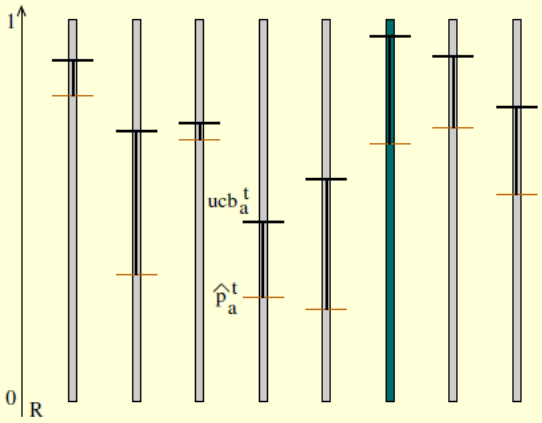
At every time step:

1. with probability (1-ε), it exploits this knowledge (greedy) by pulling the arm with the highest empirical mean (with ties broken arbitrarily)
2. with probability ε, it explores by pulling a random arm and thus ensuring that as the number of steps increases, each arm is sampled infinitely and hence our empirical mean for each arm converges to the true mean. This will ensure that our belief of the arms' means are close to the actual ones so that we don't exploit a sub-optimal arm in the exploitation step.

UCB (Upper Confidence Bound) algorithm

While the ε-greedy algorithm explores indiscriminately without any preference for algorithms that are nearly greedy or particularly uncertain, it would be better if an algorithm could select a non-greedy action on the exploration step according to their potential for actually being optimal. This can be done by taking into account both their closeness of current estimates to the greedy arm and the uncertainty in those estimates.
This can be achieved by calculating an upper confidence bound (UCB) for each arm and then sampling on the basis of these UCBs. These can be calculated as follows: (comes from the Hoeffding's inequality)

- At time t, for every arm *a*, define $\text{ucb}_a^t = \hat{p}_a^t + \sqrt{\frac{2\ln(t)}{u_a^t}}$.
- $\hat{p}_a^t$ is the empirical mean of rewards from arm *a*.
- u_a^t the number of times *a* has been sampled at time *t*.



- Sample an arm *a* for which ucb_a^t is maximal.

*ties are broken arbitrarily if two arms have same UCB

As this algorithm does not explore indiscriminately, it is expected to perform better than the ε-greedy algorithm in the long run. According to the analysis conducted in class, this algorithm achieves logarithmic regret over long horizons

Each arm is pulled once at the beginning so that $u_a^t > 0$ for UCB calculation. So, the algorithm uses UCBs to pull only after n time steps for an n-armed bandit

KL-UCB (Upper Confidence Bound) algorithm

This algorithm is very similar to the UCB algorithm with just the upper confidence bounds (ucb-kl) defined instead as follows:

$\text{ucb-kl}_a^t = \max\{q \in [\hat{p}_a^t, 1] \text{ such that } u_a^t KL(\hat{p}_a^t, q) \leq \ln(t) + c\ln(\ln(t))\}$, where $c \geq 3$.
KL-UCB algorithm: at step t, pull $\text{argmax}_a\ \text{ucb-kl}_a^t$.

*two arms having the same ucb-kl is highly unlikely. In case it happens, the first arm encountered is pulled

Here, the KL(a, b) function is the Kullback Leibler Divergence.
Each arm is pulled once at the beginning so that $u_a^t > 0$ for and $t > 1$ so $\ln(t) > 0$. So, the algorithm uses KL-UCBs to pull only after n time steps for an n-armed bandit
The value of c I've chosen in my algorithm is 3 as $c \geq 3$ and 3 performed empirically better than higher values.

This algorithm performs better than UCB as the KL inequality provides a tighter upper bound than the Hoeffding's inequality.

Thompson Sampling algorithm

This algorithm holds a belief (beta) distribution for the rewards offered by each arm and samples a value for each arm at every time step and then pulls the arm for which this value is maximal.
Here is a quick overview of the implemented algorithm:

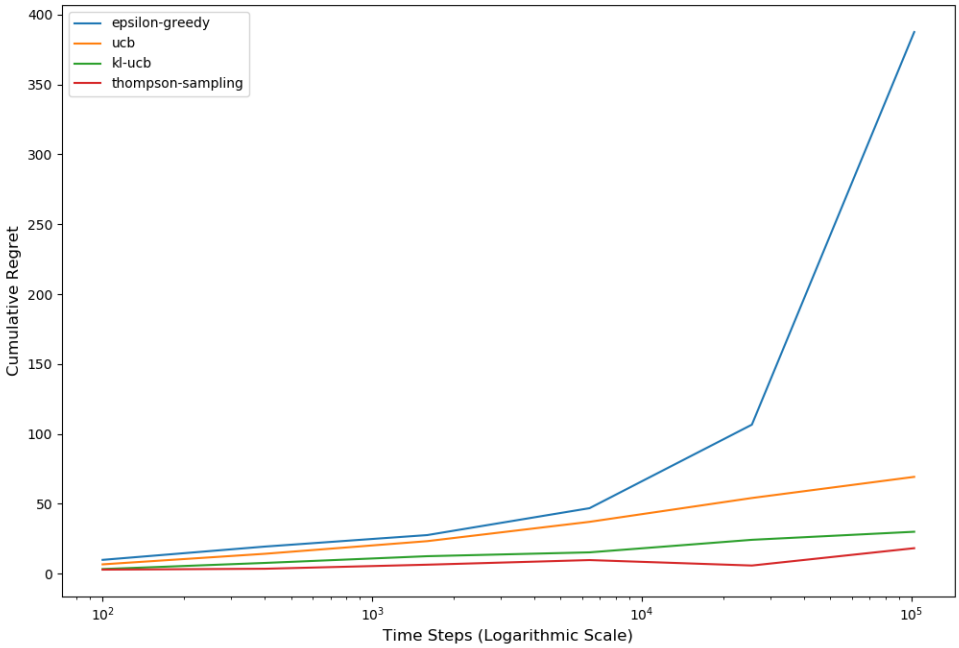
- At time t, let arm a have s_a^t successes (ones/heads) and f_a^t failures (zeroes/tails).
- $Beta(s_a^t + 1, f_a^t + 1)$ represents a "belief" about the true mean of arm a.
- Mean = $\frac{s_a^t+1}{s_a^t+f_a^t+2}$; variance = $\frac{(s_a^t+1)(f_a^t+1)}{(s_a^t+f_a^t+2)^2(s_a^t+f_a^t+3)}$.
- Computational step: For every arm a, draw a sample $x_a^t \sim Beta(s_a^t + 1, f_a^t + 1)$.
- Sampling step: Sample an arm a for which x_a^t is maximal.

*two arms having the same x_a^t is highly unlikely. In case it happens, the first arm encountered is pulled

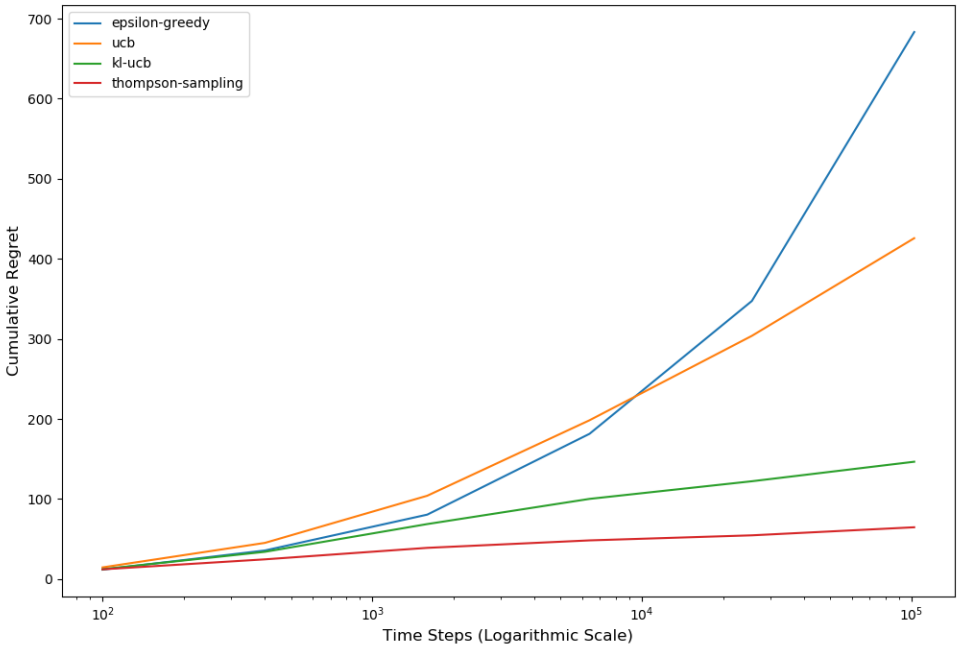
This algorithm performs the best and achieves the minimum regret.

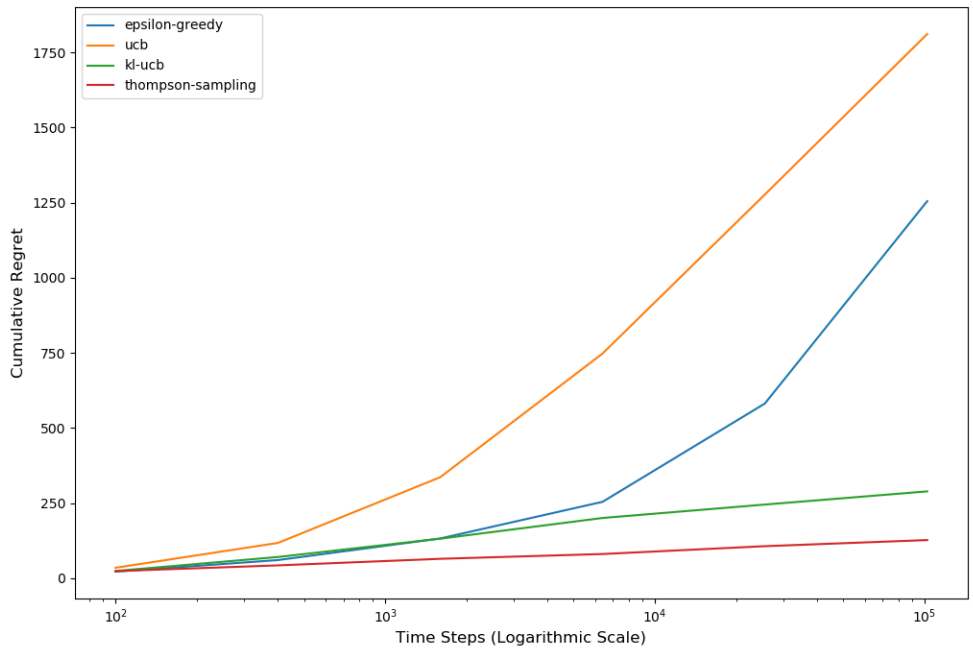
PLOTS FOR TASK 1

Instance 1



Instance 2





Observations:

1. Overall, it can be seen that ε-greedy algorithm performs worst and Thompson Sampling performs best in the long run. (For Instance-3, ε-greedy is still performing better than UCB at 100K horizon but at around 1 million pulls, UCB gives significantly lower regret as compared to ε-greedy)
2. The graphs for UCB, KL-UCB and Thompson Sampling appear linear for higher horizons for a logarithmic scale of the x-axis thus implying that these algorithms accumulate logarithmic regret in the long run. This empirical result is consistent with theory
3. ε-greedy accumulates super-logarithmic regret because even though it might have found the optimal arm to exploit, it's still might pull non-optimal arms on the exploration step with high probability so it keeps accumulating that regret and will continue to do so in perpetuity
4. We see that KL-UCB performs better than regular UCB. As mentioned before, this is because the KL inequality provides a tighter upper bound for probability of deviation from the mean than the Hoeffding's inequality
5. Thompson Sampling appears to be more robust to different kinds of multi-armed bandit instances and provides consistently good performance with similar regret trends

Task 2 – Thompson Sampling with hint

Introduction

In this task, we have been provided a random permutation of the true means of the arms of the bandit instance. We have to devise an algorithm that uses this extra knowledge ("hint") to perform better than our previous best performing algorithm, "Thompson Sampling."

Strategy and algorithm

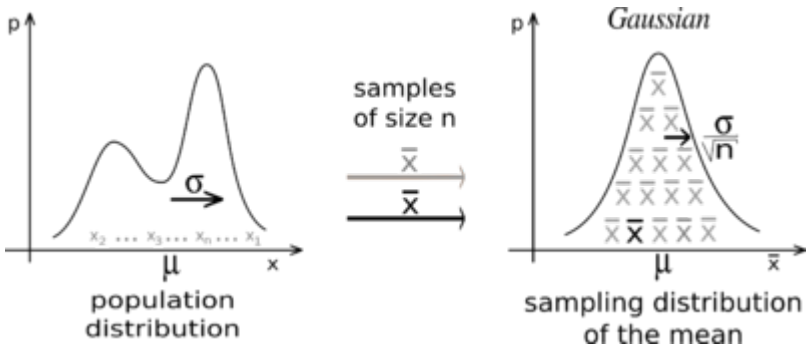
From the hint given, we can find out what the optimal arm mean value is (even though we won't know which arm it belongs to).

$$mean_{optimal} = \max([permutaion\ of\ arm\ means])$$

At some time-step, we want to sample an arm whose empirical mean is closest to the optimal mean as it is most likely to be the optimal arm. Let's approach the situation like this:

- Assume that the arm you are looking at is the optimal arm
- Then at time step t, if the count is c, then my empirical mean of this so-called "optimal arm" can be looked at as the mean of a sample of size c that was sampled from the probability distribution for rewards for the optimal arm (with mean $p^*$and variance $p^*(1-p^*)$ [standard formula for bernoulli dist.])
- By the central limit theorem (CLT), this mean will be from a gaussian distribution where the mean is p^* and the variance is $p^*(1-p^*)/n$ where n is the sample size (Here, n = c)
- Now let's calculate the value of this gaussian pdf. at the arm's empirical mean and call this value v_a
- Calculate v_a for each arm in a similar manner. We know that the larger the v_a, the closer arm a's empirical mean is to p^* and the sharper the gaussian function is
- So let's pull the arm $argmax_a(v_a)$.

*We first pull each arm once and then start this algorithm from time step n for an n armed bandit. This is to have finite variance for pdf.s



source: Wikipedia

So why does this strategy work?

If at any point, an arm is pulled but gives zero reward, it's empirical mean decreases and its count increases. That means on the next time step, the gaussian for this arm will be sharper (due to the decreased variance as a result of an increase in count), and also the empirical mean of the arm will be farther than the mean. These two factors decrease the gaussian value for later time-steps hence reducing the chance of pulling this arm

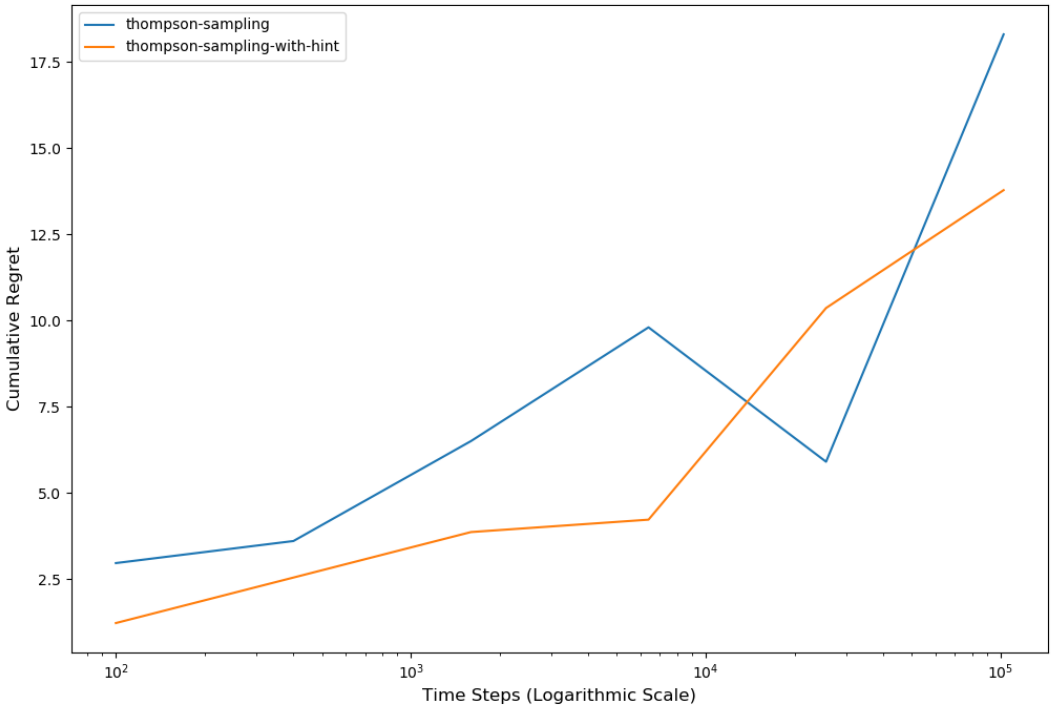
On the other hand, if the reward had been 1, the gaussian would still be sharper on the next time step (as count still increases), and the empirical mean would have also increased. These two factors lead to an increase in the value of the gaussian for later time-steps and thus increase the chance of pulling this arm.

Observations: (Based on tinkering with code and plots [next page])

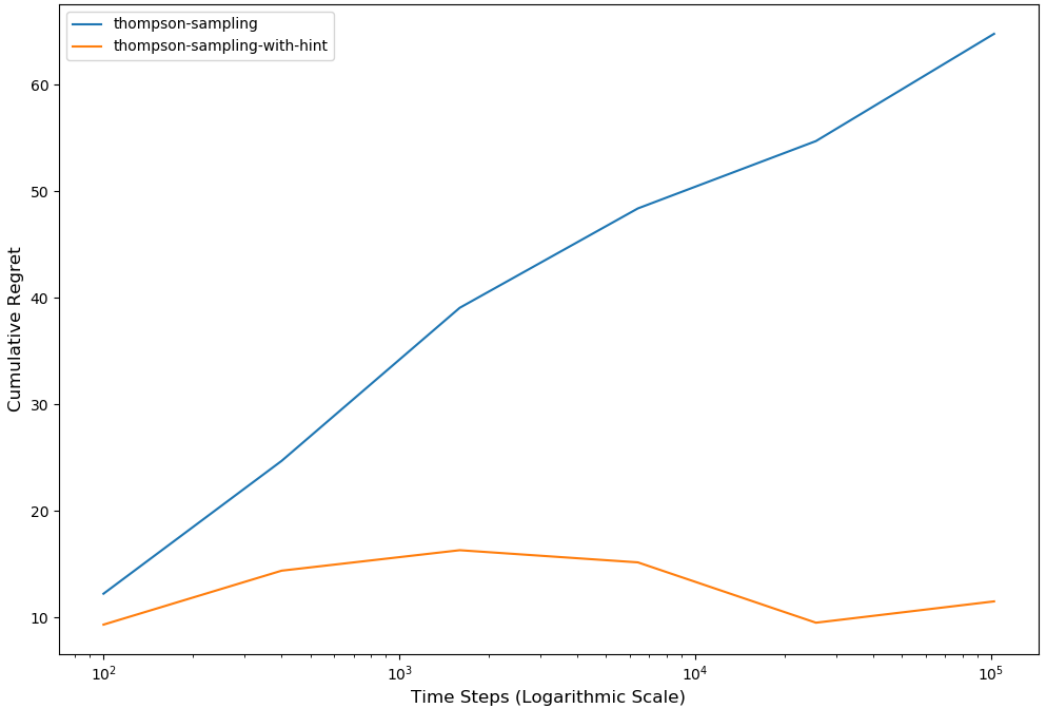
- The improved algorithm with hint performs way better than Thompson sampling at all kinds of horizons
- Using $variance^2$ as the new variance in this algorithm for calculating the gaussian values gives marginally better results (for all 3 instances) due to sharper distributions. Higher powers can lead to worse results though
- Not sure without theoretical proof, but it seems like this algorithm will give a sub-logarithmic regret over long horizons

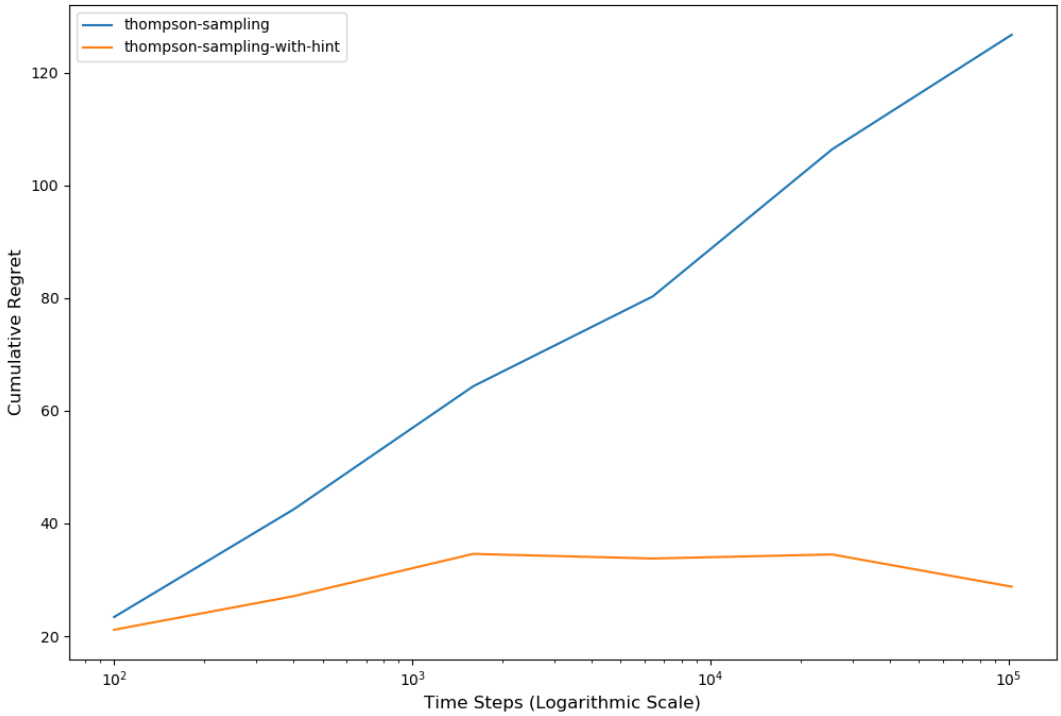
PLOTS FOR TASK 2

Instance 1



Instance 2





Task 3 – Playing with Epsilon

Introduction

We are tasked with finding if there are three epsilons such that $\epsilon_1 < \epsilon_2 < \epsilon_3$ such that cumulative regret for the ε-greedy algorithm for ϵ_2 is minimum.

Result of Analysis

The values $\boxed{\epsilon_1 = 0.0001,\ \epsilon_2 = 0.02,\ \epsilon_3 = 0.1}$ achieve this situation for all three instances. [Find proof attached on the next page]
Very high epsilons lead to unnecessary exploration even after finding the optimal arm to exploit, leading to excessive regret accumulation.
Very low epsilons can lead to the algorithm getting stuck on exploiting a sub-optimal arm, thus accumulating regret.

Instance 1

```
with open('../instances/i-1.txt') as f:
    arms = [line.strip() for line in f]
    arms = list(map(float, arms))
```

```
# epsilon = 0.0001
average = 0
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.0001).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
4075.5600000000004
```

```
# epsilon = 0.02
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.02).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
387.48
```

```
# epsilon = 0.1
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.1).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
2042.8999999999999
```

Instance 2

```
with open('../instances/i-2.txt') as f:
    arms = [line.strip() for line in f]
    arms = list(map(float, arms))
```

```
# epsilon = 0.0001
average = 0
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.0001).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
8739.859999999999
```

```
# epsilon = 0.02
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.02).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
683.3
```

```
# epsilon = 0.1
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.1).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
2101.3999999999996
```

Instance 3

```
with open('../instances/i-3.txt') as f:
    arms = [line.strip() for line in f]
    arms = list(map(float, arms))
```

```
# epsilon = 0.0001
average = 0
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.0001).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
13334.360000000002
```

```
# epsilon = 0.02
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.02).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
1254.5999999999995
```

```
# epsilon = 0.1
average = 0.
for seed in range(50):
    average += (float(epsilon_greedy(arms, seed, 102400, 0.1).split(", ")[-1])-average)/(seed+1)
print(average)
```

```
4287.6
```