Assignment 1

Ujjwal Sharma - 23M0837

Task 1

Epsilon Greedy

The epsilon-greedy algorithm was already implemented.
After each pull, the values returned are used to update both the Uta (the count of pulls) and the Pta (the estimated value) of the arm that was pulled.
During each pull, with probability ε, a random arm is selected; otherwise, the arm with the highest Pta is chosen.

UCB Algorithm

The implementation of the UCB algorithm follows the definition outlined in the slides.

After each pull, the values returned (reward and arm) are used to update both the Uta (the count of pulls) and the Pta (the estimated value) of the arm that was pulled.

Using these updated values, the UCB of the same arm is then recalculated.

During each pull, the arm with the highest UCB is selected.

KL-UCB Algorithm

The implementation of the KL-UCB algorithm follows the definition outlined in the slides.

We first define the KL-Divergence function

```
def kl_divergence(p, q):

    max_value_for_divergence = float('inf')

    if q == 0 and p == 0:

        return 0

    elif q == 0 and not p == 0:

        return max_value_for_divergence

    elif q == 1 and p == 1:

        return 0

    elif q == 1 and not p == 1:

        return max_value_for_divergence

    elif p == 0:

        return math.log(1 / (1 - q))

    elif p == 1:

        return math.log(1 / q)

    return p * math.log(p / q) + (1 - p) * math.log((1 - p) / (1 - q))
```

This definition is taken from
https://github.com/guptav96/bandit-algorithms/blob/main/algo/klucb.py

From the slides, Lecture 3, slide 12:

ucb-klt a is the solution $q \in [pta, 1]$ to $KL(pta, q) = \ln(t)+c*\ln(\ln(t))/uta$.

Also in the slides it is given that KL(p,q) increases linearly with p and q so we can use binary search to find the value of q that satisfies the equation

$$uta*KL(pta, q) \leq \ln(t) + c \ln(\ln(t))$$

After each pull, the values returned are used to update both the Uta (the count of pulls) and the Pta (the estimated value) of the arm that was pulled.

During the pull, for each arm, we search for a *q* in the range (Pta, 1] such that the above equation holds. The arm with the highest value of *q* is then selected.

In my implementation, the search for *q* is restricted to a predefined number of iterations, similar to https://github.com/guptav96/bandit-algorithms/blob/main/algo/klucb.py

I also noticed that even though the number of iterations is constant while increasing the allowed precision of (upper limit - lower limit), the time significantly increases but the regrets are small.

Here are some examples

Number of iterations: 1000, precision: 1e-2

```
(env747) PS C:\Users\ushar\STUDY\IIT Bombay\RL\Assignment 1\code> python .\autograder.py --tas
k 1 --algo kl_ucb
================== Task 1 ==================
Testcase 1
KL-UCB            : PASSED. Regret: 14.26

Testcase 2
KL-UCB            : PASSED. Regret: 74.02

Testcase 3
KL-UCB            : PASSED. Regret: 71.64

Time elapsed: 73.31 seconds
```

Number of iterations: 1000, precision: 1e-4

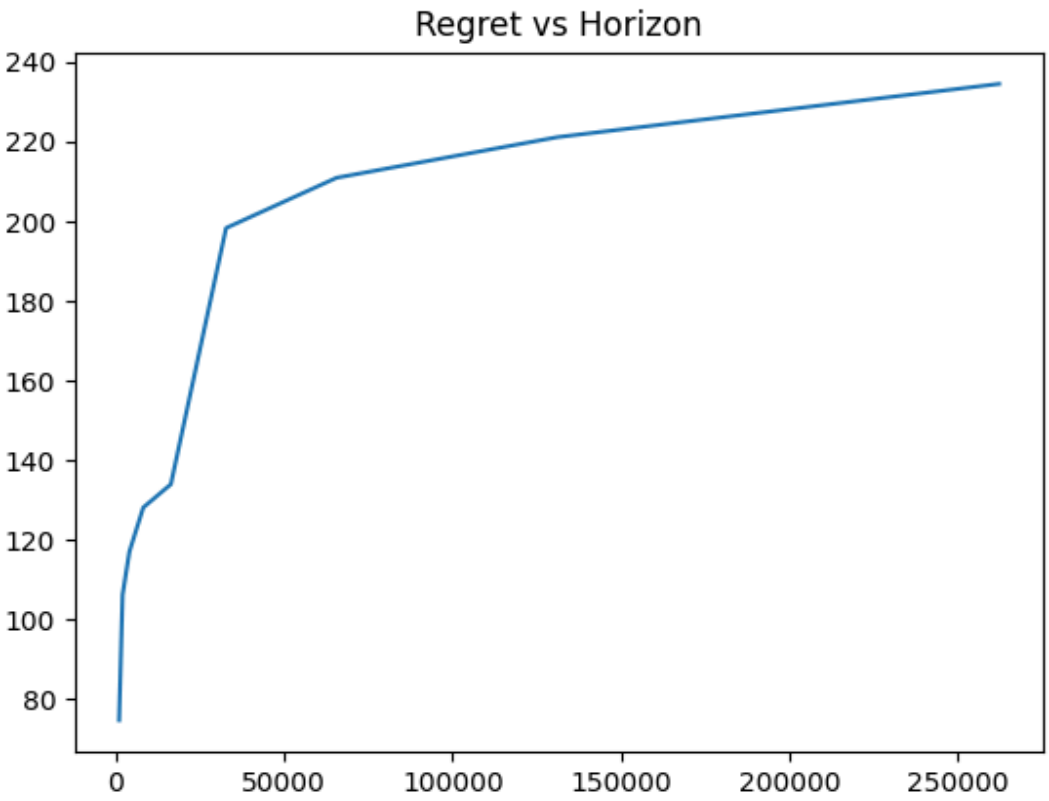
```
(env747) PS C:\Users\ushar\STUDY\IIT Bombay\RL\Assignment 1\code> python .\autograder.py --tas
k 1 --algo kl_ucb
================== Task 1 ==================
Testcase 1
KL-UCB            : PASSED. Regret: 13.78

Testcase 2
KL-UCB            : PASSED. Regret: 82.71

Testcase 3
KL-UCB            : PASSED. Regret: 52.93

Time elapsed: 167.57 seconds
```

The final Plot is for the number of iterations: 1000, precision: 1e-2



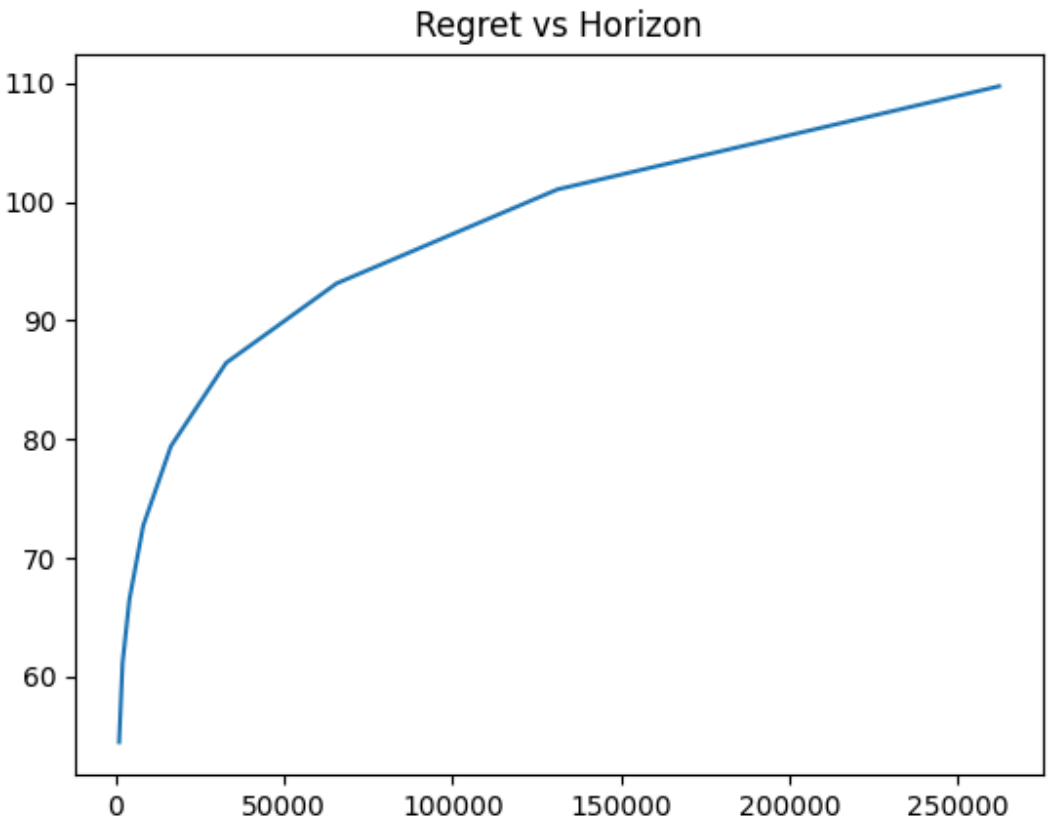
This graph illustrates the KL-UCB algorithm's arm selection process. In the early stages, when the horizon is smaller, the algorithm focuses on exploration, represented by the straight line. However, around the 50,000 horizon mark, the algorithm has explored enough and refined its value estimates, and it starts exploiting the arms with maximum estimates.

Thompson Sampling

The implementation of the Thompson Sampling algorithm follows the definition outlined in the slides.

After each pull, the values returned (reward and arm) are used to update the success or failure count for the arm that was pulled.

During the pull, we sample *n* values from the Beta distribution for each arm, where *alpha_i = (success_i + 1)* and *beta_i = (failure_i + 1)*. The arm with the highest sampled value is then selected.



This graph illustrates the sampling process in the Thompson Sampling algorithm. The exploration phase is minimal, with the algorithm quickly estimating the parameters and beginning exploitation even before reaching the 50,000 horizon mark. As shown in the curve, Thompson Sampling results in the lowest regret compared to the other three algorithms.

Task 2

Question:

This task involves dealing with a bandit instance where pulling arms has an associated cost. In this setting, you can request pulls from an oracle by providing a query set (S) which is an array of arm indices. The oracle then chooses an arm from this set uniformly at random and provides you with both the reward obtained and the arm it decided to pull. For this action, the oracle charges you a cost of 1/|S|.

Solution:

Here the problem is that we will have to select the pulls from a set of arms and after selecting we would get the reward of the arm and a penalty for selecting the arm. The goal is to maximize the reward.

Eg, if we only give a single element in the set, we would get a reward of pi but a penalty of -1 so the total reward would be pi - 1
We need to make this penalty as small as possible while making sure that we get a high reward.

Given a set {a1, a2…, ak}, the penalty is 1/k.

The expected reward from this set is $E_R(s) = \frac{\sum_{i=1}^{k} a_i}{k} - \frac{1}{k} = \frac{(\sum_{i=1}^{k} a_i) - 1}{k}$

To get a high expected reward at every step, we need to propose a set S such that $E_R(s)$ is always maximum.

The basic framework of the algorithm is to do exploration and exploitation. We saw from the previous task that Thompson sampling does this very efficiently and produces minimum regret. So I propose a variant of the **Thompson sampling algorithm**.

1. $For\ every\ pull\ of\ the\ bandit,\ return\ set\ S\ which\ has\ the\ maximum\ E_R(S)$
2. $After\ the\ pull,\ update\ the\ success/failure\ of\ the\ arm\ that\ was\ pulled.$

Maximum $E_R(s)$ can be returned as follows.

From Thompson sampling, we know that the arm that has a bigger mean will have a bigger p_i.

The algorithm is as follows

1. $At\ each\ pull,\ sample\ the\ \ Beta(s_{a_i} + 1, f_{a_i} + 1)\ (= samples\ (SS))$
2. $Sort\ the\ samples\ in\ descending\ order.$

Now the idea is to return a subset from these sorted sample values such that the expected reward becomes maximum.

3. $find\ the\ last\ element\ of\ the\ subset\ from\ sorted\ samples\ (SS)\ such\ that\ E_R(S) = \frac{(\sum_{i=1}^{k} a_i) - k}{k}\ is\ maximum$

This can be done by applying a single loop on the sorted samples and then calculating the value.

4. $Return\ such\ set\ S.$

The reason that this will give the set with maximum expected reward is that because the values are sorted in descending order, a particular subset constructed in this manner will contain the maximum values that can come inside the subset of the same size.

Implementation

```
class CostlySetBanditsAlgo(Algorithm):
    def __init__(self, num_arms, horizon):
        self.num_arms = num_arms
        self.horizon = horizon
        self.succesful_pulls = np.zeros(num_arms)
        self.failed_pulls = np.zeros(num_arms)

    def give_query_set(self):
        beta_values = np.random.beta(self.succesful_pulls + 1,
self.failed_pulls + 1)

        sorted_arms = np.argsort(beta_values)[: : -1]
        max_expected_reward = float('-inf')
        cumulative_sum = 0
        index_to_be_selected = -1
        for i in range(self.num_arms):
```

```
            cumulative_sum = cumulative_sum + beta_values[sorted_arms[i]]
            expected_reward = (cumulative_sum - 1) / (i + 1)
            if expected_reward > max_expected_reward:
                max_expected_reward = expected_reward
                index_to_be_selected = i


        return sorted_arms[:index_to_be_selected + 1]

    def get_reward(self, arm_index, reward):

        if reward == 1:
            self.succesful_pulls[arm_index] += 1
        else:
            self.failed_pulls[arm_index] += 1
```

Auto-grader output:

```
(env747) PS C:\Users\ushar\STUDY\IIT Bombay\RL\Assignment 1\code> python .\autograder.py --task 2
================== Task 2 ==================
Testcase 1
Costly Set Bandit Algorithm: PASSED. Net Reward: 3967.49

Testcase 2
Costly Set Bandit Algorithm: PASSED. Net Reward: 768.97

Testcase 3
Costly Set Bandit Algorithm: PASSED. Net Reward: 5758.81

Testcase 4
Costly Set Bandit Algorithm: PASSED. Net Reward: 13019.47

Testcase 5
Costly Set Bandit Algorithm: PASSED. Net Reward: 7962.23

Testcase 6
Costly Set Bandit Algorithm: PASSED. Net Reward: 10127.65

Testcase 7
Costly Set Bandit Algorithm: PASSED. Net Reward: 9001.74

Testcase 8
Costly Set Bandit Algorithm: PASSED. Net Reward: 7521.51

Time elapsed: 120.51 seconds
```

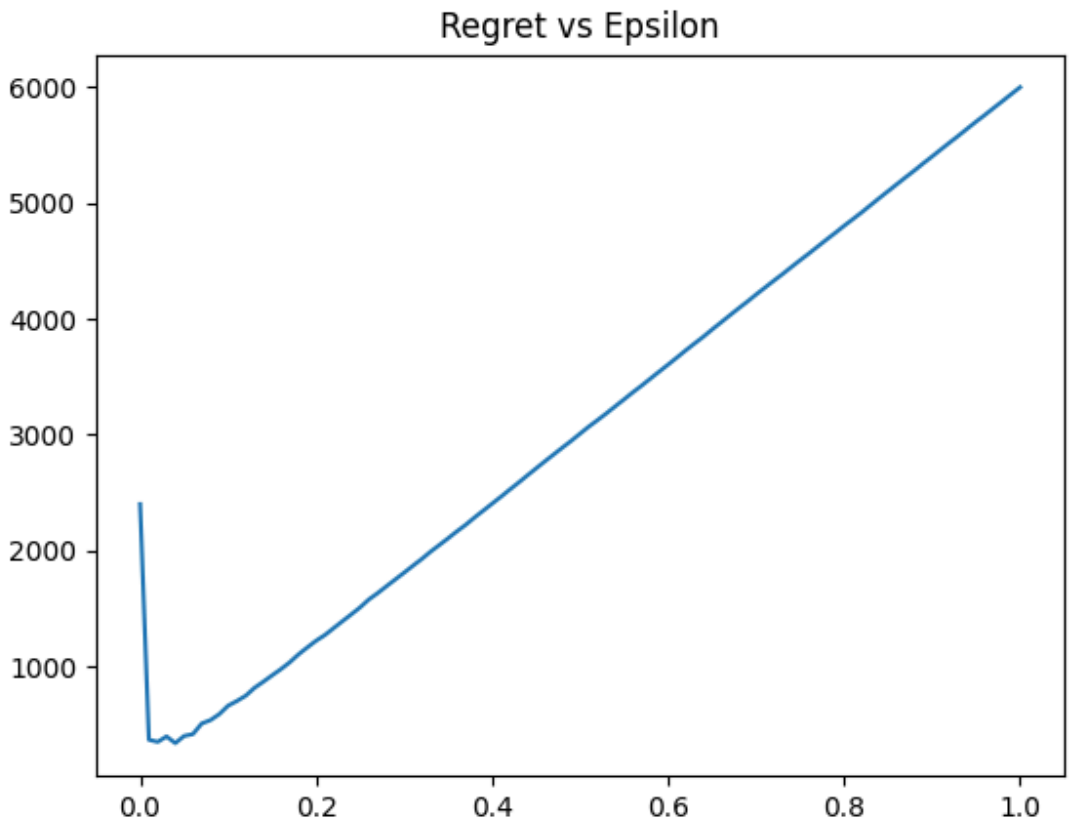
Task 3

Exploring the effect of epsilon in the e-greedy algorithm.

Tie-breaking is uniform at random.

Observations

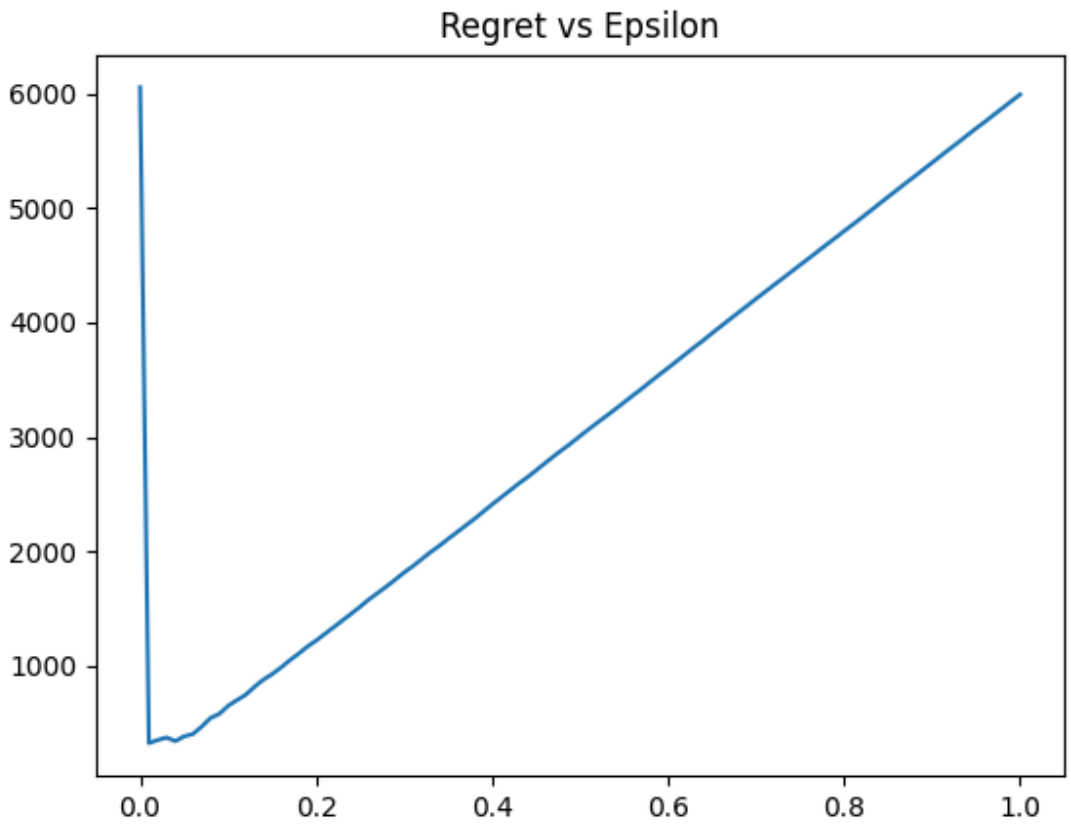
1. When epsilon is very low (0.01, 0.02), the algorithm tends to exploit the arm that appears to be the best at that moment, without sufficient exploration. As a result, we observe relatively high regret in the initial phase (premature exploitation). In the special case where epsilon equals 0, at every time step the model always selects the arm with the maximum empirical average (because the arms were pulled once the regret was also small initially, see Ablation).
2. The epsilon value reaches near-optimal performance around 0.04, with 0.04 being the ideal value in this scenario. This represents a perfect balance between exploration and exploitation. At this level, the algorithm explores enough to stabilize its estimates, bringing them closer to the true values, and the subsequent exploitation minimizes regret.
3. As epsilon increases beyond a certain threshold (0.2 in this case), the algorithm starts to explore randomly, as evidenced by the graph. When the epsilon continues to rise, the regret increases linearly with the epsilon. This occurs because the algorithm starts to prioritize random arm selections over exploiting the knowledge gained from its past estimates. In the extreme case where epsilon equals 1, the algorithm samples arms randomly at every decision point, leading to a significant increase in regret.



The figure shows Regret vs. Epsilon values when each arm is pulled once at the start.

Ablation

In this ablation study, I also experimented with a version of the code where no arms are pulled at the start. The observations made previously remain valid in this case as well. However, the key difference lies in the initial phase. When epsilon equals 0, since no arms are pulled at the start, the algorithm begins by selecting arms randomly, each with an average value of 0. As a result, it incurs a significant regret.

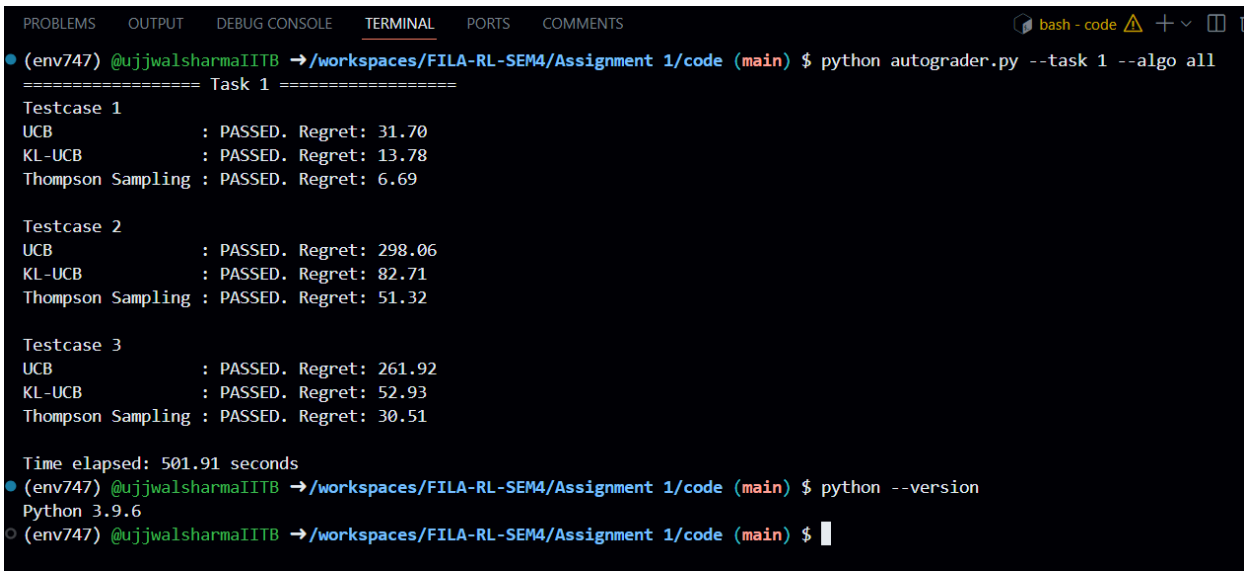


The figure shows Regret vs. Epsilon values when no arm is pulled at the start.

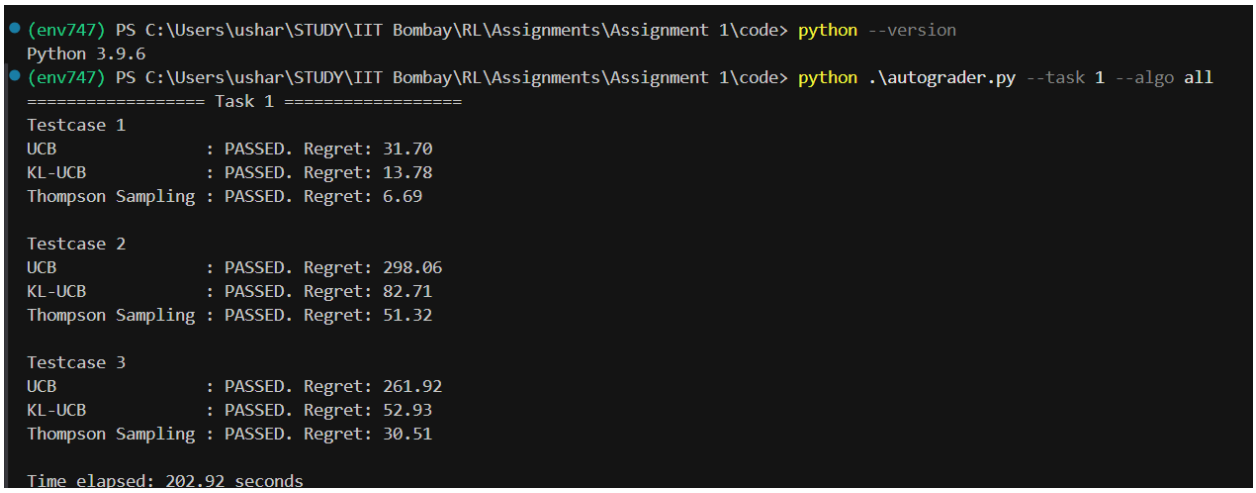
Appendix

I have tested my code on both windows and Linux using python 3.9.6

Linux



Windows



All the code can be found at
https://github.com/ujjwalsharmaIITB/FILA-RL-SEM4/tree/main/Assignment1