# Assignment 01

Lopmanudra Biswal, 23b0049

## 1 TASK 1

### 1.1 UCB (Upper Confidence Bound)

#### 1.1.1 Idea

UCB selects the arm with the highest index:

$$UCB_i = \hat{\mu}_i + \sqrt{\frac{2 \ln t}{n_i}}$$

where:

- $\hat{\mu}_i$ is the average reward of arm $i$,
- $n_i$ is the number of pulls of arm $i$,
- $t$ is the total number of pulls so far.

The second term is the **exploration bonus**: arms pulled fewer times get a higher bonus.

#### 1.1.2 Code Explanation

```
self.counts = np.zeros(num_arms)    # pulls per arm
self.values = np.zeros(num_arms)    # total rewards per
    arm
self.pulls = 0                      # total pulls
```

**Selecting an Arm.**

```
if self.counts[arm] == 0:
    return arm    # ensure every arm is tried once
```

```
ucb_values = np.zeros(self.num_arms)
for arm in range(self.num_arms):
    mean_reward = self.values[arm] / self.counts[arm]
    bonus = math.sqrt((2 * math.log(self.pulls)) / self.
        counts[arm])
    ucb_values[arm] = mean_reward + bonus

return int(np.argmax(ucb_values))
```

**Updating After Reward.**

```
self.counts[arm_index] += 1
self.values[arm_index] += reward
self.pulls += 1
```

### 1.1.3 graph

[H]

The performance of the UCB algorithm can be evaluated by plotting regret over time. Figure 3 shows how the regret grows with the horizon.

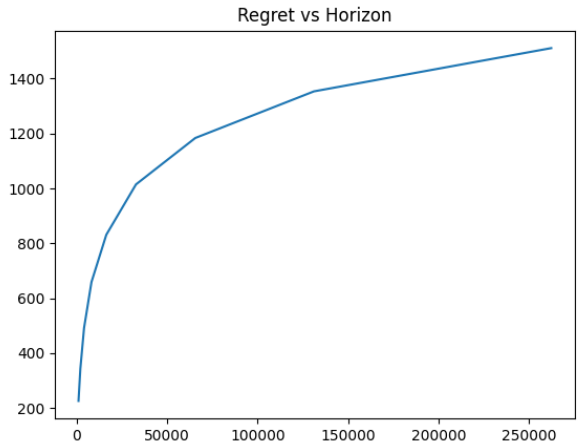


Figure 1: Regret vs Horizon for UCB algorithm.

Initially, regret grows quickly due to exploration, but as the algorithm learns the best arm, the growth rate slows down, showing efficient exploitation.

—

## 1.2 KL-UCB

### 1.2.1 Idea

KL-UCB finds the maximum $q$ such that:

$$KL(\hat{\mu}_i \,\|\, q) \leq \frac{\ln t + 3\ln(\ln t)}{n_i}.$$

This makes the bound **tighter** and especially effective for Bernoulli rewards.

### 1.2.2 Code Explanation

```
self.counts = np.zeros(num_arms)
self.values = np.zeros(num_arms)
self.estimated_means = np.zeros(num_arms)
self.pulls = 0
```

**Arm Selection.**

- If any arm has not been pulled yet, pull it once.
- Otherwise, compute:

$$\text{rhs} = \frac{\ln t + 3\ln(\ln t)}{n_i}.$$

- Perform binary search between $\hat{p}$ and 1:

```
if kl_bernoulli(p_hat, mid) > rhs:
    upper_bound = mid
else:
    lower_bound = mid
```

This finds the largest valid $q$. Then the algorithm picks the arm with the maximum KL-UCB index.

**Updating.** Same as UCB: update counts, values, means, and total pulls.

### 1.2.3 graph

[H]

The performance of the UCB algorithm can be evaluated by plotting regret over time. Figure 3 shows how the regret grows with the horizon.


Figure 2: Regret vs Horizon for $\mathrm{KL}_U CB algorithm$.

In theory, KL-UCB is expected to achieve $\mathcal{O}(\log T)$ regret and obtained graph is approximately linear as the algorithm continuously tries to learn and reduce uncertainty on how are the arms reward distributed.

—

## 1.3 Thompson Sampling

### 1.3.1 Idea

For Bernoulli bandits, use a **Beta distribution**:

$$P(\theta_i) \sim \text{Beta}(\alpha_i, \beta_i),$$

where:

- $\alpha_i$ = successes + 1,
- $\beta_i$ = failures + 1.

At each step:

1. Sample $\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$.
2. Pick the arm with the highest sample.
3. Update $\alpha_i$ or $\beta_i$ depending on reward.

### 1.3.2 Code Explanation

```
self.alpha = np.ones(num_arms)    # successes
self.beta = np.ones(num_arms)     # failures
```

**Selecting an Arm.**

```
samples = np.random.beta(self.alpha, self.beta)
return int(np.argmax(samples))
```

**Updating.**

```
if reward == 1:
    self.alpha[arm_index] += 1
else:
    self.beta[arm_index] += 1
```

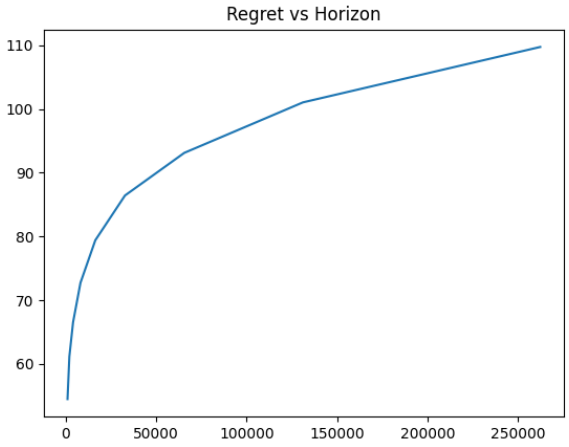


Figure 3: Regret vs Horizon for Thompson algorithm.

The regret curve decreases and flattens over time, showing that the algorithm learns to pick the best arm . Early in the run, the algorithm explores, so regret increases quickly. As more data is collected, Thompson Sampling becomes more confident and exploits the best arm, so the regret curve grows more slowly.

—

# 2 TASK 2

### 2.0.1 Idea

The policy works in three stages:

1. Explore each door once.

2. Pick the best door using expected hits:

$$\text{expected hits} = \frac{S_0}{\text{avg damage}}.$$

3. Apply threshold rules:

   - If a door is almost broken ($\leq 50$ strength left), finish it.
   - If best door requires fewer than 40 estimated hits, commit.
   - Otherwise, pick the door with smallest estimated hits.

## 2.1 Code

```
class StudentPolicy(Policy):
    def __init__(self, K: int, initial_strength: float =
        100.0,
                 rng: Optional[np.random.Generator] =
                     None):
        super().__init__(K, rng)
        self.S0 = initial_strength
        self.best_arm = None
        self.exploring = True
        self.explore_rounds = 1

    def select_arm(self, t: int) -> int:
        # Phase 1: Exploration
        for arm in range(self.K):
            if self.counts[arm] < self.explore_rounds:
                return arm

        # Phase 2: Choose best door
        if self.exploring:
            avg_damage = self.sums / np.maximum(self.
                counts, 1)
            expected_hits = np.where(avg_damage > 1e-6,
                                     self.S0 / avg_damage
                                         ,
                                     np.inf)
            self.best_arm = int(np.argmin(expected_hits))
            self.exploring = False

        # Threshold rules
        avg_damage = self.sums / np.maximum(self.counts,
            1)
        remaining = self.S0 - self.sums
        est_hits = np.where(avg_damage > 1e-8,
                            remaining / avg_damage,
                            np.inf)
```

```
        if np.min(remaining) <= 50:
            return int(np.argmin(remaining))
        if est_hits[self.best_arm] < 40:
            return self.best_arm
        return int(np.argmin(est_hits))

    def update(self, arm: int, reward: float):
        self.counts[arm] += 1
        self.sums[arm] += reward
```

## 2.2 Example Walkthrough

Suppose there are 3 doors with true (but unknown) average damages:

$$\begin{aligned}\text{Door } 0 &\to\approx 5 \text{ damage per hit},\\ \text{Door } 1 &\to\approx 2 \text{ damage per hit},\\ \text{Door } 2 &\to\approx 8 \text{ damage per hit}.\end{aligned}$$

Let the initial strength be $S_0 = 100$.

### Step 1: Exploration Phase

Each door is hit once to gather information.

- Hit Door 0 once $\to$ observe $\approx 5$ damage.
- Hit Door 1 once $\to$ observe $\approx 2$ damage.
- Hit Door 2 once $\to$ observe $\approx 8$ damage.

After this, the estimated average damages are:

$$\text{AvgDamage} = [5,\ 2,\ 8].$$

From this, the expected number of hits to break each door is:

$$\text{ExpectedHits} = \frac{S_0}{\text{AvgDamage}} = \left[\frac{100}{5},\ \frac{100}{2},\ \frac{100}{8}\right] = [20,\ 50,\ 12.5].$$

Thus, Door 2 is chosen as the best candidate.

### Step 2: Exploitation Phase

The policy now focuses mainly on Door 2, since it has the lowest expected hits.

### Step 3: Threshold Rules

As the algorithm progresses:

- If any door's estimated remaining strength is $\leq 50$, the policy finishes it immediately.
- If the best door requires fewer than 40 estimated hits, it commits fully.
- Otherwise, it continues attacking the statistically best door.

### Outcome

The algorithm:

1. explores fairly at the start,
2. identifies Door 2 as the best choice,
3. commits to Door 2 early, ensuring efficient finishing.

This example demonstrates how the Student Policy transitions smoothly from exploration to exploitation using expected hits and finishing rules.

—

# 3 TASK 3

## 3.1 Idea

KL-UCB balances exploration and exploitation by using the KL-divergence between Bernoulli distributions to form a confidence bound.

## 3.2 Code Explanation

### Initialization.

```
def __init__(self, num_arms, horizon):
    super().__init__(num_arms, horizon)
    self.num_arms = num_arms
    self.counts = np.zeros(num_arms)
    self.rewards = np.zeros(num_arms)
    self.total_pulls = 0
    self.last_arm = None
    self.repeat_left = 0
```

### Finding Optimistic Estimate.

```
def _find_q(self, p, c, n, tol=1e-6, max_iter=25):
    lower, upper = p, 1.0
    for _ in range(max_iter):
        mid = (lower + upper) / 2
        if kl_bernoulli(p, mid) > c / n:
            upper = mid
        else:
            lower = mid
    return lower
```

### Choosing an Arm.

```
def give_pull(self):
    if self.repeat_left > 0:
        self.repeat_left -= 1
        return self.last_arm

    if self.total_pulls < self.num_arms:
        self.last_arm = self.total_pulls
        self.repeat_left = 0
        return self.last_arm

    ucb_values = np.zeros(self.num_arms)
    log_total_pulls = math.log(self.total_pulls) + \
                      3 * math.log(max(1.0001, math.log(
                          self.total_pulls)))

    for arm in range(self.num_arms):
        if self.counts[arm] == 0:
            ucb_values[arm] = float("inf")
        else:
            p_hat = self.rewards[arm] / self.counts[arm]
            ucb_values[arm] = self._find_q(p_hat,
                log_total_pulls, self.counts[arm])

    self.last_arm = int(np.argmax(ucb_values))
    self.repeat_left = int(math.sqrt(self.counts[self.
        last_arm]))
    return self.last_arm
```

### Updating Rewards.

```
def get_reward(self, arm_index, reward):
    self.counts[arm_index] += 1
    self.rewards[arm_index] += reward
```

```
    self.total_pulls += 1
```

—

### 3.3 Intution

The KL-UCB algorithm refines the idea of UCB by replacing the Hoeffding-based confidence interval with one derived from the Kullback–Leibler (KL) divergence.

For a Bernoulli arm with empirical mean $\hat{\mu}_i(t)$ after $N_i(t)$ pulls, the KL-UCB index is defined as:

$$q_i(t) = \max\Big\{q \in [\hat{\mu}_i(t), 1] : KL\big(\hat{\mu}_i(t), q\big) \leq \frac{\log t + c \log\log t}{N_i(t)}\Big\}.$$

At each round $t$, the algorithm selects the arm with the largest $q_i(t)$. This method achieves regret close to the Lai–Robbins lower bound, making it more efficient than standard UCB.

### 3.4 Practical Effectiveness

- Every arm is pulled initially, and KL-UCB confidence keeps uncertainty in check.
- Focuses more on arms with higher estimated rewards as pulls increase.