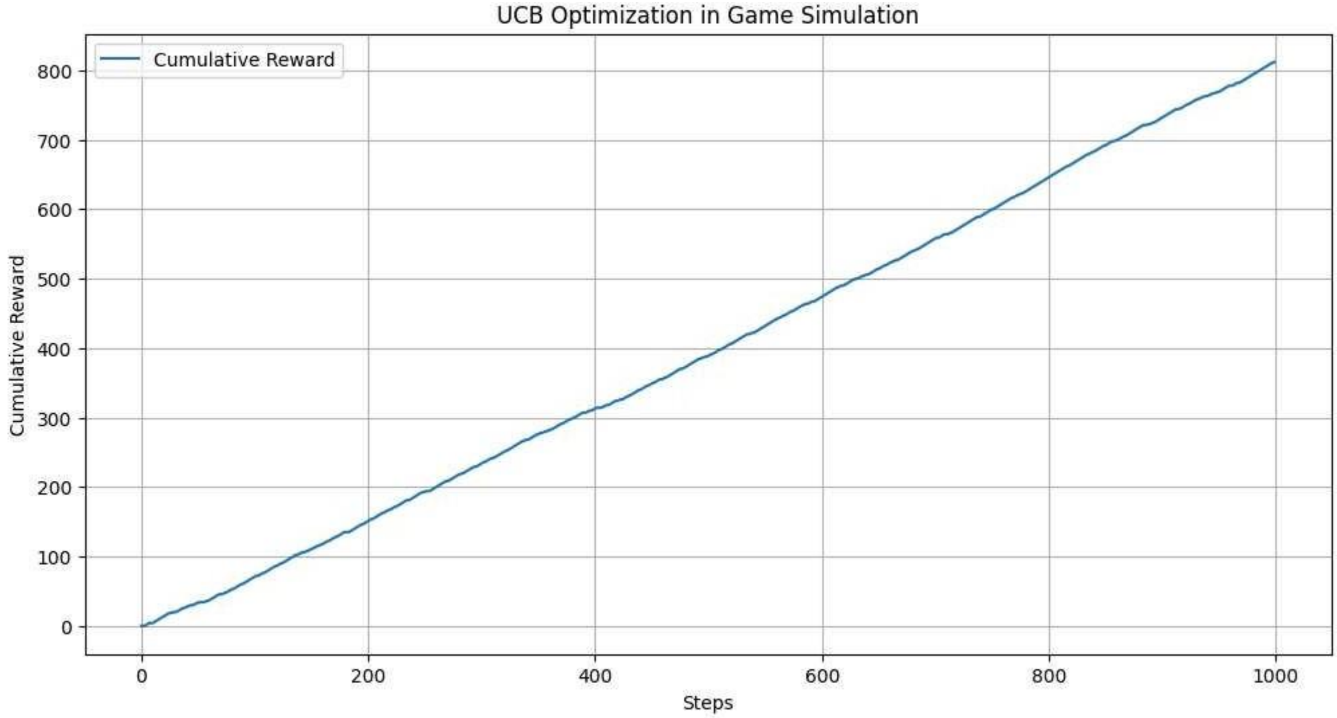
B. AKSHAJ
2211CS020071
AIML-ZETA

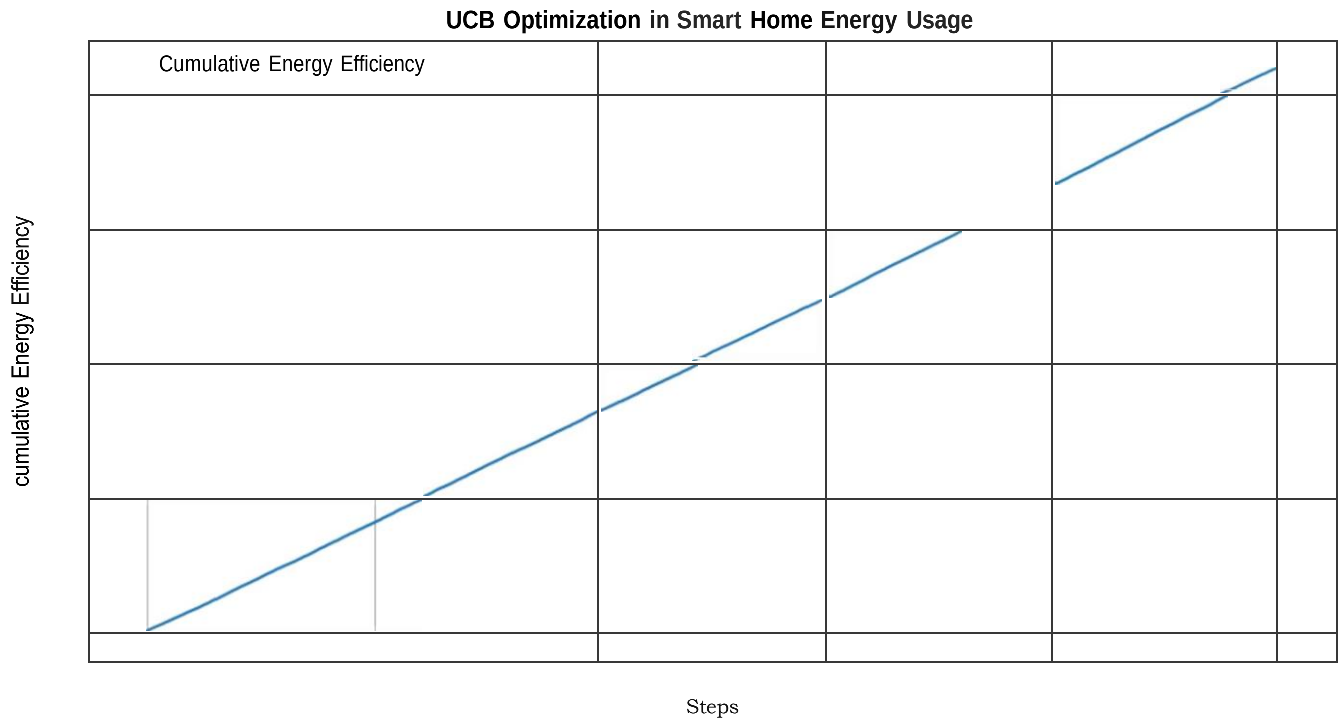
```
import numpy as np
import math
import matplotlib.pyplot as plt

# Define the UCB algorithm
class UCB:
    def finite(self, n_actions):  # Changed _init to finite
        self.n_actions - n_actions
        self.action_counts - np.zeros(n_actions)
        self.action_rewards = np.zeros(n_actions)

    # ... (rest of your UCB class code remains the same) ...

    def select_action(self, step):
        # Select action using UCB formula
        if step ‹ self.n_actions:
            return step  # Choose each action once initially

        ucb_values = [
            (self.action_rewards[i] / (self.action_counts[i] + 1e-5)) +  # Avoid division by zero
            math.sqrt(2 * math.log(step + 1) / (self.action_counts[i] + 1e-5))
            for i in range(self.n_actions)

        return np.argmax(ucb_values)

    def update(self, action, reward):
        # Update action counts and rewards
        self.action_counts[action] += 1
        self.action_rewards[action] +- reward

# Simulate the game
del simu1ate_game( n_steps, n_actions, true_reward_probs):
    ucb = UCB(n_act ions)
    total_reward = 0
    rewards = []

    for step in range(n_steps):
        action = ucb.select_action(step)
        # Simulate reward based on the true probability of the chosen action
        reward = 1 if np.random.rand() ‹ true_reward_probs[action] else 0
        ucb.update(action, reward)
        total_reward += reward
        rewards.append(total_reward)

    return rewards, ucb.action_counts

# Define par anetens
n_steps = 1000 # Total number of steps in the game
n_actions = 5  # Number of actions (e.g., doors, treasures, paths)
true_reward_probs = [0.1, 0.3, 0.5, 0.7, 0.9] P Hidden reward probabilities for each action

# Run the simulation
rewards, action_counts = simulate_game(n_steps, n_actions, true_reward_probs)

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(rewards, label-“Cumulative Reward“)
plt.xlabel(“Steps“)
plt.ylabel(“Cumulative Reward“)
plt.title(“UCB Optimization in Game Simulation")
plt.legend()
plt.grid(True)
plt.show()

# Print action counts
print(“Action counts:“, action_counts)
print(“True reward probabilities:“, true_reward_probs)
```




```
Action counts: [ 14.  24.  48. 147. 767.]
True reward probabilities: [0.1, 0.3, 0.5, 0.7, 0.9]
```

```
import numpy as np
import math
import matplotlib.pyplot as plt

# Define UCB algorithm for device modes
class UCB:
    def finite(self, n_modes): # Changed _init_ to finite
        self.n_modes = n_modes
        self.mode_counts = np.zeros(n_modes)  # Number of times each mode was chosen
        self.mode_rewards = np.zeros(n_modes)  # Sum of rewards for each mode

    def select_mode(self, step):
        # Use UCB to select the mode
        if step < self.n_modes:
            return step  # Choose each mode once initially
        ucb_values = [
            (self.mode_rewards[i] / (self.mode_counts[i] + 1e-5)) +
            math.sqrt(2 * math.log(step + 1) / (self.mode_counts[i] + ie-5))
            fon i in nange(self.n_modes)

        return np.argmax(ucb_values)

    def update(self, mode, reward):
        # Update mode counts and rewards
        self.mode_counts[mode] += I
        self.mode_rewards[mode] += reward

# Simulate the smart home energy optimization
def simulate_smart_home(n_steps, n_modes, true_efficieucy):
    ucb = UCB(n_modes)
    total_efficiency = 0
    efficiencies = []
    for step in range(n_steps):
        mode = ucb.select_mode(step)
        # Simulate energy efficiency based ou the true efficiency of the chosen mode
        efficiency = up.random.normal(loc=true_efficiency[mode], scale=0.1)
        ucb.update(mode, efficiency)
        total_efficiency += efficiency
        efficiencies.append(total_efficiency)
    return efficiencies, ucb.mode_couuts

# Define parameters
n_steps = 1000  # Number of steps in the simulation
n_modes = 4  # Number of energy modes (e.g., “low“, “medium“, “high“, “eco")
```

```
trve_efficiency = [0.6, 0.7, 0.8, 0.9)  # True average efficiency for each mode

# Run the simulation
efficiencies, mode_counts = simulate_smart_home(n_steps, n_modes, true_efficiency)

# Plot results
plt.figure(figsize=(12, 6))
pIt.plot(efficiencies, label="Cumulative Energy Efficiency")
pIt.xIabel("Steps")
pIt.yIabel("Cumulative Energy Efficiency")
plt.title("UCB Optimization in Smart Home Energy Usage")
plt.legend()
plt.grid(True)
plt.show()

# Print mode counts and true efficiencies
print("Mode counts:", mode_counts)
print("True mode efficiencies:", true_efficiency)
```




```
Mode counts: [ 68. 110. 203. 619.]
True mode efficiencies: [0.6, 0.7, 0.8, 0.9]
```

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import random

# Define the board and pieces
class ChessBoard:
    def finite(self): # Changed to  init
        self.board = np.zeros((4, 4), dtype=int)  # 4x4 grid
        self.king_pos = (3, 0)  # King's initial position
        self.pawn_positions = [(0, 1), (1, 3), (2, 2)] # Pawns' initial positions
        self.place_pieces()

    def place_pieces(self):
        self.board[self.king_pos] = 1  # King represented as 1
        for pos in self.pawn_positions:
            self.board[pos] = -1  # Pawns represented as -1

    def get_features(self):
        # Flatten board as feature vector
        return self.board.flatten()

    def move_king(self, new_pos):
        x, y = self.king_pos
        self.board[x, y] = 0 # Clear old king position
        self.king_pos = new_pos
```

```
        x, y = new_pos
        self.board[x, y] = 1  # Set new king position

    def is_valid_move(self, pos):
        x, y = pos
        return 0 <- x < 4 and 0 <- y < 4 and self.board[x, y] != 1  # Inside bounds and not the king's current position

    def generate_king_moves(self):
        x, y = self.king_pos
        moves = [(x + i, y + j) for i in [-1, 0, 1] for j in [-1, 0, 1] if (i, j) != (0, 0)1
        return [move for move in moves if self.is_valid_move(move))

# Reward function
def reward_function(board, move):
    x, y = move
    if board[x, y] == -1:  # Capture a pawn
        return 10
    else:  # Move to an empty space
        return 1

# Generate training data
def generate_training_data(n_samples):

    y = []
    for _ in range(n_samples):
        board = ChessBoard()
        moves = boand.generate_ki ng_moves()
        optimal_move - None
        max_reward = -np.inf
        for move in moves:
            reward = reward_function(board.board, move)
            if reward > max_reward:
                max_reward - reward
                optimal_move - move
        X.append(board.get_features())
        y.append(optimal_move)
    return np.array(X), np.array(y)

# Train a PAC model (decision tree classifier)
def train_pac_model(X, y):
    # Flatten move labels for multi-output classification
    y_flat = [x * 4 + y for x, y in y]
    model = DecisionTreeClassifier(max_depth=5)
    model.fit(X, y flat)
    return model

# Predict a move
def predict_move(model, board):
    move_flat = model.predict([board.get_features()])[0J
    return divmod(move_flat, 4)

# Simulate a game
def simulate_game(model, n_steps-10):
    board = ChessBoard()
    for step in range(n_steps):
        print(f“Step {step + 1}:")
        print(board.board)
        move = predict_move(model, board)
        print(f“King moves to: {move}“)
        board.move_king(move)

# Main function
if manes == "Umaink": If Changed _name_ to enamel and "_ma1n_" to "dnalng"
    # Generate training data
    X, y = generate_training_data(500)

    If Tra in  PAC model
    model = I ra1n_pac_mode1(X,  y)

    # Evaluate model accuracy
    y_flat = [x * 4 + y for x, y in y]
    predictions = model.predict(X)
    print(f“Model Accuracy: {accuracy_score(y_flat, predictions) * 100:.2f}%“)

    # Simulate a game
    simulate_game(model)
```

```
Model Accuracy: 100.00%
Step 1:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 0  0 -1  0]
 [ 1  0  0  0]]
King moves to: (2, 0)
Step 2:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -1  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 3:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -l  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 4:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -1  0]
 [ 0  0  0  0]]
thing moves to: (2, 8)
Step 5:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -l  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 6:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -1  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 7:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -l  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 8:
[[ 0 -1  0  0]
 [ 0  0  0 -1]
 [ 1  0 -1  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 9:
[[ 0 -1  0  0]
 [ 0  0  0 -I]
 [ 1  0 -l  0]
 [ 0  0  0  0]]
King moves to: (2, 0)
Step 10:
[[ 0 -1  0  0]
 [ 0  0  0 -1]
```