```
#CHESS.PY
import random
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

def generate_chess_data(n_samples=1000):
    data = []
    labels = []
    for _ in range(n_samples):
        board_state = [random.randint(0, 1) for _ in range(16)]
        optimal_move = random.randint(0, 15)
        data.append(board_state)
        labels.append(optimal_move)
    return data, labels


def train_pac_model():
    data, labels = generate_chess_data()
    X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

    model = DecisionTreeClassifier(max_depth=5)
    model.fit(X_train, y_train)

    predictions = model.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    print(f"PAC Model Accuracy: {accuracy * 100:.2f}%")

train_pac_model()
```

```
PAC Model Accuracy: 7.50%
```

```
 #UCB.PY
import math
import random

class MultiArmedBandit:
    def __init__(self, probabilities):
        self.probabilities = probabilities
        self.n_actions = len(probabilities)
        self.counts = [0] * self.n_actions
        self.values = [0.0] * self.n_actions

    def select_action(self):
        total_counts = sum(self.counts)
        if total_counts < self.n_actions:
            return total_counts
        ucb_values = [
            self.values[i] + math.sqrt(2 * math.log(total_counts) / self.counts[i])
            for i in range(self.n_actions)
        ]
        return ucb_values.index(max(ucb_values))

    def update(self, action, reward):
        self.counts[action] += 1
        n = self.counts[action]
        self.values[action] += (reward - self.values[action]) / n


def simulate_game():
    probabilities = [0.1, 0.5, 0.8]
    bandit = MultiArmedBandit(probabilities)
    total_reward = 0
    n_rounds = 100

    for _ in range(n_rounds):
        action = bandit.select_action()
        reward = 1 if random.random() < probabilities[action] else 0
        bandit.update(action, reward)
        total_reward += reward

    print(f"Total reward after {n_rounds} rounds: {total_reward}")
    print(f"Estimated values: {bandit.values}")

simulate_game()
```

```
Total reward after 100 rounds: 66
Estimated values: [0.11111111111111113, 0.5555555555555556, 0.78125]
```

```
#SMARTHOME
import math
import random

class SmartHomeUCB:
    def __init__(self, efficiency_probabilities):
        self.probabilities = efficiency_probabilities
        self.n_devices = len(efficiency_probabilities)
        self.counts = [0] * self.n_devices
        self.values = [0.0] * self.n_devices

    def select_device(self):
        total_counts = sum(self.counts)
        if total_counts < self.n_devices:
            return total_counts

        ucb_values = [
            self.values[i] + math.sqrt(2 * math.log(total_counts) / self.counts[i])
            for i in range(self.n_devices)
        ]
        return ucb_values.index(max(ucb_values))

    def update(self, device, reward):
        self.counts[device] += 1
        n = self.counts[device]
        self.values[device] += (reward - self.values[device]) / n

def simulate_smart_home():
    efficiency_probabilities = [0.7, 0.6, 0.8]
    smart_home = SmartHomeUCB(efficiency_probabilities)
    total_reward = 0
    n_steps = 100

    for _ in range(n_steps):
        device = smart_home.select_device()
        reward = 1 if random.random() < efficiency_probabilities[device] else 0
        smart_home.update(device, reward)
        total_reward += reward

    print(f"Total efficiency after {n_steps} steps: {total_reward}")
    print(f"Estimated efficiencies: {smart_home.values}")

simulate_smart_home()
```

```
Total efficiency after 100 steps: 74
Estimated efficiencies: [0.7567567567567569, 0.6250000000000001, 0.794871794871795]
```