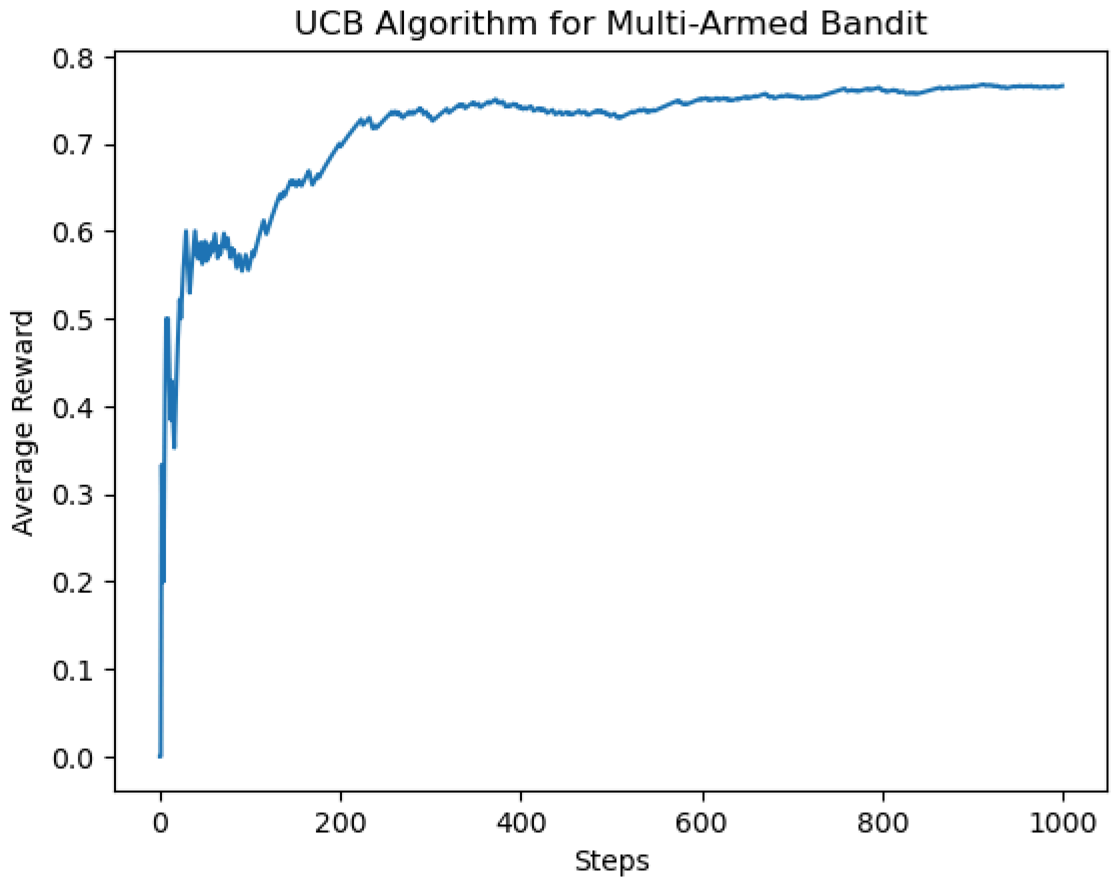
1.Implement Upper-Confience bound algorithm (UCB) in Multi Arm Banding Problem to optimize player rewards in a basic game simulation with Python Program.The game scenario involves a player choosing between different "actions" (like doors, treasures, or paths), each with a hidden reward probability. The UCB algorithm must help the game adapt dynamically to maximize the player's experience.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

class MultiArmedBandit:
    def __init__(self, num_arms, true_probs):
        self.num_arms = num_arms
        self.true_probs = true_probs
        self.estimates = np.zeros(num_arms)
        self.counts = np.zeros(num_arms)
        self.total_rewards = 0

    def select_arm(self):
        ucb_values = self.estimates + np.sqrt(2 * np.log(self.total_rewards + 1) / (self.counts + 1e-5))
        return np.argmax(ucb_values)

    def update(self, arm, reward):
        self.counts[arm] += 1
        self.total_rewards += reward
        self.estimates[arm] += (reward - self.estimates[arm]) / self.counts[arm]

    def simulate(self, num_steps):
        rewards = np.zeros(num_steps)
        for step in range(num_steps):
            arm = self.select_arm()
            reward = np.random.binomial(1, self.true_probs[arm])
            self.update(arm, reward)
            rewards[step] = reward
        return rewards

# Example setup with 3 arms and hidden probabilities for each arm
true_probs = [0.3, 0.5, 0.8]
bandit = MultiArmedBandit(num_arms=3, true_probs=true_probs)
num_steps = 1000
rewards = bandit.simulate(num_steps)
```

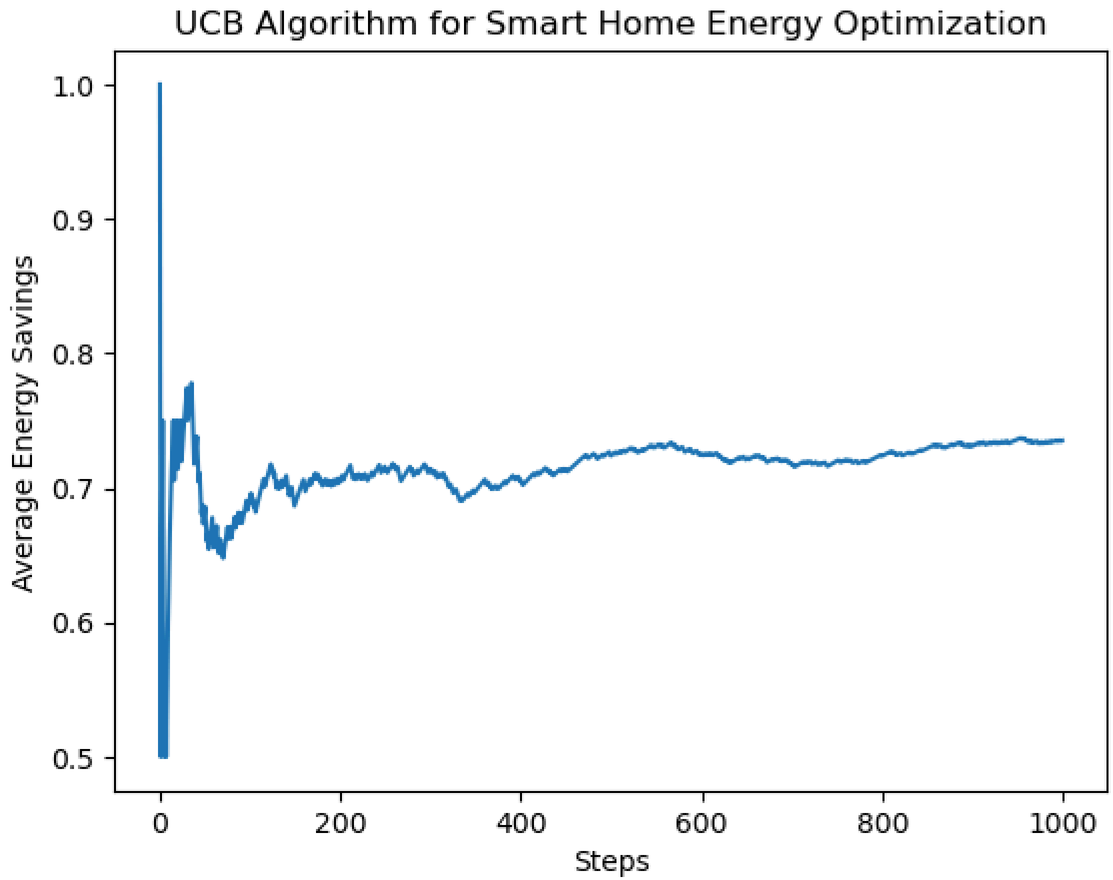
```
# Plotting the rewards over time
plt.plot(np.cumsum(rewards) / (np.arange(num_steps) + 1))
plt.xlabel('Steps')
plt.ylabel('Average Reward')
plt.title('UCB Algorithm for Multi-Armed Bandit')
plt.show()
```



2.Imagine an IoT-based smart home system that dynamically optimizes energy usage across multiple devices (e.g., air conditioner, heater, and lights). Each device has a varying energy consumption efficiency based on real-time environmental factors like temperature or occupancy. Design an UCB algorithm is used to determine which device settings (e.g., energy modes) should be prioritized to maximize energy efficiency and implement the algorithm in Python

In [2]:
```
import numpy as np

class SmartHomeOptimizer:
    def __init__(self, num_devices, efficiency_probs):
        self.num_devices = num_devices
        self.efficiency_probs = efficiency_probs
        self.estimates = np.zeros(num_devices)
        self.counts = np.zeros(num_devices)
        self.total_energy_usage = 0

    def select_device(self):
        ucb_values = self.estimates + np.sqrt(2 * np.log(self.total_energy_usage + 1) / (self.counts + 1e-5))
        return np.argmax(ucb_values)

    def update(self, device, energy_saved):
        self.counts[device] += 1
        self.total_energy_usage += energy_saved
        self.estimates[device] += (energy_saved - self.estimates[device]) / self.counts[device]

    def simulate(self, num_steps):
        energy_savings = np.zeros(num_steps)
        for step in range(num_steps):
            device = self.select_device()
            # Simulating dynamic energy savings based on real-time environment (e.g., temperature, occupancy)
            energy_saved = np.random.binomial(1, self.efficiency_probs[device])
            self.update(device, energy_saved)
            energy_savings[step] = energy_saved
        return energy_savings

# Example setup with 3 devices (AC, Heater, and Lights) and their respective energy-saving probabilities
efficiency_probs = [0.7, 0.5, 0.8]
optimizer = SmartHomeOptimizer(num_devices=3, efficiency_probs=efficiency_probs)
num_steps = 1000
energy_savings = optimizer.simulate(num_steps)
```

```
# Display energy savings over time
import matplotlib.pyplot as plt
plt.plot(np.cumsum(energy_savings) / (np.arange(num_steps) + 1))
plt.xlabel('Steps')
plt.ylabel('Average Energy Savings')
plt.title('UCB Algorithm for Smart Home Energy Optimization')
plt.show()
```



3.Develop a Chess-like game using PAC(Probably approximately correct) algorithm where the Problem set-up is as follows: Problem Setup i)Game Environment: Simplify chess to a smaller grid with basic pieces (like pawns and a king). ii)PAC Learning: Train a model to approximate a move policy that is "probably approximately correct" (i.e., likely correct within some error bounds). iii)Implementation Goals: Use supervised learning to train a model with a dataset of board states and corresponding optimal moves. Implementation: 1.The chess-like game will have a simplified 4x4 board with only a king and a few pawns. 2.PAC learning will train a simple classifier (e.g., decision tree) to predict moves.

In [3]:
```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Example board states and corresponding moves (simplified for illustration)
# Board state is represented as a flattened 4x4 grid, with 0 for empty spaces, 1 for pawns, and 2 for the king
# Optimal moves are represented as the index of the move (up, down, left, right, etc.)
board_states = [
    [0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0],
    # Add more board states here
]
optimal_moves = [1, 0]  # Corresponding optimal moves for each board state

# Flatten the board states and train the classifier
X = np.array(board_states)  # Feature set
y = np.array(optimal_moves)  # Labels (optimal moves)

# Train a decision tree to predict moves
model = DecisionTreeClassifier()
model.fit(X, y)

# Now, predict the move for a new board state
new_board_state = [0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0]  # A new board state
predicted_move = model.predict([new_board_state])

# Output the predicted move
print(f"The predicted move is: {predicted_move[0]}")
```

```
The predicted move is: 0
```