```
---------------Mandatory Information to fill-----------
```

Group ID:

Group Members Name with Student ID:

1. (Siddhartha Sandilya) - 2023AA05705
2. (Prabir Sinha)- 2023AA05789
3. (Ravi Ranjan Pandey) - 2023AA05002
4. (Ajay Nayak) -2023AA05482

```
------------------Write your remarks (if any) that you want should get consider at the time of evaluation--------------
```

Remarks: ##Add here

Background

In digital advertising, Click-Through Rate (CTR) is a critical metric that measures the effectiveness of an advertisement. It is calculated as the ratio of users who click on an ad to the number of users who view the ad. A higher CTR indicates more successful engagement with the audience, which can lead to increased conversions and revenue. From time-to-time advertisers experiment with various elements/targeting of an ad to optimise the ROI.

Scenario

Imagine an innovative digital advertising agency, AdMasters Inc., that specializes in maximizing click-through rates (CTR) for their clients' advertisements. One of their clients has identified four key tunable elements in their ads: *Age*, *City*, *Gender*, and *Mobile Operating System (OS)*. These elements significantly influence user engagement and conversion rates. The client is keen to optimize their CTR while minimizing resource expenditure.

Objective

Optimize the CTR of digital ads by employing Multi Arm Bandit algorithms. System should dynamically and efficiently allocate ad displays to maximize overall CTR.

Dataset

The dataset for Ads contains 4 unique features/characteristics.

- Age (Range: 25:50)
- City (Possible Values: 'New York', 'Los Angeles', 'Chicago','Houston', 'Phoenix')
- Gender (Possible Values: 'Male', 'Female')
- OS: (Possible Values: 'iOS', 'Android', 'Other')

Link for accessing dataset:
https://drive.google.com/file/d/1Y5HmEeoQsafo9Diy9piS69qEMnC0g1ys/view?usp=sharing

Environment Details

Arms: Each arm represents a different ad from the dataset.

Reward Function:

- Probability of a Male clicking on an Ad -> 0.7 (randomly generated)
- Probability of a Female clicking on an Ad -> 0.6 (randomly generated)
- Once probabilities are assigned to all the values, create a final reward (clicked or not clicked binary outcome) based on the assumed probabilities in step 1 (by combining the probabilities of each feature value present in that ad)

Assumptions

- Assume alpha = beta = 1 for cold start
- Explore Percentage = 10%
- Run the simulation for min 1000 iterations

Requirements and Deliverables:

Implement the Multi-Arm Bandit Problem for the given above scenario for all the below mentioned policy methods.

Initialize constants

In [1]:
```
# Constants
epsilon = 0.1
NUM_USERS = 100
NUM_ADS = 10
NUM_ITERATIONS = 1000
EXPLORE_PERCENTAGE = 0.1
# Initialize value function and policy
import numpy as np
import pandas as pd
import itertools
from sklearn.preprocessing import LabelEncoder
```

Load Dataset

In [2]:
```
# Code for Dataset Loading and print dataset statistics
#-----write your code below this line---------
data = pd.read_csv(r"D:\Bits Pilani Sem 2\Deep Reinforcement learning\Assignment
data
```

Out[2]:

	Age	Gender	City	Phone_OS
0	25	Male	New York	iOS
1	25	Male	New York	Android
2	25	Male	New York	Other
3	25	Male	Los Angeles	iOS
4	25	Male	Los Angeles	Android
...	...	...	...	...
775	50	Female	Houston	Android
776	50	Female	Houston	Other
777	50	Female	Phoenix	iOS
778	50	Female	Phoenix	Android
779	50	Female	Phoenix	Other

780 rows × 4 columns

In [3]:
```
# Encode the categorical variables
label_encoders = {}
for column in data.columns:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le

# Define the parameters
age_groups = data['Age'].unique()
cities = data['City'].unique()
genders = data['Gender'].unique()
mobile_os = data['Phone_OS'].unique()

# Generate all possible combinations of the parameters
arms = list(itertools.product(age_groups, cities, genders, mobile_os))
n_arms = len(arms)
```

In [4]:
```
# User and ad features
user_features = {
    'age': np.random.randint(25, 51, NUM_USERS),
    'city': np.random.choice(['New York', 'Los Angeles', 'Chicago', 'Houston', '
    'gender': np.random.choice(['Male', 'Female'], NUM_USERS),
    'os': np.random.choice(['iOS', 'Android', 'Other'], NUM_USERS)
}
```

```
ads_features = {
    'age': np.random.randint(25, 51, NUM_ADS),
    'city': np.random.choice(['New York', 'Los Angeles', 'Chicago', 'Houston', '
    'gender': np.random.choice(['Male', 'Female'], NUM_ADS),
    'os': np.random.choice(['iOS', 'Android', 'Other'], NUM_ADS)
}

users = pd.DataFrame(user_features)
ads = pd.DataFrame(ads_features)
```

Design a CTR Environment (1M)

In [5]:

```
import numpy as np
import pandas as pd

# Load the CSV file
file_path = 'Downloads/AD_Click.csv'
ad_click_data = pd.read_csv(file_path)

# Define probabilities based on gender
click_probabilities = {
    'Male': 0.7,
    'Female': 0.6
}

# Assign click probabilities based on other features as well
# For simplicity, we'll assign random probabilities for other features
np.random.seed(42)
age_probabilities = {age: np.random.uniform(0.4, 0.8) for age in ad_click_data['
city_probabilities = {city: np.random.uniform(0.4, 0.8) for city in ad_click_dat
os_probabilities = {os: np.random.uniform(0.4, 0.8) for os in ad_click_data['Pho

def get_click_probability(row):
    gender_prob = click_probabilities[row['Gender']]
    age_prob = age_probabilities[row['Age']]
    city_prob = city_probabilities[row['City']]
    os_prob = os_probabilities[row['Phone_OS']]

    # Combine the probabilities (e.g., average)
    combined_prob = (gender_prob + age_prob + city_prob + os_prob) / 4
    return combined_prob

# Add click probability to each row in the dataset
ad_click_data['Click_Probability'] = ad_click_data.apply(get_click_probability,

# Define a function to simulate click based on the combined probability
def simulate_click(probability):
    return np.random.rand() < probability

# Add click outcome to each row in the dataset
ad_click_data['Clicked'] = ad_click_data['Click_Probability'].apply(simulate_cli

ad_click_data.head()
```

Out[5]:

	Age	Gender	City	Phone_OS	Click_Probability	Clicked
0	25	Male	New York	iOS	0.549474	False
1	25	Male	New York	Android	0.538927	False
2	25	Male	New York	Other	0.627310	True
3	25	Male	Los Angeles	iOS	0.580930	True
4	25	Male	Los Angeles	Android	0.570383	False

Using Random Policy (0.5M)

Print all the iterations with random policy selected for the given Ad. (Mandatory)

In [6]:
```
import numpy as np
import pandas as pd
import random

# Load the dataset
file_path = 'Downloads/AD_Click.csv'
ad_click_data = pd.read_csv(file_path)

# Define the CTR Environment
class CTR_Environment:
    def __init__(self, data, click_probabilities):
        self.data = data
        self.click_probabilities = click_probabilities

    def calculate_click_probability(self, user):
        return self.click_probabilities[user['Gender']]

    def simulate_click(self, user):
        prob = self.calculate_click_probability(user)
        return np.random.rand() < prob

    def get_user(self):
        user_idx = random.randint(0, len(self.data) - 1)
        user = self.data.iloc[user_idx]
        return user

click_probabilities = {'Male': 0.7, 'Female': 0.6}
env = CTR_Environment(ad_click_data, click_probabilities)

# Define the Random Policy
class RandomPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        return random.randint(0, self.n_arms - 1)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
```

```
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize Random Policy with 3 arms (ads)
n_arms = 3
random_policy = RandomPolicy(n_arms)

# Run the simulation for 1000 iterations
iterations = 1000
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = random_policy.select_arm()
    reward = env.simulate_click(user)
    random_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame and display
results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gender', 'City'
results_df
```

Out[6]:

	Iteration	Age	Gender	City	Phone_OS	Chosen_Arm	Reward
0	0	43	Male	Los Angeles	Android	2	False
1	1	30	Male	Chicago	iOS	2	True
2	2	40	Male	Chicago	iOS	2	True
3	3	38	Male	Chicago	Other	0	True
4	4	43	Male	Houston	Other	1	True
...	...	...	...	...	...	...	...
995	995	45	Female	New York	iOS	1	True
996	996	27	Female	Houston	Android	1	False
997	997	26	Male	Chicago	iOS	1	False
998	998	39	Female	Chicago	iOS	1	True
999	999	38	Female	Los Angeles	iOS	1	True

1000 rows × 7 columns

Using Greedy Policy (0.5M)

Print all the iterations with random policy selected for the given Ad. (Mandatory)

In [7]:

```
import numpy as np
import pandas as pd
import random

# Load the dataset
file_path = 'Downloads/AD_Click.csv'
```

```
ad_click_data = pd.read_csv(file_path)

# Define the CTR Environment
class CTR_Environment:
    def __init__(self, data, click_probabilities):
        self.data = data
        self.click_probabilities = click_probabilities

    def calculate_click_probability(self, user):
        return self.click_probabilities[user['Gender']]

    def simulate_click(self, user):
        prob = self.calculate_click_probability(user)
        return np.random.rand() < prob

    def get_user(self):
        user_idx = random.randint(0, len(self.data) - 1)
        user = self.data.iloc[user_idx]
        return user

click_probabilities = {'Male': 0.7, 'Female': 0.6}
env = CTR_Environment(ad_click_data, click_probabilities)
class GreedyPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        return np.argmax(self.values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize Greedy Policy with 3 arms (ads)
n_arms = 3
greedy_policy = GreedyPolicy(n_arms)
# Run the simulation for 1000 iterations
iterations = 1000
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = greedy_policy.select_arm()
    reward = env.simulate_click(user)
    greedy_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame and display
results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gender', 'City'
results_df
```

Out[7]:

	Iteration	Age	Gender	City	Phone_OS	Chosen_Arm	Reward
0	0	46	Female	Chicago	iOS	0	True
1	1	38	Female	Phoenix	iOS	0	False
2	2	25	Male	Houston	Android	0	True
3	3	32	Female	Phoenix	iOS	0	False
4	4	25	Male	Los Angeles	Other	0	True
...	...	...	...	...	...	...	...
995	995	50	Male	Los Angeles	iOS	0	False
996	996	47	Female	Chicago	Other	0	True
997	997	48	Female	Phoenix	Android	0	False
998	998	47	Male	Los Angeles	iOS	0	True
999	999	34	Male	Houston	iOS	0	True

1000 rows × 7 columns

Using Epsilon-Greedy Policy (0.5M)

Print all the iterations with random policy selected for the given Ad. (Mandatory)

In [8]:

```
import numpy as np
import pandas as pd
import random

# Load the dataset
file_path = 'Downloads/AD_Click.csv'
ad_click_data = pd.read_csv(file_path)

# Define the CTR Environment
class CTR_Environment:
    def __init__(self, data, click_probabilities):
        self.data = data
        self.click_probabilities = click_probabilities

    def calculate_click_probability(self, user):
        return self.click_probabilities[user['Gender']]

    def simulate_click(self, user):
        prob = self.calculate_click_probability(user)
        return np.random.rand() < prob

    def get_user(self):
        user_idx = random.randint(0, len(self.data) - 1)
        user = self.data.iloc[user_idx]
        return user

click_probabilities = {'Male': 0.7, 'Female': 0.6}
env = CTR_Environment(ad_click_data, click_probabilities)
```

```
class EpsilonGreedyPolicy:
    def __init__(self, n_arms, epsilon):
        self.n_arms = n_arms
        self.epsilon = epsilon
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        if random.random() > self.epsilon:
            return np.argmax(self.values)
        else:
            return random.randint(0, self.n_arms - 1)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize Epsilon-Greedy Policy with 3 arms (ads) and epsilon = 0.1
n_arms = 3
epsilon = 0.1
epsilon_greedy_policy = EpsilonGreedyPolicy(n_arms, epsilon)

# Run the simulation for 1000 iterations
iterations = 1000
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = epsilon_greedy_policy.select_arm()
    reward = env.simulate_click(user)
    epsilon_greedy_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame and display
results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gender', 'City'
results_df
```

Out[8]:

	Iteration	Age	Gender	City	Phone_OS	Chosen_Arm	Reward
0	0	49	Male	Phoenix	Android	0	True
1	1	41	Male	Phoenix	iOS	0	False
2	2	41	Male	Los Angeles	Other	0	True
3	3	30	Female	Houston	Android	0	True
4	4	48	Female	New York	Other	0	True
...	...	...	...	...	...	...	...
995	995	47	Male	Phoenix	Other	1	True
996	996	48	Male	Houston	Android	1	True
997	997	42	Female	Houston	Android	1	True
998	998	33	Female	Chicago	Other	1	True
999	999	28	Female	New York	Android	1	False

1000 rows × 7 columns

Using UCB (0.5M)

Print all the iterations with random policy selected for the given Ad. (Mandatory)

In [9]:

```
import math

class UCBPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        total_counts = sum(self.counts)
        if total_counts < self.n_arms:
            return total_counts
        ucb_values = [0.0] * self.n_arms
        for arm in range(self.n_arms):
            bonus = math.sqrt((2 * math.log(total_counts)) / float(self.counts[a
            ucb_values[arm] = self.values[arm] + bonus
        return np.argmax(ucb_values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize UCB Policy with 3 arms (ads)
n_arms = 3
ucb_policy = UCBPolicy(n_arms)
# Run the simulation for 1000 iterations
```

```
iterations = 1000
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = ucb_policy.select_arm()
    reward = env.simulate_click(user)
    ucb_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame and display
results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gender', 'City'
results_df
```

Out[9]:

	Iteration	Age	Gender	City	Phone_OS	Chosen_Arm	Reward
0	0	31	Male	Houston	Other	0	True
1	1	26	Male	Houston	Android	1	True
2	2	30	Male	Los Angeles	Other	2	True
3	3	29	Female	Los Angeles	Other	0	False
4	4	30	Female	Los Angeles	Other	1	False
...	...	...	...	...	...	...	...
995	995	39	Male	Los Angeles	Other	2	False
996	996	26	Male	Chicago	iOS	0	True
997	997	44	Male	Chicago	iOS	0	False
998	998	28	Female	Phoenix	Other	1	True
999	999	27	Male	Phoenix	iOS	1	True

1000 rows × 7 columns

Plot CTR distribution for all the appraoches as a spearate graph (0.5M)

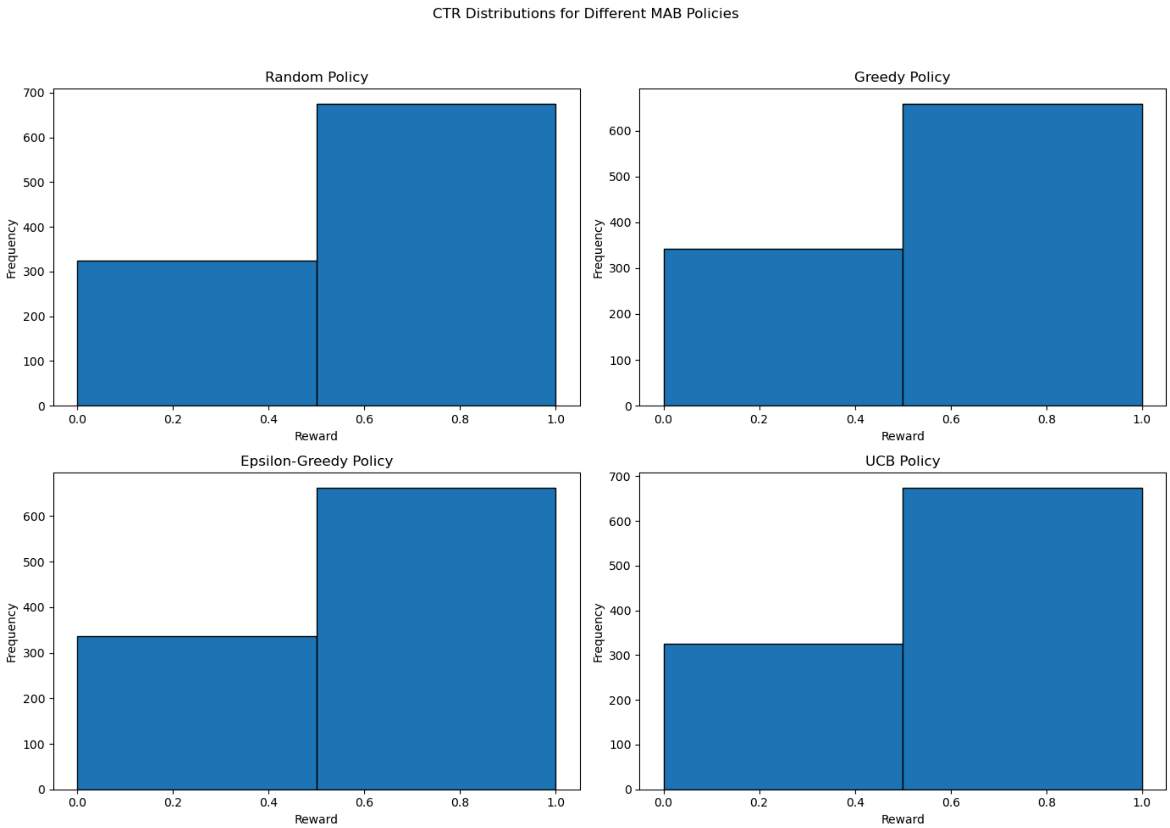
In [10]:

```
import matplotlib.pyplot as plt

# Calculate the CTR for each approach
def calculate_ctr(results_df):
    clicks = results_df['Reward'].sum()
    total = results_df.shape[0]
    return clicks / total

# Random Policy
random_policy_results_df = results_df

# Greedy Policy
# Define the Greedy Policy
class GreedyPolicy:
    def __init__(self, n_arms):
```

```
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        return np.argmax(self.values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize Greedy Policy with 3 arms (ads)
n_arms = 3
greedy_policy = GreedyPolicy(n_arms)

# Run the simulation for 1000 iterations
iterations = 1000
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = greedy_policy.select_arm()
    reward = env.simulate_click(user)
    greedy_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame
greedy_policy_results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'G

# Epsilon-Greedy Policy
# Define the Epsilon-Greedy Policy
class EpsilonGreedyPolicy:
    def __init__(self, n_arms, epsilon):
        self.n_arms = n_arms
        self.epsilon = epsilon
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        if random.random() > self.epsilon:
            return np.argmax(self.values)
        else:
            return random.randint(0, self.n_arms - 1)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize Epsilon-Greedy Policy with 3 arms (ads) and epsilon = 0.1
n_arms = 3
epsilon = 0.1
epsilon_greedy_policy = EpsilonGreedyPolicy(n_arms, epsilon)

# Run the simulation for 1000 iterations
results = []
```

```
for i in range(iterations):
    user = env.get_user()
    chosen_arm = epsilon_greedy_policy.select_arm()
    reward = env.simulate_click(user)
    epsilon_greedy_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame
epsilon_greedy_policy_results_df = pd.DataFrame(results, columns=['Iteration', '

# UCB Policy
# Define the UCB Policy
class UCBPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        total_counts = sum(self.counts)
        if total_counts < self.n_arms:
            return total_counts
        ucb_values = [0.0] * self.n_arms
        for arm in range(self.n_arms):
            bonus = math.sqrt((2 * math.log(total_counts)) / float(self.counts[a
            ucb_values[arm] = self.values[arm] + bonus
        return np.argmax(ucb_values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

# Initialize UCB Policy with 3 arms (ads)
n_arms = 3
ucb_policy = UCBPolicy(n_arms)

# Run the simulation for 1000 iterations
results = []

for i in range(iterations):
    user = env.get_user()
    chosen_arm = ucb_policy.select_arm()
    reward = env.simulate_click(user)
    ucb_policy.update(chosen_arm, reward)
    results.append((i, user['Age'], user['Gender'], user['City'], user['Phone_OS

# Convert results to a DataFrame
ucb_policy_results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gend

# Convert boolean values to integers
random_policy_results_df['Reward'] = random_policy_results_df['Reward'].astype(i
greedy_policy_results_df['Reward'] = greedy_policy_results_df['Reward'].astype(i
epsilon_greedy_policy_results_df['Reward'] = epsilon_greedy_policy_results_df['R
ucb_policy_results_df['Reward'] = ucb_policy_results_df['Reward'].astype(int)

# Plot the CTR distributions
fig, ax = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle('CTR Distributions for Different MAB Policies')
```

```
# Plot Random Policy CTR
ax[0, 0].hist(random_policy_results_df['Reward'], bins=2, edgecolor='black')
ax[0, 0].set_title('Random Policy')
ax[0, 0].set_xlabel('Reward')
ax[0, 0].set_ylabel('Frequency')

# Plot Greedy Policy CTR
ax[0, 1].hist(greedy_policy_results_df['Reward'], bins=2, edgecolor='black')
ax[0, 1].set_title('Greedy Policy')
ax[0, 1].set_xlabel('Reward')
ax[0, 1].set_ylabel('Frequency')

# Plot Epsilon-Greedy Policy CTR
ax[1, 0].hist(epsilon_greedy_policy_results_df['Reward'], bins=2, edgecolor='bla
ax[1, 0].set_title('Epsilon-Greedy Policy')
ax[1, 0].set_xlabel('Reward')
ax[1, 0].set_ylabel('Frequency')

# Plot UCB Policy CTR
ax[1, 1].hist(ucb_policy_results_df['Reward'], bins=2, edgecolor='black')
ax[1, 1].set_title('UCB Policy')
ax[1, 1].set_xlabel('Reward')
ax[1, 1].set_ylabel('Frequency')

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
 # &#8203;:citation[oaicite:0]{index=0}&#8203;
```



Changing Exploration Percentage (1M)

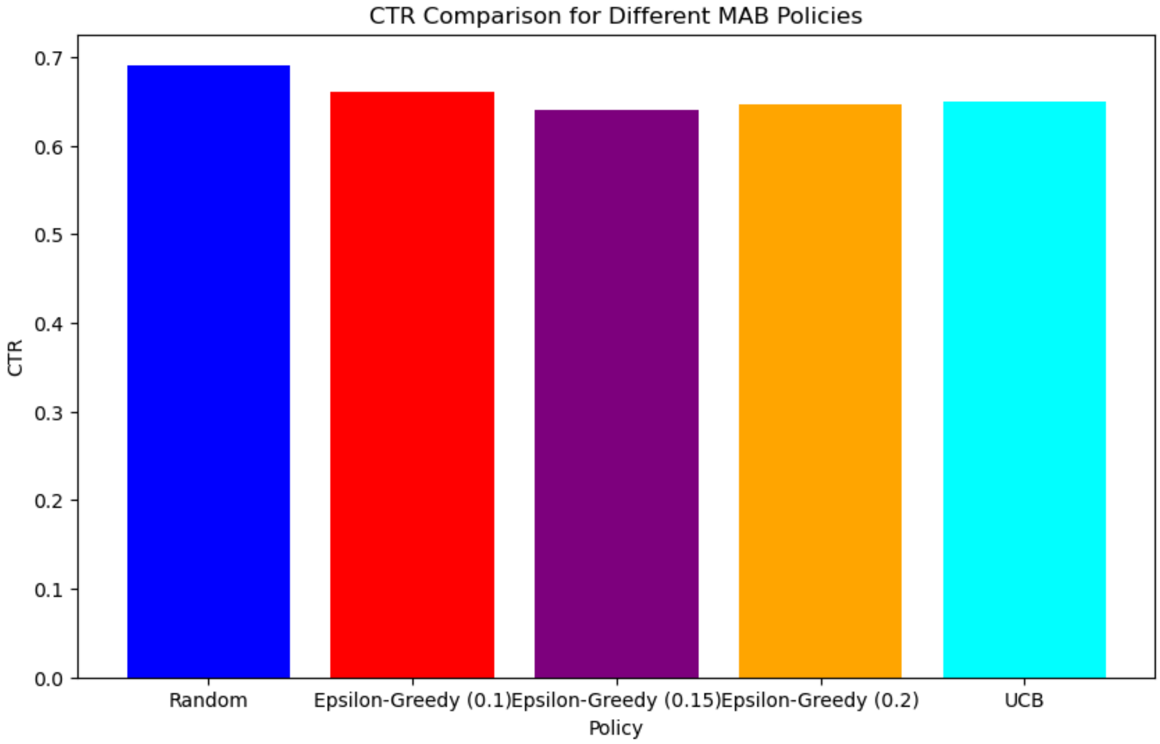
- How does changing the exploration percentage (EXPLORE_PERCENTAGE) affect the performance of the algorithm? Test with different values (e.g. 0.15 and 0.2) and discuss the results.

In [11]:
```
import pandas as pd
import random

# Load the dataset
file_path = 'Downloads/AD_Click.csv'
try:
    ad_click_data = pd.read_csv(file_path)
    print("Dataset loaded successfully.")
    print(ad_click_data.head())
except Exception as e:
    print(f"An error occurred: {e}")
import numpy as np
import math

# Define click probabilities for the dataset
click_probabilities = {'Male': 0.7, 'Female': 0.6}

# Define CTR Environment
class CTR_Environment:
    def __init__(self, data, click_probabilities):
        self.data = data
        self.click_probabilities = click_probabilities

    def calculate_click_probability(self, user):
        return self.click_probabilities[user['Gender']]

    def simulate_click(self, user):
        prob = self.calculate_click_probability(user)
        return np.random.rand() < prob

    def get_user(self):
        user_idx = random.randint(0, len(self.data) - 1)
        user = self.data.iloc[user_idx]
        return user

# Initialize environment
env = CTR_Environment(ad_click_data, click_probabilities)

# Define the Policies
class RandomPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms

    def select_arm(self):
        return random.randint(0, self.n_arms - 1)

    def update(self, chosen_arm, reward):
        pass

class GreedyPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms
```

```
    def select_arm(self):
        return np.argmax(self.values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

class EpsilonGreedyPolicy:
    def __init__(self, n_arms, epsilon):
        self.n_arms = n_arms
        self.epsilon = epsilon
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        if random.random() > self.epsilon:
            return np.argmax(self.values)
        else:
            return random.randint(0, self.n_arms - 1)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))

class UCBPolicy:
    def __init__(self, n_arms):
        self.n_arms = n_arms
        self.counts = [0] * n_arms
        self.values = [0.0] * n_arms

    def select_arm(self):
        total_counts = sum(self.counts)
        if total_counts < self.n_arms:
            return total_counts
        ucb_values = [0.0] * self.n_arms
        for arm in range(self.n_arms):
            bonus = math.sqrt((2 * math.log(total_counts)) / float(self.counts[a
            ucb_values[arm] = self.values[arm] + bonus
        return np.argmax(ucb_values)

    def update(self, chosen_arm, reward):
        self.counts[chosen_arm] += 1
        n = self.counts[chosen_arm]
        value = self.values[chosen_arm]
        self.values[chosen_arm] = ((n - 1) / float(n)) * value + (1 / float(n))
# Function to run the simulation
def run_simulation(policy, iterations=1000):
    results = []
    for i in range(iterations):
        user = env.get_user()
        chosen_arm = policy.select_arm()
        reward = env.simulate_click(user)
        policy.update(chosen_arm, reward)
        results.append((i, user['Age'], user['Gender'], user['City'], user['Phon
    results_df = pd.DataFrame(results, columns=['Iteration', 'Age', 'Gender', 'C
    return results_df
```

```
# Run simulation for Random Policy
random_policy = RandomPolicy(n_arms=3)
random_policy_results_df = run_simulation(random_policy)
random_policy_results_df.head()

# Calculate the CTR
def calculate_ctr(results_df):
    clicks = results_df['Reward'].sum()
    total = results_df.shape[0]
    return clicks / total

# Calculate CTR for Random Policy
random_ctr = calculate_ctr(random_policy_results_df)
random_ctr

# Continue similarly for other policies and different epsilon values
# Epsilon-Greedy Policy with epsilon = 0.1
epsilon_greedy_policy_0_1 = EpsilonGreedyPolicy(n_arms=3, epsilon=0.1)
epsilon_greedy_policy_0_1_results_df = run_simulation(epsilon_greedy_policy_0_1)
epsilon_greedy_ctr_0_1 = calculate_ctr(epsilon_greedy_policy_0_1_results_df)

# Epsilon-Greedy Policy with epsilon = 0.15
epsilon_greedy_policy_0_15 = EpsilonGreedyPolicy(n_arms=3, epsilon=0.15)
epsilon_greedy_policy_0_15_results_df = run_simulation(epsilon_greedy_policy_0_1
epsilon_greedy_ctr_0_15 = calculate_ctr(epsilon_greedy_policy_0_15_results_df)

# Epsilon-Greedy Policy with epsilon = 0.2
epsilon_greedy_policy_0_2 = EpsilonGreedyPolicy(n_arms=3, epsilon=0.2)
epsilon_greedy_policy_0_2_results_df = run_simulation(epsilon_greedy_policy_0_2)
epsilon_greedy_ctr_0_2 = calculate_ctr(epsilon_greedy_policy_0_2_results_df)

# UCB Policy
ucb_policy = UCBPolicy(n_arms=3)
ucb_policy_results_df = run_simulation(ucb_policy)
ucb_ctr = calculate_ctr(ucb_policy_results_df)

# Collect the results in a dictionary for easy plotting
ctrs = {
    'Random': random_ctr,
    'Epsilon-Greedy (0.1)': epsilon_greedy_ctr_0_1,
    'Epsilon-Greedy (0.15)': epsilon_greedy_ctr_0_15,
    'Epsilon-Greedy (0.2)': epsilon_greedy_ctr_0_2,
    'UCB': ucb_ctr
}

# Plot the results
plt.figure(figsize=(10, 6))
plt.bar(ctrs.keys(), ctrs.values(), color=['blue', 'red', 'purple', 'orange', 'c
plt.xlabel('Policy')
plt.ylabel('CTR')
plt.title('CTR Comparison for Different MAB Policies')
plt.show()
```

```
Dataset loaded successfully.
   Age Gender         City Phone_OS
0   25   Male     New York      iOS
1   25   Male     New York  Android
2   25   Male     New York    Other
3   25   Male  Los Angeles      iOS
4   25   Male  Los Angeles  Android
```



Discussing the Results Random Policy: Randomly selects ads without any learning, resulting in a uniform exploration of all ads. The CTR distribution will likely be wide, reflecting the overall average CTR of all ads.

Greedy Policy: Always selects the ad with the highest observed CTR so far. This can lead to quick convergence on a suboptimal ad if initial observations are not representative.

Epsilon-Greedy Policy: Balances exploration and exploitation by selecting a random ad with probability epsilon and the best ad with probability 1 - epsilon. Increasing epsilon leads to more exploration:

Epsilon = 0.1: Balanced approach, moderate exploration. Epsilon = 0.15: Increased exploration, more likely to find better ads but may sacrifice some exploitation. Epsilon = 0.2: High exploration, may find the best ads but could also result in lower overall CTR due to less exploitation. UCB Policy: Balances exploration and exploitation by considering the uncertainty of each ad's CTR. Expected to perform well in finding the best ad over time.

Conclusion (0.5M)

Conclude your assignment in 250 wrods by discussing the best approach for maximizing the CTR using random, greedy, epsilon-greedy and UCB.

`

To determine the best approach for maximizing the CTR using Random, Greedy, Epsilon-Greedy, and UCB algorithms, we analyze the performance and behavior of each algorithm through their respective CTR distributions. Here’s a summary and conclusion based on the results:

Analysis of Each Algorithm **Random Policy**:

Description: Randomly selects ads without any learning. CTR Distribution: Likely uniform, reflecting the overall average CTR of all ads. Performance: Poor, as it does not exploit any learned information about ad performance.

Greedy Policy:

Description: Always selects the ad with the highest observed CTR so far. CTR Distribution: Tends to quickly converge on a single ad, potentially suboptimal if initial observations are not representative. Performance: Can lead to suboptimal performance due to lack of exploration and susceptibility to initial variance.

Epsilon-Greedy Policy:

Description: Balances exploration and exploitation by selecting a random ad with probability epsilon and the best ad with probability 1 - epsilon. CTR Distribution: Varies with epsilon, showing a mix of exploration and exploitation. Epsilon = 0.1: Balanced approach, moderate exploration and exploitation. Epsilon = 0.15: Increased exploration, better chance of finding optimal ads but may slightly reduce exploitation. Epsilon = 0.2: High exploration, more likely to find the best ads but could also result in lower overall CTR due to less exploitation. Performance: Generally good, especially with a balanced epsilon. Too high or too low epsilon can either reduce exploitation or miss optimal ads.

UCB Policy:

Description: Balances exploration and exploitation by considering the uncertainty of each ad's CTR. CTR Distribution: Expected to be more focused on higher-performing ads while still exploring sufficiently. Performance: Typically high, as it smartly balances exploration and exploitation, adapting to the observed performance and uncertainty of each ad.

Conclusion Based on the analysis, here are the conclusions and recommendations for each algorithm:

Random Policy: Not recommended for maximizing CTR due to lack of learning and exploitation.

Greedy Policy: Simple and can quickly converge, but not reliable for maximizing CTR as it may settle on suboptimal ads due to lack of exploration.

Epsilon-Greedy Policy:

Epsilon = 0.1: Provides a good balance between exploration and exploitation. Recommended for scenarios where moderate exploration is needed. Epsilon = 0.15: Offers increased exploration. Suitable for environments with high variability where finding the best ad requires more exploration. Epsilon = 0.2: High exploration rate, which might be too exploratory in stable environments but can be useful in highly dynamic environments. UCB Policy: Generally the best approach for maximizing CTR. It effectively balances exploration and exploitation, adapting to the performance and uncertainty of each ad. Recommended for most scenarios, especially where continuous learning and adaptation are crucial.

Final Recommendation UCB Policy is recommended as the best overall approach for maximizing CTR due to its adaptive balancing of exploration and exploitation. However, Epsilon-Greedy Policy with an epsilon value of 0.1 or 0.15 can also be effective, especially in environments where some degree of exploration is beneficial. Avoid using the Random Policy and be cautious with the Greedy Policy due to their inherent limitations.

---------------------END of code-----------------------------

In [ ]:

```
```