```
import gymnasium as gym
import numpy as np

env = gym.make('Blackjack-v1', natural=True)

num_episodes = 10000
returns = {}
policy = {}

def sample_policy(observation):
    return 0 if np.random.rand() < 0.5 else 1  # 0: hit, 1: stand

for episode in range(num_episodes):
    state, info = env.reset()
    done = False
    episode_data = []

    while not done:
        action = sample_policy(state)
        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        episode_data.append((state, action, reward))
        state = next_state

    G = 0
    for state, action, reward in reversed(episode_data):
        G = reward + G
        if (state, action) not in returns:
            returns[(state, action)] = []
        returns[(state, action)].append(G)

for key in returns:
    print(f"State: {key[0]}, Action: {key[1]}, Returns: {returns[key]}")
```

✦

```
State: (12, 3, 1), Action: 1, Returns: [1.0]
```

```
State: (14, 8, 1), Action: 0, Returns: [-1.0, -1.0, -1.0, -1.0]
State: (12, 6, 1), Action: 0, Returns: [1.0]
State: (12, 4, 1), Action: 0, Returns: [-1.0]
/usr/local/lib/python3.12/dist-packages/jupyter_client/session.py:203: DeprecationWarning: datetime.datetime.utcnow() is depr
  return datetime.utcnow().replace(tzinfo=utc)
```

```
import gymnasium as gym

env = gym.make('Blackjack-v1', natural=True)
```

```
import numpy as np

returns = {}


Q = np.zeros((32, 11, 2, 2))
N = np.zeros((32, 11, 2, 2))

num_episodes = 50000
EPSILON = 0.1
```

```
import random

def policy(state):
    """
    Implements an epsilon-greedy policy.

    Args:
        state: A tuple representing the current state (player_sum, dealer_card, usable_ace).

    Returns:
        An integer representing the action (0 for hit, 1 for stand).
    """

    state_index = (state[0], state[1], state[2])

    if random.random() < EPSILON:
        return random.randint(0, N_ACTIONS - 1)  # Explore
    else:

        if state_index[0] < Q.shape[0] and state_index[1] < Q.shape[1] and state_index[2] < Q.shape[2]:
             return np.argmax(Q[state_index])
        else:

             return random.randint(0, N_ACTIONS - 1)
```

```


for episode in range(num_episodes):

    state, info = env.reset()
    done = False
    episode_data = []

    while not done:
        action = policy(state)

        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        episode_data.append((state, action, reward))
        state = next_state


    pass
```

```


G = 0
for state, action, reward in reversed(episode_data):
    G = reward + GAMMA * G

    state_index = (state[0], state[1], state[2])


    if (state_index, action) not in returns:
        returns[(state_index, action)] = []

    returns[(state_index, action)].append(G)
    N[state_index][action] += 1
    Q[state_index][action] = np.mean(returns[(state_index, action)])
```

```
for episode_num in range(num_episodes):

    episode_data = []


    state, info = env.reset()
    done = False


    while not done:
        action = policy(state)

        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        episode_data.append((state, action, reward))
        state = next_state

    G = 0
    for state, action, reward in reversed(episode_data):
        G = reward + GAMMA * G

        state_index = (state[0], state[1], state[2])

        if state_index[0] < Q.shape[0] and state_index[1] < Q.shape[1] and state_index[2] < Q.shape[2]:
            if (state_index, action) not in returns:
                returns[(state_index, action)] = []

            returns[(state_index, action)].append(G)

            N[state_index][action] += 1

            if N[state_index][action] > 0:
                 Q[state_index][action] = np.mean(returns[(state_index, action)])


optimal_policy = np.argmax(Q, axis=3)
```

```
total_reward = 0

num_eval_episodes = 10000

for episode_num in range(num_eval_episodes):
    state, info = env.reset()
    done = False
    episode_reward = 0

    while not done:
        state_index = (state[0], state[1], state[2])
        if state_index[0] < Q.shape[0] and state_index[1] < Q.shape[1] and state_index[2] < Q.shape[2]:
             action = np.argmax(Q[state_index])
        else:

             action = np.argmax(Q[state_index])


        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        episode_reward += reward
        state = next_state

    total_reward += episode_reward

average_reward = total_reward / num_eval_episodes

print(f"Average reward over {num_eval_episodes} evaluation episodes: {average_reward}")
```

```
Average reward over 10000 evaluation episodes: -0.0779
```