Lab Assignment: Reinforcement learning

Lab team: J07

Name (member 1): Héctor Tablero Díaz

Name (member 2): Álvaro Martínez Gamo

- Please include your full name at the beginning of all submitted files.
- Make sure the presentation is well-structured: the report will be evaluated not only for correctness, but also for clarity, conciseness, and completeness.
- Make use of figures and tables to summarize the results and illustrate the discussions.
- If external material is used, the sources must be cited.
- Include references in APA format https://pitt.libguides.com/citationhelp/apa7. Lack or poorly formatted references can be penalized.
- A generative AI tool can be used for consultation. You must specify the tool used in your report.
- You are not allowed to use a generative AI tool to generate code.

Submit a single `.zip` file, whose name has the format `AA3_2024_2025_P03_teamCode_lastName1_lastName2.zip` The name must not include graphical accents, spaces, uppercase letters, or special characters.

For example: `AA3_2024_2025_P03_V03_munyoz_deLaRosa.zip`

This compressed file must include the following files:

- This Python notebook with the solutions of the exercises. The notebook should include only code snippets, figures, tables, derviations and explanations (with LaTex if necessary) in Markdown cells. Handwritten material can be included in the Python notebook as images. Functions should be defined in a separate `.py` file, not in the notebook.
- The necessary `.py` and addional files to ensure the Python notebook code can be executed sequentially without errors.
- A PDF file generated from the notebook (Export the notebook as an HTML file. Open the HTML file in a Browser and print it as a PDF file).

Make sure that all the code cells can be executed squentially without errors (Kernel -> Restart & Run All). Exectution and formatting errors will be penalized.

The grade of this lab assignment is based on

- This submission (50 %).
- An individual in-class exam (50%).

Evaluation criteria:

- [6 points] Quality of the report (correctness, clarity, conciseness, completeness).
- [3 points] Quality of the code (correctness, adherence to a Python style guide -for instance, Google's-, comments, functional decomposition).
- [1 point] References.

Training a reinforcemnt learning agent at the gymnasium 🚀

📚 RL-environment:

- Python and NumPy

- Gymnasium

🎮 Environment:

- FrozenLake-v1

🖥️ Pygame:

https://www.pygame.org/wiki/about

Exercise 1: The gymnasium

TO DO: Complete the gymnasium tutorials

- Basic usage: https://gymnasium.farama.org/introduction/basic_usage/
- Training an RL-agent: https://gymnasium.farama.org/introduction/train_agent/

In [3]:
```
%load_ext autoreload
%autoreload 2

import numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
import imageio
import os

from tqdm.notebook import tqdm
import rl_utils
import time
from IPython.display import display, clear_output


from reinforcement_learning import (
    sarsa_learning,
    q_learning,
    greedy_policy,
    epsilon_greedy_policy,
)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [4]:
```
# Install packages if needed
# %pip install pygame
# %pip install gymnasium
```

Exercise 2: Q-learning

We're now ready to code our Q-Learning algorithm 🔥

Step 0: Set up and understand Frozen Lake environment

Let's begin with a simple 4x4 map and non-slippery, meaning the agent always moves in the desired direction.

We add a parameter called `render_mode` that specifies how the environment should be visualised. In our case because we **want to record a video of the environment at the end, we need to set render_mode to rgb_array**.

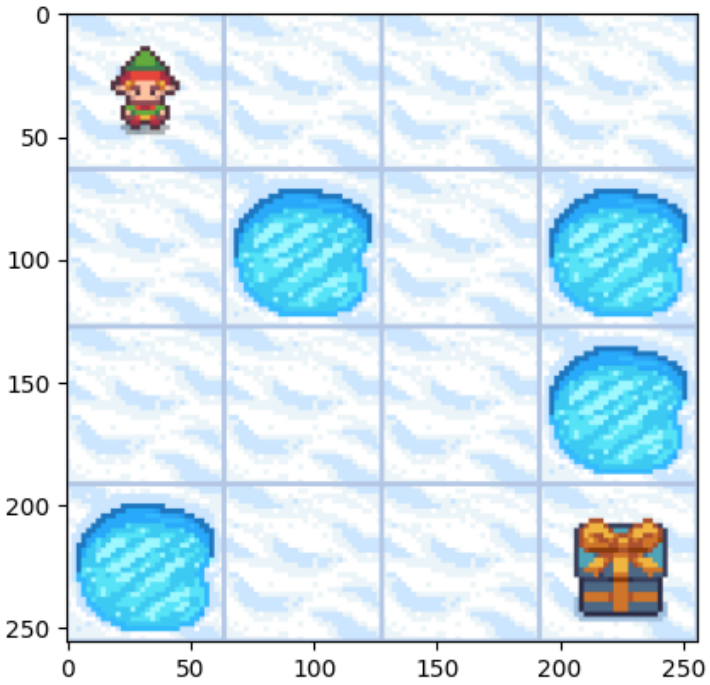
As explained in the documentation "rgb_array": Return a single frame representing the current state of the environment. A frame is a np.ndarray with shape (H, W, 3) representing RGB values for an H (height) times W (width) pixel image.

In [5]:
```
small_environment = gym.make(
    'FrozenLake-v1',
    map_name='4x4',
    is_slippery=False,
    render_mode='rgb_array',
)
```

Let's see what the environment looks like:

In [6]:
```
state, info = small_environment.reset()  # observation state
action_names = {0: 'Left', 1: 'Down', 2: 'Right', 3: 'Up'}

print(state)
print(info, '(Probability that the action has led to the current state)')
n_states = small_environment.observation_space.n
print("There are ", n_states, " possible states")

n_actions = small_environment.action_space.n
print("There are ", n_actions, " possible actions")
print(action_names)

fig, ax = plt.subplots()
game_image = ax.imshow(small_environment.render())
```

```
0
{'prob': 1} (Probability that the action has led to the current state)
There are  16  possible states
There are  4  possible actions
{0: 'Left', 1: 'Down', 2: 'Right', 3: 'Up'}
```



In [7]:
```
# Generate a random observed state
print("Observation state (randomly selected)", small_environment.observation_space.sample())

# Generate a random action from the current state
print("Action (randomly selected):", small_environment.action_space.sample())
```

```
Observation state (randomly selected) 5
Action (randomly selected): 0
```

In [8]:
```
#  Generate an episode

obsevation, info = small_environment.reset()  # state state_state
fig, ax = plt.subplots()
game_image = ax.imshow(small_environment.render())


MAX_STEPS = 100
refresh_rate = 1 # in (1 / seconds)

episode_over = False
n_steps = 0

while not episode_over and n_steps < MAX_STEPS:
    n_steps += 1
    action = small_environment.action_space.sample()

    state, reward, terminated, truncated, info = small_environment.step(action)
    episode_over = terminated or truncated

    ax.set_title(
        'Step: {}  State: {}  Reward: {} Action:{}'.format(
            n_steps,
            state,
            reward,
            action_names[action],
        )
    )

    display(fig)
    time.sleep(1.0 / refresh_rate)
    clear_output(wait=True)  # Clear previous output
    game_image.set_data(small_environment.render())

small_environment.close()
```

Step 1: Greedy and Epsilon greedy policies

Since Q-Learning is an **off-policy** algorithm, we have two policies. This means we're using a **different policy for acting and updating the value function**.

- Epsilon-greedy policy (acting policy)

- Greedy-policy (updating policy)

The greedy policy will also be the final policy we'll have when the Q-learning agent completes training. The greedy policy is used to select an action using the Q-table.

Epsilon-greedy is the training policy that handles the exploration/exploitation trade-off.

- With *probability 1 - ε* : **we do exploitation** (i.e. our agent selects the action with the highest state-action pair value).
- With *probability ε*: we do **exploration** (trying a random action).

As the training continues, we progressively **reduce the epsilon value since we will need less and less exploration and more exploitation.**

TO DO: Implement these policies in `reinforcement_learning.py`

Step 2: Train the RL agent 🏃

TO DO: Implement the Q-learning algorithm in `reinforcement_learning.py`

Define the hyperparameters for the learning process ⚙️

The exploration related hyperparamters are some of the most important ones.

- We need to make sure that our agent **explores enough of the state space** to learn a good value approximation. To do that, we need to have progressive decay of the epsilon.
- If you decrease epsilon too fast (too high decay_rate), **you take the risk that your agent will be stuck** in a local optimum, since your agent didn't explore enough of the state space and hence can't solve the problem.

In [9]:
```
# Training hyperparameters

n_training_episodes = 1000
max_steps = 100 # Maximum number of steps per episode
learning_rate = 0.7
gamma = 0.95 # Discount factor

# Exploration parameters
max_epsilon = 1.0  # Initial exploration probability
min_epsilon = 0.05  # Minimum exploration probability
decay_rate = 0.0005 # Exponential decay rate for the exploration probability
```

In [10]:
```
# Initialize Q-table
Qtable_small = np.zeros(
    (
        small_environment.observation_space.n,
        small_environment.action_space.n
    )
)


# Learn Q-table
Qtable_small = q_learning(
    small_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
```

```
    Qtable_small,
)
```

```
0%|          | 0/1000 [00:00<?, ?it/s]
```

Let's see what our Q-Learning table looks like now 👀

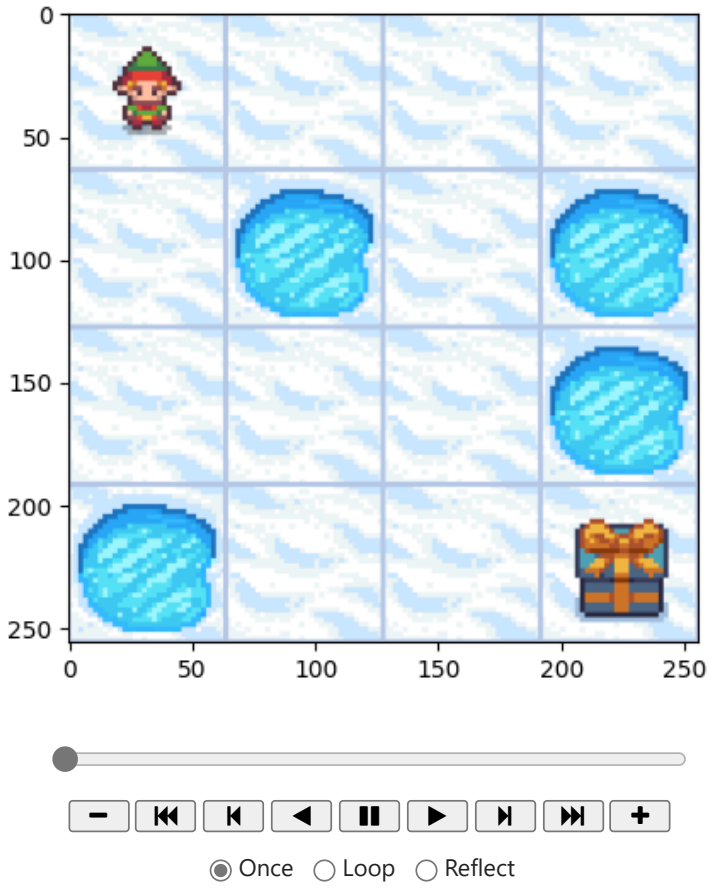
In [11]:
```
Qtable_small
```

Out[11]:
```
array([[0.73509189, 0.77378094, 0.77378094, 0.73509189],
       [0.73509189, 0.        , 0.81450625, 0.77378094],
       [0.77378094, 0.857375  , 0.77378094, 0.81450625],
       [0.81450625, 0.        , 0.77378094, 0.77378094],
       [0.77378094, 0.81450625, 0.        , 0.73509189],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.9025    , 0.        , 0.81450625],
       [0.        , 0.        , 0.        , 0.        ],
       [0.81450625, 0.        , 0.857375  , 0.77378094],
       [0.81450625, 0.9025    , 0.9025    , 0.        ],
       [0.857375  , 0.95      , 0.        , 0.857375  ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.9025    , 0.95      , 0.857375  ],
       [0.9025    , 0.95      , 1.        , 0.9025    ],
       [0.        , 0.        , 0.        , 0.        ]])
```

...and what our agent is doing!

In [12]:
```
frames = rl_utils.generate_greedy_episode(small_environment, Qtable_small)
rl_utils.show_episode(frames, interval=250)
```

Out[12]:



A more challenging problem

We're ready now to find our way in more challenging environments 💥

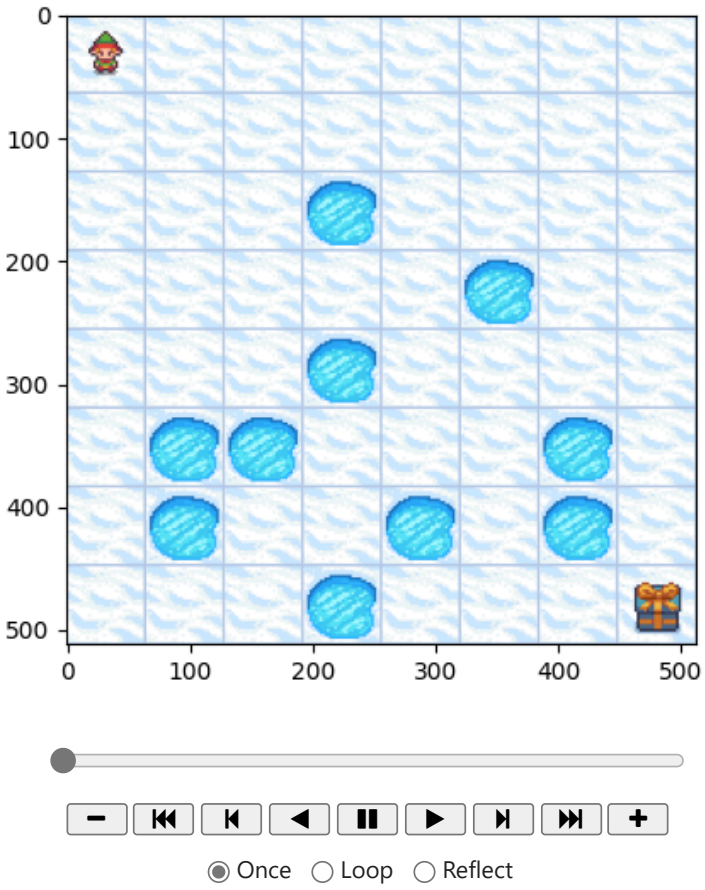
In [13]:
```
large_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=False,
    render_mode='rgb_array'
)

n_states = large_environment.observation_space.n
print("There are ", n_states, " possible states")

n_actions = large_environment.action_space.n
print("There are ", n_actions, " possible actions")
```

```
There are  64  possible states
There are  4  possible actions
```

In [ ]:
```
# TO DO: Training hyperparameters
n_training_episodes = 10000
max_steps = 700
learning_rate = 0.7
gamma = 0.99
max_epsilon = 1.0
min_epsilon = 0.1
decay_rate = 0.00001


# Initialize Q-table
Qtable_large = np.zeros(
    (
        large_environment.observation_space.n,
        large_environment.action_space.n
    )
)

# Learn Q-table
Qtable_large = q_learning(
    large_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    Qtable_large,
)
```

In [15]:
```
frames = rl_utils.generate_greedy_episode(large_environment, Qtable_large)
rl_utils.show_episode(frames, interval=250)
```

Out[15]:



Slippery environment

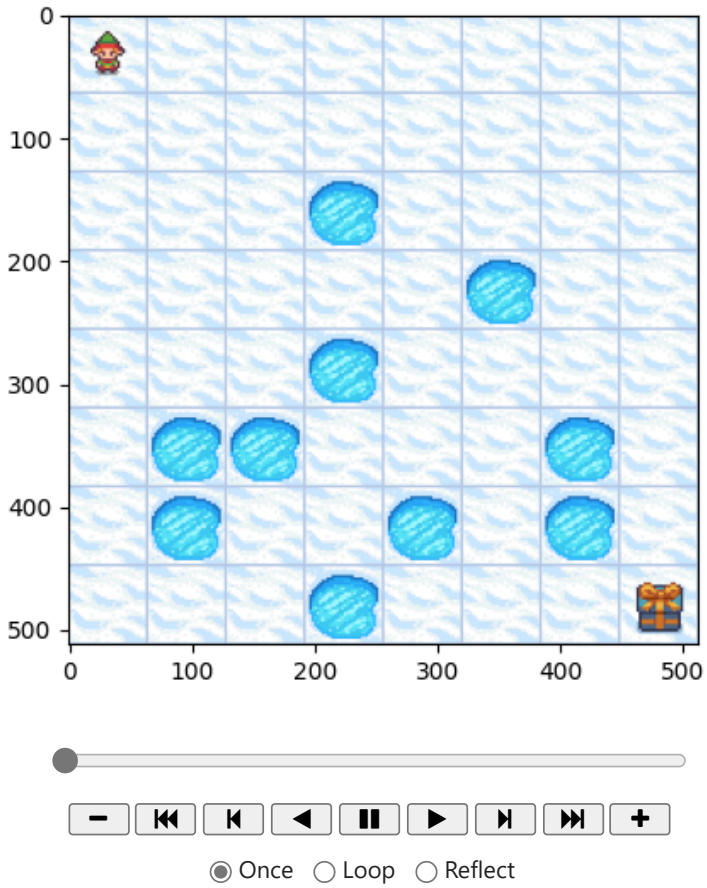
Our environment is now slippery, meaning the agent sometimes slips and moves in an unintended direction

In [16]:
```
slippery_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=True,
    render_mode='rgb_array',
)
```

To visualize the challenges in this environment, let's make our agent go right several times and see what happens:

In [17]:
```
go_right = []
action = 2  # go-right
n_steps = 20
state, info = slippery_environment.reset()
for i in range(n_steps):
    go_right.append(slippery_environment.render())  # Capture current frame (RGB array)
    slippery_environment.step(action)

rl_utils.show_episode(go_right, interval=250)
```

Out[17]:



Let's see what happens when we train our agent in the same way as before.

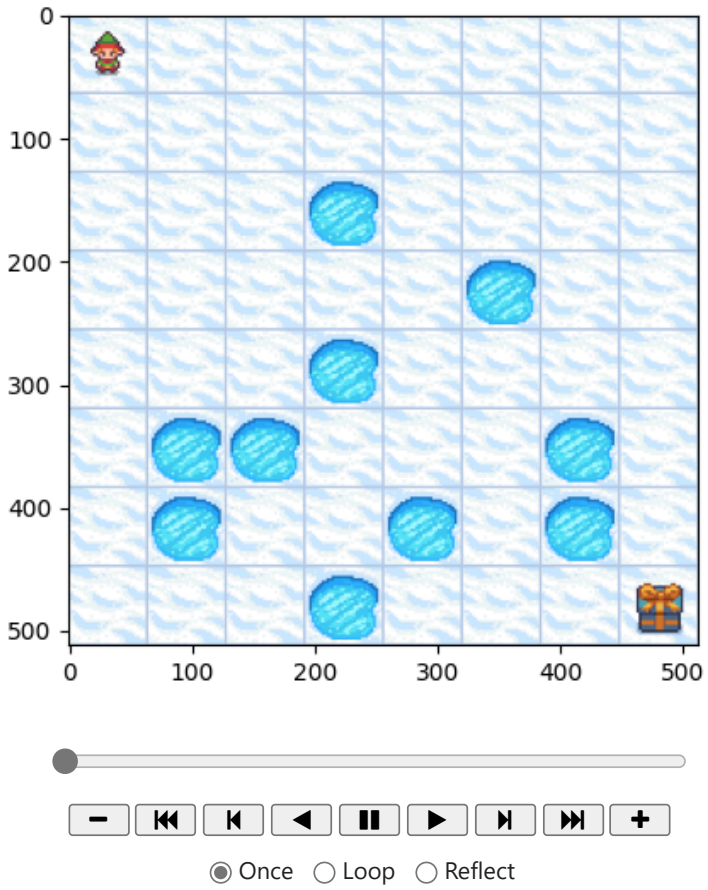
> Procedemos a volver a cambiar los hiperparametros de tal forma que el agente llegue al objetivo con el atributo de `is_slippery` en **True**.

In [ ]:
```
# TO DO: Training hyperparameters
################################################################################################
n_training_episodes = 100000
max_steps = 800
learning_rate = 0.1
gamma = 0.995
max_epsilon = 1.0
min_epsilon = 0.5
decay_rate = 0.00001
################################################################################################


# Initialize Q-table
Qtable_slippery = np.zeros(
    (
        slippery_environment.observation_space.n,
        slippery_environment.action_space.n
    )
)

# Learn Q-table
Qtable_slippery = q_learning(
    slippery_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
```

```
    decay_rate,
    Qtable_slippery,
)
```

In [19]:
```
frames = rl_utils.generate_greedy_episode(slippery_environment, Qtable_slippery)
rl_utils.show_episode(frames, interval=250)
```

Out[19]:



> Podemos obervar como el agente tiene una mayor dificultad de encontrar el objetivo debido a la dificultad añadida por el `enviroment`.

Exercise 3: Train the RL agent using SARSA 🏃

TO DO: Implement the SARSA learning algorithm in `reinforcement_learning.py`

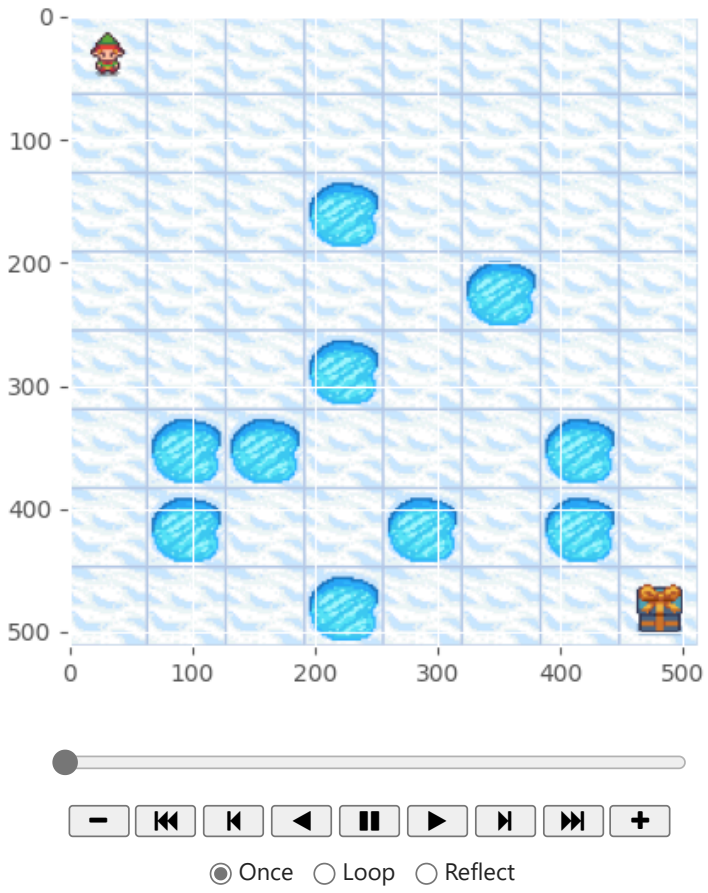
> Procedemos a replicar el entrenamiento anterior pero ahora usando el algoritmo `SARSA`.

In [ ]:
```
large_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=False,
    render_mode='rgb_array'
)

# Hiperparámetros optimizados
n_training_episodes  = 10000
max_steps = 700
learning_rate = 0.1
gamma = 0.995
max_epsilon = 1.0
min_epsilon = 0.5
```

```
decay_rate = 0.0001


# Inicializar Q-table
Qtable_large_sarsa = np.zeros(
    (
        large_environment.observation_space.n,
        large_environment.action_space.n
    )
)


# Aprender Q-table
Qtable_large_sarsa = sarsa_learning(
    large_environment,
    n_training_episodes,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    max_steps,
    Qtable_large_sarsa,
)
```

In [27]:
```
frames = rl_utils.generate_greedy_episode(large_environment, Qtable_large_sarsa)
rl_utils.show_episode(frames, interval=250)
```

Out[27]:



Exercise 4: Compare SARSA and Q-learning

Answer these questions:

- What is "episode reward" in RL and how is it related to the performance of an RL agent?
- What is "episode length" in RL and how is it related to the performance of an RL agent?
- What is "training error" in RL and how is it related to the performance of an RL agent?
    - Provide an explicit expression of the training error in the cases explored

1. Recompensa de Episodio en RL (episode reward)

Definición:
Suma total de recompensas acumuladas por un agente desde el estado inicial hasta el terminal en un episodio.

Fórmula:

$$\text{Recompensa de Episodio} = \sum_{t=0}^{T} r_t$$

Variables:

- r_t: Recompensa en el paso t.
- T: Total de pasos en el episodio.

Relación con el Rendimiento:

- **Valores altos = Mejor rendimiento**: Indica políticas exitosas para maximizar recompensas.
- **Seguimiento de progreso**: Monitoreo de cambios durante el entrenamiento.
- **Indicador de convergencia**: Estabilización sugiere aprendizaje completo.
- **Comparación de políticas**: Métrica para evaluar algoritmos/configuraciones.

2. Longitud de Episodio en RL (episode length)

Definición:
Número de pasos realizados por el agente antes de terminar un episodio.

Relación con el Rendimiento:

Tipo de Entorno	Longitud Ideal	Explicación
Entornos con objetivos	Corta (⬇️)	Camino eficiente al objetivo.
Entornos de "supervivencia"	Larga (⬆️)	Evita terminación prematura.
Entornos con penalización/paso	Corta (⬇️)	Minimiza recompensas negativas.

Caso Específico (Frozen Lake):

- Longitudes cortas = Política eficiente puesto que evita desvíos innecesarios.

3. Error de Entrenamiento en RL (training error)

Definición:
Diferencia entre las estimaciones de valor del agente y el valor objetivo (error TD).

Fórmulas por Algoritmo:

- **Q-learning**:

$$\text{Error TD} = r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)$$

- **SARSA**:

$$\text{Error TD} = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Variables:

- γ: Factor de descuento.
- $Q(s_t, a_t)$: Valor estimado de la acción a_t en estado s_t.

Error Cuadrático Medio (MSE):

$$\text{MSE} = \frac{1}{T} \sum_{t=0}^{T-1} (\text{Error TD}_t)^2$$

Relación con el Rendimiento:

- **Disminución del error**: Indica convergencia del aprendizaje.
- **Estabilidad**: Error estable = Política consistente.
- **Progreso**: Tasa de reducción refleja velocidad de aprendizaje.
- **Comparación algorítmica**: Diferencias en error revelan estabilidad dinámica.

Exercise 5: Compare SARSA and Q-learning

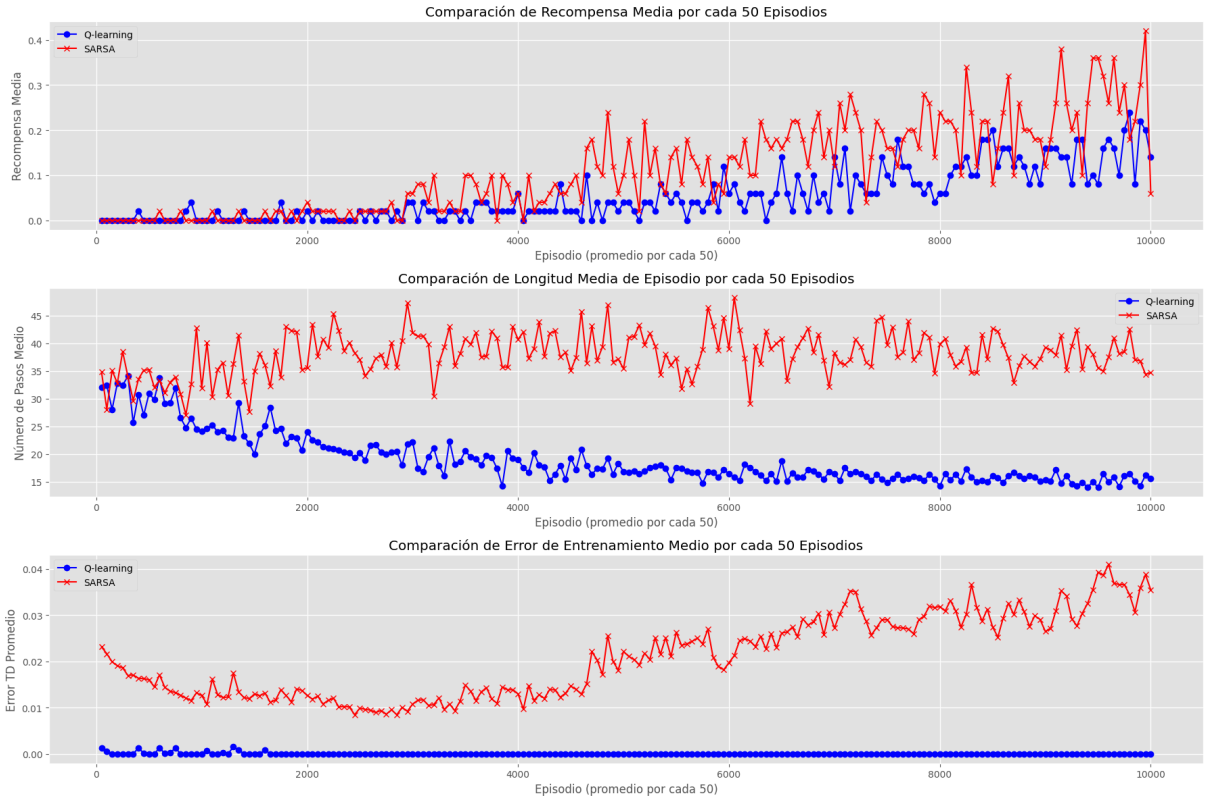
- Use matplotlib to compare the learning curves of SARSA and Q-learning, in terms of
  - episode reward.
  - episode length.
  - training error
- Discuss the results obtained.

> Ahora procedemos a comparar la curva de ambos algoritmos en los terminos explicados en el ejercicio anterior y por facilidad de cómputo y posterior discusión de los resultados, se va a usar el environment con el atributo `is_slippery=False`.

In [39]:
```python
from rl_utils import run_experiments

large_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=False,
    render_mode='rgb_array'
)

q_learning_param = {
    'n_training_episodes': 10000,
    'max_steps': 700,
    'learning_rate': 0.7,
    'gamma': 0.99,
    'max_epsilon': 1.0,
    'min_epsilon': 0.1,
    'decay_rate': 0.0001,
    'Q_table': Qtable_large # tabla obtenida en celdas anteriores
}

sarsa_param = {
    'n_training_episodes': 10000,
    'max_steps': 700,
    'learning_rate': 0.1,
    'gamma': 0.995,
    'max_epsilon': 1.0,
    'min_epsilon': 0.5,
    'decay_rate': 0.0001,
    'Q_table_sarsa' : Qtable_large_sarsa  # tabla obtenida en celdas anteriores
```

```
}

run_experiments(large_environment, q_learning_param, sarsa_param)
```

```
Entrenando algoritmo Q-learning...
  0%|          | 0/10000 [00:00<?, ?it/s]
Entrenando algoritmo SARSA...
  0%|          | 0/10000 [00:00<?, ?it/s]
```




```
Experimentos completados y gráficas generadas.

Resumen de resultados (promedios de los últimos 50 episodios):
Q-learning - Recompensa media: 0.14, Longitud media: 15.56, Error medio: 0.0000
SARSA - Recompensa media: 0.06, Longitud media: 34.68, Error medio: 0.0355
```

Discusión sobre las diferencias entre **SARSA** y **Q-learning**

- Análisis de la Recompensa Media:

En la primera gráfica, observamos que **SARSA** (🔴) logra consistentemente **recompensas más altas** que **Q-learning** (🔵), especialmente después de los **4000 episodios**. Esta diferencia se hace más pronunciada hacia el final del entrenamiento.

> **Conclusión rápida:** SARSA consigue mejores resultados en entornos donde el riesgo debe ser gestionado cuidadosamente.

Esta diferencia se debe fundamentalmente a la naturaleza **on-policy** de SARSA frente a la **off-policy** de Q-learning:

- 🔹 **SARSA (on-policy)**:
  Aprende la política que sigue durante el entrenamiento, **considerando el comportamiento exploratorio**, volviéndose más **conservador** y evitando riesgos innecesarios.

- 🔹 **Q-learning (off-policy)**:
  Aprende la política óptima **independientemente** de la política seguida durante la exploración, apostando por **rutas más arriesgadas** que pueden ser menos estables.

> En este entorno específico (*FrozenLake*), **SARSA** parece encontrar un mejor equilibrio entre **exploración** y **explotación**.

- Análisis de la Longitud Media de Episodio:

La segunda gráfica muestra una clara divergencia:

- **Q-learning (🔵)** tiende a episodios **más cortos** (aproximadamente **15 pasos**).
- **SARSA (🔴)** mantiene episodios **más largos** (alrededor de **35-40 pasos**).

Interpretación:

1. **Q-learning**:
    - Encuentra rutas más **directas** hacia la meta.
    - Prioriza caminos **rápidos** pero **arriesgados**.
2. **SARSA**:
    - Prefiere **rutas conservadoras** y **más largas**.
    - Evita **estados peligrosos** como los agujeros en *FrozenLake*.

> En general Q-learning encuentra caminos más cortos pero peligrosos, por el contrario, SARSA prefiere trayectorias más largas pero seguras, maximizando el éxito.

- Análisis del Error de Entrenamiento Medio:

La tercera gráfica muestra una diferencia importante:

- **SARSA (🔴)** presenta **errores TD** significativamente **mayores y crecientes**.
- **Q-learning (🔵)** mantiene errores **cercanos a cero**.

Explicación:

1. **SARSA**:
    - Conciliar **exploración** y **explotación** genera **inconsistencias** entre las predicciones y los resultados reales.
    - El error es alto porque **se sigue adaptando** constantemente.
2. **Q-learning**:
    - Optimiza directamente hacia la política **óptima teórica**.
    - **Converge** más rápidamente a una política **estable**, aunque pueda ser **subóptima globalmente**.

> **Importante:** Un mayor error en SARSA no implica peor rendimiento; al contrario, refleja su capacidad de **adaptarse** dinámicamente.

Conclusión General:

Estas gráficas ilustran cómo la **naturaleza on-policy** de **SARSA** frente a la **off-policy** de **Q-learning** afecta a el **comportamiento** de aprendizaje, el **ritmo de convergencia** y el **rendimiento final** en entornos complejos como *FrozenLake*.

> **Como hemos visto en las slides de clase, esta gráfica ilustra que:**
> **SARSA** apuesta por seguridad y adaptabilidad.
> **Q-learning** apuesta por rapidez y optimalidad teórica.

Exercise 6: Train a deep Q-learning agent (optional: extra point)

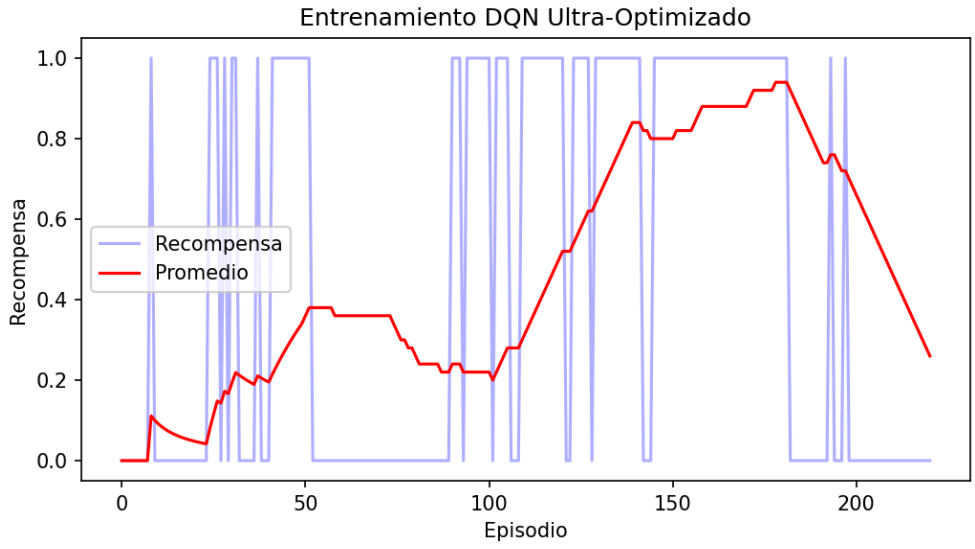
> Por motivos computacionales y de tiempo, hemos procedido a crear un agente en un environment 4x4 y con una baja optimización de todos los procesos con el objetivo de ver resultados en poco tiempo a pesar de no ser lo mas óptimo con el fin de realizar una brebe exploración a cerca del `deep Q-learning`.

In [1]:
```
from deep_agent import ultra_optimized_main

ultra_optimized_main()
```

```
Iniciando entrenamiento ultra-optimizado...
Entrenando durante máximo 500 episodios (con early stopping)...
  0%|          | 0/500 [00:00<?, ?it/s]
Episodio 1/500, Promedio: 0.00, Epsilon: 1.00
Episodio 50/500, Promedio: 0.34, Epsilon: 0.01
Episodio 100/500, Promedio: 0.22, Epsilon: 0.01
Episodio 150/500, Promedio: 0.80, Epsilon: 0.01
Episodio 200/500, Promedio: 0.68, Epsilon: 0.01
Early stopping en episodio 221 - No hay mejora durante 20 episodios
Entrenamiento completado en 221 episodios
  0%|          | 0/10 [00:00<?, ?it/s]
Evaluación: Puntuación media: 0.00, Éxito: 0.00% (0/10)

Rendimiento bajo. Entrenando brevemente...
Entrenando durante máximo 100 episodios (con early stopping)...
  0%|          | 0/100 [00:00<?, ?it/s]
Episodio 1/100, Promedio: 0.00, Epsilon: 1.00
Episodio 50/100, Promedio: 0.00, Epsilon: 0.01
Episodio 100/100, Promedio: 0.00, Epsilon: 0.01
Entrenamiento completado en 100 episodios

Evaluando agente final...
  0%|          | 0/50 [00:00<?, ?it/s]
Evaluación: Puntuación media: 0.24, Éxito: 24.00% (12/50)
Evaluación final: Tasa de éxito: 24.00%
```

Se han dejado las salidas a pesar de no renderizar la barra de progreso puesto que contiene información valiosa del proceso de entrenamiento del modelo.

> A continuación se muestra el resultado del entrenamiento del agente.



A pesar de todas las limitaciones con las que ha operado el agente, ha conseguido obtener ciertos resultados satisfactorios en poco tiempo.

Esto nos hace ver que si aumentamos la capacidad de cómputo y buscamos una mayor optimización de parametros e hiperparametros, podemos llegar a conseguir unos resultados altamente precisos.