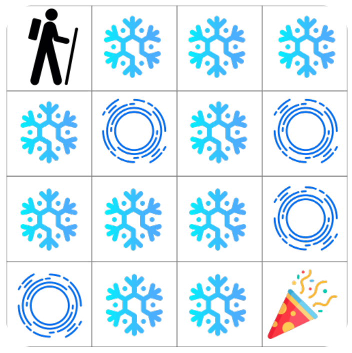
TAXI DRIVER

BOOTSTRAP

TAXI DRIVER

Preliminary

You will be playing an agent skiing on a frozen lake.
It's a simple game, so you can also implement the program yourself in any language you want.



✓ You're skiing in a grid world of size `n` x `m`.
✓ Starting from top left (0,0), you must go to bottom right.
✓ You can go in four directions, except when on the edges.
✓ Each cell is either :

- **F** for frozen (you can go on it) ;
- or **H** for hole (if you step into it, you fall and end up at the start point again).

The adjacent maze for instance is described the following way:

```
SFFF
FHFH
FFFH
HFFG
```

The major difficulty is that when you're are on a frozen cell, it is not guaranteed that you'll go in the direction you chose. If you choose right for instance, you might go in another direction.

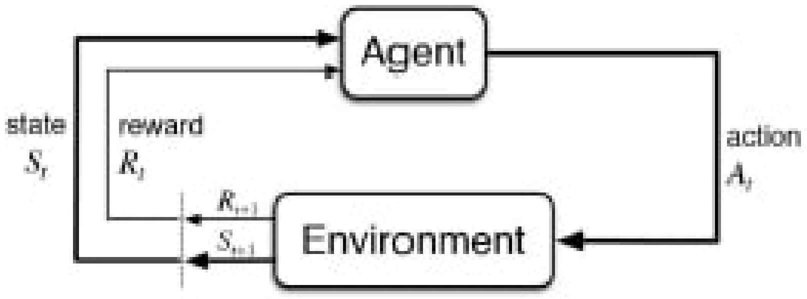
Before starting this bootstrap, import the *Gym* library and create the `Frozen Lake` environment. Render the lake.

Study briefly this library and write a random deplacement agent, to take ownership of the library.

Reinforcement Learning Basics

Let's solve this game using RL (Reinforcement Learning).

If you are not familiar with RL, get information about it!



There are 3 main things you need to define when starting a RL project:

- ✓ **States**: different states of your games ;
- ✓ **Actions**: actions the agent can take ;
- ✓ **Rewards**: rewards for choosing good actions.

Rewards will largely impact the performances of your system since your agent will choose actions that give them the best rewards.

Define and modelize the game **states**, **actions** and **rewards**.

Models and policies

There are two main types of RL algorithms:

1. *Model-based* which requires a *transition probability distribution* ;
2. *Model-free*, which does not require such a table.

There may be policies (that describes the probabilities for each possible *action* from each possible *state*) to describe the decisions to be taken for both types.

Algorithms may be on-policy (where the total reward is estimated only using the policy) or off-policy (when the total reward is estimated through several possible actions), use greedy policies or not.

Policies define which actions to take. But it's important to keep exploring while learning to optimize the total reward. In RL, as in some aspects of life, it's a trade-off between exploration and exploitation.

Exploitation will lead to the best local action but will prevent new explorations.
Exploration allows finding new actions with new states never discovered, and possibly better actions to take.

Epsilon-greedy method.

It is essential that you understand the meaning of these concepts.
Take time to investigate.

The game we suggested to you being **episodic**, there are two possible ways of learning: waiting until the end of one game to update the policy, or updating it after each action reward step (called Temporal Difference Learning).

Q-learning and SARSA algorithms

Implement both a **Q-learning** algorithm (a model-free off-policy algorithm) and **SARSA** (State Action Reward State Action), which is an on-policy algorithm.

Don't worry about optimization and parameter tuning.

For each algorithm, implement a function or a class to train your agent on a specific number of episodes (*n*) and then test it on other episodes (*m*).

If *n* is too low, the variance will be too important. If *n* is too high, it will take a long time to compute. Find a trade off between complexity and quality of measurement.

Output should display as much info as possible: average training time, average rewards, average number of steps, time for training, proportions of won episodes...

Compare your results between both algorithms (with several values of *n* and *m*), your random agent and a brute-force method.

Conclude about this benchmark.

Deep Learning

Q-learning and SARSA algorithms may not be efficient enough.
After investigating about the principles of Deep Learning, implement Deep Q-Learning.

Neural network...

v 1.4