Part 1: Gridworld Environment (10 points)

The `Dungeon` class in `iml/dungeon.py` defines a simple dungeon/gridworld environment. The most important attribute of this class is `transitions`, a three-dimensional array describing the dynamics of an agent moving around in our dungeon.

Given a number `action` in the set $\{0, 1, 2, 3\}$ encoding a direction the agent chooses to move, `transitions[action]` is a matrix of transition probabilities. Specifically, the value of this matrix at row i and column j is the probability that performing the given action will move us into state j given that we were previously in state i.

States are positions within a dungeon. Some states are unreachable because they are unoccupied by walls or holes. When our agent tries to walk into a wall, they don't move. When our agent tries to walk into a hole, they get teleported to the starting position. The dungeon floor may be slippery, so the agent sometimes moves in a direction it did not intend to. The natural goal in this environment is to get from the starting position `start` to a goal position `end` as quickly as possible.

Remember that a *policy* is a rule specifying what actions the agent should make as a (potentially stochastic) function of its current state. For a given policy π, let $V_\pi(i)$ be the expected number of steps that an agent following policy π will take to reach the goal state. For example, $V_\pi(\text{end}) = 0$, and $V_\pi(i) = 1$ if we can always get from i to the end state in one step. Finally, we define $V(i)$ to be the minimum value of $V_\pi(i)$ attained by any policy π.

(a) For `dungeon = make_ring_dungeon()`, compute our probability distribution over states after starting at `dungeon.start` and making 10 random actions. Display it using `dungeon.show()`. (2 points)

(b) For `dungeon = make_ring_dungeon()`, compute the exact state values $V_\pi(i)$ under the random policy. Display them. (3 points)

(c) Compute and display the state values $V(i)$ for `make_big_dungeon()`. (3 points)

(d) Based on the optimal state values $V(i)$, compute an optimal policy for the big dungeon. Display it using `dungeon.show_policy()`. (2 points)

Part 2: Tic-Tac-Toe (10 points)

Now let's solve the simple game of tic-tac-toe using Q-learning.

To define an agent, inherit from the `Player` class defined in `tic_tac_toe.py`. Then you can use the `rollout()` method to simulate games against an opponent who makes random moves.

Your agent will always play first, with the X's. When it is your agent's turn, the `play()` method is called with a string representation of the game state. This method should return a number in $\{0, ..., 8\}$, indicating which square it wants to play in. As soon as the game ends, the `outcome()` method informs the agent of the game's result. An outcome of 1 means it won, 0 means it drew, -1 means it lost.

In addition to the usual win and draw conditions of tic-tac-toe, the game is marked a loss when the agent chooses to play in a position that is already occupied.

(a) Using the language of state value functions, explain how you would normally solve tic-tac-toe. What would you say is the value of the starting state? (2 points)

(b) Implement an agent that learns tic-tac-toe through Q-learning. By whatever means necessary, ensure that the trained agent wins at least 99% of the time in a trial of $100,000$ games against the random opponent. (5 points)

(c) When playing against the random opponent, are some choices of starting moves better than others? (3 points)

Bonus: Multi-Armed Bandits (4 points)

Suppose that we present an agent with 100 arms which a priori are indistinguishable. At each time step i, it is allowed to choose an arm $a(i) \in \{1, 2, \ldots, 100\}$, after which it may receive a reward $r(i)$. For all but one arm, the agent has even odds of receiving a reward of 1, and otherwise receives 0. The remaining arm is special and provides a reward of 1 with probability 2/3. How can we design an agent that receives as much reward as possible over a limited timeframe?

The ϵ-greedy algorithm is one simple approach to this problem. Another approach which often performs better is called Thompson sampling. Thompson sampling recommends that we select each actions with probability equal to the probability, conditional on our previous trials, that it is the action with maximum expected reward.

(a) Implement Thompson sampling for this problem. Over 100 independent trials with 5000 iterations each, report the empirical fractions of times that the agent chose the special arm. (4 points)