AI Gamer Tile Puzzle:

The Eight Puzzle consists of a *3 x 3* board of sliding tiles with a single empty space. For each configuration, the only possible moves are to swap the empty tile with one of its neighboring tiles. The goal state for the puzzle consists of tiles 1-3 in the top row, tiles 4-6 in the middle row, and tiles 7 and 8 in the bottom row, with the empty space in the lower-right corner.

In this section, you will develop two solvers for a generalized version of the Eight Puzzle, in which the board can have any number of rows and columns. If you wish to use the provided GUI for testing, described in more detail at the end of the section, then your implementation must adhere to the recommended interface. However, this is not required, and no penalty will imposed for using a different approach.

A natural representation for this puzzle is a two-dimensional list of integer values between *0* and $r \cdot c - 1$ (inclusive), where *r* and *c* are the number of rows and columns in the board, respectively. In this problem, we will adhere to the convention that the *0*-tile represents the empty space.

1. **[0 points]** In the `TilePuzzle` class, write an initialization method `__init__(self, board)` that stores an input board of this form described above for future use. You additionally may wish to store the dimensions of the board as separate internal variables, as well as the location of the empty tile.

2. **[0 points]** *Suggested infrastructure.*

   In the `TilePuzzle` class, write a method `get_board(self)` that returns the internal representation of the board stored during initialization.

```
>>> p = TilePuzzle([[1, 2], [3, 0]])
>>> p.get_board()
[[1, 2], [3, 0]]
```

```
>>> p = TilePuzzle([[0, 1], [3, 2]])
>>> p.get_board()
[[0, 1], [3, 2]]
```

   Write a top-level function `create_tile_puzzle(rows, cols)` that returns a new `TilePuzzle` of the specified dimensions, initialized to the starting configuration. Tiles *1* through $r \cdot c - 1$ should be arranged starting from the top-left corner in row-major order, and tile *0* should be located in the lower-right corner.

```
>>> p = create_tile_puzzle(3, 3)
>>> p.get_board()
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
>>> p = create_tile_puzzle(2, 4)
>>> p.get_board()
[[1, 2, 3, 4], [5, 6, 7, 0]]
```

   In the `TilePuzzle` class, write a method `perform_move(self, direction)` that attempts to swap the empty tile with its neighbor in the indicated direction, where valid inputs are limited to the strings `"up"`, `"down"`, `"left"`, and `"right"`. If the given direction is invalid, or if the

move cannot be performed, then no changes to the puzzle should be made. The method should return a Boolean value indicating whether the move was successful.

```
>>> p = create_tile_puzzle(3, 3)
>>> p.perform_move("up")
True
>>> p.get_board()
[[1, 2, 3], [4, 5, 0], [7, 8, 6]]
```

```
>>> p = create_tile_puzzle(3, 3)
>>> p.perform_move("down")
False
>>> p.get_board()
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

In the `TilePuzzle` class, write a method `scramble(self, num_moves)` which scrambles the puzzle by calling `perform_move(self, direction)` the indicated number of times, each time with a random direction. This method of scrambling guarantees that the resulting configuration will be solvable, which may not be true if the tiles are randomly permuted. *Hint: The `random` module contains a function `random.choice(seq)` which returns a random element from its input sequence.*

In the `TilePuzzle` class, write a method `is_solved(self)` that returns whether the board is in its starting configuration.

```
>>> p = TilePuzzle([[1, 2], [3, 0]])
>>> p.is_solved()
True
```

```
>>> p = TilePuzzle([[0, 1], [3, 2]])
>>> p.is_solved()
False
```

In the `TilePuzzle` class, write a method `copy(self)` that returns a new `TilePuzzle` object initialized with a deep copy of the current board. Changes made to the original puzzle should not be reflected in the copy, and vice versa.

```
>>> p = create_tile_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.get_board() == p2.get_board()
True
```

```
>>> p = create_tile_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.perform_move("left")
>>> p.get_board() == p2.get_board()
False
```

In the `TilePuzzle` class, write a method `successors(self)` that yields all successors of the puzzle as (direction, new-puzzle) tuples. The second element of each successor should be a new `TilePuzzle` object whose board is the result of applying the corresponding move to the current board. The successors may be generated in whichever order is most convenient, as long as successors corresponding to unsuccessful moves are not included in the output.

```
>>> p = create_tile_puzzle(3, 3)
>>> for move, new_p in p.successors():
...     print move, new_p.get_board()
... ... print move, new_p.get_board() up [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
left [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
down [[1, 2, 3], [4, 7, 5], [6, 0, 8]]
```

```
>>> b = [[1,2,3], [4,0,5], [6,7,8]]
>>> p = TilePuzzle(b)
>>> for move, new_p in p.successors():
... print move, new_p.get_board()
up [[1, 0, 3], [4, 2, 5], [6, 7, 8]]
left [[1, 2, 3], [0, 4, 5], [6, 7, 8]]
right [[1, 2, 3], [4, 5, 0], [6, 7, 8]]
```

3. **[20 points]** In the `TilePuzzle` class, write a method `find_solutions_iddfs(self)` that yields all optimal solutions to the current board, represented as lists of moves. Valid moves include the four strings `"up"`, `"down"`, `"left"`, and `"right"`, where each move indicates a single swap of the empty tile with its neighbor in the indicated direction. Your solver should be implemented using an iterative deepening depth-first search, consisting of a series of depth-first searches limited at first to *0* moves, then *1* move, then *2* moves, and so on. You may assume that the board is solvable. The order in which the solutions are produced is unimportant, as long as all optimal solutions are present in the output.

   Hint: This method is most easily implemented using recursion. First define a recursive helper method `iddfs_helper(self, limit, moves)` that yields all solutions to the current board of length no more than `limit` which are continuations of the provided move list. Your main method will then call this helper function in a loop, increasing the depth limit by one at each iteration, until one or more solutions have been found.

```
>>> b = [[4,1,2], [0,5,3], [7,8,6]]
>>> p = TilePuzzle(b)
>>> solutions = p.find_solutions_iddfs()
>>> next(solutions)
['up', 'right', 'right', 'down', 'down']
```

```
>>> b = [[1,2,3], [4,0,8], [7,6,5]]
>>> p = TilePuzzle(b)
>>> list(p.find_solutions_iddfs())
[['down', 'right', 'up', 'left', 'down',
'right'], ['right', 'down', 'left',
'up', 'right', 'down']]
```

4. **[20 points]** In the `TilePuzzle` class, write a method `find_solution_a_star(self)` that returns an optimal solution to the current board, represented as a list of direction strings. If multiple optimal solutions exist, any of them may be returned. Your solver should be implemented as an A* search using the Manhattan distance heuristic, which is reviewed below. You may assume that the board is solvable. During your search, you should take care not to add positions to the queue that have already been visited. It is recommended that you use the `PriorityQueue` class from the `Queue` module.

   Recall that the Manhattan distance between two locations *(r_1, c_1)* and *(r_2, c_2)* on a board is defined to be the sum of the componentwise distances: *|r_1 - r_2| + |c_1 - c_2|*. The Manhattan distance heuristic for an entire puzzle is then the sum of the Manhattan distances between each tile and its solved location.

```
>>> b = [[4,1,2], [0,5,3], [7,8,6]]
>>> p = TilePuzzle(b)
>>> p.find_solution_a_star()
['up', 'right', 'right', 'down', 'down']
```

```
>>> b = [[1,2,3], [4,0,5], [6,7,8]]
>>> p = TilePuzzle(b)
>>> p.find_solution_a_star()
['right', 'down', 'left', 'left', 'up',
 'right', 'down', 'right', 'up', 'left',
 'left', 'down', 'right', 'right']
```

The arguments `rows` and `cols` are positive integers designating the size of the puzzle.