Module 5 Homework - Recursive path exploration

Use recursion to model path searching in two board games. The basic approach is the same for both games:

- Choose one of the available next steps
- Recursively explore all options for that step, until you find a solution or dead end

Game 1 - Stepping Circle

Model a circular board game. The board consists of a series of tiles with numbers on them. The numbers represent how many spaces you can move from a given tile, and you can move clockwise (CW) or counter-clockwise (CCW). It's okay to loop around - moves that go before the first tile or after the last are valid.

The goal is to reach the final tile (the tile 1 counter-clockwise from the start).

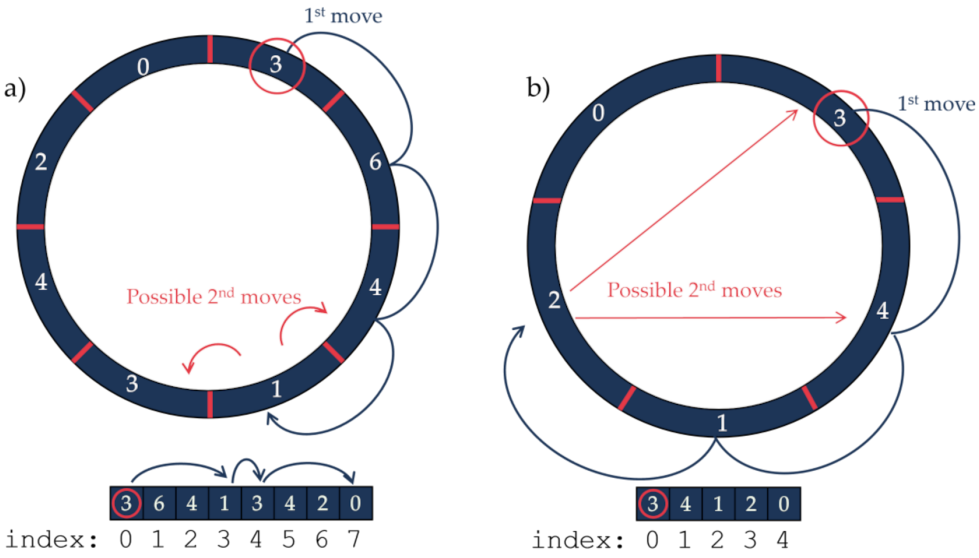

Figure 1: (a) `[3, 6, 4, 1, 3, 4, 2, 0]` is solveable in 3 moves. (b) `[3, 4, 1, 2, 0]` is unsolveable.

`SolvePuzzle.py`

- `solve_puzzle(board)` returns a boolean denoting if `board` is solveable.

```
>>> solve_puzzle([3, 6, 4, 1, 3, 4, 2, 0])
True
>>> solve_puzzle([3, 4, 1, 2, 0])
False
```

`TestSolvePuzzle.py`

Write unittests for a few puzzles. Make sure to test that your algorithm works with boards that require a mixture of CW and CCW moves, boards that require circling past the end, and boards that are unsolveable.

Keep the boards for these tests relatively small (<= 5 spaces) to make debugging easier

Tips

- Use memoization to to avoid infinite loops
- You can assume the numbers on tiles are non-negative integers (0 is valid, and may appear on any tile)
- The modulo operator `%` is helpful for finding indices when you loop around

Game 2 - Knight and Pawns

Use recursion to determine if a knight starting in a given square on a chessboard can capture all pawns on that board using only moves that capture a pawn (moves to spaces without a pawn, or spaces with pawns that have already been captured, are not allowed):

```
>>> #  0 1 2 3 4 5 6 7
>>> #0 - - - - - - - - Row 0: (0, 0), (0, 1), (0, 2), ... (0, 7)
>>> #1 - - - p - - - -
>>> #2 - p - - - p - -
>>> #3 - - - k - - - -
>>> #4 - - p - - - - -
>>> #5 - - - - - p - -
>>> #6 - - - p - - - -
>>> #7 - - - - - - - - -Row 7: (7, 0), (7, 1), (7, 2), ... (7, 7)
>>> k_idx = (3, 3)                                          # Initial knight index
>>> p_idxs = {(1, 3), (2, 1), (2, 5), (4, 2), (5, 5), (6, 3)}   # Set of pawn indices
>>> solveable(p_idxs, k_idx)
True
```

`TestKnights.py`

Starter code for `TestKnights.py` includes some guidance on tests.

`Knights.py`

- `valid_moves(k_idx)` - returns a set of tuples representing all valid moves for a knight at `k_idx`.

  ```
  >>> valid_moves((3, 3))
  {(2, 5), (1, 4), (2, 1), (1, 2), (4, 1), (5, 2), (5, 4), (4, 5)}
  ```

- `solveable(p_idxs, k_idx)` - returns a boolean denoting whether `p_idxs` and `k_idx` represent a solveable board

Files to Submit

Submit the following (plus any additional files necessary for your code to run)

- `TestSolvePuzzle.py`
- `SolvePuzzle.py`
- `TestKnights.py`
- `Knights.py`