Module 5 Lab - Recursive Board Game

Model a circular board game consisting of numbered tiles. The numbers represent how many tiles 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).

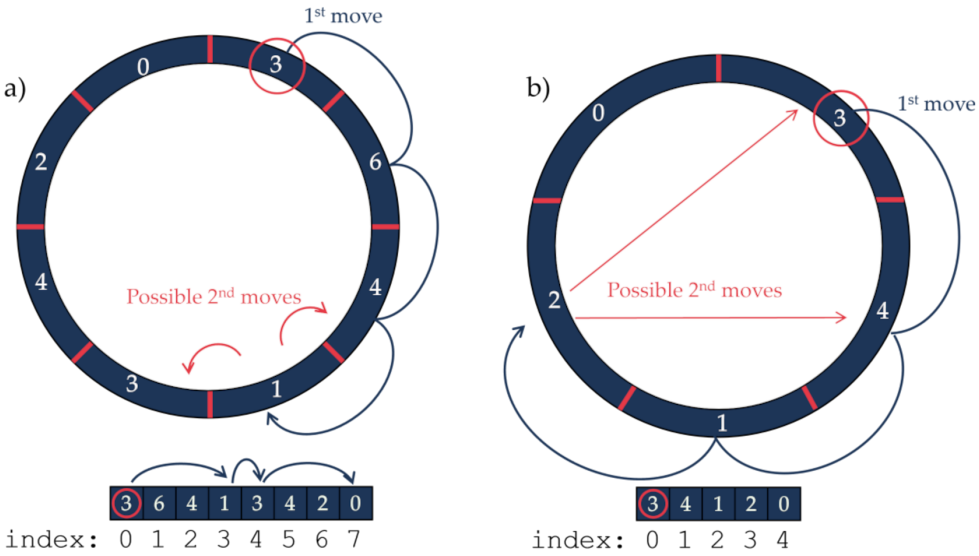

Figure 1: (a) `[3, 6, 4, 1, 3, 4, 2, 0]` is solveable in 3 moves. (b) `[3, 4, 1, 2, 0]` is unsolveable. While not shown, moving 3 spaces CCW at the start would also be a valid first move.

`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
```

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
- Add unittests to `TestSolvePuzzle.py` to help debug

Submission

At a minimum, submit the following files:

- `solve_puzzle.py`
- `TestSolvePuzzle.py`

Students must submit by the due date (typically Friday at 11:59 pm EST) to receive credit.

More Guidance

The above may be enough to get you started. If you get stuck, the rest of this document includes some more structured guidance.

Initialization

We want our user to call a function using e.g. `solve_puzzle(L)`. We'll then want to choose an initial index and create an empty set to keep track of which indices we've already visited, so a helper function is appropriate here:

```
def solve_puzzle(L):
    """Add a docstring"""
    return _solve_puzzle(L, idx=0, visited=set())

def _solve_puzzle(L, idx, visited):
    """Add a docstring"""
```

Recursive structure

Our general recursive algorithm for branching-path problems like this is:

1) Are we at a base case? If so, return appropriate value.
    - Here, our base case is if our *index* is equal to *the index of the last item in the list*. In this case, we should return `True` to denote that this board is solvable.
2) If not, calculate all possible next steps
    - Here, this is two values: the clockwise and counterclockwise indices.
3) Explore each path. If a solution is found, return `True`. If no paths are solveable, return `False` - there is no valid solution starting from this index.

    ```
    if _solve_puzzle(CWPARAMS): return True
    elif _solve_puzzle(CCWPARAMS): return True
    return False
    ```

    The above 3 lines can be combined into a single statement using the `or` operator.

Parameter Sharing

When we pass a mutable collection (like a list or set) as an argument in a function, python really just passes a pointer to that collection. This means that multiple levels of the recursive function will be working on the *same* object (see Figure 2 below).

The last attribute in each function on the stack is another function call, whose value is denoted by a question mark. This represents that the function object is waiting to resume until the above function returns a value.

Memoization

In this problem, we should memoize the indices we have visited. It probably makes the most sense to do this at the top of `_solve_puzzle`:

- check if we've already visited this index, and return `False` if so.

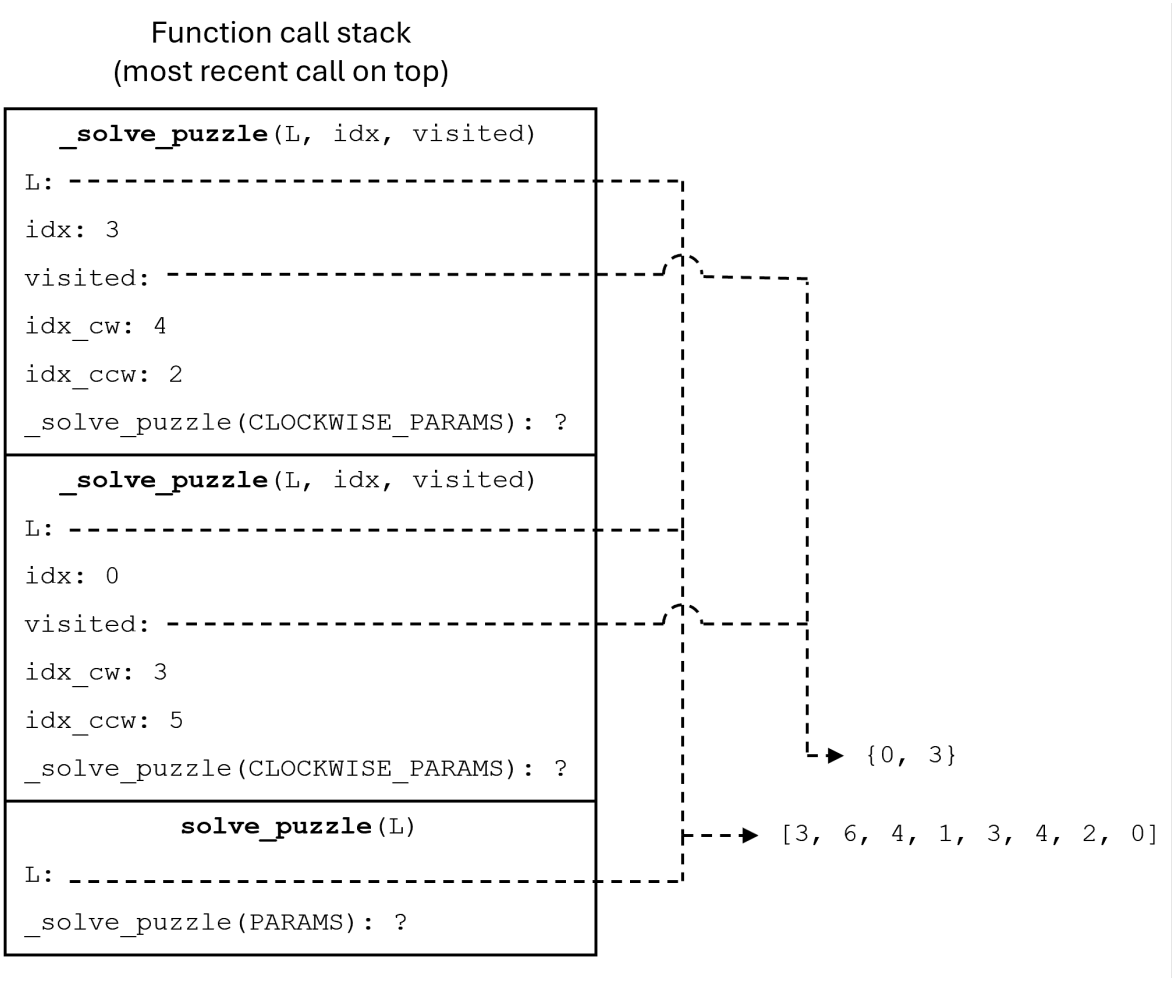

Figure 2: The function call stack just before the third call to `_solve_puzzle()` with the example board shows how multiple functions share access to the same mutables. Adding an item to the set `visited` in one function is immediately reflected in all the others.

- Otherwise, add this index to visited set, then do the rest of the algorithm.

```
if idx in visited: return False
visited.add(idx)
```

Calculating Moves

There is a chance that our move will bring us to a position greater than our final index (moving forwards past the edge of the board) or less than zero (moving backwards before the start). The modulo operator is a convenient way to "loop around." Consider a board with 5 items, and "position" values that grow from 0 to 11:

position	position % 5
0	0
1	1
2	2
3	3
4	4
5	0
6	1
7	2
8	3
9	4
10	0
11	1

Modulos of negative numbers work similarly:

position	position % 5
0	0
-1	4
-2	3
-3	2
-4	1
-5	0

In general, we can find the relative position of a looping index for a list of `n` items by calculating `position % n`:

```
idx_cw = (idx+L[idx]) % len(L)
idx_ccw = (idx-L[idx]) % len(L)
```