Module 5 Lab - Recursive Board Game

Many of the problems used to teach recursion are not efficiently solved with recursion - summing numbers, finding fibonaccis, and calculating factorials all have elegant but inefficient recursive solutions.

Here, we look at a problem type where recursive solutions tend to be genuinely useful: exploring branching paths until we hit a dead end or find a solution. We'll see this a lot in graphs later this semester, and it often comes up when modelling games.

The basic problem abstraction is this:

- we have multiple options of what to do at each step
- we want to see if *any* series of steps exists that connect a starting point to a valid solution

Stepping Game

We will model a circular board game. The board consists of a series of tiles with numbers on them.

- The number on a tile represents the number of tiles you can move from there, forwards (clockwise) or backwards (counter-clockwise).
- It's okay to "circle 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). This is the only valid solution - **finding a tile with "0" is not necessarily a valid solution, since non-final tiles may contain 0.**

Not all boards will be solvable. See below for a solveable and an unsolveable example.

Deliverables

`TestSolvePuzzle.py` First, write tests. This helps solidify the rules of the game in your mind and ensures you will know when you get a working algorithm.

Tests will not directly impact your grade, but you will not be able to debug your code without them.

- Test at least 4 puzzles:
  - solveable using only clockwise moves
    - make sure counter-clockwise moves do not produce a valid solution here
  - solveable using only counter-clockwise moves
    - make sure clockwise moves do not produce a valid solution here
  - requires a mixture of clockwise and counter-clockwise moves to solve
    - make sure a purely clockwise or purely counter-clockwise step-sequence do not produce a valid solution here
  - unsolveable
- Use the Unittest package
- Keep the boards for these tests relatively small (<= 5 spaces) to make debugging easier

`solve_puzzle.py`

- contains a function `solve_puzzle()`:

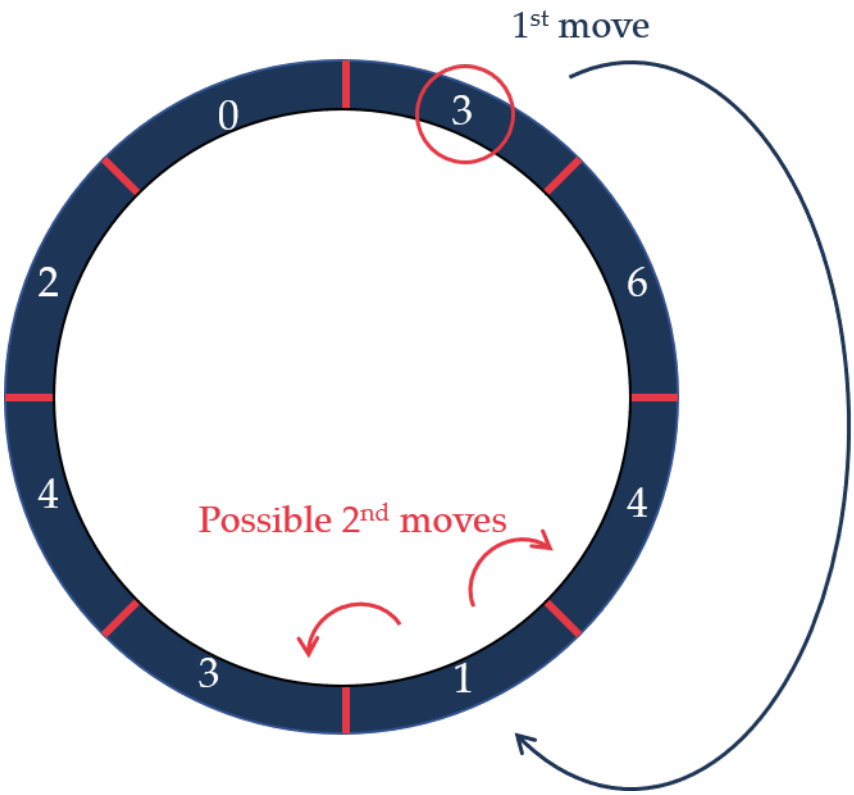

Figure 1: Solveable game: `[3, 6, 4, 1, 3, 4, 2, 0]`

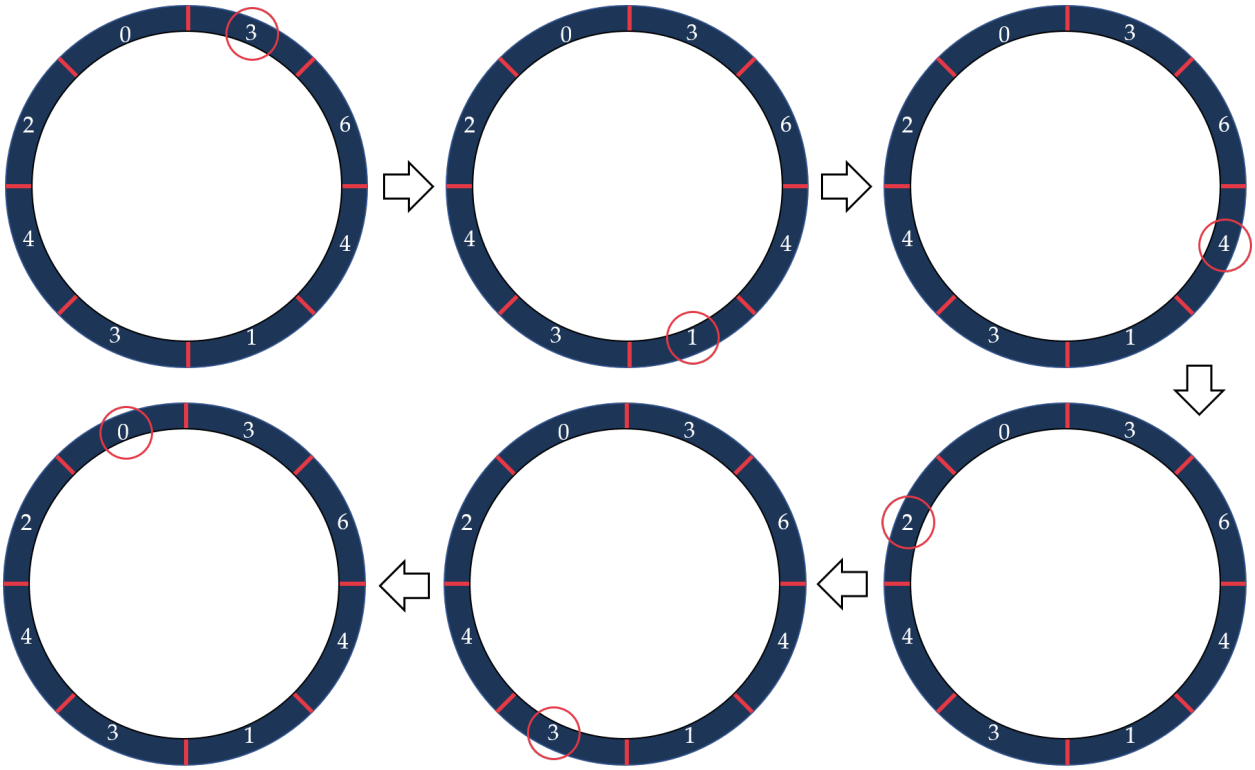

Figure 2: Step by step solutions to the above game

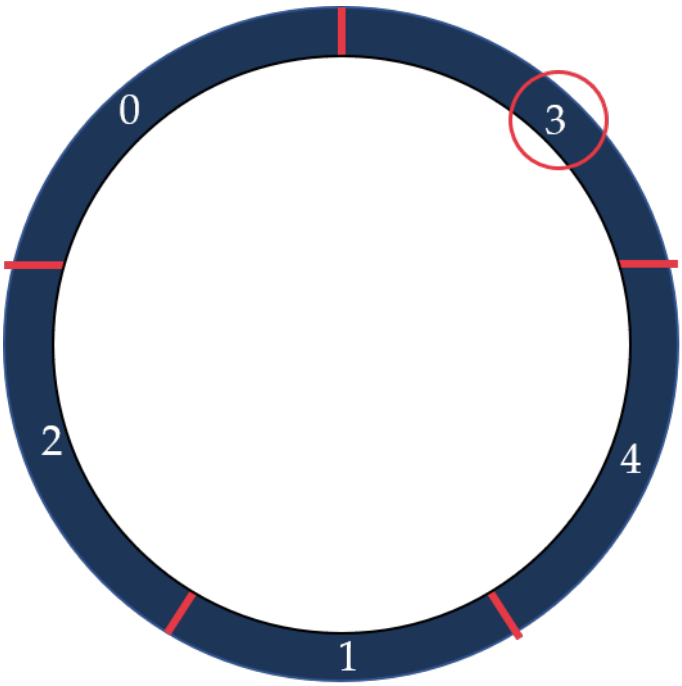

Figure 3: Unsolveable game: `[3, 4, 1, 2, 0]`

- Input: a list representing the board. The first item in the list is the starting position, with subsequent items denoting the board in a clockwise fashion. See the figures above for an example of list representations of boards.
- Output: A boolean (`True` or `False`) denoting if the puzzle is solveable.

Tips

- You will need memoization to to avoid infinite loops. If you use a helper function, avoid using an empty mutable collection (like an empty set set) as a default value.

```
def solve_puzzle(L, visited=None):  # No helper function
    if visited is None: # initialize

    # the rest of your work here
```

```
def solve_puzzle(L): # Using a helper function
    visited = set()
    return _solve_puzzle(L, visited)

def _solve_puzzle(L, visited):
    # the rest of your work here
```

- You can assume the numbers on tiles are non-negative integers (0 is valid).

Submission

At a minimum, submit the following files:

- `solve_puzzle.py`
- `test_solve_puzzle.py`

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

Grading

This assignment is entirely auto-graded.

Feedback

If you have any feedback on this assignment, please leave it here.