Objectives

* Practice with Recursion, memorization, and Dynamic Programming

Due Date

This assignment is due on Thursady, February 29, 2024, by 11:00 pm.

**Remarks:**

* It is expected you develop test cases for your code.
* When you are asked to hand in code, you cut-and-paste the definition(s) from your code and paste into this Word document immediately following the activity.
* Include with your code several test case examples and your results.
* This homework is to be individual work.

Value

This assignment is worth 20 points.

Activities

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 provided examples for a solveable and an unsolveable puzzle.

1. Write your tests for your programming using unittest and in the file TestSolvePuzzle.py. Coming up with and writing these tests helps solidify the rules of the game in your mind and ensures you will know when you get a working algorithm. The example puzzles provided can help in developing these tests.

* 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.
  + Solvable using 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
  + Unsolveable
* Keep the boards for these tests relatively small (<= 5 spaces) to make debugging easier

1. Write your function solve\_puzzle(board) in solve\_puzzle.py. See some tips following the figures.

* The board parameter is a list containing the numbers of the board. The first number in the list is the starting point with other numbers going around in a clockwise direction. Lists examples are given in the example figures.
* The function returns True if the board is solveable and False if it is not. For extra credit: return the list of moves to make reflecting the indices being linked. For example, the solution to the example below would be: [0, 3, 2, 6, 4, 7]. Clearly, the solution always starts with zero and ends with the length of the board minus one.

A diagram of a circular object

Description automatically generated

A diagram of a game

Description automatically generated

A blue circle with red lines

Description automatically generated

**Tips:**

* You will need memoization 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. Rather, use None and check if the parameter is set to None and create an empty mutable collection.

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