HW 05: A Maze without Walls (part 2)

In this homework, we will extend our code from the lab to make it do a couple more tricks including the ability to solve it with dynamic programming. We will write a new class `PuzzleDP` that will extend our class from the homework. Put the new class in a file called `puzzle_dp.py`. For your submission, be sure to zip together both your `puzzle.py` file and also `puzzle_dp.py`.

Find the path

In the lab, we used recursion to find if a path exists from `start` to `end` in the puzzle. Now, we're going to write a new method called `pathsolve` that returns a valid path from `start` to `end`. The code will work mostly like the code from the lab. The tricky part is to understand how the path is constructed as the recursive calls return.

As it is quicker to append to a list, you can append the current state to the list as you return from recursive calls and reverse the final list. The other possibility is to use a deque. Python has an efficient, built-in `deque` data structure that supports adding to the front with a method called `appendleft`. Don't forget to import it: `from collections import deque`.

Begin from the end

A classic trick for solving a maze is to start from the end and work backwards. It's a little tricky to do this with this puzzle, because one square can be reached by many other squares.

Write a method called `reverse` that returns a dictionary mapping positions ((row,column)-tuples) to sets of positions that can reach that position. Using this dictionary, it will be possible to find paths by going backwards.

Find distances using dynamic programming

Now, we're going to modify our old approach to remove the recursion entirely. Here's the idea. We're going to build a table of solutions that is like the original puzzle grid, but each entry stores the distance to the `end`. After we have filled up the whole table, we will be able to quickly return the

length of the shortest path from a given position to `end`.

Override the `solve` method from the lab so that it uses the `distances` method. Notice that you can now handle much larger instances than would work for the recursive version. For example, the following example is no problem.

```
# 100 x 100 grid of all ones.
p = PuzzleDP([[1]*100]*100)
p.solve((0,0), ((99,99)))
```

Such an example easily overflows the stack in the recursive version of the problem.

Summary:

You `PuzzleDP` class should extend `Puzzle` from lab.
It should implement `solve`, `pathsolve`, `distances`, and `reverse`.
You should not need to modify the provided `distances` code. Your code should be included in a module `puzzle_dp.py`.