Feedback — Homework 4

Help Center

Thank you. Your submission for this homework was received.

You submitted this homework on **Thu 17 Mar 2016 1:03 AM CST**. You got a score of **100.00** out of **100.00**.

"*That which does not kill us makes us stronger.*" - Friedrich Nietzsche

The original Fifteen puzzle

For our final mini-project, we will build a program that solves the Fifteen puzzle which was popularized in the 1800's by Loyd. The original version of the Fifteen puzzle consists of fifteen tiles numbered one to fifteen that lie on a 4×4 grid. Since there are sixteen spaces in the grid, one of the spaces is empty.

This version of the puzzle starts in an initial configuration where the tiles numbered one to four lie on the top row (left to right), tiles five to eight lie on the next row down, tiles nine to twelve lie on the third row, and tiles thirteen to fifteen lie on the bottom row with the blank space starting in the lower right corner.

To manipulate the puzzle, the player can slide a tile adjacent to the blank space into the blank space. Repeating this process scrambles the order of the tiles. The challenge in the Fifteen puzzle is to bring the tiles back into its initial configuration.

A modified version of the Fifteen puzzle

Before discussing how to solve the Fifteen puzzle, first, we will make a minor modification to the game design of the puzzle that simplifies reasoning about the game. In the original game, the tiles are numbered one to fifteen with the blank space lying at the lower right. In this version, the blank space can be viewed as logically corresponding to the number sixteen.

From a computational point of view, this design has two drawbacks: the tile numbers start at one and the number logically associated with the blank space depends on the size of the puzzle. A better design would be to start with the blank space in the upper left-hand corner and associate the number zero with it. Now, the remaining fifteen tiles are placed as usual in ascending order from left to right and top to bottom.

This **program template** implements an interactive implementation of this modified version of the Fifteen puzzle. Running the template initializes the puzzle in its *solved* configuration with the blank space (represented by the light blue tile with number zero) in the upper left corner. You can use the arrows keys to swap tile zero with its neighboring tiles. With a fairly small number of key presses, one can scramble the board so that it is difficult to return the board to the solved configuration. Your task for this week is to implement a solver for this modified version of the Fifteen puzzle.

To begin, we suggest that you spend a few minutes examining the template in detail. The main component of the template is a `Puzzle` class that provides the functionality for our template. The contents of the board are represented as a 2D list of numbers. In particular, note that the

number of the tile currently positioned in the row `row` and column `col` of the puzzle is `self._grid[row][col]`. Here, row zero is the topmost row and column zero is the leftmost column with indices increasing from left to right and top to bottom.

Modeling solutions to the Fifteen puzzle

Our goal for both the homework and mini-project will be to add several methods to the `Puzzle` class that automatically solve the puzzle after it has been scrambled. The solution will be a sequence of tile swaps between tile zero and its neighbors. At this point, we must make a critical decision: *How will we represent solutions to the puzzle?*

One possibility would be to record the path of tile zero as it moves through the puzzle during the solution process. This choice is awkward for two reasons: the position of tile zero is a pair of indices, which requires more indexing. Also, building an inconsistent path for the tile is possible (i.e; a bug could lead to a path that is not physically possible).

Instead, we have chosen to encode the path of tile zero as a string whose entries are either `"l"`, `"r"`, `"u"`, and `"d"`. These letters correspond to swaps of tile zero with its left, right, up, and down neighbors, respectively. This design choice will allow us to specify relative movements of tile zero without worrying about its position and record these movements more compactly as a string.

Moreover, illegal movements such as attempting to swap tile zero across the border of the puzzle can be handled easily, allowing for simpler debugging. To experiment with this model, use the arrow keys to move tile zero around the puzzle and then click the "Print moves" button. The corresponding movement string will be printed in the console. We recommend that you use this functionality when creating your own puzzle configurations for testing.

Question 1

Understanding move strings

A 4×4 puzzle in its solved configuration is shown below. Which configuration is the result of applying the move string `"drdr"` to the puzzle?

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Your Answer	Score	Explanation
○		

1	2	3	0
4	5	6	7
8	9	10	11
12	13	14	15

1	5	2	3
4	6	10	7
8	9	0	11
12	13	14	15

0	4	2	3
5	1	6	7
8	9	10	11
12	13	14	15

4	1	2	3
5	9	6	7
8	10	0	11
12	13	14	15

✔ 10.00 Yes, this is the configuration after applying `"drdr"`.

Total 10.00 / 10.00

Question 2

Which move string updates the puzzle from the configuration shown on the left to the configuration shown on the right?

1	2	3	7
5	4	9	6
8	0	10	11
12	13	14	15

5	1	2	7
4	8	3	6
0	9	10	11
12	13	14	15

Note that on the left, the tiles ten to fifteen are in their correct locations. On the right, tile nine has also been moved to its correct location.

Hint: From the solved (initial) configuration, enter the move string `"ddrdrudlulurrrlldluurrrdllldr"` in the input field to generate the left configuration.

Your Answer		Score	Explanation
○ `"urrullldduld"`			
○ `"ruldrulld"`			
○ `"ruldrul"`			
◉ `"urulldduld"`	✔	10.00	Yes, this move string places tile nine in its correct location.
Total		10.00 / 10.00	

Question 3

Solving a *2 x 2* puzzle

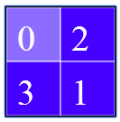
For the next three problems, we will focus on exploring the behavior of a 2×2 puzzle. The size of the puzzle is passed to the initializer for the `Puzzle` class as a height and a width. Modify the last line of the template to create a 2×2 puzzle.

Now, from the solved configuration, enter the move string `"rdlu"` repeatedly. How many times do you need to enter this string to return the puzzle to its solved configuration?

Your Answer		Score	Explanation
○ 5			
○ 1			
○ 4			
◉ 3	✔	10.00	Yes, applying this move string three times returns the puzzle to its solved configuration.
Total		10.00 / 10.00	

Question 4

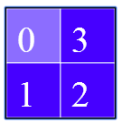
Starting from the configuration shown below, which move string returns the 2×2 puzzle to its solved configuration?



Your Answer		Score	Explanation
◉ `"rdlu"`	✔	10.00	Yes. This string returns the puzzle to its solved configuration.
○ `"rlrl"`			
○ `"dudu"`			
○ `"drul"`			
Total		10.00 / 10.00	

Question 5

For configuration shown below, which of the following move strings return the puzzle to its solved configuration?



Your Answer		Score	Explanation
☐ `"druldrul"`	✔	2.50	This move string does not return the puzzle to its solved configuration.
☐ `"rdlu"`	✔	2.50	This move string does not return the puzzle to its solved configuration.
☑ `"rdlurdlu"`	✔	2.50	This move string also returns the puzzle to its solved configuration.

☐ `""`	✔	2.50	This string does nothing to the puzzle.
Total		10.00 / 10.00	

Question 6

The overall strategy for solving the Fifteen puzzle

With the preliminaries out of the way, we now describe how to solve the general $m \times n$ version of the Fifteen puzzle. The solution process consists of repeatedly repositioning tiles into their solved positions in the puzzle. We refer to each instance of this process as "solving" for a tile at a specified position in the puzzle.

The solution process has three phases:

1. We first solve the bottom $m - 2$ rows of the puzzle (in a bottom to top order) to reduce the problem to that of solving a $2 \times n$ puzzle.
2. We then solve the rightmost $n - 2$ columns of the top two rows of the puzzle (in a right to left order) to reduce the problem to that of solving a 2×2 puzzle.
3. Finally, we then solve this 2×2 puzzle based on the observations in problems #3-5.

Invariants for the Fifteen puzzle

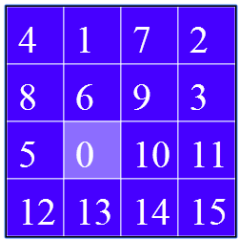
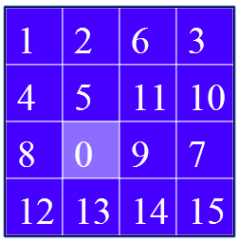
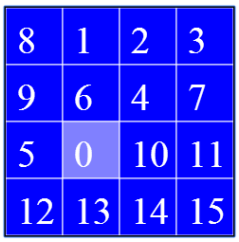
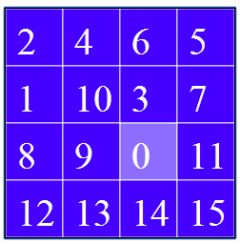
In the next four problems, we will explore one particular strategy for implementing phase one. The key to this strategy will be to develop an invariant that reflects the state of the puzzle during phase one and then implement solution methods that maintain this invariant.

Phase one will have one invariant `lower_row_invariant(i, j)` which is true prior to solving for the tile at position (i, j) (where $i > 1$). This invariant consists of the following conditions:

- Tile zero is positioned at (i, j).
- All tiles in rows $i + 1$ or below are positioned at their solved location.
- All tiles in row i to the right of position (i, j) are positioned at their solved location.

Problem: Which of the configurations below satisfy the invariant `lower_row_invariant(2, 1)`?

Your Answer		Score	Explanation
☑	✔	3.00	This configuration satisfies the invariant.

			
☐ 	✔	2.00	The tiles to the right of the zero tile are not solved.
☑ 	✔	3.00	This configuration satisfies the invariant.
☐ 	✔	2.00	This configuration has the tile zero in the wrong position.
Total		10.00 / 10.00	

Question 7

Solving for tiles in the lower rows

In phase one, we will implement two solution methods for positions in the lower rows. The method `solve_interior_tile(i, j)` will solve for all positions except for those in the left column ($j > 0$). The method `solve_col0_tile(i)` will solve for positions in the leftmost column.

The solution method `solve_interior_tile(i, j)` is related to the invariants as follows: If the invariant `lower_row_invariant(i, j)` is true prior to execution of `solve_interior_tile(i, j)`, the invariant `lower_row_invariant(i, j - 1)` should be true after execution of this method. In short, the solution method should update the puzzle so

the invariant is still true.

Following the examples in the notes on invariants, the execution trace of the solver can be annotated with assertions of the form:

```
...
assert my_puzzle.lower_row_invariant(i,j)
my_puzzle.solve_interior_tile(i, j)
assert my_puzzle.lower_row_invariant(i, j - 1)
...
```

where `my_puzzle` is the name of the puzzle being solved.

Problem: Which annotated execution trace captures the relationship between the solution method `solve_col0_tile` and the invariant `lower_row_invariant`? Remember that once the entire ith row is solved, the solution process then proceeds to the rightmost column of the $i-1$st row. You may assume that the puzzle is $m \times n$.

Your Answer	Score	Explanation
○ ... assert my_puzzle.lower_row_ invariant(i, 0) my_puzzle.solve_interior_tile (i, 0) assert my_puzzle.lower_row_ invariant(i - 1, n - 1) ...		
◉ ... assert my_puzzle.lower_row_ invariant(i, 0) my_puzzle.solve_col0_tile(i) assert my_puzzle.lower_row_ invariant(i - 1, n -1) ...	✔ 10.00	Correct. The solution process should continue on the last tile on the next row up.
○ ... assert my_puzzle.lower_row_ invariant(i, 0) my_puzzle.solve_col0_tile(i) assert my_puzzle.lower_row_ invariant(i, n - 1) ...		

```
...
assert my_puzzle.lower_row_
invariant(i, 0)
my_puzzle.solve_col0_tile(i)
assert my_puzzle.lower_row_
invariant(i - 1, n)
...
```

Total	10.00 / 10.00

Question 8

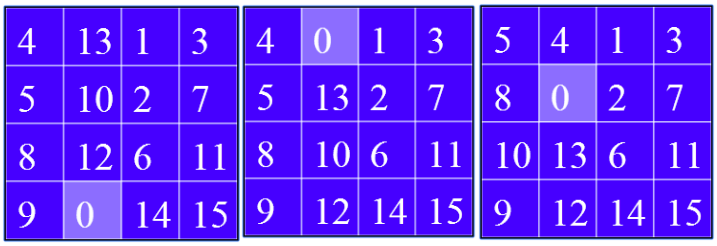
Implementing `solve_interior_tile`

We are now ready to formulate the basic algorithm for `solve_interior_tile(i, j)`. Given a target position (i, j), we start by finding the current position of the tile that should appear at this position to a solved puzzle. We refer to this tile as the *target tile*.

While moving the target tile to the target position, we can leverage the fact that `lower_row_invariant(i, j)` is true prior to execution of `solve_interior_tile(i, j)`. First, we know that the zero tile is positioned at (i, j). Also, the target tile's current position (k, l) must be either above the target position ($k < i$) or on the same row to the left ($i = k$ and $l < j$).

Our solution strategy will be to move the zero tile up and across to the target tile. Then we will move the target tile back to the target position by applying a series of cyclic moves to the zero tile that move the target tile back to the target position one position at a time. Our implementation of this strategy will have three cases depending on the relative horizontal positions of the zero tile and the target tile.

The three images below show an example in which the target tile (with number 13) is directly above the target position. The left image shows the configuration at the start of `solve_interior_tile(3, 1)`, the middle image shows the configuration after the zero tile has been moved to the target tile's current position using the move string `"uuu"`, and the right image shows the configuration after the target tile has been moved down one position towards the target position using the move string `"lddru"`



Problem: Starting from the configuration on the right, which move string completes the solution process for this position and updates the puzzle to a configuration where `lower_row_invariant(3, 0)` is true?

Your Answer		Score	Explanation
◉ `"lddruld"`	✔	10.00	Yes. Repeat the previous move string and then move the zero tile left and down. Note that we moved the zero tile around the left of the target tile to ensure that we did not disturb previously solved tiles.
○ `"rddlu"`			
○ `"lddru"`			
○ `"lddrulddru"`			
Total		10.00 / 10.00	

Question Explanation

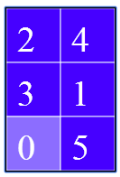
You can reach the left configuration from the unscrambled configuration via the move string `"dddruldrulurdruulddd"`.

Question 9

Solving a *3 x 2* puzzle

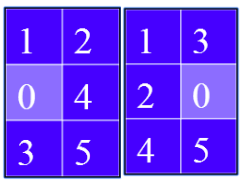
Our solution strategy for `solve_interior_tile` fails for positions in the leftmost column of the puzzle since we lack a free column on the left of the target position. For the leftmost column, the method `solve_col0_tile` will use a solution process that is similar to that of a 3×2 puzzle.

As a motivating example, imagine that we have used `solve_interior_tile(2, 1)` to position the five tile correctly. The example below shows a typical configuration that satisfies `lower_row_invariant(2, 0)`.



The problem here is that, unless the four tile happens to be above the zero tile, there is no way to swap the four tile into its correct position without temporarily moving the five tile.

In this case, the solution is move the zero tile up and to right and then reposition the four tile above the five tile with the zero tile to its left. The move string for this update can be generated in a manner similar to the process used in `solve_interior_tile`. The configuration on the left below shows the result of this process.



From this left configuration, we can apply a fixed move string that generates the configuration shown on the right in which the four and five tiles are at their desired location while leaving the zero tile above the five tile.

Problem: Which move string below updates the puzzle from the left configuration into the right configuration above?

Your Answer		Score	Explanation
`"rdlurdlu"`			
`"druldruldru"`			
`"r"`			
`"ruldrdlurdluurddlur"`	✔	10.00	Yes, this move string updates the puzzle as desired.
Total		10.00 / 10.00	

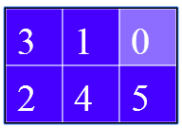
Question Explanation

You may reach the topmost configuration via the move string `"drdluruldrdluruldrulddruld"`. From there, you can reach the lower left configuration via the string `"uruld"`.

Question 10

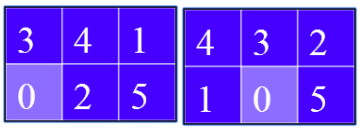
Solving a *2 x 3* puzzle

In phase two, we solve the top two rows of the puzzle, one column at a time from right to left. The basic strategy here is similar to that of solving a 3×2 puzzle. In the configuration below, we have already positioned the five tile correctly, with the zero tile positioned above it.



To solve the right column of the puzzle, we must correctly position the two tile next. The issue here is that, unless the two tile ends up to the left of the zero tile, there is no way to swap the two tile into its correct position without temporarily moving the five tile.

In this case, the solution is to move the zero tile over and down and then use a variant of `solve_interior_tile` to position the two tile at $(1, 1)$ with the zero tile at $(1, 0)$.



From this left configuration, we can apply a fixed move string that generates the configuration shown on the right. In this configuration, the two and five tiles are in their desired positions with the zero tile positioned to the left of the five tile, ready for the nex step in the solution process.

Problem: Which move string below updates the left configuration into the right configuration?

Your Answer		Score	Explanation
◉ `"urdlurrdluldrruld"`	✔	10.00	Yes. This move string performs the desired update.
○ `"r"`			
○ `"urrdl"`			
○ `"urrd"`			
Total		10.00 / 10.00	

Question Explanation

You can reach the topmost configuration via the move string `"rrdluldrru"`. From there, you can reach the middle configuration via the string `"ldl"`.