Homework 4

Retourner à la semaine 4

10/10 points obtenus
(100%)

Quiz réussi !

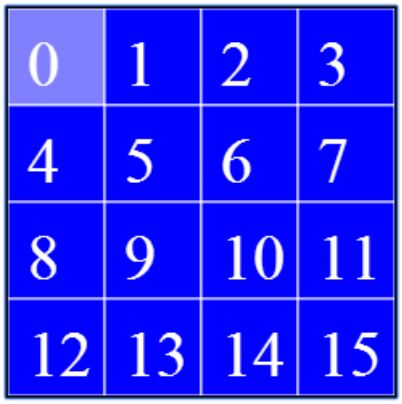
1 / 1
points

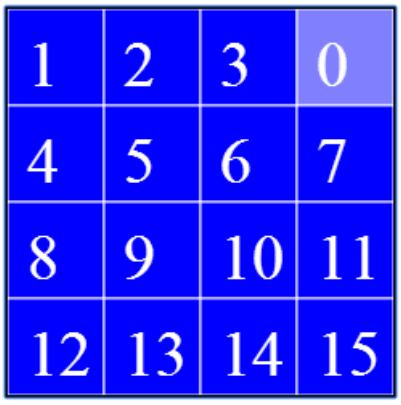
1.
"That which does not kill us makes us stronger." - Friedrich Nietzsche

This final homework will introduce you to the process of creating an automated solver for the Fifteen puzzle (our final mini-project). If you have not already done so, you should review this class reading on the basics of the Fifteen puzzle and only then attempt the homework. To begin, we consider how to model solutions to the puzzle.

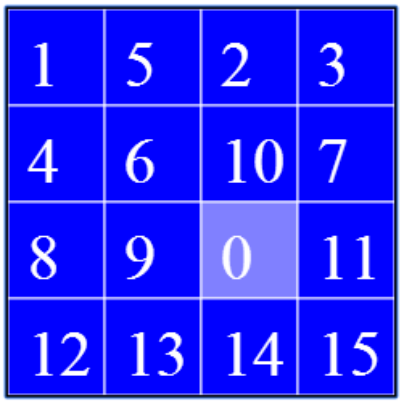
Understanding move strings

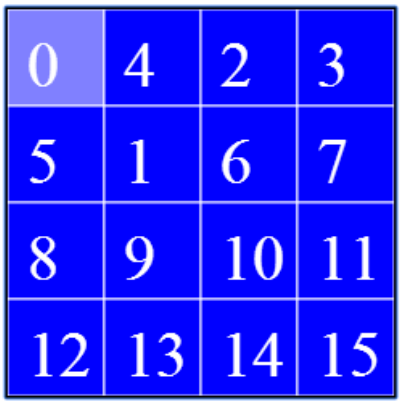
A 4×4 puzzle in its solved configuration is shown below. Which configuration is the result of applying the move string `"drdr"` to the puzzle?

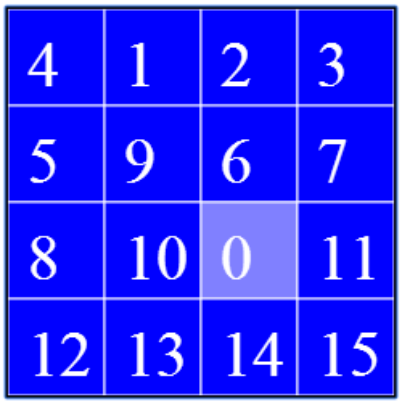


- 

-







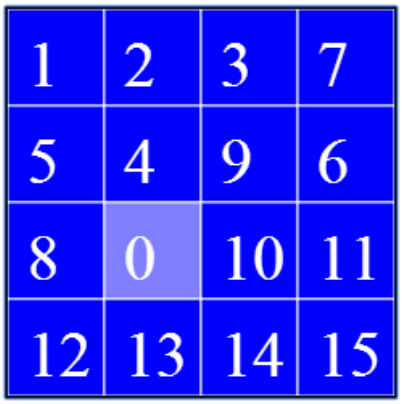
Correct
Yes, this is the configuration after applying "`drdr`".

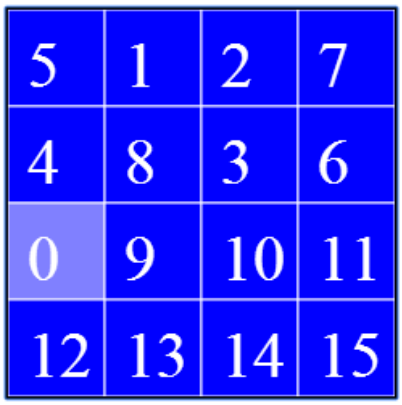
---

1 / 1 points

2.

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





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.

- "`ruldrul`"
- "`urrullldduld`"
- "`ruldrulld`"
- "`urullddruld`"

Correct
Yes, this move string places tile nine in its correct location.

---

1 / 1 points

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?

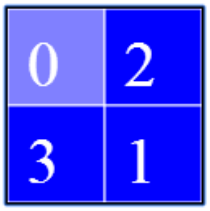
- 5
- 2
- 1
- 3 ✓

Correct
Yes, applying this move string three times returns the puzzle to its solved configuration.

---

1 / 1 points

4.
Starting from the configuration shown below, which move string returns the 2×2 puzzle to its solved configuration?



- "rdlu" ✓

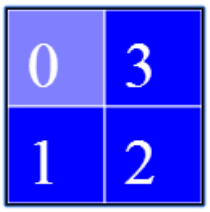
Correct
Yes. This string returns the puzzle to its solved configuration.

- "drul"
- ""
- "rlrl"

---

1 / 1 points

5.
For configuration shown below, which of the following move strings return the puzzle to its solved configuration?



- [x] "drul"

Correct
This move string is the shortest string that updates the puzzle to its solved configuration.

☐ "rdul"

L'élément désélectionné est correct

☐ "rdlu"

L'élément désélectionné est correct

☐ "druldrul"

L'élément désélectionné est correct

✔ 1 / 1 points

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

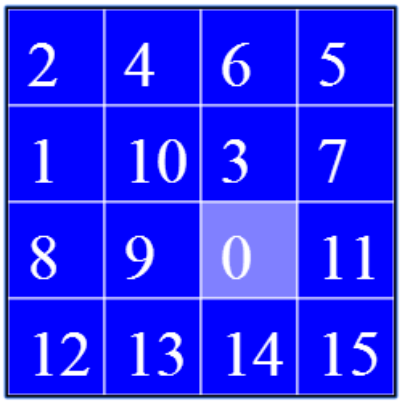
In the next four problems, we will explore one particular strategy for implementing phase one. The key to this strategy will be to develop an invariant that reflects the state of the puzzle during phase one and then implement solution methods that maintain this invariant.

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:

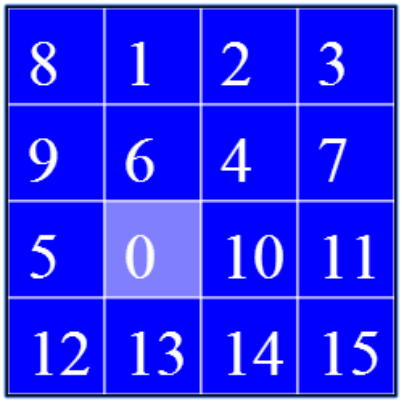
- Tile zero is positioned at (i, j).
- All tiles in rows $i + 1$ or below are positioned at their solved location.
- All tiles in row i to the right of position (i, j) are positioned at their solved location.

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

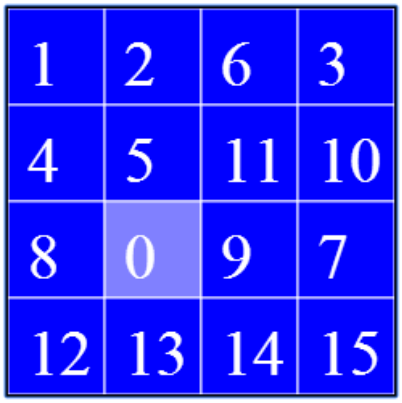
☐



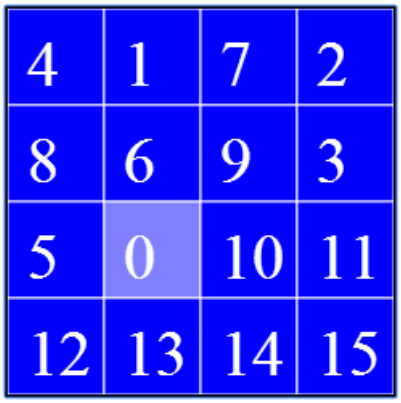
L'élément désélectionné est correct

☐ 

Correct
This configuration satisfies the invariant.

☐ 

L'élément désélectionné est correct

☐ 

Correct
This configuration satisfies the invariant.

---

1 / 1 points

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 $m \times n$.

```
...
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_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)
...
```

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

Correct
Correct. The solution process should continue on the last tile on the next row up.

1 / 1 points

8.

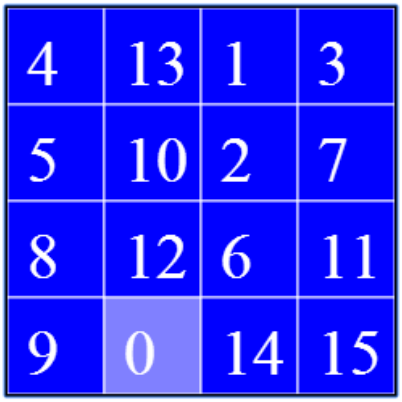
Implementing `solve_interior_tile`

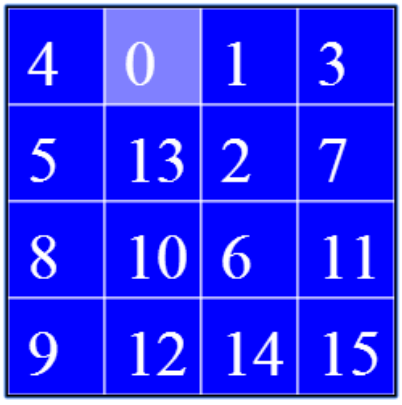
We are now ready to formulate the basic algorithm for `solve_interior_tile(i, j)`. Given a target position (i, j), we start by finding the current position of the tile that should appear at this position to a solved puzzle. We refer to this tile as the *target tile*.

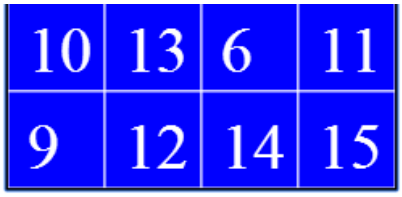
While moving the target tile to the target position, we can leverage the fact that `lower_row_invariant(i, j)` is true prior to execution of `solve_interior_tile(i, j)`. First, we know that the zero tile is positioned at (i, j). Also, the target tile's current position (k, l) must be either above the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

- "lddruld"

Correct
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.

- "lddrulddru"
- "rddlu"
- "lddru"

1 / 1 points

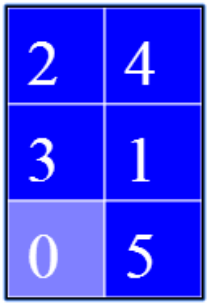
9.

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.

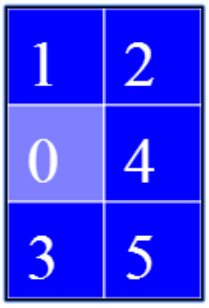
For the leftmost column, the method `solve_col0_tile` will use a solution process that is similar to that of a 3×2 puzzle.

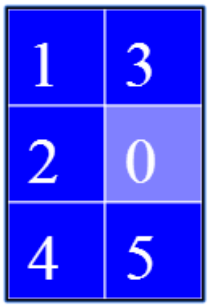
As a motivating example, imagine that we have used `solve_interior_tile`$(2, 1)$ to position the five tile correctly. The example below shows a typical configuration that satisfies `lower_row_invariant`$(2, 0)$.



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?

- `"ruldrdlurdluurddlur"`

Correct
Yes, this move string updates the puzzle as desired.

- "r"
- "rdlurdlu"
- "druldruldru"

---

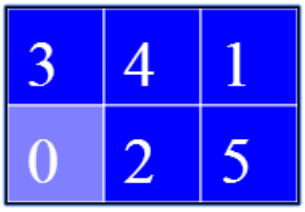
1 / 1 points

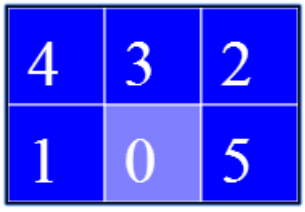
10.
Solving a *2 x 3* puzzle

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?

- "urdlurrdluldrruld"

Correct
Yes. This move string performs the desired update.

- "urrd"
-