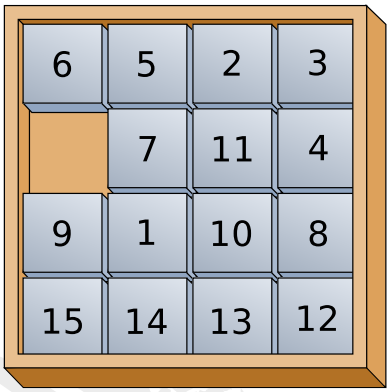
# Homework 1: Solving the Sliding Puzzle With A*

Your job is to create an AI to solve a "sliding puzzle" (like the 15-puzzle or 9-puzzle) using the A* algorithm. Given any puzzle, you must find the sequence of moves to put all the numbers in order, with the gap in the lower-right square.




**A 15-Puzzle.** This sliding puzzle is solvable in 31 moves.

A program has been provided for you, which is called `slidingpuzzle.py`. If the proper graphics libraries are on your computer, it will display an animated puzzle that will then be solved. The program takes two command-line arguments:

- The name of a file containing an unsolved puzzle.
- Optionally, either `-n` or `--nographics`. If one of these is present, it will turn off graphics and and display the solution in the terminal.

The `.puz` files it accepts are text files of $m \times n$ puzzles, containing the proper numbers separated by spaces. The gap in the puzzle is represented by the `.` character. For example, the puzzle in the figure would be:

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

Your job is to create a Python file called `solver.py`. This file will have one public function, called `solve()`. `solve()` will take one argument: a 2D row-major list containing a puzzle. The gap will be represented by a `0`. It will return one of two things:

- If the puzzle is not solvable, it will return `None`.
- If the puzzle is solvable, it will return a solution of minimum length. This will be a list of `str`s, each of which is either `'U'`, `'D'`, `'L'`, or `'R'`, representing an "up", "down", "left", or "right" move. For the puzzle shown here, one possible solution is:
  `['U','U','L','L','D','D','R','U','R','U','L','D','R','D','D','L','U',`
  `'R','D','L','L','L','U','U','R','U','R','D','L','L','U']`

Since you are using A*, you will probably also want to create a `State` class, which designates a state of the puzzle (that is, a current layout of all the tiles). Because you will be putting this `State` inside of a closed list, it will need to be hashable for efficiency. This means that you will need to make sure that your `State` object includes a "magic" `__hash__()` method. The `heapq` module will also be useful, as it provides functions to alter a list as if it were a priority queue.

A* requires an admissible heuristic, that estimates how close any given state is to the goal state. An easy heuristic to use is to relax the problem so that tiles can move through each

other, and count the number of moves that would be necessary. For example, in the given puzzle the `1` tile must move left once and up twice, for 3 moves. Adding this up for every tile, we find that this heuristic estimates this state to be 19 moves away from the goal state. (Although this heuristic works for small puzzles, it can be a problem for larger puzzles and puzzles with longer solutions. See below.)

A sliding puzzle contains 2 orbits, each of which is a collection of states that are mutually reachable from each other via any number of moves. Each is the same size. This means that any random state is only 50% likely to be solvable. Here is an algorithm to determine if the goal state is reachable from any arbitrary state.

1. Flatten the 2D list into a 1D list and remove the `0`. So the given puzzle would become: `[6, 5, 2, 3, 7, 11, 4, 9, 1, 10, 8, 15, 14, 13, 12]`
2. Count the number of *inversions* in this list. This is the number of elements *after* a given element that are *less than* that element. So in this case, tile `6` has 5 values after it, that are less than it. Tile `10` has only 1 such value. The total number of inversions here is 28.
3. If the puzzle's width is odd, the puzzle is solvable if and only if the total number of inversions is even.
4. If the puzzle's width is even, you must add the number of inversions to the number of rows away from the bottom that the gap must move. (So if the gap is in the bottom row, add 0. If the gap is in the penultimate row, add 1. And so on.) The puzzle is solvable if and only if this sum is even. Thus, the given puzzle is solvable since $28 + 2 = 30$.

The reason this works is because horizontal moves don't change the number of inversions, while vertical moves do. A vertical move will change the number of inversions by an odd number if the puzzle width is even, and by an even number if the width is odd. The goal state has 0 inversions.

- For odd-width puzzles, no combination of moves can possibly change the number of inversions from even to odd, or vice versa. Thus if the initial number of inversions is odd, it is impossible to reach the goal state.
- For even-width puzzles, we add the number of vertical moves the gap must make. This value is 0 for the goal state, and any vertical move will change this sum by an even number. Thus if the initial sum is odd, no combination of moves can possibly reach the goal state.

**Extra credit:** There are better heuristics, that provide a better estimation of distance to goal, that are still admissible. This will allow your solver to solve more complicated and larger puzzles. Find one of these heuristics online, and implement it.