Submitting Information:

- Use the code I provided for each problem. DON'T DELETE ANY FUNCTION
- Submit your work on Canvas.
- The deadline is Dec 7th at 6:00PM
- Follow the guidelines in Project 4 rubric

The Problem. Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
4   2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
7   8  6        7  8  6        7  8  6        7  8  6        7  8

 initial                                                     goal
```

Best-first search. We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *state* of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of *priority function* for a state. We consider two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the state. Intuitively, a state with a small number of blocks in the wrong position is close to the goal state, and we prefer a state that have been reached using a small number of moves.
- *Manhattan priority function.* The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the state.

For example, the Hamming and Manhattan priorities of the initial state below are 5 and 10, respectively

```
8  1  3        1  2  3     1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
4     2        4  5  6     ----------------------     ----------------------
7  6  5        7  8        1  1  0  0  1  1  0  1     1  2  0  0  2  2  0  3

 initial          goal          Hamming = 5 + 0           Manhattan = 10 + 0
```

We make a key oberservation: to solve the puzzle from a given state on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank tile when computing the Hamming or Manhattan priorities.)

Consequently, as soon as we dequeue a state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

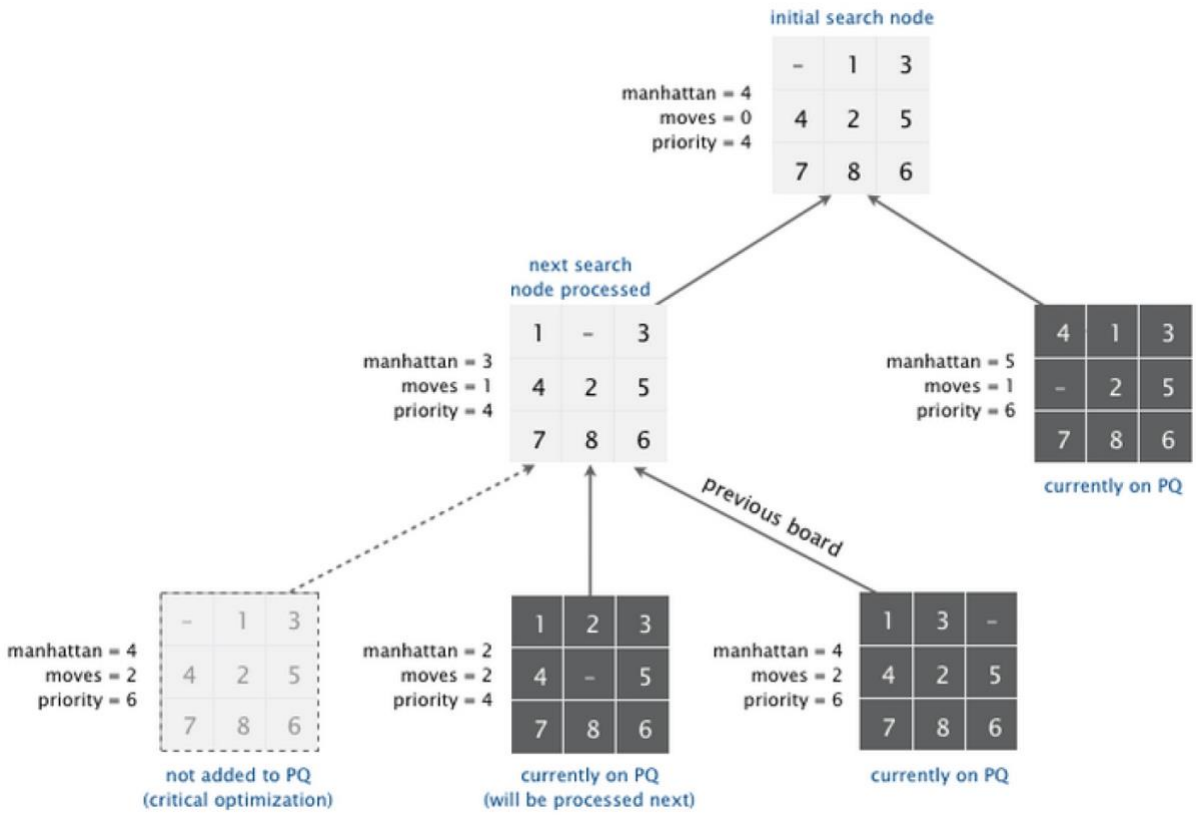
A critical optimization. After implementing best-first search, you will notice one annoying feature: states corresponding to the same board position are enqueued on the priority queue many times. To prevent unnecessary exploration of useless states, when considering the neighbors of a state, don't enqueue the neighbor if its board position is the same as the previous state.

```
 8  1  3       8  1  3      8  1  3
 4     2       4  2         4     2
 7  6  5       7  6  5      7  6  5

 previous       state       disallow
```

A Second Optimization To avoid recomputing the Manhattan distance of a board (or, alternatively, the Manhattan priority of a solver node) from scratch each time during various priority queue operations, compute it at most once per object; save its value in an instance variable; and return the saved value as needed. This caching technique is broadly applicable: consider using it in any situation where you are recomputing the same quantity many times and for which computing that quantity is a bottleneck operation.

Game Tree One way to view the computation is as a game tree, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a priority queue; at each step, the A* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).



Detecting Unsolvable Puzzles Not all initial boards can lead to the goal board by a sequence of legal moves, including the two below:

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

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: those that lead to the goal board; and those that cannot lead to the goal board. Moreover, we can identify in which equivalence class a board belongs without attempting to solve it.

- Odd board size. Given a board, an inversion is any pair of tiles i and j where i < j but i appears after j when considering the board in row-major order (row 0, followed by row 1, and so forth).

```
1 2 3             1 2 3             1 2 3             1 2 3             1 2 3
4 5 6   =>        4 5 6 =>          4   6   =>          4 6   =>        4 6 7
8 7               8   7             8 5 7             8 5 7             8 5

1 2 3 4 5 6 8 7   1 2 3 4 5 6 8 7   1 2 3 4 6 8 5 7   1 2 3 4 6 8 5 7   1 2 3 4 6 7 8 5

inversions = 1    inversions = 1    inversions = 3    inversions = 3    inversions = 3
         (8 -7)            (8 -7)   (6 -5 8-5 8 -7)   (6 -5 8-5 8 -7)   (6 -5 7-5 8 -5)
```

If the board size N is an odd integer, then each legal move changes the number of inversions by an even number. Thus, if a board has an odd number of inversions, then it cannot lead to the goal board by a sequence of legal moves because the goal board has an even number of inversions (zero).

The converse is also true: if a board has an even number of inversions, then it can lead to the goal board by a sequence of legal moves.

```
          1 3               1   3               1 2 3               1 2 3             1 2 3
        4 2 5         =>    4 2 5         =>    4   5       =>      4 5       =>      4 5 6
        7 8 6               7 8 6               7 8 6               7 8 6             7 8
1 3 4 2 5 7 8 6   1 3 4 2 5 7 8 6   1 2 3 4 5 7 8 6   1 2 3 4 5 7 8 6   1 2 3 4 5 6 7 8

inversions = 4    inversions = 4    inversions = 2    inversions = 2    inversions = 0
(3 -2 4-2 7-6 8 -6)(3 -2 4-2 7-6 8 -6)  (7 -6 8 -6)       (7 -6 8 -6)
```

- Even board size. If the board size N is an even integer, then the parity of the number of inversions is not invariant. However, the parity of the number of inversions plus the row of the blank square is invariant: each legal move changes this sum by an even number. If this sum is even, then it cannot lead to the goal board by a sequence of legal moves; if this sum is odd, then it can lead to the goal board by a sequence of legal moves.

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

 blank row = 1     blank row = 1     blank row = 2     blank row = 2     blank row = 3
inversions = 6    inversions = 6    inversions = 3    inversions = 3    inversions = 0
--------------    --------------    --------------    --------------    --------------
       sum = 7           sum = 7           sum = 5           sum = 5           sum = 3
```

Problem 1. (Board Data Type) Create an immutable data type Board in Board.java with the following API:

method	description
`Board(int[][] tiles)`	constructs a board from an N-by-N array of tiles
`int tileAt(int i, int j)`	returns tile at row i, column j (or 0 if blank)
`int size()`	returns board size N
`int hamming()`	returns number of tiles out of place
`int manhattan()`	returns sum of Manhattan distances between tiles and goal
`boolean isGoal()`	is this board the goal board?
`boolean isSolvable()`	is this board solvable?
`boolean equals(Board that)`	does this board equal *that*?
`Iterable<Board> neighbors()`	returns all neighboring boards
`String toString()`	returns string representation of this board (in the output format specified below)

Performance requirements. You may assume that the constructor receives an N-by-N array containing the N^2 integers between 0 and N^2 - 1, where 0 represents the blank square. Your implementation should support all Board methods in time proportional to N^2 (or better) in the worst case, with the exception that isSolvable() may take up to N^4 in the worst case.

```
$ java-algs4 Board data/puzzle05.txt
5
5
false
true
3
0 1 3
4 2 6
7 5 8
3
4 1 3
2 0 6
7 5 8
3
4 1 3
7 2 6
0 5 8
```

Problem 2. (Solver Data Type) Create an immutable data type Solver in Solver.java with the following API:

method	description
`Solver(Board initial)`	finds a solution to the initial board (using the $A^\star$ algorithm)
`int moves()`	returns the minimum number of moves to solve initial board
`Iterable<Board> solution()`	returns a sequence of boards in a shortest solution

Corner cases. The constructor should throw a java.lang.NullPointerException if the initial board is null and a java.lang.IllegalArgumentException if the initial board is not solvable.

Your program should work correctly for arbitrary N-by-N boards (for any 1 <= N <= 32768), even if it is too slow to solve some of them in a reasonable amount of time.

```
$ java-algs4 Solver data/puzzle05.txt
Minimum number of moves = 5
3
0 1 3
4 2 6
7 5 8
3
1 0 3
4 2 6
7 5 8
3
1 2 3
4 0 6
7 5 8
3
1 2 3
4 5 6
7 0 8
3
1 2 3
4 5 6
7 8 0
```

Data The data directory contains a number of sample input les representing boards of different sizes. The input (and output) format for a board is the board size N followed by the N-by-N board, using 0 to represent the blank square.

```
$ more data/puzzle04.txt
3
0 1 3
4 2 5
7 8 6
```

Acknowledgements *This project is an adaptation of the 8 Puzzle assignment developed at Princeton University by Robert Sedgewick and Kevin Wayne*

Hints and Clarifications

Project goal: write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm

Problem 1.
Instance variables

- Tiles in the board, `int[][] tiles`
- Board size, `int N`
- Hamming distance to the goal board, `int hamming`
- Manhattan distance to the goal board, `int manhattan`

Helper method `int blankPos()`

- Return the position (in row-major order) of the blank (zero) tile; for example, if N = 3 and the blank tile is in row i = 1 and column j = 2, the method should return 5

Helper method `int inversions()`

- Return the number of inversions

Helper method `int[][] cloneTiles()`

- Clone and return `this.tiles`

Board(int[][] tiles)

- Initialize the instance variables `this.tiles` and `this.N` to tiles and the number of rows in tiles respectively
- Calculate the Hamming and Manhattan distances of this board and the goal board, in the instance variables hamming and `manhattan` respectively

`int tileAt(int i, int j)`

- Return the tile at row i and column j

`int size()`

- Return the board size

`int hamming()`

- Return the Hamming distance to the goal board

`int manhattan()`

- Return the Manhattan distance to the goal board

`boolean isGoal()`

- Return `true` if this board is the goal board, and `false` otherwise

`boolean isSolvable()`

- Return `true` if this board is solvable, and `false` otherwise

`boolean equals(Board that)`

- Return `true` if this board equals that, and `false` otherwise

`Iterable<Board> neighbors()`

- For each possible neighboring board (determined by the position of the blank tile), clone the tiles of this board, exchange the appropriate tile with the blank tile in the clone, make a Board object from the clone, and enqueue it into a queue of Board objects
- Return the queue

Problem 2

Instance variables

- Sequence of boards in a shortest solution, `LinkedStack<Board>` solution
- Minimum number of moves to solve the initial board, `int moves`

Helper `SearchNode` type representing a node in the game tree

- Instance variables: the board represented by this node, `Board board`; number of moves it took to get to this node from the initial node (containing the initial board), `int moves;` and the previous search node, `SearchNode previous`
- `SearchNode(Board board, int moves, SearchNode previous)`: initialize instance variables appropriately

Helper `int HammingOrder.compare(SearchNode a, SearchNode b)`

- Return a comparison of the `a.board.hamming() + a.moves` and `b.board.hamming() + b.moves`

Helper `int ManhattanOrder.compare(SearchNode a, SearchNode b)`

- Return a comparison of the `a.board.manhattan() + a.moves` and `b.board.manhattan() + b.moves`

`Solver(Board initial)`

- Create a `MinPQ<SearchNode>` object pq (using Manhattan ordering), initialize solution, and insert initial search node into pq
- As long as pq is not empty
  - Remove the minimum (call it node) from pq
  - If the board in node is the goal board, obtain moves and solution from it and break
  - Otherwise, iterate over the neighboring boards, and for each neighbor board that is different from the previous, insert a new `SearchNode` object into pq, built using appropriate arguments

`int moves()`

- Return the minimum number of moves to solve the initial board

`Iterable<Board> solution()`

- Return the sequence of boards in a shortest solution

Data

The data directory contains a number of sample input les representing boards of different sizes; the input (and output) format for a board is the board size N followed by the N-by-N board, using 0 to represent the blank square

```
$ more data/puzzle04.txt
3
0 1 3
4 2 5
7 8 6
```

The visualization client `SolverVisualizer` takes the name of a file as command-line argument, and

- Uses your Solver and Board data types to solve the sliding block puzzle defined by the input file
- Renders a graphical animation of your program's output
- Uses the `Board.manhattan()` to display the Manhattan distance at each stage of the solution

```
$ java-algs4 SolverVisualizer data/puzzle04.txt
```