The Problem The purpose of this assignment is to write a program to solve the 8-puzzle problem (and its natural generalizations) using the $A^\star$ 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 Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the $A^\star$ search algorithm. We define a *search node* of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to a goal board. The success of this approach hinges on the choice of *priority function* for a search node. We consider two priority functions:

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

For example, the Hamming and Manhattan priorities of the initial search node 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 observation: To solve the puzzle from a given search node 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 tile that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each tile must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Hamming or Manhattan priorities.) Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

A Critical Optimization Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the previous search node.

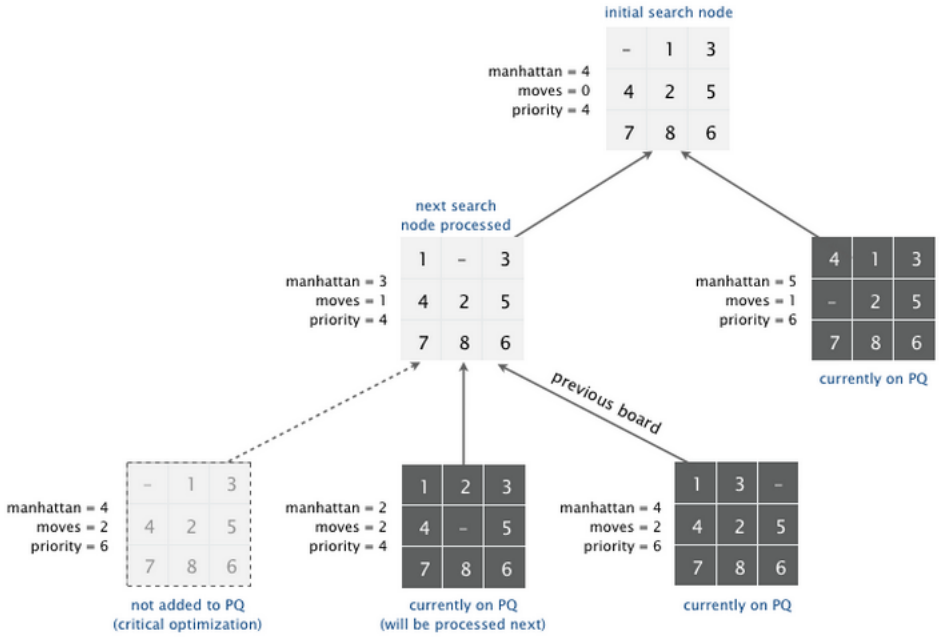
```
8  1  3        8  1  3        8  1        8  1  3     8  1  3
4     2        4  2           4  2  3     4     2     4  2  5
7  6  5        7  6  5        7  6  5     7  6  5     7  6

previous     search node      neighbor    neighbor     neighbor
                                         (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)`	construct a board from an N-by-N array of tiles
`int tileAt(int i, int j)`	tile at row i, column j (or 0 if blank)
`int size()`	board size N
`int hamming()`	number of tiles out of place
`int manhattan()`	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()`	all neighboring boards
`String toString()`	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 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
$ java Board data/puzzle4x4-unsolvable.txt
12
13
false
false
4
 3  2  0  8
 1  6  4 12
 5 10  7 11
 9 13 14 15
4
 3  2  4  8
 1  6 12  0
 5 10  7 11
 9 13 14 15
4
 3  2  4  8
 1  6  7 12
 5 10  0 11
 9 13 14 15
4
 3  2  4  8
 1  0  6 12
 5 10  7 11
 9 13 14 15
```

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

method	description
`Solver(Board initial)`	find a solution to the initial board (using the $A^\star$ algorithm)
`int moves()`	the minimum number of moves to solve initial board
`Iterable<Board> solution()`	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.

```
$ java 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
```

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

Data The `data` directory contains a number of sample input files 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
```

Visualization Client The program `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 SolverVisualizer data/puzzle04.txt
```

Files to Submit

1. `Board.java`
2. `Solver.java`
3. `report.txt`

Before you submit:

- Make sure your programs meet the input and output specifications by running the following command on the terminal:

  ```
  $ python3 run_tests.py -v [<problems>]
  ```

  where the optional argument `<problems>` lists the problems (`Problem1`, `Problem2`, etc.) you want to test, separated by spaces; all the problems are tested if no argument is given.

- Make sure your programs meet the style requirements by running the following command on the terminal:

  ```
  $ check_style <program>
  ```

  where `<program>` is the `.java` file whose style you want to check.

- Make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes

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