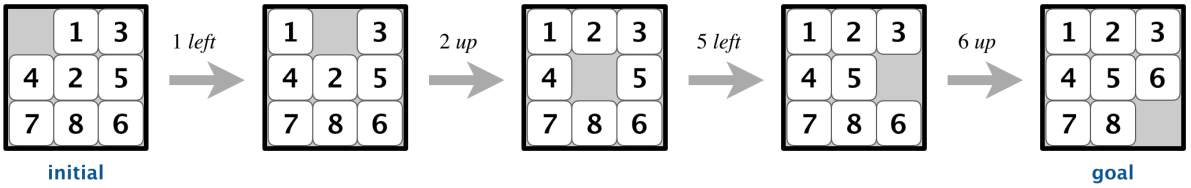
Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

1	7	3
2	5	8
	6	4

The problem. The 8-puzzle is a sliding puzzle that is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8, plus a blank square. The goal is to rearrange the tiles so that they are sorted by label in row-major order, using as few moves as possible. You are permitted to slide tiles either horizontally or vertically into the blank square. The following diagram shows a sequence of moves from an *initial board* (left) to the *goal board* (right).



Board data type. To begin, create a data type that models an *n*-by-*n* board with sliding tiles. Implement an immutable data type Board with the following API:

```
public class Board {

   // create a board from an n-by-n array of tiles,
   // where tiles[row][col] = tile at (row, col)
   public Board(int[][] tiles)

  // string representation of this board
  public String toString()

  // sum of Manhattan distances between tiles and goal
  public int manhattan()

}
```

Note: you are free to add additional methods that you like or need to do so.

Constructor. You may assume that the constructor receives an *n*-by-*n* array containing a permutation of the n^2 integers between 0 and $n^2 - 1$, where 0 represents the blank square. You may also assume that $2 \le n \le 32{,}768$.

String representation. The toString() method returns a string composed of 1 line with all the tiles in row-major order, using 0 to designate the blank square, with tiles separated by spaces ***only.***

board

1 0 3 4 2 5 7 8 6

string representation

Manhattan distance. To measure how close a board is to the goal board, we define a notion of distance. The *Manhattan distance* between a board and the goal board is the sum of the Manhattan distances (sum of the vertical and horizontal distance) from the tiles to their goal positions.

board

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

Manhattan = 10
(1 + 2 + 2 + 2 + 3)

goal

Unsolvable boards. Note that not all boards are solvable (see the additional material for sufficient and necessary conditions that make a board solvable). For the sake of the project, assume that only solvable boards are given in input. Note: make sure that you use solvable boards to test your implementation!

A* search. Now, we describe a solution to the 8-puzzle problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *search node* of the game to consist of the following elements: a board, the number of moves made to reach the board, and a pointer to its parent in the game tree (defined below). First, insert the initial search node (the initial board, 0 moves, and a null pointer) into a priority queue. Then, remove from the priority queue the search node with the minimum priority, and insert onto the priority queue all *neighboring search nodes,* namely the nodes that can be reached in one move from the dequeued search node, which are its children in the game tree. Repeat this procedure until the search node dequeued corresponds to the goal board.

The efficacy of this approach hinges on the choice of *priority function* for a search node. We consider the following priority function:

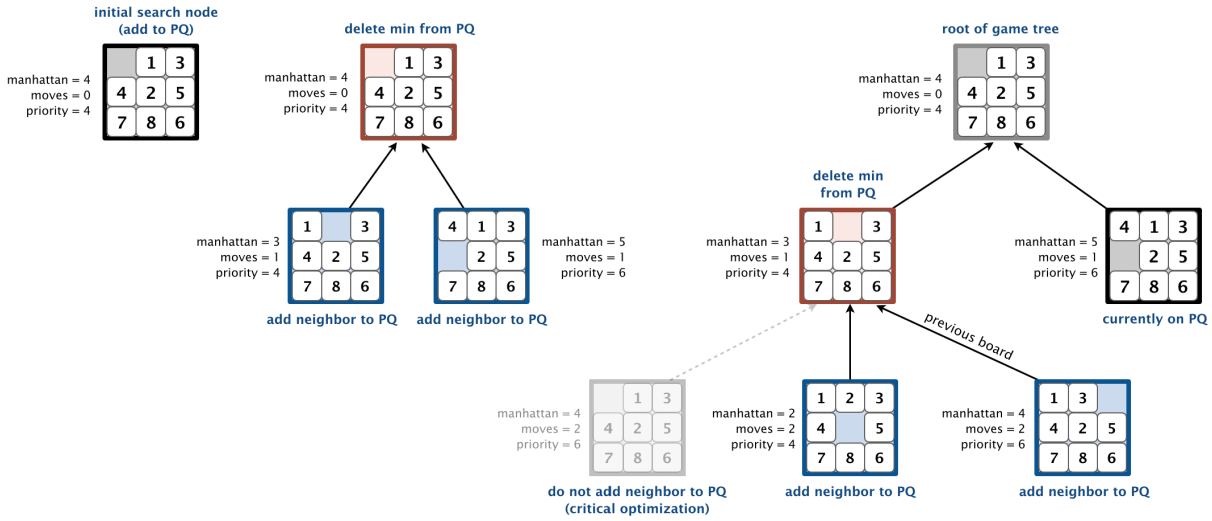
- The *Manhattan priority function* is the Manhattan distance of a board plus the number of moves made so far to get to the search node. Intuitively, a search node with a

small Manhattan distance is close to the goal, and we prefer a search node if it has been reached using a small number of moves.

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 the Manhattan priority function. 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* moves. (Challenge for the mathematically inclined: prove this fact.)

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).

For example, the following diagram illustrates the game tree after each of the first three steps of running the A* search algorithm on a 3-by-3 puzzle using the Manhattan priority function.



Note that for the purpose of this project, it is sufficient to represent the game tree solely through the pointers that connect each search node to its parent.

Solver data type. In this part, you will implement A* search to solve *n*-by-*n* slider puzzles. Create a class Solver with the following API:

```
public class Solver {

   // test client (see below)
   public static void main(String[] args)

}
```

Note: you are free to add additional methods that you like or need.

Implementation requirement. To implement the A* algorithm, you may use the PriorityQueue class from java.util for the priority queue (see also Section 9.3.5 of the textbook).

Test client. Your test client should take the name of an input file as a command-line argument and print the minimum number of moves to solve the puzzle and the sequence of boards from the initial one to the solution. The input file contains the board size *n*, followed by the string representation of the initial board (a string composed of 1 line with all the tiles in row-major order, using 0 to designate the blank square, with tiles separated by spaces).

```
~/Desktop/8puzzle> more puzzle04.txt
3
0  1  3  4  2  5  7  8  6

~/Desktop/8puzzle> java Solver puzzle04.txt
4
0  1  3  4  2  5  7  8  6
1  0  3  4  2  5  7  8  6
1  2  3  4  0  5  7  8  6
1  2  3  4  5  0  7  8  6
1  2  3  4  5  6  7  8  0
```

Deliverables. Submit the files Board.java and Solver.java to the dedicated section in the Moodle Exam site. You may not call any library functions other than those in java.lang, java.io, and java.util**.** *Important:* the file names must be exactly the ones above. Make sure that your program compiles and runs (with the output formatted as described above) using the commands

```
javac Solver.java

java Solver puzzleID.txt
```

otherwise, we may not be able to evaluate your submission.

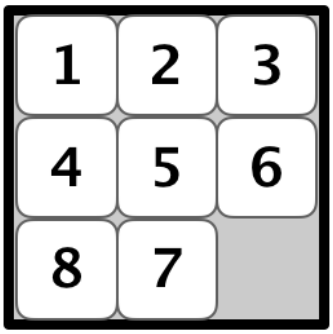
Additional info. This project is adapted from the one described at https://coursera.cs.princeton.edu/algs4/assignments/8puzzle/specification.php. You may check there for additional material, ideas, and suggestions.

Grading. The grading scheme will be the following:

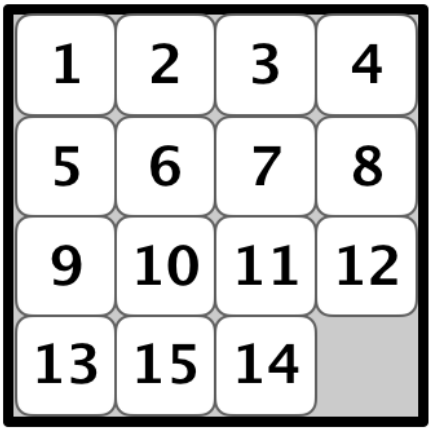
- 1 point if your solution solves the instances in a test set (unknown to you), independently of how long it required
- 2 points if your solution is in the top-20 of all solutions in terms of running time
- 3 points if your solution is in the top-10 of all solutions in terms of running time

ADDITIONAL MATERIAL

Detecting unsolvable boards. Not all initial boards can lead to the goal board by a sequence of moves, including these two:



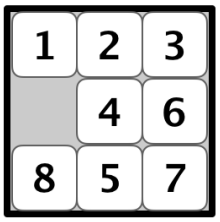
unsolvable



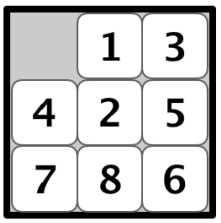
unsolvable

Remarkably, we can determine whether a board is solvable *without* solving it! To do so, we count *inversions*, as described next.

- *Inversions.* 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).

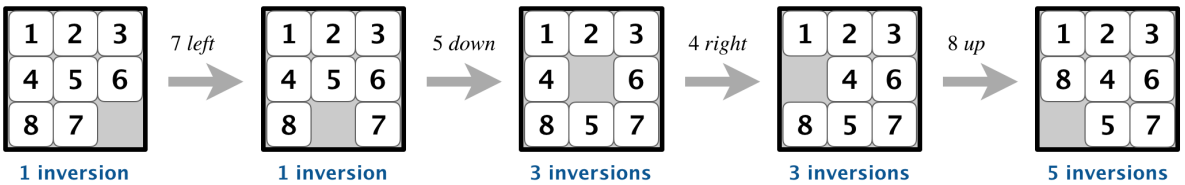


row-major order: 1 2 3 4 6 8 5 7
3 inversions: 6-5, 8-5, 8-7

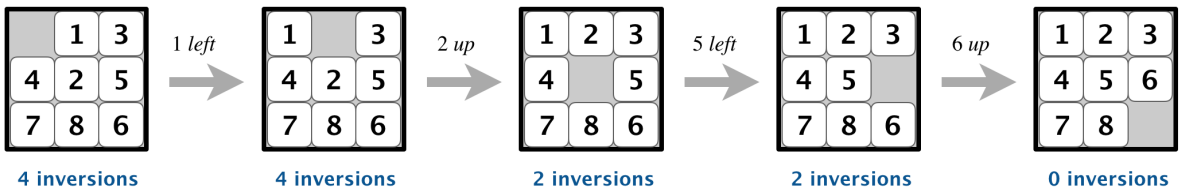


row-major order: 1 3 4 2 5 7 8 6
4 inversions: 3-2, 4-2, 7-6, 8-6

- *Odd-sized boards.* First, we'll consider the case when the board size *n* is an odd integer. In this case, each move changes the number of inversions by an even number. Thus, if a board has an odd number of inversions, it is *unsolvable* because the goal board has an even number of inversions (zero).

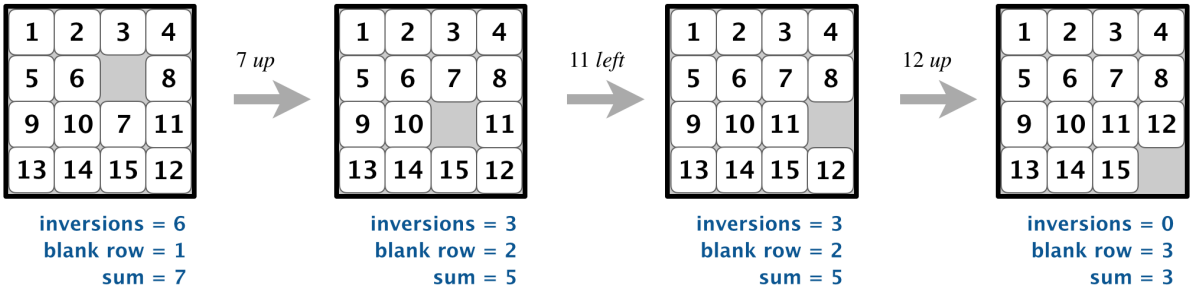


It turns out that the converse is also true: if a board has an even number of inversions, then it is *solvable*.



In summary, when *n* is odd, an *n*-by-*n* board is solvable if and only if its number of inversions is even.

- *Even-sized boards.* Now, we'll consider the case when the board size *n* is an even integer. In this case, 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 (indexed starting at 0) is invariant: each move changes this sum by an even number.



That is, when *n* is even, an *n*-by-*n* board is solvable if and only if the number of inversions plus the row of the blank square is odd.