Exercises

Exercise 1. (*Certify Heap*) Implement the static method `isMaxHeap()` in `CertifyHeap.java` that takes an array `a` of `Comparable` objects (excluding `a[0] = *`) and returns `true` if `a` represents a max-heap, and `false` otherwise.

```
>_ ~/workspace/project4
$ java CertifyHeap
* M A X H E A P
<ctrl-d>
false
$ java CertifyHeap
<ctrl-d>
* A A E H M P X
false
$ java CertifyHeap
<ctrl-d>
* X P M H E A A
true
```

Exercise 2. (*Ramanujan's Taxi*) Srinivasa Ramanujan was an Indian mathematician who became famous for his intuition for numbers. When the English mathematician G. H. Hardy came to visit him one day, Hardy remarked that the number of his taxi was 1729, a rather dull number. To which Ramanujan replied, "No, Hardy! It is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways." Verify this claim by writing a program `Ramanujan1.java` that accepts n (int) as command-line argument and writes to standard output all integers less than or equal to n that can be expressed as the sum of two cubes in two different ways. In other words, find distinct positive integers a, b, c, and d such that $a^3 + b^3 = c^3 + d^3 \leq n$.

```
>_ ~/workspace/project4
$ java Ramanujan1 10000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 2^3 + 16^3 = 9^3 + 15^3
```

Directions:

- Use four nested `for` loops, with these bounds on the loop variables: $0 < a \leq \sqrt[3]{n}$, $a < b \leq \sqrt[3]{n - a^3}$, $a < c \leq \sqrt[3]{n}$, and $c < d \leq \sqrt[3]{n - c^3}$

Do not explicitly compute cube roots, and instead use `x * x * x < y` in place of `x < Math.cbrt(y)`.

Exercise 3. (*Ramanujan's Taxi Redux*) Write a program `Ramanujan2.java` that uses a minimum-oriented priority queue to solve the problem from Exercise 2.

```
>_ ~/workspace/project4
$ java Ramanujan2 10000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 9^3 + 15^3 = 2^3 + 16^3
```

Directions:

- Initialize a min-PQ `pq` with pairs $(1, 2), (2, 3), (3, 4), \ldots, (i, i + 1)$, where $i < \sqrt[3]{n}$
- While i is not empty:
  - Remove the smallest pair (call it *current*) from `pq`.
  - Print the previous pair (k, l) and current pair (i, j) if $k^3 + l^3 = i^3 + j^3 \leq n$.
  - If $i < \sqrt[3]{n}$, insert the pair $(i, j + 1)$ into `pq`.

Again, do not explicitly compute cube roots, and instead use `x * x * x < y` in place of `x < Math.cbrt(y)`.

Problems

Goal The purpose of this project is to write a program to solve the 8-puzzle problem (and its natural generalizations) using the $A^\star$ search algorithm.

The Problem 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 sum of the Hamming distance (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 distance (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.

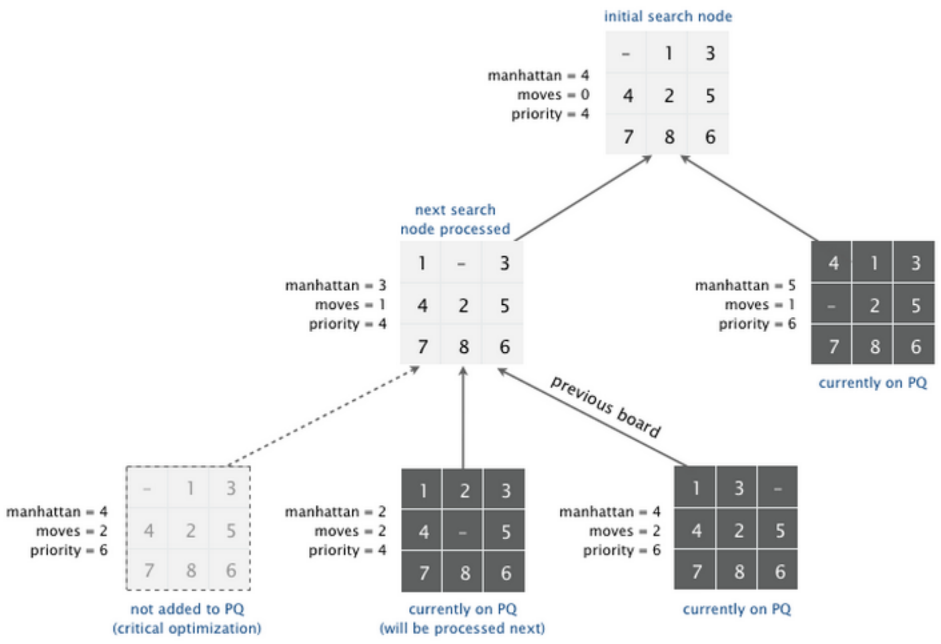
A Critical Optimization Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board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

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    =>    8  4  6
                    8  7             8     7          8  5  7          8  5  7             5  7

row-major order:    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 8 4 6 5 7

                    inversions = 1   inversions = 1   inversions = 3   inversions = 3   inversions = 5
                    (8-7)            (8-7)            (6-5 8-5 8-7)    (6-5 8-5 8-7)    (8-4 8-6 8-5 8-7 6-5)
```

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

row-major order:     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*) Implement an immutable data type called `Board` to represent a board in an n-puzzle, supporting the following API:

Board	
`Board(int[][] tiles)`	constructs a board from an $n \times n$ array; `tiles[i][j]` is the tile at row i and column j, with 0 denoting the blank tile
`int size()`	returns the size of this board size
`int tileAt(int i, int j)`	returns the tile at row `i` and column `j`
`int hamming()`	returns Hamming distance between this board and the goal board
`int manhattan()`	returns the Manhattan distance between this board and the goal board
`boolean isGoal()`	returns `true` if this board is the goal board, and `false` otherwise
`boolean isSolvable()`	returns `true` if this board solvable, and `false` otherwise
`Iterable<Board> neighbors()`	returns an iterable object containing the neighboring boards of this board
`boolean equals(Object other)`	returns `true` if this board is the same as `other`, and `false` otherwise
`String toString()`	returns a string representation of this board

Performance Requirements

- The constructor should run in time $T(n) \sim n^2$, where n is the board size.
- The `size()`, `tileAt()`, `hamming()`, `manhattan()`, and `isGoal()` methods should run in time $T(n) \sim 1$.
- The `isSolvable()` method should run in time $T(n) \sim n^2 \log n^2$.
- The `neighbors()` and `equals()` methods should run in time $T(n) \sim n^2$.

```
>_ ~/workspace/project4

$ java Board data/puzzle05.txt
The board (3-puzzle):
 4  1  3
    2  6
 7  5  8
Hamming = 5, Manhattan = 5, Goal? false, Solvable? true
Neighboring boards:
 4  1  3
 7  2  6
    5  8
----------
    1  3
 4  2  6
 7  5  8
----------
 4  1  3
 2     6
 7  5  8
----------
$ java Board data/puzzle4x4-unsolvable1.txt
The board (4-puzzle):
 3  2  4  8
 1  6    12
 5 10  7 11
 9 13 14 15
Hamming = 12, Manhattan = 13, Goal? false, Solvable? false
Neighboring boards:
 3  2  4  8
 1  6  7 12
 5 10    11
 9 13 14 15
----------
 3  2     8
 1  6  4 12
 5 10  7 11
 9 13 14 15
----------
 3  2  4  8
 1  6 12
 5 10  7 11
 9 13 14 15
----------
 3  2  4  8
 1     6 12
 5 10  7 11
 9 13 14 15
----------
```

Before you write any code, make sure you thoroughly understand the concepts that are central to solving the 8-puzzle and its generalizations using the $A^\star$ algorithm. Compute the following for the two initial boards A and B shown below:

A

4	1	3
	2	6
7	5	8

B

1	2	3
4	6	5
7	8	

1. Hamming distance of the board to the goal board
2. Manhattan distance of the board to the goal board
3. Neighboring boards of the board
4. Row-major order of the board
5. Position of the blank tile (in row-major order) in the board
6. Number of inversions (excluding the blank tile) for the board
7. Is the board solvable? Explain why or why not

8. A shortest solution for the board, if one exists

Directions:

- 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`.
  - Position of the blank tile in row-major order, `int blankPos`.
- `private int[][] cloneTiles()`
  - Return a defensive copy of the tiles of the board.
- `Board(int[][] tiles)`
  - Initialize the instance variables `this.tiles` and `n` to `tiles` and the number of rows in `tiles` respectively.
  - Compute the Hamming/Manhattan distances to the goal board and the position of the blank tile in row-major order, and store the values in the instance variables `hamming`, `manhattan`, and `blankPos` respectively.
- `int size()`
  - Return the board size.
- `int tileAt(int i, int j)`
  - Return the tile at row `i` and column `j`.
- `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 the board is the goal board, and `false` otherwise.
- `boolean isSolvable()`
  - Create an array of size $n^2 - 1$ containing the tiles (excluding the blank tile) of the board in row-major order.
  - Use `Inversions.count()` to compute the number of inversions in the array.
  - From the number of inversions, compute and return whether the board is solvable.
- `Iterable<Board> neighbors()`
  - Create a queue `q` of `Board` objects.
  - For each possible neighbor of the board (determined by the blank tile position):
    * Clone the tiles of the board.
    * Exchange an appropriate tile with the blank tile in the clone.
    * Construct a `Board` object from the clone, and enqueue it into `q`.
  - Return `q`.
- `boolean equals(Board other)`
  - Return `true` if the board is the same as `other`, and `false` otherwise.

Problem 2. (*Solver Data Type*) Implement an immutable data type called `Solver` that uses the $A^\star$ algorithm to solve the 8-puzzle and its generalizations. The data type should support the following API:

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

Corner Cases

- The constructor should throw a `NullPointerException("board is null")` if *board* is `null` and an `IllegalArgumentException("board is unsolvable")` if *board* is unsolvable.

```
>_ ~/workspace/project4
$ java Solver data/puzzle05.txt
Solution (5 moves):
 4  1  3
    2  6
 7  5  8
----------
    1  3
 4  2  6
 7  5  8
----------
 1     3
 4  2  6
 7  5  8
----------
 1  2  3
 4     6
 7  5  8
----------
 1  2  3
 4  5  6
 7     8
----------
 1  2  3
 4  5  6
 7  8
----------
$ java Solver data/puzzle4x4-unsolvable1.txt
Unsolvable puzzle
```

Directions:

- Instance variables:
  - Minimum number of moves needed to solve the initial board, `int moves`.
  - Sequence of boards in a shortest solution of the initial board, `LinkedStack<Board> solution`.
- `Solver :: SearchNode` (represents a node in the game tree)
  - Instsance variables:
    * The board represented by this node, `Board board`.
    * Number of moves it took to get to this node from the initial node, `int moves`.
    * The previous search node, `SearchNode previous`.
  - `SearchNode(Board board, int moves, SearchNode previous)`
    * Initialize instance variables appropriately.
  - `int compareTo(SearchNode other)`
    * Return a comparison of the search node with other , based on the sum: Manhattan distance of the board in the node plus the number of moves to the node (from the initial search node).
- `Solver(Board initial)`
  - Create a `MinPQ<SearchNode>` object `pq` and insert the initial search node into it
  - As long as `pq` is not empty:

  * Remove the smallest node (call it `node`) from `pq`.
  * If the board in `node` is the goal board, extract from the node the number of moves in the solution and the solution and store the values in the instance variables `moves` and `solution` respectively, and break.
  * Otherwise, iterate over the neighboring boards of `node.board`, and for each neighbor that is different from `node.previous.board`, insert a new `SearchNode` object into `pq`, constructed using appropriate values.

- `int moves()`
  - Return the minimum number of moves needed to solve the initial board.
- `Iterable<Board> solution()`
  - Return the sequence of boards in a shortest solution of the initial board.

Data and Test Programs The `data` directory contains a number of sample input files representing boards of different sizes; for example

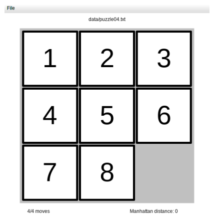
```
>_ ~/workspace/project4

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

The program `SolverVisualizer` accepts the name of an input file as command-line argument, and using your `Board` and `Solver` data types graphically solves the sliding block puzzle defined by the file

```
>_ ~/workspace/project4

$ java SolverVisualizer data/puzzle04.txt
```



The program `PuzzleChecker` accepts the names of an input files as command-line arguments, creates an initial board from each file, and writes to standard output: the filename, minimum number of moves to reach the goal board from the initial board, and the time (in secs) taken; if the initial board is unsolvable, a "–" is written for the number of moves and time taken

```
>_ ~/workspace/project4

$ java PuzzleChecker data/puzzle*.txt
filename                                  moves     time
----------------------------------------------------
data/puzzle00.txt                             0     0.00
data/puzzle01.txt                             1     0.00
data/puzzle02.txt                             2     0.00
data/puzzle03.txt                             3     0.00
data/puzzle04.txt                             4     0.00
data/puzzle05.txt                             5     0.00
...
data/puzzle47.txt                            47     9.41
data/puzzle48.txt                            48     2.13
data/puzzle49.txt                            49    19.63
data/puzzle4x4-unsolvable1.txt               --       --
data/puzzle50.txt                            50    12.31
```

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

Files to Submit

1. `CertifyHeap.java`
2. `Ramanujan1.java`
3. `Ramanujan2.java`
4. `Board.java`
5. `Solver.java`
6. `report.txt`

Before you submit your files, make sure:

- You do not use concepts outside of what has been taught in class.
- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project4
$ check_style src/*.java
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.