Project 4 (8 Puzzle) Checklist

Prologue

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

Files:

- ⇝ `project4.pdf` (project description)
- ⇝ `project4.zip` (starter files for the exercises/problems, `report.txt` file for the project report, and test data files)

Exercises

Exercise 1. (*Certify Heap*) Implement the static method `maxOrderedHeap()` in `CertifyHeap.java` that takes an array `a[]` of `Comparable` objects and returns `true` if `a[]` represents a maximum-ordered heap and `false` otherwise. Your implementation must be linear.

```
>_ ~/workspace/project4

$ java edu.umb.cs210.p4.CertifyHeap
O T H R P S O A E I N G
<ctrl-d>
false
$ java edu.umb.cs210.p4.CertifyHeap
O T S R P N O A E I H G
<ctrl-d>
true
```

Exercises

CertifyHeap.java

```java
package edu.umb.cs210.p4;

import stdlib.StdIn;
import stdlib.StdOut;

public class CertifyHeap {
    // Return true of v is less than w and false otherwise.
    private static <T extends Comparable<T>> boolean less(T v, T w) {
        return (v.compareTo(w) < 0);
    }

    // Return true if a[] represents a maximum-ordered heap
    // and false otherwise.
    protected static <T extends Comparable<T>> boolean maxOrderedHeap(T[] a) {
        int N = a.length;

        // For each node 1 <= i <= N / 2, if i is less than
        // either of its children, return false, meaning a[]
        // does not represent a maximum-ordered heap.
        // Otherwise, return true.
        ...
    }

    // Test client. [DO NOT EDIT]
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        StdOut.println(maxOrderedHeap(a));
    }
}
```

Exercises

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 takes a command-line argument N and prints out 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 $a^3 + b^3 = c^3 + d^3$.

```
>_ ~/workspace/project4

$ java edu.umb.cs210.p4.Ramanujan1 40000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 2^3 + 16^3 = 9^3 + 15^3
13832 = 2^3 + 24^3 = 18^3 + 20^3
39312 = 2^3 + 34^3 = 15^3 + 33^3
32832 = 4^3 + 32^3 = 18^3 + 30^3
20683 = 10^3 + 27^3 = 19^3 + 24^3
```

Hint: Use four nested `for` loops, with these bounds on the loop variables: $0 < a \le \sqrt[3]{N}$, $a < b \le \sqrt[3]{N - a^3}$, $a < c \le \sqrt[3]{N}$, and $c < d \le \sqrt[3]{N - c^3}$; do not explicitly compute cube roots, and instead use `x * x * x < y` in place of `x < Math.cbrt(y)`.

Exercises

Ramanujan1.java

```java
package edu.umb.cs210.p4;

import stdlib.StdOut;

public class Ramanujan1 {
    public static void main(String[] args) {
        ...
    }
}
```

Exercises

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.

- Initialize a minimum-oriented priority queue `pq` with pairs $(1, 2)$, $(2, 3)$, $(3, 4)$, $\dots$, $(i, i+1)$, where $i < \sqrt[3]{N}$
- While `pq` is not empty
  - Remove the (current) pair (i, j) such that $i^3 + j^3$ is the smallest
  - Print the previous pair (k, l) and current pair (i, j) if $k^3 + l^3 = i^3 + j^3 \leq N$
  - If $j < \sqrt[3]{N}$, insert the pair $(i, j+1)$ into `pq`

```
>_ ~/workspace/project4

$ java edu.umb.cs210.p4.Ramanujan2 40000
1729 = 1^3 + 12^3 = 9^3 + 10^3
4104 = 2^3 + 16^3 = 9^3 + 15^3
13832 = 18^3 + 20^3 = 2^3 + 24^3
20683 = 19^3 + 24^3 = 10^3 + 27^3
32832 = 18^3 + 30^3 = 4^3 + 32^3
39312 = 15^3 + 33^3 = 2^3 + 34^3
```

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

Exercises

Ramanujan2.java

```java
package edu.umb.cs210.p4;

import dsa.MinPQ;
import stdlib.StdOut;

public class Ramanujan2 {
    // A data type that encapsulates a pair of numbers (i, j)
    // and the sum of their cubes, ie, i^3 + j^3.
    private static class Pair implements Comparable<Pair> {
        private int i;          // first element of the pair
        private int j;          // second element of the pair
        private int sumOfCubes; // i^3 + j^3

        // Construct a pair (i, j).
        Pair(int i, int j) {
            this.i = i;
            this.j = j;
            sumOfCubes = i * i * i + j * j * j;
        }

        // Compare this pair to the other by sumOfCubes.
        public int compareTo(Pair other) {
            return sumOfCubes - other.sumOfCubes;
        }
    }

    public static void main(String[] args) {
        ...
    }
}
```

Problems

The guidelines for the project problems that follow will be of help only if you have read the description of the project and have a general understanding of the problems involved. It is assumed that you have done the reading.

Problems

Problem 1. (*Board Data Type*) Create an immutable data type `Board` 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 the tile at row i and column j, or 0 if blank
`int size()`	returns the board size n
`int hamming()`	returns the number of tiles that are out of place
`int manhattan()`	returns the sum of Manhattan distances between tiles and goal
`boolean isGoal()`	returns `true` if this is the goal board, and `false` otherwise
`boolean isSolvable()`	returns `true` if this board solvable, and `false` otherwise
`boolean equals(Board that)`	returns `true` if this board equals *that*, and `false` otherwise
`Iterable<Board> neighbors()`	returns all neighboring boards, as an iterable object
`String toString()`	returns a string representation of this board (in the output format specified below)

Problems

Exercise: Consider the 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	

- $\rightsquigarrow$ What are the neighboring boards of A and B?
- $\rightsquigarrow$ Calculate the Hamming distances of A and B to the goal board.
- $\rightsquigarrow$ Calculate the Manhattan distances of A and B to the goal board.
- $\rightsquigarrow$ Calculate the number of inversions in A and B.
- $\rightsquigarrow$ Are A and B solvable? Explain why or why not.

Problems

Hints

- 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`
- `private 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 6
- `private int inversions()`
  - Return the number of inversions
- `private int[][] cloneTiles()`
  - Clone and return `this.tiles`

Problems

- `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
- `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 to the goal board, and store the distances in the instance variables `hamming` and `manhattan` respectively
- `int tileAt(int i, int j)`
  - Return the tile at row `i` and column `j`

Problems

- `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()`
  - Create a queue `q` of `Board` objects
  - 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 `q`
  - Return `q`

Problems

Problem 2. (*Solver Data Type*) Create an immutable data type `Solver` 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, as an iterable object

Hints

- Instance variables
  - Sequence of boards in a shortest solution, `LinkedStack<Board> solution`
  - Minimum number of moves to solve the initial board, `int moves`
- `Solver` :: `SearchNode` (represents 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

Problems

- `Solver :: HammingOrder :: int compare(SearchNode a, SearchNode b)`
  - Return a comparison of the `a.board.hamming() + a.moves` and `b.board.hamming() + b.moves`
- `Solver :: ManhattanOrder :: int 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 values
- `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

Problems

The data directory contains a number of sample input files representing boards of different sizes; for example

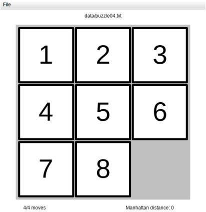
```
>_ ~/workspace/project4

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

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

```
>_ ~/workspace/project4

$ java edu.umb.cs210.p4.SolverVisualizer data/puzzle04.txt
```



Epilogue

Use the template file `report.txt` to write your report for the project

Your report must include:

- Time (in hours) spent on the project
- Difficulty level (1: very easy; 5: very difficult) of the project
- A short description of how you approached each problem, issues you encountered, and how you resolved those issues
- Acknowledgement of any help you received
- Other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Epilogue

Before you submit your files:

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

```
>_ ~/workspace/project4
$ check_style <program>
```

  where `<program>` is the fully-qualified name of the program

- Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and running time

- Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Files to submit:

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