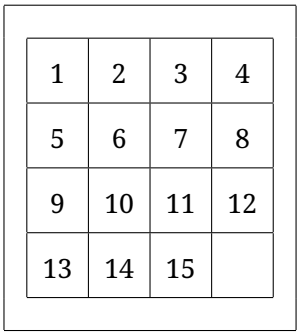
Object Orientation – Spring 2018 **Assignment 6**

1 Sliding Games

A standard sliding game has $N \times N$ squares with pieces numbered from 1 to $N^2 - 1$. The last square is left blank, so you can move the adjacent pieces into the resulting gap. The goal of the sliding game is to convert a random initial combination to a final configuration where all the pieces are in an ascending order by sliding pieces around. The picture shows a solved 4×4 game:



In every state of the game, there are at most 4 legal moves. We can indicate these moves by the moving direction of the gap: *North*, *east*, *south* and *west*. As you can see, the configuration above has only two legal moves: The gap can be shifted to the *north* or *west* only.

2 Goal of the assignment

In this assignment, you are supposed to implement an algorithm that finds the (preferably shortest) solution of an N by N sliding game. An algorithm that could be implemented here is breadth-first search, which was explained in the Intro AI course.
After completing this exercise you should be able to:

- Use lists and queues to implement a search algorithm;
- use the breadth-first search strategy (or some other strategy) to traverse a search space;
- redefine and use the hash-method;
- use the *best-first* optimisation of breadth-first search;
- use priority queues from the Java API.

3 Instructions

Sketch out a breadth-first searching algorithm on paper before you actually write and test your code. It is often handy to use small scale example problems during code development. For instance, start with a 2×2 or 3×3 board with an almost completed set up.

4 Problem sketch

You need a *queue* to implement a breadth-first search strategy. This data type has the characteristic feature that the elements that stand in the queue the longest are released the first - just as a queue at the supermarket check out. New candidates are placed at the end of the queue and

need to wait their turn. To solve this sliding puzzle you should go on as follows: as long as the queue of configurations is not empty, you take out the first element. If this puzzle is identical to the solution, you're done. Otherwise, you will inspect all immediate successors of the current configuration. These offspring configurations are placed in the back of the waiting queue. They will be processed automatically as soon as the configurations are already in the queue have been dealt with. This algorithm does not only work for sliding games, but for various other puzzles involving search strategies as well.

Therefore, we abstract from the sliding games and define a more general interface which our concrete puzzle will implement:

```
public interface Configuration extends Comparable<Configuration> {
    public abstract Configuration parent();

    public abstract Collection<Configuration> successors();

    public abstract boolean isSolution();

    public default List<Configuration> pathFromRoot(){
        throw new UnsupportedOperationException( "pathFromRoot: not supported yet."
            );
    }
}
```

We will discuss in a minute why this interface implements the (standard Java) interface `Comparable`. At first instance, you can let NetBeans (or Eclipse) generate the default implementation.

To represent the sliding game itself, we will use the class `SlidingGame`. The configuration of the game is kept by as a two-dimensional array of integers. Next to that, the position of the gap is stored explicitly in a field within the class so we do not have to search through the two dimensional array every time.

The class we provided is far from being complete. A number of methods and attributes needs to be added to get the program working.

```
public class SlidingGame implements Configuration {
    public static final int N = 3, SIZE = N * N, HOLE = SIZE;
    private int[][] board;
    private int holeX, holeY;

    public SlidingGame( int[] start ) {
        board = new int[N][N];

        assert start.length == N*N: "Length of specified board incorrect";

        for( int p = 0; p < start.length; p++ ) {
            board[p % N][p / N] = start[p];
            if ( start[p] == HOLE ) {
                holeX = p % N;
                holeY = p / N;
            }
        }
    }

    @Override
    public String toString() { ... }

    @Override
    public boolean equals(Object o) { ... }

    @Override
```

```
    public boolean isSolution () { ... }

    @Override
    public Collection<Configuration> successors () { ... }

    @Override
    public Configuration parent() { ... }
}
```

Our *breadth-first solver* looks like the following:

```
public class Solver
{
    Queue<Configuration> toExamine;

    public Solver( Configuration g ) {
        ...
    }

    public String solve() {
        while ( ! toExamine.isEmpty() ) {
            Configuration next = toExamine.remove();
            if ( next.isSolution() ) {
                return "Success!";
            } else {
                 for ( Configuration succ: next.successors() ) {
                     toExamine.add ( succ );
                }
            }
        }
        return "Failure!";
    }

}
```

Naturally, it does not make much sense to revisit configurations that were already evaluated. Adding such an intermediate configuration to the queue again will definitely not give a shorter solution. It might even lead to an infinite loop. To prevent that, you should store all intermediate solutions that you found.

5 Assignment parts:

Complete the program in the way described below:

5.1 The `SlidingGame` class

Complete the class named `SlidingGame`. Since this class is an extension of class `Configuration`, you will need to implement at least the following methods:

1. `isSolution`: indicates whether the respective game is solved or not.
2. `parent`: returns the predecessor of the current configuration. *Hint*: add an attribute to the class to keep track of the predecessor.
3. `successors`: returns the direct successors of the current configuration. Store them in any datatype that implements the `Collection` interface - such as a `LinkedList` or an `ArrayList`.

5.2 The path to the solution

The output of the solver we implemented above is rather disappointing; all you get to see is whether your game can be solved or not. In the first case, you might be curious about the solution itself: which pieces do you need to move to get to the end configuration? In order to present such solution to the user, you need to be able to trace back the direct predecessor of each intermediate step. We use the `parent` method for that purpose.

In order to reconstruct the path from the initial configuration to the current state, the default method

```
public default List<Configuration> pathFromRoot(){}
```

is added to the interface `Configuration`. The current implementation is a stub: You should implement it the proper way.

5.3 Observed states

To prevent (an infinite loop of) repetitions during your search you need to keep track of the states you visited already. The easiest way to do that is to store them in a list of states. You can then use the `contains` method to determine whether or not a state is already present in the list. To keep this part of your program flexible, it is advisable to use the `Collection` interface. Next to `contains`, this interface also contains the method `add`, which can be used to add new configurations.

Storing all the states requires a lot of storage space. If you get a message stating that the entire *heap space* is used, you might have made a mistake that led to an infinite search loop. If your program is correct and you get a memory error anyway (e.g. in a 2×2 game), you can assign more memory space to your program using the option `-Xmx1g` . In NetBeans you can find this option in the *File* menu in the *project properties* window, under *run - VM options* With the given option, you will size up the heap space to a maximum of 1 GB.

5.4 A basic solution

Use the given ingredients to write a Java program that tries to solve a given puzzle. Once a solution is found, the program should display the path of all intermediate configurations that lead to the solution in the right order.

By the way: not all games are solvable. By swapping two adjacent pieces, the puzzle changes its *parity* - and you can never retrieve the original position by moving the pieces. So, in the case that an unsolvable game is presented to your program, it should print a message accordingly.

5.5 Handling intermediate solutions more efficiently

The number of solvable configurations for an $n \times n$ puzzle is $(n^2)!/2$. Even for a modest 3×3 puzzle, this adds up to a total of 181440 configurations. Before your program concludes that a given puzzle is unsolvable, it will have created a list of all reachable configurations. To then see whether another puzzle is new, it will need to run through the entire list again. For each new puzzle, this comes down to a workload of $O(n)$.

This process of checking for configuration reoccurrences can be sped up by using a *hash table*. A hash table stores elements based on a unique *hash value*. The hash value is also needed to determine whether the table already contains a given element. In practice, this searching process has a complexity of $O(1)$ - which is a considerable improvement compared to the $O(n)$ complexity of a normal list search.

On a side note: It is no problem if two different puzzles have an identical hash value. The hash table will take care of this and store both puzzles without slowing down the searching progress in a significant way.

Java includes libraries that implement sets based on hash tables, such as *HashSet*. The *HashSet* library uses the method `public int hashCode()` to determine a hash value. This method is

part of the class `Object`, and hence inherited by every class. As is often the case, the standard implementation of this method is not very useful and you should implement a better version.

If two objects are equal (according to the (`Object`) method `equals`, then calling the `hashCode` method on each of the two objects must produce the same result. It is not required that if two objects are unequal, calling the hashCode method on each of the two objects must yield distinct results. However, you should be aware that producing distinct hash values for unequal objects may improve the performance of hash tables.

The numerical nature of our game lends itself to use the locations and the values of the pieces to compute a hash value. You can use the following formula to do the calculations:

$$\text{hashwaarde} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{board}[x][y] \cdot 31^{x+y \cdot N}$$

Given that a `HashSet`, just as a list, implements the `Collection` interface, you will need to change almost nothing else in the structure of your program.

5.6 A more effective searching algorithm

If you implement the algorithm described above, you will notice that the capabilities of the solver are rather small: as soon as your program gets fed a somewhat more complex puzzle, it will use too much time or memory space to find a solution. There are a few ways we can improve the algorithm.

A rather obvious improvement would be to change the breath–first search to a best–first search that examines the best successor states first. We can easily achieve this by placing the best elements at the front of the queue.

However, we do not actually know the best successive states until we solved the puzzle and know the path to the final configuration. Luckily, we can use a heuristic to estimate the goodness of an intermediate configuration. The *Manhattan distance* is the number of squares a piece has to cover vertically and horizontally to get to a desired location (its place in the end configuration). It is recommended to store the Manhattan distance for every puzzle so you do not have to recalculate it.

To implement the best–first search strategy, you just have to replace our generic queue by a `PriorityQueue`. This data structure will place the best elements in the front all by itself. To determine the position for a new element, this queue uses the method `public int compareTo()` from the `Comparable<T>` interface. This explains why `Configuration` extends the `Comparable<Configuration>` interface. For our games, we can have `compareTo` return the difference in Manhattan distance.

All in all, the following properties should hold for the `compareTo` method:

- `sgn(x.compareTo(y))== -sgn(y.compareTo(x))` for all `x` and `y`. The function `sgn` is the *sign function*: it returns 0 if the argument is zero, 1 if the argument is greater than zero, -1 if the argument is less than zero.
- The comparison needs to be transitive. This, for example, means that if `x.compareTo(y)>0` and `y.compareTo(z)>0`, then `x.compareTo(z)>0`.
- It is sometimes useful to make sure that `x.compareTo(y)==0` if `x.equals(y)`, but it does not need to be.

  In this this exercise, for instance, two puzzles with the same Manhattan distance (which means that `compareTo` returns 0) can still have different states. On the other hand, we want to ensure that `x.compareTo(y)==0` for all `x.equals(y)`.

6 Supplementary files

All code fragments that we showed you in this assignment can be found on Blackboard. You may use these for your own code.

7 Hand in

Before Sunday March 18th, 23:59, on Blackboard.

Hand in your Java files on Blackboard before **Sunday March 18th, 23:59**. Do not forget to add your and your partner's names and student numbers.