420-406-AB Data Structures
Winter 2020

April 24th
Worth 11%

Assignment VI: Recursion

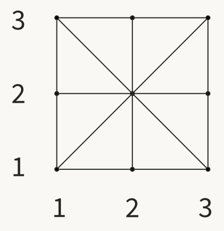
(Due on Check on Léa)

In this assignment, you will design a class to represent a simple 2-player board game, then use this class to list all possible board configurations as result of playing the game. Your algorithm will use recursion.

1 Three Men's Morris Board

The game we will use is called "Three Men's Morris", which is itself a very simplified version of the game "Nine Men's Morris".

Rules. An empty "Three Men's Morris" board looks like this:



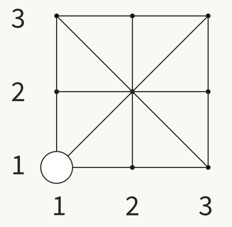
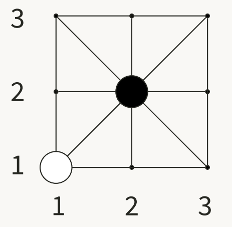
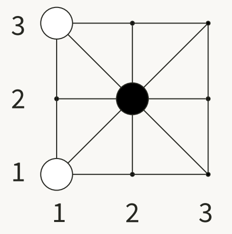
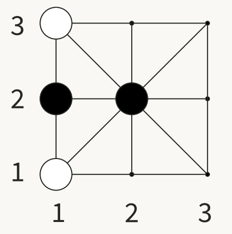
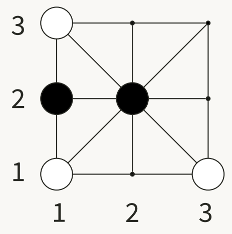
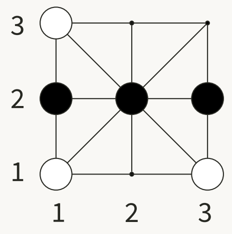
The playable positions are numbered both horizontally, called *rank*, and vertically, called *file*. Here are the game rules: [1]

> Each player has three pieces. The winner is the first player to align their three pieces on a line drawn on the board. There are 3 horizontal lines, 3 vertical lines and 2 diagonal lines.
>
> The board is empty to begin the game, and players take turns placing their pieces on empty intersections. Once all pieces are placed (assuming there is no winner by then), play proceeds with each player moving one of their pieces per turn. A piece may move to any vacant point on the board, not just an adjacent one.

We will implement the placement phase only.

[1] `https://en.wikipedia.org/wiki/Three_men%27s_morris`

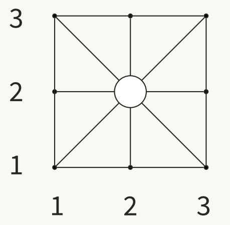
Example 1:

○ plays (1,1) 	● plays (2,2) 
○ plays (1,3) 	● plays (1,2) 
○ plays (3,1) 	● plays (3,2) 

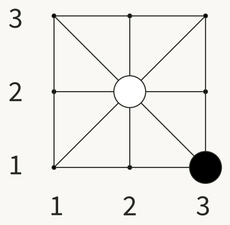
The placement phase ends with ● as winner.

Example 2:

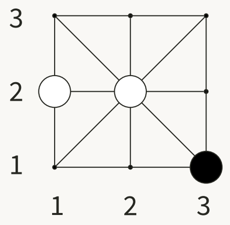
○ plays (2,2)



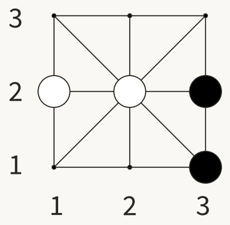
● plays (3,1)



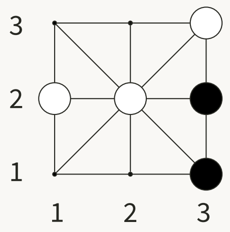
○ plays (1,2)



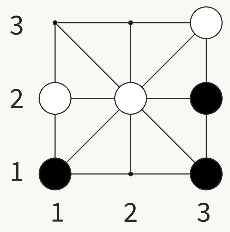
● plays (3,2)



○ plays (3,1)



● plays (1,1)



The placement phase ends with no winner.

Implementation

Design a class `ThreeMensMorris` that implements the rules of “Three Men’s Morris” (the placement phase only). Your class will implement the following `Game` interface:

```
public interface Game {
    /**
     * Play the current piece at the specified position
     * @param file The file of
     * @param rank
     * @return True if the move was performed, false otherwise.
     */
    boolean play(int file, int rank);

    /**
     * Determine the winner of the game.
     * @return The winner of the game, or NONE if there is no winner yet.
     */
    Token winner();
}
```

Example 1. The first example above will look like this in code:

```
Game board = new ThreeMensMorris();
game.play(1, 1);
assertEquals(Game.Token.None, game.winner());
game.play(2, 2);
game.play(1, 3);
game.play(3, 1);
game.play(3, 2);
assertEquals(Game.Token.Black, game.winner());
```

2 Copying Boards

It will be difficult to list all board configurations without making a *copy* (also called a *clone*) of a board[2]. Invoking this copy operation will generate an identical, but distinct object. For example,

```
ThreeMensMorris board1 = new ThreeMensMorris();
ThreeMensMorris board2 = board1.copy();
assertTrue(board1.equals(board2));
assertFalse(board1 == board2);
```

To structure our copy operation, have your `ThreeMensMorris` class implement the `Copyable<T>` interface:

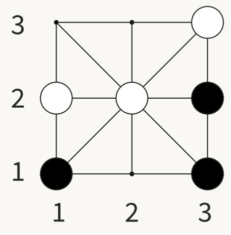
```
public interface Copyable<T> {
    /**
     * Get a copy (clone) of the object.
     * @return The copy.
     */
    T copy();
}
```

[2]Java has it's own mechanism for cloning objects using a method called `clone()` and an interface called `Cloneable`. It does not use generics and so there's some casting involved when using these. Take a look at the documentation for these if you are interested: `https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html` `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone-`

3 Standard operations.

`toString()`

Implement a `toString()` method that prints the board state, the turn and the winner of the board. For example, this board,



will be represented as the `String`:

```
[● ●○○●  ○] turn=● winner=NONE
```

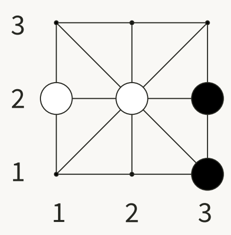
`equals( .. )` and `hashCode()`

Implement the methods `equals( .. )` and `hashCode()`. You can use the IntelliJ default implementations.

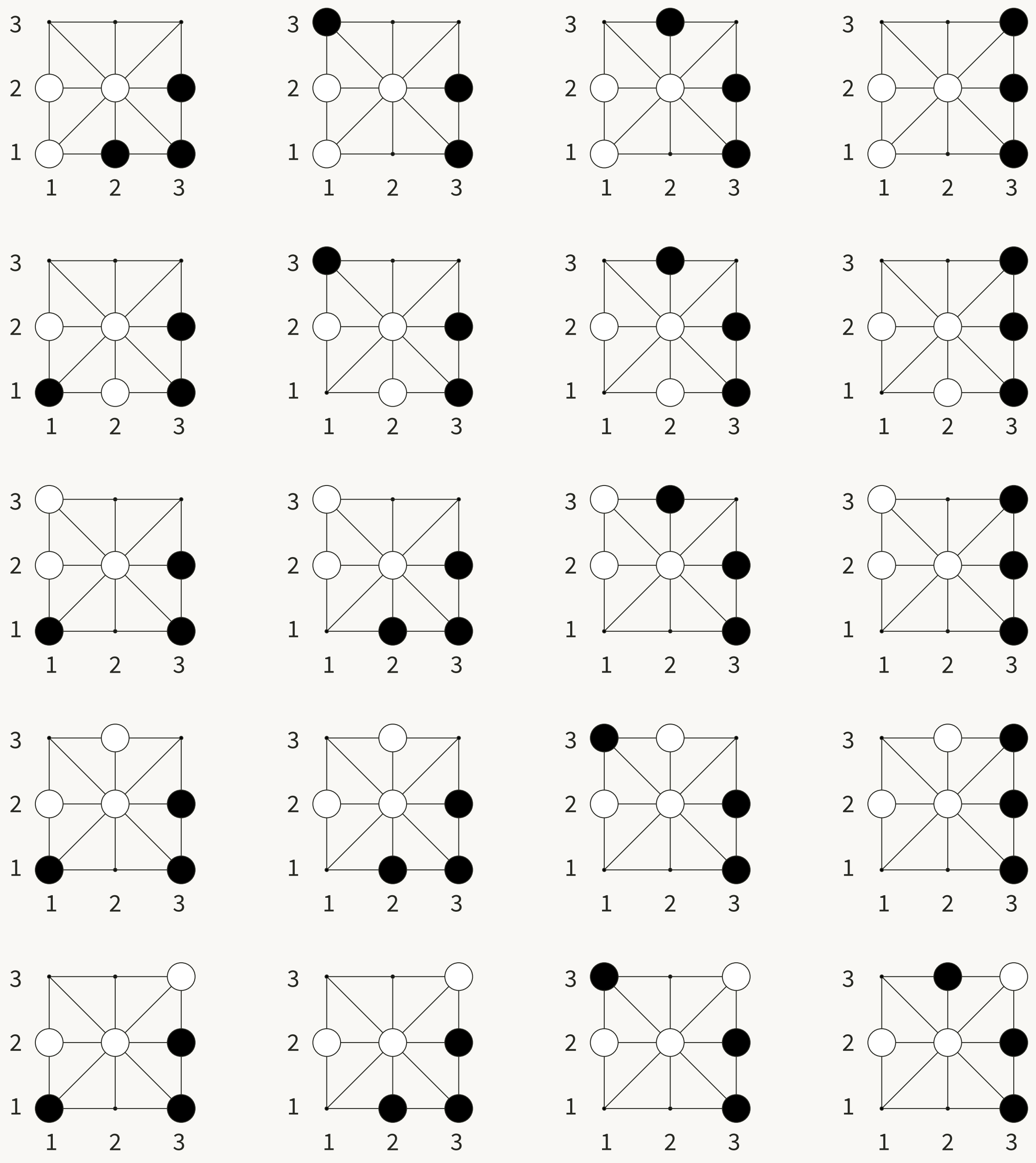
4 Listing All Boards (At the End of the Placement Phase)

Write a recursive operation that lists of all possible boards *at the end of the placement phase*, i.e.: boards that have been played until either someone wins or all pieces have been placed.

Example. Starting with the empty initial board this operation will result in too many board configurations to trace. It will be easier to verify your algorithm starting from a partially played board. For example, starting from this board:

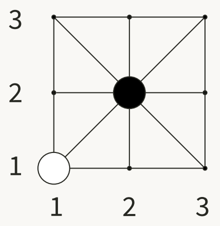


There are 20 configurations at the end of the placement phase:



4.1 Algorithm

Use recursion to implement this. Here is a way of looking at the problem that might help. For the current board, say for example it



we want collect all the boards that result from placing a ○ token at position $(2, 1)$, as well as the boards that result from instead placing a ○ token at $(3, 1)$, and so on…

4.2 Structure

Your solution must implement this following method (in class `Main`):

```
public static List<ThreeMensMorris> generate(ThreeMensMorris initial)
```

Typically, it is really inefficient to combine lists over and over again inside of a recursive method. I strongly suggest you include a recursive ”helper” method with an accumulator like we see in class. Something like this maybe:

```
private static void generateHelper(ThreeMensMorris current, List<ThreeMensMorris> acc)
```

but in the end you should structure the recursion as you see fit!

4.3 Skipping Duplicate Boards

The ideal solution would only contain a board configuration once. For example, these two move sequences result in the same board:

```
[(1,1), (2,2), (1,3)]
[(1,3), (2,2), (1,1)]
```

Optimize your algorithm to only prevent the inclusion of duplicate boards in the list, but do so as efficiently as possible. Hint: think about how a data structure could help here, also you might need more add base cases!

5 Requirements

- Your class `ThreeMensMorris` implements the interfaces `Game` and `Copyable<T>`.
- Your class `ThreeMensMorris` has implementations of `toString()`, `equals()` and `hashCode()`, you use the provided enum `Token` in your `toString()` implementation to show ○ and ● tokens.
- Provide an implementation of the method `generate( .. )` that uses recursion (can be a call the a recursive helper method).