The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Dr Tilman Dingler & Dr Thuang Pham
Semester 2, 2020

Assignment 2
Bonus Task
5pm (AEST), October 15, 2020

1 Optional Bonus Task

This final task is optional! Note that it is a challenging task and it takes quite some time to complete. You may find it more worthwhile spending this time on assignments of other subjects. But if not, you're in for a treat. If you are successful in this task, you can earn back 1.5 marks that you may have lost in assignment 1. The total mark for assignment 2 can, therefore, be 16.5. However, the total mark for the two assignments combined cannot exceed 30.

2 Advanced Nim game, Victory Guaranteed Strategy

In this task, you implement an advanced Nim game which has different rules from the original Nim game. Similar to the implementation of AI players, the polymorphism (inheritance and interface) provided by Java is expected to be demonstrated in your program. That is, both the original Nim game and this new advanced Nim game are a specialized *game* with different rules. Either the inheritance or the interface can be chosen to reflect this relationship; the decision is up to you to suit your own design. Since, in this bonus section, you are going to have two types of games, the `removeStone()` method in the human player and AI player should also have two implementations, i.e., in game instances of different types of game, different `removeStone()` implementations are used, particularly for the AI player. For this advanced game setting, you can use the name `advancedMove()` for the method. The rules of the advanced Nim game are as follows:

The major rule in this advanced Nim game is that a player is **only allowed to remove 1 stone or 2 adjacent stones in each move**. Here, adjacent stones are stones that are neighbors to each other. Hence, the upper bound of stones to be removed is always 2. However, the requirement on the **adjacent stones** makes the position of the stones matter, i.e., removing the same number of stones at different positions may produce different game states. While in the original Nim game, stones are conceptually the same, i.e., removing the same number of stones always produces the same game state. For example, given 5 stones represented as $< * * * * * >$,

- in the original Nim game removing 2 stones will always produce the state $< * * * >$

- in the advanced Nim game, depending on which stones are removed, removing 2 stones produces multiple states. In the following example, note that state 2 differs from state 3 because in state 2 the first two remained stones cannot be removed in a single move since they are not adjacent while in state 3 the first two remained stones can be removed in a single move.

    1. removing the first and the second, results to $< \times \times * * * * >$
    2. removing the second the third, results to $< * \times \times * * * >$
    3. removing the third and the fourth, results to $< * * \times \times * * >$
    4. removing the fourth and the fifth, results to $< * * * \times \times >$

Similar to the original Nim game, each player takes turns to remove either one stone or two adjacent stones. **A player wins if he / she removed the last stone**.

There should be a new command defined in the `Nimsys`, named `startadvancedgame`, which will commence a new advanced game following the above rules. The syntax of this new command is `startadvancedgame initialstones, username1, username2`.

During the game, each player enters the move in the form of two integers `position number`, indicating the position and the number of stones to be removed. The stones left are displayed on one line, each stone is printed in the form of `<position, *>` if the stone is presented or `<position, x>` if the stone has already been removed.

Here is an example of the expected display of this command:

```
$startadvancedgame 10, lskywalker, artoo

Initial stone count: 10
Stones display: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Player 1: Luke Skywalker
Player 2: R2 D2

10 stones left: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
3 2

8 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
R2's turn - which to remove?

6 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

4 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
R2's turn - which to remove?

2 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,x> <9,x> <10,*>
Luke's turn - which to remove?
5 1

1 stones left: <1,x> <2,x> <3,x> <4,x> <5,x> <6,x> <7,x> <8,x> <9,x> <10,*>
R2's turn - which to remove?

Game Over
R2 D2 wins!
```

```
$
```

Any invalid input listed below shall be caught by **try / catch syntax**, and a line stating `Invalid move.` should be printed out:

- invalid position; given N stones when the game commences, the position should be $[1, N]$, any other values are invalid;
- invalid number of stones; **the number should be either 1 or 2**, any other values are invalid;
- the stones at the specified positions have already been removed

Here is an example output:

```
6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
12 1

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 3

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
2 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
```

After implementing the game rules, you are also required to design the strategy for the AI player. Because the game rules change in the advanced Nim game, the victory guaranteed strategy designed for the original Nim game does not work in this new game. Your task here is to implement the AI in this new game. The hints are: if the AI player **is the first one who moves** when the game commences, there is **always a victory guaranteed strategy** for the AI player. If the AI player **is not the first one who moves** when the game commences, it is still **possible** for the AI player to win as long as the rival player does not play exactly following the winning strategy. Hence, the implementation of the strategy for the AI player in this game should enable the AI player to:

(a) win if it moves first when game commences,

(b) win if it moves second, given that the rival player does not follow exactly the wining strategy.

The details of the strategy are left for you to design. To simplify the code design, you may assume **a maximum of 11 stones** in each game played. **In order to get bonus marks, you need to implement the strategy to meet all of the below requirements**:

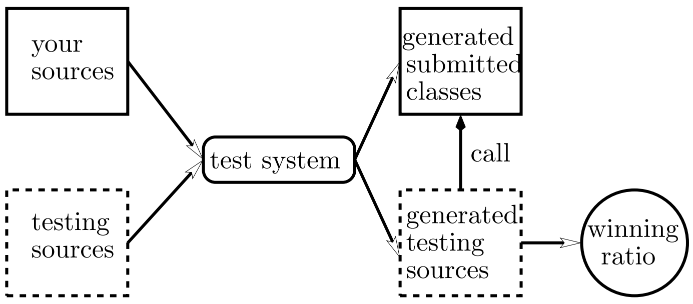

Figure 1: How testing on the advanced Nim game solution works; the testing procedure of the task in Section 2, the submitted codes will be compiled and run together with our testing sources, and the generated testing classes will generate the winning ratio of the submitted solution.

(a) Your AI player class who can play the advanced game should be named as `NimAIPlayer`, please note this name is case sensitive and mandatory;

(b) Your AI player class should have a constructor with **no parameters**;

The victory guaranteed strategy designed by you should be implemented in this `advancedMove()` method. This method will be invoked when we test the correctness of your implementation of the victory guaranteed strategy.

Overall, your AI player class will look like:

```
public class NimAIPlayer ...
{
// you may further extend a class or implement an interface
// to accomplish the task in Section 2.6
public NimAIPlayer() {}
...
public String advancedMove(boolean[] available, String lastMove)
{
// the implementation of the victory
// guaranteed strategy designed by you
...
}
...
}
```

Here, the `boolean[] available` represents the stones remained to be removed, `true` as remained and `false` as removed, e.g., $< * \times \times \times * >$ can be represented as [true false false false true]. The `lastMove` represents the last move made by the rival player, e.g., if the rival player removed the second stone in the last turn, then `lastMove` is of value "2 1", if the second and the third stones are removed in the last turn, then `lastMove` is of value "2 2". The `advancedMove()` method returns the move chosen by your AI player this turn, in the form of `position number`. For example, if your AI player chooses to remove the first and the second stones, the returned string should be "1 2".

Your solution will be evaluated using test cases in three cases:

1. Your AI player moves first to play against a dummy player, who moves randomly;
2. Your AI player moves first to play against an oracle AI player, who enumerates all the possibilities and try best to win;
3. Your AI player moves second to play against a dummy player, who moves randomly;

The solution will be assessed based on the wining ratio of your solution in the three cases. Figure 1 shows the testing procedure of a submitted solution. Specifically, for three cases, case 1, case 2 and case 3, 0.5 marks will be granted ONLY if your AI winning ratio is 100% in each case, suggesting your AI player passes all the test cases. Otherwise, a non-100% ratio for any case will not get any marks for that particular case.

As described earlier, it is important that your design makes good use of object-oriented design principles. This is particularly relevant when it comes to implementing the actual gameplay. **In a real game, game proceeds with each player performing the action of removing some number of stones from the game. Your design should reflect this structure.**

Don't worry about changing your output for singular/plural entities. Simply always use plural entities, i.e., you should output '1 games', '1 wins', '1 stones', etc.

The solution will be assessed based on the wining ratio of your solution in the three cases. Figure 1 shows the testing procedure of a submitted solution. Specifically, for three cases, case 1, case 2 and case 3, 0.5 marks will be granted ONLY if your AI winning ratio is 100% in each case, suggesting your AI player passes all the test cases. Otherwise, a non-100% ratio for any case will not get any marks for that particular case.

3 Submission

The same procedure and rules apply as described in the assignment 2 specification document.

4 Individual Work

The rules of academic integrity apply to the bonus task as much as they do for all assignments.