SUPINFO Academic Dept.

SUPINFO International University

INSTITUTE OF INFORMATION TECHNOLOGY

Algorithms and programming in Python

First Mini - Project

Two little strategy games

Version 1.0
Last update: 24/10/2018
Use: Students/Staff
Author: Laurent GODEFROY/PN

SUMMARY

1 PREAMBLE

This project is **individual**.

Any form of plagiarism or using codes available online or any other type of support, even partial, is prohibited and will cause a 0, a cheater mention, and even a disciplinary board.

You must upload an archive with the format « .zip » containing the source code of your project.

There is no viva for this mini-project.

A grading **scale** is given at the end of this subject.

The goal of this mini-project is to program two little combinatory and abstract strategy games in Python.

2 NIMBLE

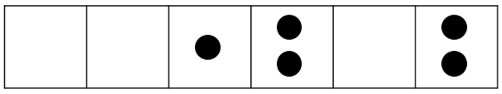
2.1 THE RULES OF THIS GAME

Important note: no code is requested in this sub part. We will only explain the rules.

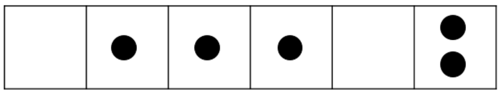
Two players confront each other on a one-dimensional board of n squares. At the beginning, each square contains a random number of pawns (this number can be 0). At each turn, a player chooses a square containing at least one pawn, and moves one of the pawns to one of the squares on the left, *i.e.* a square of strictly lower index. This square can be empty or can already contains one or more pawns.

The winner is last player able to move a pawn. In other words, as soon as a player cannot move, that player loses. It corresponds with the situation where all the pawns are on the first square of the board.

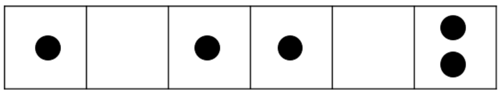
Let's give an example of the game with a board of 6 squares :



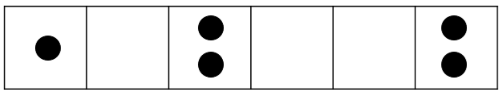
The first player moves a pawn of the square number 4 to the square number 2 :



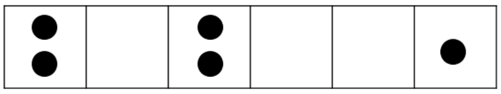
The second player moves a pawn of the square number 2 to the square number 1 :



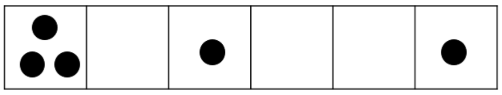
The first player moves a pawn of the square number 4 to the square number 3 :



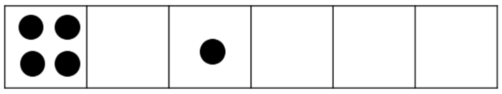
The second player moves a pawn of the square number 6 to the square number 1 :



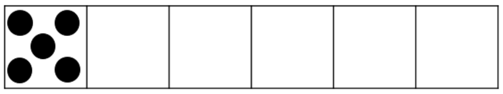
The first player moves a pawn of the square number 3 to the square number 1 :



The second player moves a pawn of the square number 6 to the square number 1 :



The first player moves a pawn of the square number 3 vers la case n°1 :



The second player cannot move pawns anymore. He has lost.

2.2 PROGRAMMING THIS GAME IN PYTHON

It is highly recommended to read this part before starting to code.

Important notes :

- We can possibly implement subroutines in addition to those asked.
- The board will be a one-dimensional list of positive integers or null, and each integer of this list represents the number of pawn that the corresponding square contains.
- Instead calculate regularly the dimension of the list, we will prefer pass it as parameter of our subroutines.
- The example of visual rendering is just indicative. You can improve it.

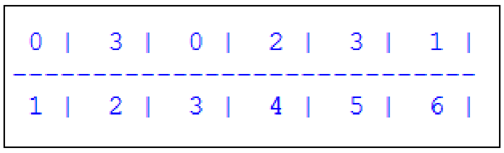
Notations of the subroutines' parameters that we will implement :

- « board » : a one-dimensional list of positive integers or null that represents the board of the game.
- « n » : strictly positive integer equal to the number of elements of « board ».
- « p » : strictly positive integer equal to the maximum number of pawns that can contain a square at the initialization of the game.

- « i » : any integer.
- « j » : any integer.

Implement the following subroutines in a file called “nimble.py“ :

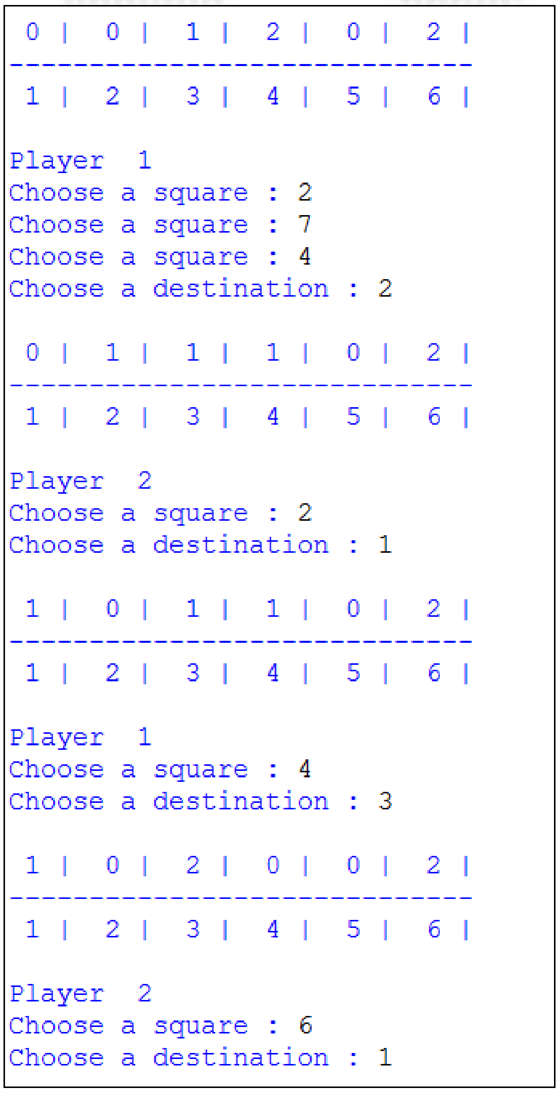
- A function « newBoard(n,p) » that returns a one-dimensional list and represents the initial state of a board with **n** squares. There will be on each square a random number of pawns between **0** and **p**.
- A procedure « display(board,n) » that prints the board on the python’s console. We will reprensent each square by the number of pawns that the square contains, and we will print below his index. Example :



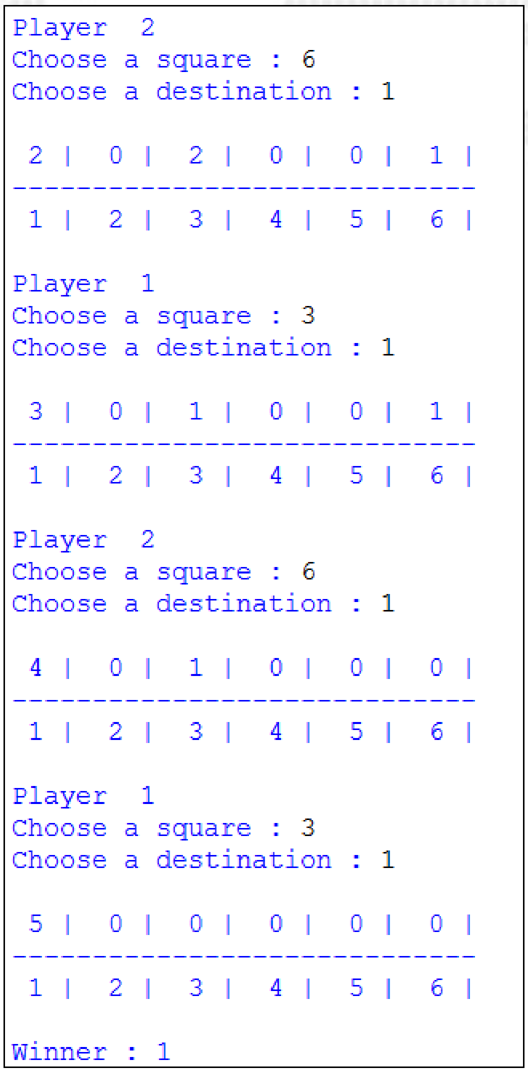
```
 0 |  3 |  0 |  2 |  3 |  1 |
------------------------------
 1 |  2 |  3 |  4 |  5 |  6 |
```

- A function « possibleSquare(board,n,i) » that returns **True** if **i** is the number of a square that contains a movable pawn. Else, it returns **False**.
- A function « selectSquare(board,n) » that permits the player to enter an index of a square that contains a movable pawn. We will suppose that such a square exist and we will not test it here. As long as this number is not valid because of the rules and the dimension of the board, we will ask to enter another number. Finally, the function will return this number.
- A function « possibleDestination(board,n,i,j) » where we suppose that **i** is the number of a square wich contents a movable pawn. This function returns **True** if **j** is the number of the square where the pawn of number **i** can move. Else, it returns **False**.
- A function « selectDestination(board,n,i) » where we suppose that **i** is the number of square wich contents a movable pawn. This function permits to enter the number of square where the pawn of coordinates **i** can move. We will suppose that such a square exists and we will not test it here. As long as this number is not valid because of the rules and the dimension of the board, we will ask to enter another number. Finally, the function will return this number.
- A procedure « move(board,n,i,j) » where we suppose that **i** is the number of square wich contents a movable pawn, and that **j** is the number of his destination. This procedure realizes the displacement.
- A function « lose(board,n) » that returns **False** if it exists at least a movable pawn on the board. Else, it returns **True**.
- A procedure « nimble(n,p) » that will use the previous subroutines (and others if it’s needed) in order to permit two players to confront a complete game on a board of **n** squares, each one containing maximum **p** pawns.

Here is the example of the sub-part 2.1, but this time with our program :



```
 0 |  0 |  1 |  2 |  0 |  2 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  1
Choose a square : 2
Choose a square : 7
Choose a square : 4
Choose a destination : 2

 0 |  1 |  1 |  1 |  0 |  2 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  2
Choose a square : 2
Choose a destination : 1

 1 |  0 |  1 |  1 |  0 |  2 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  1
Choose a square : 4
Choose a destination : 3

 1 |  0 |  2 |  0 |  0 |  2 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  2
Choose a square : 6
Choose a destination : 1
```



```
Player  2
Choose a square : 6
Choose a destination : 1

 2 |  0 |  2 |  0 |  0 |  1 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  1
Choose a square : 3
Choose a destination : 1

 3 |  0 |  1 |  0 |  0 |  1 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  2
Choose a square : 6
Choose a destination : 1

 4 |  0 |  1 |  0 |  0 |  0 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Player  1
Choose a square : 3
Choose a destination : 1

 5 |  0 |  0 |  0 |  0 |  0 |
-----------------------------
 1 |  2 |  3 |  4 |  5 |  6 |

Winner : 1
```

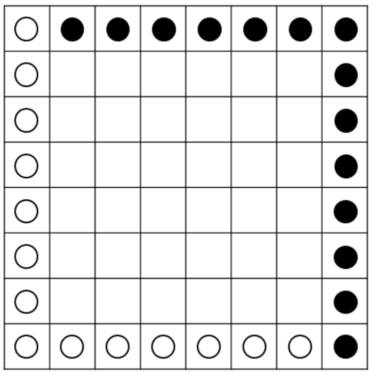
3 MING MANG

3.1 THE RULES OF THIS GAME

Importantes notes : no code is requested in this sub part, where we will explain the rules.

It is an ancestral tibetan game. Two players confront on square board of dimension n, *i.e.* the board contains n rows et n columns. The first player has white pawns and the second player has black pawns.

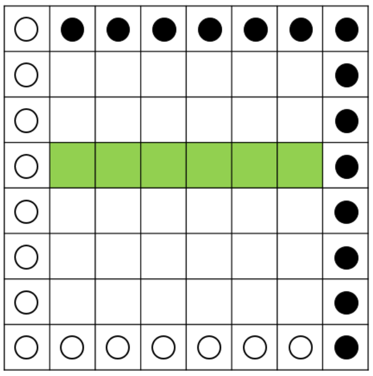
At the beginning of the game, the pawns are positioned as follows (example with a board of 8 rows and 8 columns) :



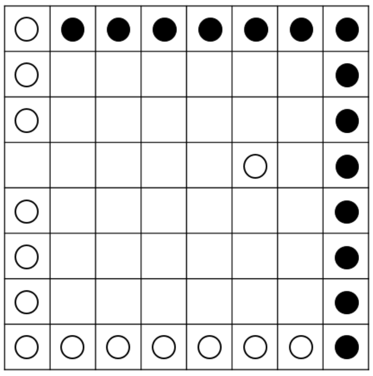
There is white pawns (respectively blacks) on the first (respectively last) columns and on the last (respectively first) row except for the last (respectively first) square.

Whites begin, and then alternatively the players move one of their pawns to an empty square located on the same row or on the same column of the square, with the constraint that there are only empty squares between the departure position and the arrival position. So, we cannot jump above another pawn whether it’s the same color or not.

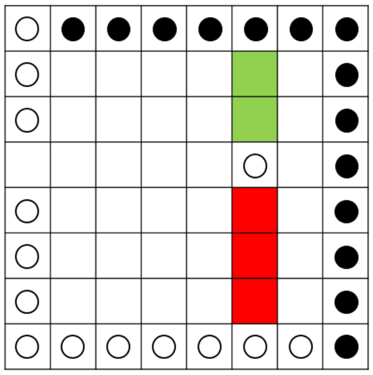
For example with this initial configuration as below, if the first player decides to move his pawn located on the fourth row and the first column, he will able to do it only on the green squares :



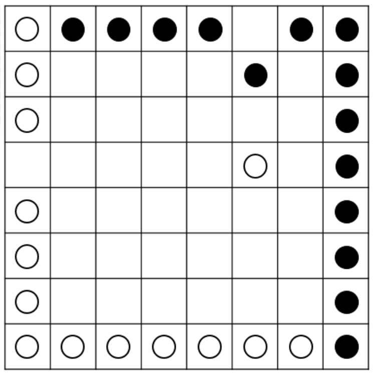
For example, the square located on the sixth column :



Then, if the second player decides to move his pawn located on the first row and the sixth column, he will be able to move only on the green squares and not on the red squares :



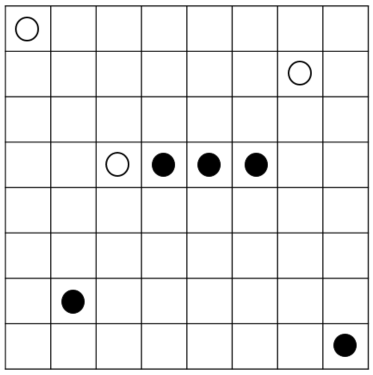
For example, the square located on the second row :



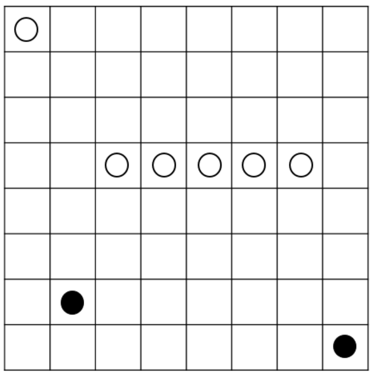
And so on.

If after the move of a white pawn (respectively black), one or more black pawns (respectively whites) are stuck between two white pawns (respectively blacks), these black pawns (respectively whites) are captured and their color changes. In the case where several pawns are stuck, they must be on successive squares and must be located on the same row or on the same column.

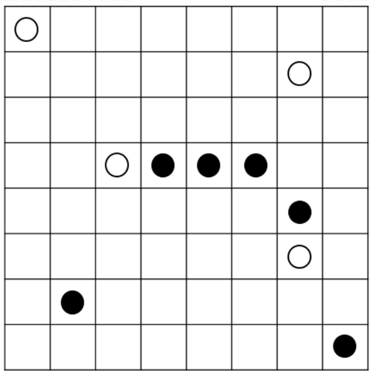
Let’s consider the following configuration :



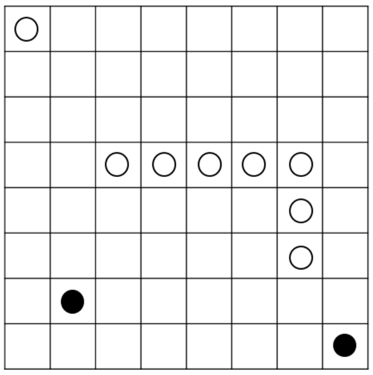
If the player with the white pawns move the pawn located on the second row and the seventh column to the square located on the fourth row and the seventh column, he captures three black pawns :



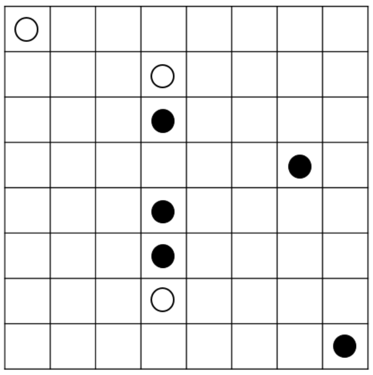
Note that catches can be done in several directions simultaneously. Let's consider the following configuration :



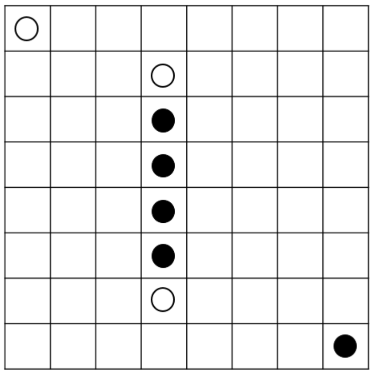
If the player with the white pawns move the pawn located on the second row and the seventh column to the square located on the fourth row and the seventh column, he captures four black pawns in both directions :



On the other hand, if during a move a pawn forms a group of friend pawns stuck by two enemy pawns, there is no capture. Let's consider the following configuration :



If the player with the black pawns move his pawn located on the fourth row and on the seventh column to the square located on the fourth row and the fourth column, there is no capture :



The winner is the last player able to move a pawn. In other words, as soon as a player does not have anymore pawns or pawns able to move, he has lost.

3.2 PROGRAMMING THIS GAME IN PYTHON

It is highly recommended to read this part before starting to code.

Importantes notes :

- We can possibly implement these subroutines in addition to those asked.
- The board will be a two-dimensional list of integers equal to 0, 1 or 2. An empty square will be represented by a 0, a pawn of the first player will be represented by a 1 and a pawn of the second player by a 2.
- Instead of re-calculating again and again the dimension of this list, we will passed in paramater of our subroutines.
- The example of visual rendering is just indicative. You can improve it.

Notations of the subroutines' parameters that we will implement :

- « board » : a two-dimensional list of integers equals to 0, 1 or 2 that represents the game board.
- « n » : strictly positive integer equal to the number of rows and columns of the « board ».
- « player » : integer that represents the player where it's the turn (for example, 1 for the first player and 2 for the second).
- « i » : any integer.
- « j » : any integer.
- « k » : any integer.
- « l » : any integer.

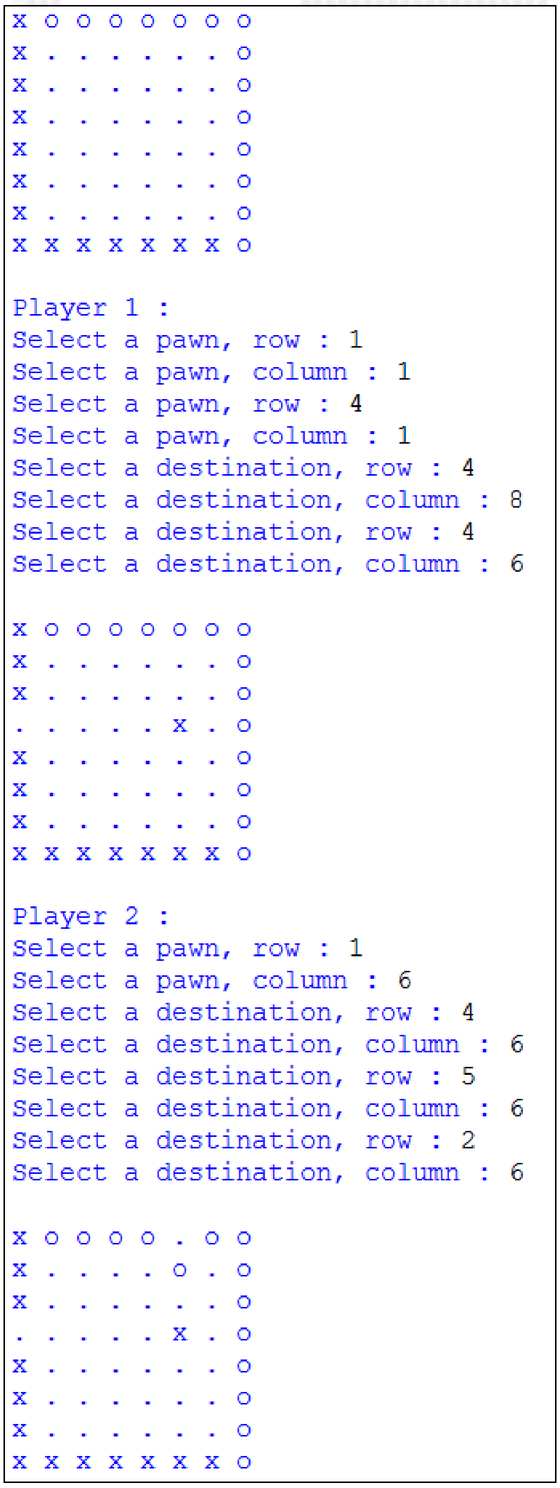
Implement the following subroutines in a file called "mingMang.py" :

- A function « newBoard(n) » that returns a two-dimensional list representing the initial state of a game board with **n** rows and **n** columns.
- A procedure « display(board,n) » that print the board on the python console. We will represent an empty square by a '.', a white pawn by a 'x' and a black pawn by a 'o'. Initially, a board of 8 rows and 8 columns will be print as follows :

```
x o o o o o o o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x x x x x x x o
```

- A function « possiblePawn(board,n,player,i,j) » that returns **True** if **i** and **j** are the coordinates of a pawn that the player **player** can move. Else it returns **False**.
- A function « selectPawn(board,n,player) » that permits to the player **player** to enter the coordinates of a moveable pawn. We will suppose that such a square exists, so we won't test it here. As long as the coordinates are not valid because of the rules and because of the dimension of the board, we will ask him to enter other two numbers. Finally, the function will return the coordinates.
- A function « possibleDestination(board,n,i,j,k,l) » where we suppose that **i** and **j** are the coordinates of a pawn able to move. This function returns **True** if **k** and **l** are the coordinates of a square where the pawn of coordinates **i** and **j** can move. Else it returns **False**.
- A function « selectDestination(board,n,i,j) » where we suppose that **i** and **j** are the coordinates of a pawn able to move. This function permits to enter the coordinates of a square where the pawn of coordinates **i** and **j** can move to We will suppose that this pawn exists, so we won't test it here. As long as the coordinates are not valid because of the rules and because of the dimension of the board, we will ask him to enter other two numbers. Finally, the function will return the coordinates.
- A procedure « move(board,n,player,i,j,k,l) » where we suppose that **i** and **j** are the coordinates of a pawn that the player **player** can move, and that **k** and **l** are the coordinates of its destination. This procedure realises the move and the possible resulting captures.
- A function « lose(board,n,player) » that will returns **False** if the player **player** can move one of his pawn. Else it returns **True**.
- A procedure « mingMang(n) » will use the previous subroutines (and others if it's needed) to permits to 2 players to play a complete game on a board of **n** rows and **n** columns.

Here is the example of the subpart 3.1, but this time with our program :



```
x o o o o o o o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x x x x x x x o

Player 1 :
Select a pawn, row : 1
Select a pawn, column : 1
Select a pawn, row : 4
Select a pawn, column : 1
Select a destination, row : 4
Select a destination, column : 8
Select a destination, row : 4
Select a destination, column : 6

x o o o o o o o
x . . . . . . o
x . . . . . . o
. . . . . x . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x x x x x x x o

Player 2 :
Select a pawn, row : 1
Select a pawn, column : 6
Select a destination, row : 4
Select a destination, column : 6
Select a destination, row : 5
Select a destination, column : 6
Select a destination, row : 2
Select a destination, column : 6

x o o o o . o o
x . . . . o . o
x . . . . . . o
. . . . . x . o
x . . . . . . o
x . . . . . . o
x . . . . . . o
x x x x x x x o
```

3.3 Bonus

Implement another version of this game (we will keep the previous functional parts), where a player is not allowed to make a move that make a configuration of the game already played.

4 INDICATIVE SCALE

This scale may change and is only **indicative**.

- Part 2 : 10 points
- Part 3 : 30 points

It makes a total of 40 points brought proportionally to a grade based on 20 points.