1. Problem Specification:

This is a board game played on a 5x5 matrix, using input from human against the computer.

Player-1 (human) places token ‘1’ anywhere on the grid
Player-2 (computer) places token ‘2’ anywhere on the grid

The objective for player-1 to win the game is to form a square of 1’s, i.e.

1 1
1 1

The objective for the computer to win the game is not to consider forming a square of 2’s, i.e.

2 2
2 2

but to prevent player-1 from forming a square of 1’s.

Play alternates between the two players: Player-1 always goes first in any new game
Player-1 is prompted to place their token (1);
Player-2 randomly generates a position to place its token (2)

The game ends when either a square of 1’s is formed (hence player-1 is the winner)
or
there are no more spaces left in which to place a token and no square of 1’s has been achieved.
Player-2 is considered to have won if no square of 1’s is formed.

2. Possible outcomes

a) Possible winning combination:
The computer wins – not because of the square of 2's as shown, but because it has blocked player-1 from forming a square of 1's

1	1	2	2	1
2	1	1	1	1
1	1	2	2	2
1	2	2	1	1
2	2	2	2	1

b) Possible winning combination:
Player-1 wins – a square of four 1's has been formed before the grid was full up

1	1	2	2	0
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0
2	0	2	0	0

3. Implementation

Start the program with a call to 'main( )'. No menu is needed.

a) Create a two player game where each player 'selects' positions in a 5 x 5 grid. The grid is initialised with every position set to 0.

b) Players take it in turn to play: player-1 always goes first
Player-1 can change any 0 to a 1
Player-2 can change any 0 to a 2

c) Player-1 cannot occupy a square if it is occupied by a player-2 token:
Player-2 can occupy any square and can overwrite a player-1 token

d) Input of a value outside of the grid range would not be allowed:
However no validation is needed

e) The program should declare the winner

f) Provide some analysis of game-play – see 6(b) Testing

g). Provide a design solution to the shortcoming of the chosen algorithm in pseudocode only – see 6(c) Testing
DON'T IMPLEMENT THE PSEUDOCODE

h) Testing can be kept to a minimum – see 6 Testing

i) **There is no need to pseudocode the implementation**
but see (7) above

4. Game play

Example I/O

a) Scenario: New game

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

b) Player-1: input square to play: 25
or
Player-1: input row number: 4
Player-1: input column number: 4

You could consider changing the index to read 1 to 5 for ease for the user?

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 1

c) Player-2: input square to play: 1 #randomly generated

2 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 1

As this number is randomly generated, you may not want to bother with a prompt?

If you do want a prompt, use input() to hold execution until return is pressed

d) Player-1: input square to play: 1

Square is occupied. Try again: 25

Square is occupied. Try again: 24

2 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 1 1

You do not have to limit the number of tries here. Sooner or later, the user must enter a valid (empty) square.

You could amend the design and allow player-1 to over-ride their original token … if there's any point to this?

e) Player-2: input square to play: 25

2 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 1 2

Player-2 is allowed to over-ride a player-1 token, but not vice-versa

f) Several player-moves later . . .

Player-1: input square to play: 19

2 2 0 0 0

2 0 0 0 0

0 0 0 0 0

0 0 1 1 1

0 0 1 1 2

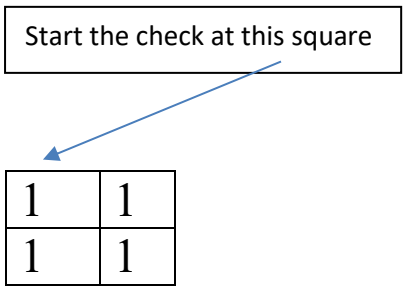
Player-1 wins! End of game

5. NOTES for implementation:

The game-play as it stands will always favour player-1. Player-2 (computer) is simply randomising its playing position.
To balance this, we are allowing the computer to over-ride any player-1 tokens, but not vice-versa and we are employing a restricted algorithm.

a. Don't change the algorithm – use it in your code: We want to analyse it:

In order to test the 'playability' of the game, we will implement it with a restricted algorithm to define a square of 1's. This should give us proof of concept so we can expand it later (but not in this assignment).
Square = take top left-hand token and see if there is the same token to the right, underneath it and diagonally to the right:



In other words, every time a token '1' is placed, check the three squares: (i) the one below it, (ii) the one to its to the right and (iii) the one below it and to the right

b. Enhance the game play to improve the balance: Allow player-2 on each turn to place 2 (or more?) tokens.

6. Testing:

a) In your .pdf file, display your test data as below

Test #	Input	Output	Comments
1	player-1 – single move player-2 – single move	Show the end grid	
2	player-1 – single move player-2 – double move	Show the end grid	
3	player-1 – single move player-2 - triple move	Show the end grid	
etc			

Note: you may have to resort to physically drawing the output grid to show the above results rather than wait for the game to end naturally

b) In a few sentences, (probably easier to do this underneath the test table):
In your opinion, is a better balance produced by allowing player-2 multiple entries at each turn?

Provide an analysis of the game-play:
Don't overdo it – do enough to be able to make a conclusion:
Explain how many tests you ran under what conditions:
e.g.
#1: player-1 single move, player-2 single move:
Three games – player-1 wins all easily

#2: player-1 single move, player-2 double move:
Three games – player-1 wins all – harder

#3: Three games – player-1 wins 2, player-2 wins 1

#4: Conclusion/Comments:

(There is no 'correct answer' to this: We are looking for some analytical ability rather than exact science.
Your colleagues may come to a different conclusion – it depends on the dispersion of the random tokens)

c) Carrying on under the experiment in (b), have a look at the algorithm:

In a short sentence, identify its primary weakness under the certain condition where the algorithm fails;

We only want to use this type of algorithmic pattern for square detection: Don't expand the algorithm to <u>all</u> the potential squares around it: i.e.

	1			

	1			

	1			

	1			

We like this approach: (i) check below it, (ii) check to the right, (iii) check below it and to the right

For the current algorithm:
def square()
input token = ‘1’
square = (input-position and position (bottom) and position (right) and position (diag-right)) = 1

(d) For your amended algorithm to correct the failure under a certain condition. Identify it and output a pseudocode solution:
def (alg-extend):
input token’1’
if token at . . .
then invert the square
square: input-position and . . . etc

e) There is no requirement for ‘graphical’ output. Use text-based output to display the grid

f) Keep validation to a minimum