Computer Science

COMPSCI 130 S2

Assignment

Due: **Friday, 30th September 2022, 11:59pm**
Worth: **6% of the final mark**

Introduction

"Connect Four" (also known as Four Up, Plot Four, Find Four, Four in a Row, Four in a Line, etc.) is a popular two-player game where players choose a colour and then take turns dropping one coloured disk from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own disks. The image on the right shows an example.

In this assignment, you are going to implement a variation of this game and a simple AI which allows you to play against the computer.

Warning

DO NOT SUBMIT SOMEONE ELSE'S WORK:

- The work done on this assignment must be your own work. Think carefully about any problems you come across and try to solve them yourself before you ask anyone for help.
- Under no circumstances should you take or pay for an electronic copy of someone else's work. This will be penalised heavily.
- Under no circumstances should you give a copy of your work to someone else
- The School of Computer Science uses copy detection tools on the files you submit. If you copy from someone else, or allow someone else to copy from you, this copying will be detected, and disciplinary action will be taken.
- To ensure you are not identified as cheating you should follow these points:
    - Always do individual assignments by yourself.
    - Never give another person your code.
    - Never put your code in a public place (e.g., forum, your web site).
    - Never leave your computer unattended. You are responsible for the security of your account.
    - Ensure you always remove your USB flash drive from the computer before you log off.

Your name, UPI and other notes

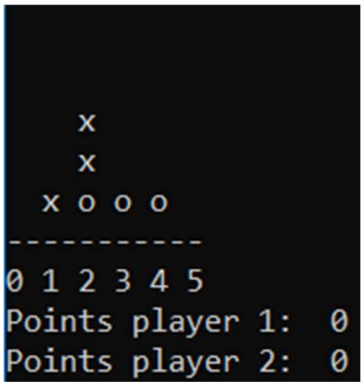
- All files should also include your name and ID in a comment at the beginning of the file.
- All your files should be able to be compiled without requiring any editing.
- All your files should include good layout structure

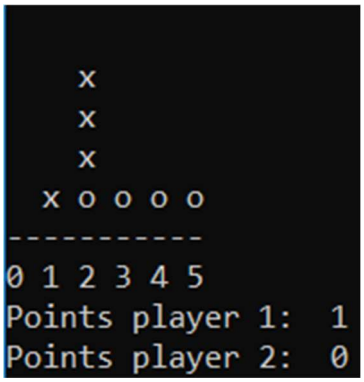
The modified game rules

Our game board (the "grid") has a square shape with n rows and n columns, where $1 \leq n \leq 10$ can be specified when initialising the game (note that for n<4, no player can score 4-in-a-row). We number the columns from left to right and refer to them as slots (since for the real-world version of the game they have a "slot" at the top where we insert the disks).
The rows are numbered from bottom to top, i.e., the bottom row is 0, the row above it 1 etc.
We refer to the position of a disk in the board as a tuple (column, row).
We represent the disks of player 1 (human player) with circles (o) and the disks of player 2 (computer player) with crosses (x).

The image below on the left shows a game board of size 6 x 6 after 3 moves each by the human and computer players. The numbers 0 to 5 at the bottom are the columns (slots). The image below on the right shows the same game board after one more move by the human player (insert disk into slot 5) and by the computer player (insert disk into slot 2).

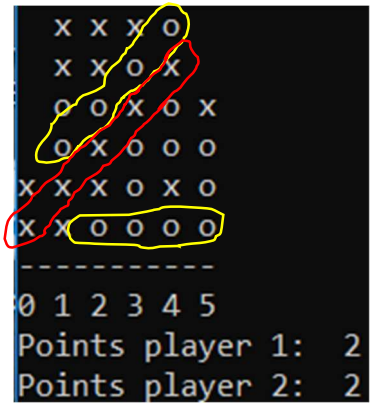
The human player has disks (circles) at positions (2,0), (3,0), (4,0), and (5,0), i.e., the disks occupy the bottom row from column 2 to 5. The computer player has disks (crosses) at positions (1,0), (2,1), (2,2) and (2,3).





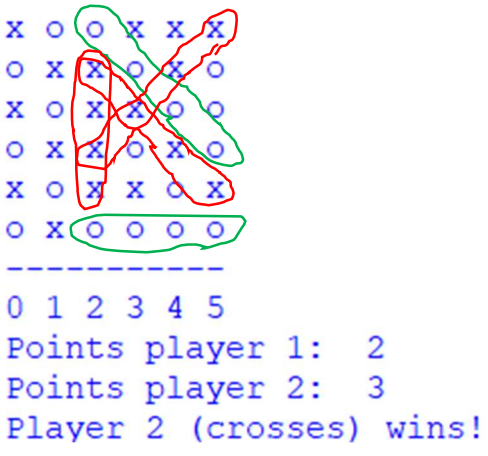
If a player has k connected pieces (horizontal, vertical or diagonally), where $k > 4$, then this counts as k-3 lines of 4 connected pieces.

For example, in the image on the right, player 1 (circles) has two winning lines: One horizontal line of four disks from (2,0) [column 2, row 0 (bottom)] to (5,0) [column 5, row 0] and one diagonal line of four disks from (1,2) to (4,5).
Player 2 (crosses) has a line of 5 connected disks which goes from position (0,0) to (4,4). Since the line has 5 connected pieces this counts as (5-3)=2 connected lines of 4 (one from (0,0) to (3,3) and one from (1,1) to (4,4)).



The game continues until the entire board is full (i.e., there are n x n disks) and **the player with the most lines of 4 connected pieces wins**.

Example: in the image on the right, player 1 (circles) has two lines of 4 connected pieces: a horizontal line from (2,0) to (5,0) and a diagonal line from (2,5) to (5,2).
Player 2 (crosses) has three lines of 4 connected pieces: a vertical line from (2,1) to (2,4), a diagonal line from (2,2) to (5,5), and a diagonal line from (2,4) to (5,1).
Hence, player 2 is the winner.



Section 1: The class GameBoard – Basic Functionality (10 marks)

The class **GameBoard** is used to implement the game board (grid). In Section 1 you will implement basic functionality including adding a disk into the board, testing whether a game is finished, and displaying the game board.

The class **GameBoard** is defined as follows:

```
class GameBoard:
    def __init__(self, size):
        self.size = size
        self.num_entries = [0] * size
        self.items = [[0] * size for i in range(size)]
        self.points = [0] * 2
```

The variable **size** specifies the size of the game board, i.e. the number **size** of columns and rows. You can assume that 1≤ **size** ≤10 and you don't need to test for this.
The variable **num_entries** is a list, which specifies the number of disks in each column. The initial value for each column is zero.
The game board itself is represented by the 2D array `items`. This is a list of lists. The first dimension refers to the column number and the second dimension to the row number, i.e. **items[i][j]** is the value of the item in the i-th column and j-th row. Each item is initialised with the value 0 (meaning, no disk). The value 1 will represent a disk of player 1 and the value 2 a disk of player 2.
The variable **points** represents the points (number of lines of 4 connected pieces) of each player: **points[0]** are the points of player 1, and **points[1]** are the points of player 2.

Copy the class GameBoard above into the CodeRunner answer box, and implement the following methods:

`num_free_positions_in_column(self, column):`

Returns the number of free positions in column **column**. For example, if we have a board of size 6 x 6 then at the start of the game the method should return the value 6 for each column; after a player inserted a disk into a column the value should be 5 for that column etc.

`game_over(self):`

Returns **True** if the game is over and otherwise **False**. A game is over if none of the columns has any free positions.

`display(self):`

Displays the game board with the rows displayed from top to bottom (**size**-1 down to 0) and the columns from left to right (0 to **size**-1). Items with value 0 are displayed as empty spaces, items with value 1 (player 1) are displayed as circles (small letter 'o'), items with value 2 (player 2) are displayed as crosses (small letter 'x'). Between each item is one space in the horizontal direction. Under the items is a horizontal line (displayed using minus signs) and below that the column numbers. Finally, at the bottom are the number of points of each player.

Example:

The code below inserts items into the game board and updates the items in each column (note: this will be done later by an **add** method). The picture on the right illustrates the output of the **display** method and **num_free_positions_in_column** method.

```
board = GameBoard(4)
board.items[2][0]=1
board.items[2][1]=2
board.num_entries[2]=2
print("Num free items in column 1:",board.num_free_positions_in_column(1))
print("Num free items in column 2:",board.num_free_positions_in_column(2))
board.display()
```

```
Num free items in column 1: 4
Num free items in column 2: 2


   x
   o
-------
0 1 2 3
Points player 1: 0
Points player 2: 0
```

Section 2: The class GameBoard – Insert a disk into the game board (10 marks)

Use your implementation of the class GameBoard from section 1.
NOTE: You need to implement section 1 correctly to complete this task!

First add the method below – you will implement it in section 3, but since we need it for this section, we will use this incomplete implementation as a placeholder:

```
    def num_new_points(self, column, row, player):
        return 0
```

Your task is to implement the method

```
add(self, column, player):
```

in the class **GameBoard**. The method inserts a disk into the slot **column** of the game board. If the column is full (**num_entries** ≥ size) or doesn't exist (**column** < 0 or **column** >= **size**) the method returns **False**. Otherwise, the method determines the first available row number (use the **num_entries** variable), updates the game board value for **item[column][row],** increments **num_entries** for that column by one, and adds the number of new points (from the method **num_new_points**) for player **player** created by adding the disk. The method then returns **True**.

NOTE: The dummy implementation of the method **num_new_points** returns currently always zero, i.e., in the test cases for this question, the players will always have zero points. In the next section you will implement this method so that it calculates the new 4-in-1-row sequences created by inserting a disk into a slot.

Example:

```
board = GameBoard(3)
board.add(0,1)
board.add(1,2)
board.add(1,1)
board.add(1,2)
board.add(1,1)
print("number disks in each slot is: ",board.num_entries)
board.display()
```

results in the output:

```
Number disks in each slot is:  [1, 3, 0]
  x
  o
o x
-----
0 1 2
Points player 1: 0
Points player 2: 0
```

Note that column 1 has only 3 entries, even though we added 4 disks. The reason for this is, that the size of the game board is 3. Hence the last add command `board.add(1,1)` returns **False** and does not modify the game board.

Section 3: The class GameBoard – Calculate how many 4-in-1-row sequences an inserted disk creates (20 marks)

Use your implementation of the class GameBoard from section 1 and 2.
NOTE: You need to implement section 1 and 2 correctly to complete this task!

Your task is to implement the method

```
num_new_points(self,column,row,player):
```

in the class **GameBoard**. The method calculates how many 4-in-a-row sequences a newly inserted disk by player **player** at position (**column, row**) creates. You can assume that the game board has been correctly updated, i.e., it contains a disk at the given position.

NOTE: Only calculate the newly created 4-in-a-row sequences. You need to check for horizontal, vertical, and diagonal sequences.

Example 1:

Executing the code below on the left, results in the output shown on the right.

```
board = GameBoard(5)
board.add(0,1)
board.add(0,2)
board.add(1,1)
board.add(1,2)
board.add(2,1)
board.add(2,2)
board.add(3,1)
board.add(3,2)
board.add(4,1)
print("Newly created 4-in-a-row sequences: ",board.num_new_points(4,0,1))
board.display()
```

```
Newly created 4-in-a-row sequences:  1
 x x x x
 o o o o o
 ---------
 0 1 2 3 4
Points player 1: 2
Points player 2: 1
```

Note that after the first eight inserted disks, both players have one 4-in-a-row sequence (player 1 from (0,0) to (3,0), and player 2 from (0,1) to (3,1)). The method call **board.add(4,1)** inserts for player 1 a disk into slot 4 and creates one new 4-in-a-row sequence in horizontal direction from (1,0) to (4,0).

Example 2:

Inserting a disk can create multiple new 4-in-a-row sequences as shown in this example. The last command (player 1 inserts a disk into slot 3) creates two 4-in-a-row sequences: a horizontal one from (0,0) to (3,0), and a diagonal one from (0,3) to (3,0).

```
board = GameBoard(5)
board.add(0,1)
board.add(0,2)
board.add(1,1)
board.add(1,2)
board.add(2,1)
board.add(0,2)
board.add(2,1)
board.add(2,2)
board.add(1,1)
board.add(1,2)
board.add(0,1)
board.add(0,2)
board.add(3,1)
print("Newly created 4-in-a-row sequences: ",board.num_new_points(3,0,1))
board.display()
```

```
Newly created 4-in-a-row sequences:  2
 x
 o x
 x o x
 x x o
 o o o o
 ---------
 0 1 2 3 4
Points player 1: 2
Points player 2: 0
```

BACKGROUND: In sections 4 and 5 we will implement a simple AI, which enables a human player to compete with the computer. The computer selects a move based on a simple heuristic:

1) It tries to find a free slot as close to the middle of the board as possible
2) It always selects a move which enables it to complete the maximum 4-in-a-row sequences. If there are multiple such moves it selects the free slot as close to the middle as possible.
3) If there is no move enabling the computer to create a 4-in-a-row sequence, then it selects a move blocking the maximum possible 4-in-a-row sequences for the human player. If there are multiple such moves it selects the free slot as close to the middle as possible.

NOTE: Our heuristic only uses a 1-move look ahead (one move of the human and computer player). We could improve the AI considerably by looking multiple moves ahead. This requires recursion and a tree data structure, which we will only discuss towards the end of this semester. For an example of such a "game tree" see: https://en.wikipedia.org/wiki/Minimax#Example_2.

Section 4: The class GameBoard – Calculate all free slots as close to the middle of the board as possible (20 marks)

NOTE: You only need the methods init() and num_free_positions_in_column(column) from the class GameBoard (see section 1) to complete this task!

Your task is to implement the method

```
free_slots_as_close_to_middle_as_possible(self):
```

in the class **GameBoard**. The method returns a list of all free slots (i.e., columns which are not full yet), in the order of their distance to the middle of the board. If there are two such columns, the smaller column number comes first.

Example 1:

```
-------
0 1 2 3
```

Assume we have an empty board of size 4 as shown above. The middle of the board is (0+3)/2=1.5. Since both column number 1 and 2 have the same distance to 1.5 the first free slot is the smaller number, i.e., 1 followed by 2. Slots 0 and 3 also have the same distance to 1.5, so again we choose the smaller column number first, i.e. 0. Hence the returned list of free slots is [1, 2, 0, 3].

Example 2:

Now assume we have a board of size 4, where column 2 is full, as illustrated on the right. The middle of the board is (0+3)/2=1.5. Since columns number 1 and 2 have the same distance to 1.5, the first free slot is the smaller number, i.e. 1. However, column 2 is full, so is not part of the resulting list. Hence the returned list of free slots is [1, 0, 3].

```
  x
  o
  x
  o
-------
0 1 2 3
```

Example 3:

Now assume we have a board of size 7, where columns 2 and 3 are full, as illustrated on the right. The middle of the board is (0+6)/2=3. However, that column is full, so not part of the result. The next closest column numbers to 3 are 2 and 4. Column 2 is full, so the closest free slot to the middle is 4. The next closest column number are 1 and 5, and then 0 and 6. Hence the returned list of free slots is [4, 1, 5, 0, 6].

```
x o
o x
x o
o x
x o
o x
x o
-------------
0 1 2 3 4 5 6
```

Section 5: The class GameBoard – Find slot resulting in the maximum number of points (20 marks)

NOTE: This task requires correct implementations of most of sections 1-4!

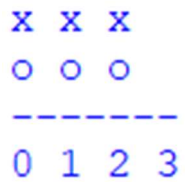
Your task is to implement the method

```
column_resulting_in_max_points(self, player):
```

in the class **GameBoard**. The method computes the slot into which the player **player** needs to insert a disk in order to obtain the maximum number of points (i.e., maximum number of 4-in-a-row sequences). If multiple such slots exist, the closest to the middle of the board is chosen (using the method you implemented in section 4). The return value is a tuple **(slot_number_for_max_points, max_points).**

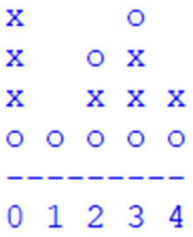
Example 1:

Assume we have a board of size 4 as shown on the right.
If we call the method for player 1 (circle) then the best move is slot 3, since it is the only move resulting in a new 4-in-1-row sequence. Hence the return value of the method should be (3, 1).
If we call the method for player 2 (crosses) then no move will result in a new 4-in-1-row sequence. Hence all moves result in the same maximum number of points (i.e., zero) and the method should return the slot as close to the middle as possible, i.e. ,1. Hence the return value of the method should be (1, 0).



Example 2:

Assume we have a board of size 5 as shown on the right.
If we call the method for player 1 (circle) then the best move is slot 1, since it is the only move resulting in a new 4-in-1-row sequence. Hence the return value of the method should be (1, 1).
If we call the method for player 2 (crosses) then slot 0 would result in one new 4-in-a-row sequence. However, slot 1 results in two new 4-in-a-row sequences ((0,1) to (3,1) and (1,1) to (4,1)). Hence the return value of the method should be (1, 2).



PLAYING THE GAME :-)

After implementing sections 1-5 you can play 4-in-a-row against the computer, by downloading the file FourInARowGameWithAI.py from Canvas (Assignments -> Assignment) and inserting your implementation of the GameBoard class.

Section 6: Creative Extension (10 marks)

Extend your program in a "cool" way, e.g., using a more attractive graphical representation, more functionalities, or better usability (e.g., clearly indicate the existing 4-in-a-row sequences, a help method etc.)
Please explain your extension in the file AssCool.pdf and submit a separate source file titled AssCool.py. The markers will evaluate your submission using the following criteria:

- Explanation (how detailed, clear, and insightful is your explanation?)
- Novelty and creativity (how creative / interesting / novel / "cool" is your solution?)
- Technical difficulty (what is the complexity of your solution in terms of software development and/or algorithm development skills reflected in your solution?)

Submission

Submit your assignment online via the Assignment Dropbox (https://adb.auckland.ac.nz/) at any time before the due date. Submit the following files:

1. **Ass.py** (Python source file with the solution for sections 1-5)
2. **AssCool.py** (Python source file with the solution for section 6)
3. **AssCool.pdf** (pdf-file with your explanations for the extension described in section 6)

Remember to include your name, UPI and a comment at the beginning of each file you create or modify. You may make more than one submission but note that every submission that you make replaces your previous submission. Submit ALL your files in every submission. Only your very latest submission will be marked. Please double check that you have included all the files required to run your program and AssCool.pdf in the zip file before you submit it.

Note 1. We will only mark your submitted code. Your testing results on CodeRunner will not be counted. Note that the CodeRunner tests were created to help you finding common errors, but they do not represent an exhaustive test. The markers will use different test cases and code which passes all CodeRunner test cases might still fail the marker test cases.
Note 2: Your program must compile and run to gain any marks. Your solution in Ass.py must run on CodeRunner.

MARKING GUIDE

TOTAL: 100 marks

Section 1: The class GameBoard – Basic Functionality (20 marks)	**10 marks**
Correct implementation of the method **num_free_positions_in_column**	**2 marks**
Correct implementation of the method **game_over**	**2 marks**
Correct implementation of the method **display**	**6 marks**

Section 2: Insert a disk into the game board (10 marks)	**10 marks**
Method **add** works correctly for an empty game board	**2 marks**
Method **add** works always correctly	**8 marks**

Section 3: Calculate how many 4-in-1-row sequences an inserted disk creates	**20 marks**
Method **num_new_points** correctly identifies 4-in-a-row sequences in horizontal direction	**5 marks**
Method **num_new_points** correctly identifies 4-in-a-row sequences in vertical direction	**5 marks**
Method **num_new_points** correctly identifies 4-in-a-row sequences in diagonal direction	**10 marks**

Section 4: Calculate all free slots as close to middle of board as possible	**20 marks**
Method **free_slots_as_close_to_middle_as_possible** works correctly for empty game boards	**5 marks**
Method **free_slots_as_close_to_middle_as_possible** works correctly for all cases	**15 marks**

Section 5: Find slot resulting in maximum number of points	**20 marks**
Method **column_resulting_in_max_points** fulfils test case 1 in CodeRunner	**5 marks**
Method **column_resulting_in_max_points** fulfils test case 2 in CodeRunner	**5 marks**
Method **column_resulting_in_max_points** always works correctly	**10 marks**

Section 6: creative extension	**10 marks**
Explanation (how detailed, clear and insightful is your explanation in the file A2cool.pdf?)	3 **marks**
Novelty and creativity (how creative / interesting / novel / "cool" is your solution?)	4 **marks**
Technical difficulty (what is the complexity of your solution in terms of software development and/or algorithm development skills reflected in your solution?)	3 **marks**

Section 7: Coding style (for the entire submission)	**10 marks**
Appropriate comments at the top of programs	1 **marks**
Easy to read	4 **marks**
Follows variable naming conventions	5 **marks**