

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



COMPUTER ARCHITECTURE – CO2007

Assignment

FOUR IN A ROW

Student: 2153488 – Nguyen Minh Khue

HO CHI MINH CITY, OCTOBER 2022

ASSIGNMENT SPECIFICATIONS

1. Introduction

Four in a Row is the classic two player game where you take turns to place a counter in an upright grid and try and beat your opponent to place 4 counters in a row.

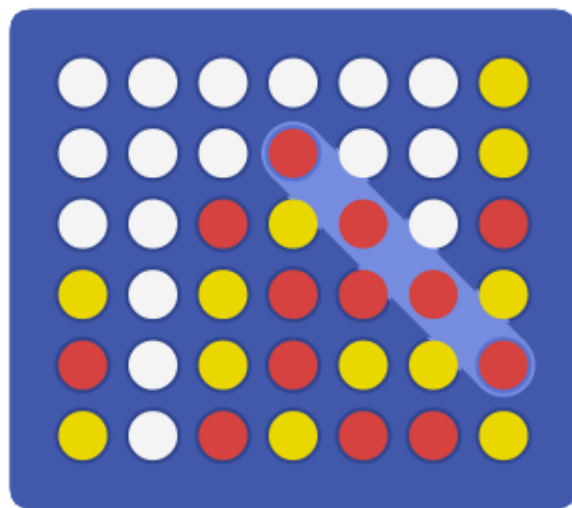


Figure 1: Four in a Row. [1]

The game is played with a seven-column and six-row grid, which is arranged upright. The starting player is randomly chosen, pick a game piece color (yellow or red) and can place a piece in any column. Each player then alternately takes a turn placing a piece in any column that is not already full. The piece fall straight down, occupying the lowest available spot within the column or be stopped by another piece. The aim is to be the first of the two players to connect four pieces of the same colour vertically, horizontally or diagonally (an example is shown in Figure 1). If each cell of the grid is filled and no player has already connected four pieces, the game ends in a draw, so no player wins.

2. Goals

My objective when creating this game is not only to do an assignment to get high grades, but also to make a friendly and fun games which I can share and play with my friends and close ones. With that being said, I tried to make the game as funny and playful as possible. All the dialogues in this game are made by me and only me so it was based on my interest. I hope that anyone who plays this game will feel joyful and relaxed when interacting with it and have quality time playing this game!

3. Implementation

➤ Dialogues

No game can be completed without an interesting interface, so I made a lot of dialogues so that players can keep up with the game without any confusion. My dialogues include:

- Intro of game
 - ◆ For the intro I will use a text file containing the interface, and use procedures to open file and read from file (for this I will use a read-buffer, refs in figure 2).
- Print out game's Board (most important)
 - ◆ Simple looping will print out '|' first, then goes the element 'x', 'o' or '*' if empty. After printing 7 times (width), this will print out an endline '\n'.
- Prompt for each player
 - ◆ Telling which player to start, tell them to choose a piece.
- Conditions that players must follow
 - ◆ Players can only choose from 1-7. Invalid moves whether column already full or column out of range will be discussed below.
- Warning when players take an invalid move and the type of error they made
 - ◆ Players will have a message showing which error they made so that they can avoid making the same mistake again.
- Number of warnings they have left
 - ◆ Show each player how many warnings they have left before being disqualified.
- Question the player if they want to undo their move (if they still can)
- Number of undoes a player has
- Show victory player
- Indicate game tie if there's no valid move left
 - ◆ I will use a variable to keep track of the game play, this variable will increase after a move. And if it reaches the limit moves (height*width), a Game Tie Procedure will be called.
- Question of players want to restart
- Print out greets when game ends.

- ```
intro: .asciiz "/Users/khuenguyen/Desktop/ASSCA/intro.txt"
outtro: .asciiz "/Users/khuenguyen/Desktop/ASSCA/outtro.txt"
buffer_intro: .space 10000
buffer_outtro: .space 10000
undoPlayer1: .asciiz "Player 1 undo ? Yes-y||No-n\n"
undoPlayer2: .asciiz "Player 2 undo ? Yes-y||No-n\n"
totalUndos: .asciiz "Total undos left: "
gameTie: .asciiz "No more moves available. Game tied !!\n(0 o 0)\n"
randomPlayer1: .asciiz "\nPlayer 1 will start the game\n"
randomPlayer2: .asciiz "\nPlayer 2 will start the game\n"
pickColor: .asciiz "Pick a piece: x or o: "
repickColor: .asciiz "\nC'mon game haven't even started and you already made a wrong input.\nTry again (-_-)\n"
player1Prompt: .asciiz "\nPlayer 1's turn, choose a column(1-7): "
player2Prompt: .asciiz "\nPlayer 2's turn, choose a column(1-7): "
invalidInput: .asciiz "Invalid input: Choose from 1 to 7!!\n"
invalidInputFull: .asciiz "Invalid input: Column_Full."
firstViolation: .asciiz "Try again, 2 chances left!(n(==)=)\n"
secondViolation: .asciiz "Try another column, last chance!\n \(\(^)\)/\n"
disqualified: .asciiz "You violated 3 times !!!(X _ X)\n"
player1Win: .asciiz "Player 1 wins!<(^_^)>\n"
player2Win: .asciiz "Player 2 wins!<(^_^)>\n"
restartPrompt: .asciiz "Press 1 to restart\n"
*txt
```

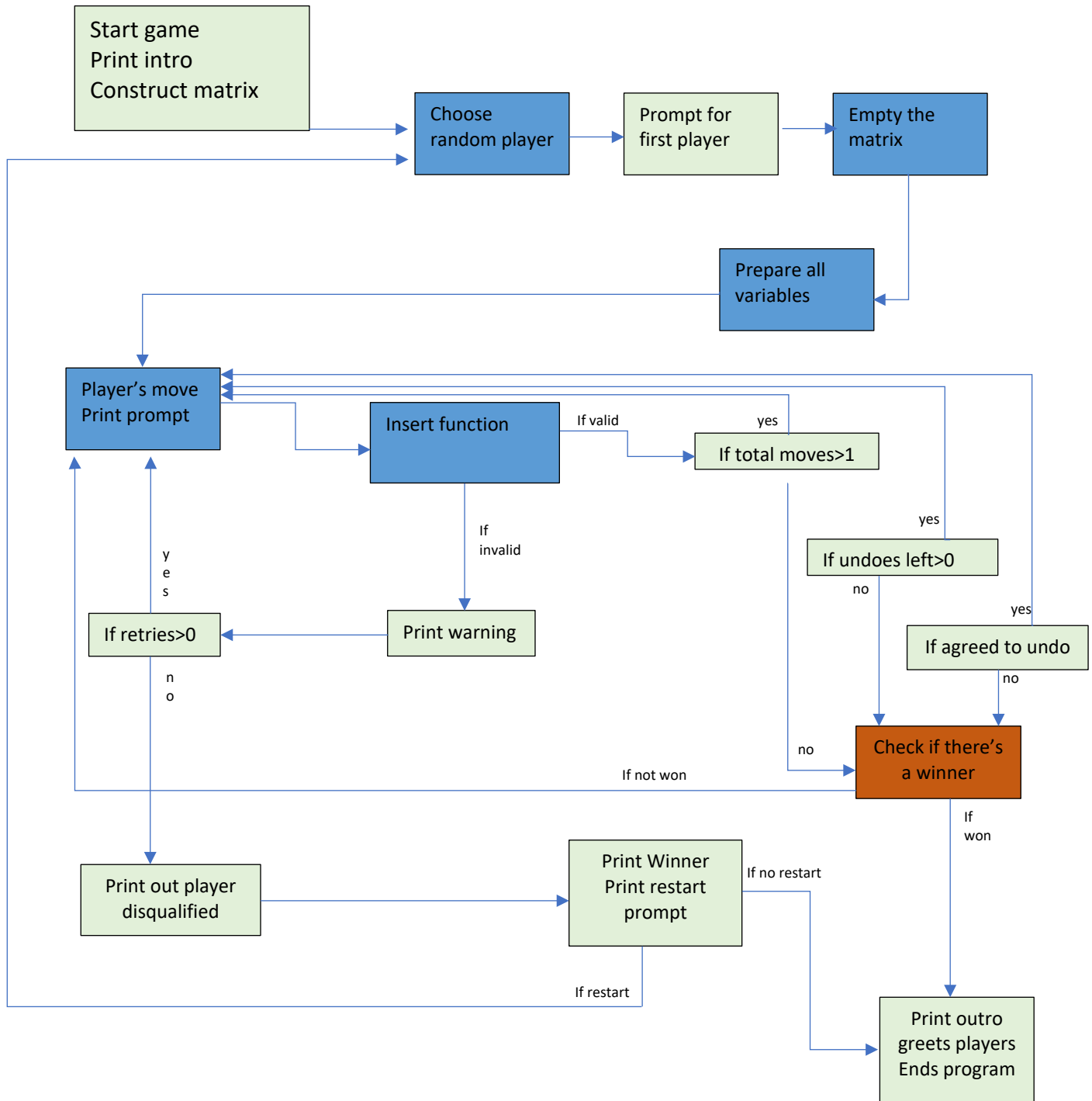
[illegible]




|                                                                                                                                                                                                                                                                                                                                                                                                                                                  |  |  |  |  |  |  |  |  |  |                                                                                                                                                                                                                                                                                                                                                                                                                                                  |  |  |  |  |  |  |  |  |  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--|--|--|--|--|--|--|--|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--|--|--|--|--|--|--|--|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                  |  |  |  |  |  |  |  |  |  |                                                                                                                                                                                                                                                                                                                                                                                                                                                  |  |  |  |  |  |  |  |  |  |
| <p>Diagram of a 10-bit shift register with feedback. The register has 10 stages labeled 1 to 10. Stage 10 is the output. Stage 10 is connected to a feedback loop that goes through a 2-to-1 multiplexer. The multiplexer has two inputs: the output of stage 10 and the output of stage 9. The output of the multiplexer is connected to stage 1. The multiplexer is controlled by a signal 'f'. The output of the register is labeled 'f'.</p> |  |  |  |  |  |  |  |  |  | <p>Diagram of a 10-bit shift register with feedback. The register has 10 stages labeled 1 to 10. Stage 10 is the output. Stage 10 is connected to a feedback loop that goes through a 2-to-1 multiplexer. The multiplexer has two inputs: the output of stage 10 and the output of stage 9. The output of the multiplexer is connected to stage 1. The multiplexer is controlled by a signal 'f'. The output of the register is labeled 'f'.</p> |  |  |  |  |  |  |  |  |  |

Computer Architecture - Semester 1 (2022 - 2023)

#### 4. Algorithm

##### ➤ Pseudocode:



As demonstrated in the diagram above, the principle of the game is quite simple and straight forward, all the light green  boxes are simple command **printing prompt and messages**, while the light blue boxes  represents some more **complex looping and branching** and then finally the heavy orange  box marks the most sophisticated procedure call of the game, and also one of the most important – **checking if a player has just made a winner move.**

➤ Details:

Throughout the process of making of the game, I will store some of the important information in the \$s0-\$s7 register, the information includes:

- Base address of matrix in: \$s0
- Current player in: \$s1 (number 1 for player 1 or 2 for player 2)
- Current piece by that player in \$s2 ('x' or 'o')
- Number of moves 2 players has made: \$s3
- Number of warnings player 1 has: \$s4
- Number of warnings player 2 has: \$s5
- Number of undoes player 1 has: \$s6
- Number of undoes player 2 has: \$s7

- Matrix:

Clearly we need a 2-d array for this game (row and column). Mips only support linear array structure so we will improvise, use a 1-d array of size 42 to store 42 space of matrix, and access elements of which row and column using formulas:  $\text{matrix}[i][j] = \text{matrix} + j * \text{width} + i$ . (matrix contains address of array).

- Choose starting player randomly

For choosing a random player to start the game, I will generate a number by calling syscall after load 42 into \$v0, the number generated will be from 0-99, after adding another 100 in to the number, I take modulo 2 from it. If result is 0, player 1 is chosen to start, else player 2 will start the game

```
li $a1, 100 #Here you set $a1 to the max bound.
li $v0, 42 #generates the random number.
syscall
add $a0, $a0, 100 #Here you add the lowest bound
addi $t0, $zero, 2
div $a0, $t0
mfhi $s1
addi $s1, $s1, 1
beq $s1, 1, player1Start
beq $s1, 2, player2Start
```

After a starting player has been choose, program will jump to choosing piece, this is straight forward (only enter 'x' or 'o'). if ad invalid piece is entered, a funny message will print out.

```

Player 2 will start the game
Pick a piece: x or o: q
Game haven't even started and you already made a wrong input.
Try again (-_-)
Pick a piece: x or o: |

```

- Choose column

When a valid piece is given, the game will officially start. First procedure of game is player choosing column. Players are asked to choose a column between 1 to 7.

Simple conditions will be used to check whether a column  $\leq 0$  or  $> 7$  is entered. If an invalid move is made (whether column is full or out of range) program will jump to procedure based on which player has made the wrong move (info in \$s1).

After program has known which player has made wrong move, procedure managing error will examine \$s4 or \$s5 based on which player is wrong (those 2 registers store total warnings left for each player). If there are still 3 warnings left, first warning will be print out and jump back to choosing column. If there are 2 warnings left, print out warning and tell players they only have 1 chance left, jump back to choosing column. If warning is only 1 left, print which player has been disqualified and jump the End Game procedure.



```

invalidCol:
 li $v0,4
 la $a0,invalidInput
 syscall
 beq $s1,1,player1Wrong
 beq $s1,2,player2Wrong
 player2Wrong:
 addi $s5,$s5,-1
 beq $s5,2,firstTime
 beq $s5,1,secondTime
 beq $s5,0,noMore
 player1Wrong:
 addi $s4,$s4,-1
 beq $s4,2,firstTime
 beq $s4,1,secondTime
 beq $s4,0,noMore
 firstTime:
 li $v0,4
 la $a0,firstViolation
 syscall
 j gameplay
 secondTime:
 li $v0,4
 la $a0,secondViolation
 syscall
 j gameplay
 noMore:
 li $v0,4
 la $a0,disqualified
 syscall
 beq $s1,1,player2W
 j player1W

```

“firstTime”, “secondTime” and “noMore” are labels representing functions that will be executed if player still have 3, 2 and 1 warnings left respectively. At “noMore”, game will print out disqualified player, and “player1W” or “player2W” will print winner and question to restart game.

- Insert piece into matrix



After a valid input column is entered, program calls for insert function. Insert function will run from bottom up of matrix and stop at first index that is empty (ascii value =0). Here I subtract column with 1 because matrix is 0\_indexed.

If found an empty space, “drop” function will be called, this function basically change empty element at that location to ‘x’ or ‘o’ depending on argument \$a2 (store current piece).

Else if offset is negative and program hasn’t found an empty space, program will jump to “fullRow” procedure. This procedure will print out full\_row error and jump to error managing branch mentioned before.

```
j undo
insert: # address in a1. col in a0, element in a2
 addi $t0,$zero,height
 addi $t0,$t0,-1
 mul $t0,$t0,width
 add $t0,$t0,$a0
 addi $t0,$t0,-1
 findspace:
 add $t1,$a1,$t0
 lb $t1,0($t1)
 beq $t1,$zero,drop
 addi $t0,$t0,-width
 slt $t2,$t0,$zero
 beq $t2,1,fullRow
 j findspace
 drop:
 add $t1,$a1,$t0
 sb $a2,0($t1)
 move $a1,$s0
 move $a3,$a2
 move $a2,$t0
 j undo
```

- Undo:

Before jumping to checking winner, procedures called “player1Repick” or “player2Repick” (based on \$s1) if total moves are more than 1 and number of undoes stored in \$s6-7 are greater than 0.

Before undo prompt, game will print out matrix so that player can see clearly whether they want a repick or not. If that player can still make undoes, procedure “repick1” or “repick2” will asked player to enter ‘y’ to repick, or any other key to continue. If ‘y’ is entered, decrease undo chance stored \$s6-7 by 1 and jump back to Choose column, if not jump to check win.

- Check Win:

After function “drop” function is called, it will jump to check win function. Before the actual checking for winner, this function will see if total \$s3 is less than 42, if yes it will cancel checking and jump to “tieGame” procedure. If “tieGame” hasn’t occurred, actual checking for winner to proceed.

During checking, I will save most recent piece played in register \$a3, address of array in \$a1 and offset of most recent move in \$a2. Throughout the process of checking I will only change the number added to offset to get to the next position, not the offset itself. Doing so I won’t have to worry about offset of current move being changes.

Check win is quite complex because it has to run through 4 types of check, which are:

- ◆ Check win vertically:

This particular checking will only be called if index inserted is in row less than 5 (max row is 7). This makes sense because in four in a row, pieces will only be dropped down from high to low, if you want to have 4 pieces stacked on each other the last index will be at row 4 at least.

Check downwards will be straight forward, we increase the offset by 7 (to get to the next row) and check if it is the same with the element we already stored in \$a3. If yes jump again and increase result \$v0, if no, jump to the branch placed in check win function called “aftercheckDown”. After check down will check result stored in \$v0 after checking down to see if it is equal to 4. If it is jump to End Game procedure. If not continue checking horizontally.

- ◆ Check win horizontally:

This procedure will check to the right of the offset by increasing it by 1. If element is same with what stored in \$a3, continue increasing \$v0, else jump to check to the left. If it has reached to the limit of that row or encounter a different piece at checking position, it will jump to check left.

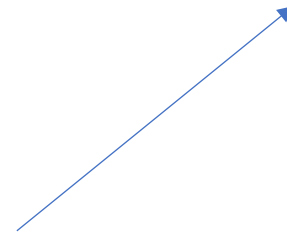
Checking left is quite the same as checking to the right, except for when I reaches minimum index of that row or encounter different element, it will jump to a function place in check win called “aftercheckLR” (LR is left-right).

“aftercheckLR” will check if \$v0 is equal to 4 or not. If yes it jump to end end, if no jumps to next checking procedure.

◆ Check win diagonally upwards:

This is more complex than the 2 previous checking. Diagonal checking upwards will first check in the upright direction first. Checking in the upright direction will decrease offset by -6 to get to the first index on the right of the row above. Reaching limit in checking upright is not just if offset is below 0, if you consider it checking upright will also stop if it has already reached the last column on the right. The offset of every element on is column will always be divisible by 7 if added 1 in, as shown in the table below.

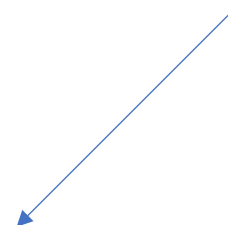
|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| 35 | 36 | 37 | 38 | 39 | 40 | 41 |



If there is a different element or it has reached the limit on the right, program will jump to checking to the down left direction, else continue increasing \$v0 and jump to next up right position.

Checking down left will have bound opposite to up right – the first column. Besides from checking whether offset is greater than number of elements in the matrix, it also has to check whether reached to first column or not, this column will have offset divisible by 7, as shown below.

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| 35 | 36 | 37 | 38 | 39 | 40 | 41 |



Continue jumping to next down left position if offset still in boundaries and element is the same with \$a3. If any condition fails, jump to “aftercheckURDL” (URDL means up right - down left)

If \$v0 is 4 jump to end game, else jump to final checking.

◆ Check win diagonally downwards:

This final checking is similar to the previous one (diagonally upwards). Therefore the principle works the same except for the bound for up left direction is the same for down left boundaries and it also has to be greater than 0. Boundaries for checking down right will be less than 42 and the final column (similar to checking up right).

• End game:

If a winner's move is found at any of the checkings above, “endgame” procedure will be called, print out current player stored in \$s1, and ask whether players want to restart.

• Toggle player:

In case there hasn't been a winner, a procedure will be called at the end of “checkwin” to toggle between player 1 and player 2, it also toggles the piece ‘x’ and ‘o’ to the other one. After toggling, new current player and piece is updated, game jump back to insert function for current player.

• Tie game:

If total number of moves in \$s3 is greater than or equal to 42 (means 42 moves has been made – no more empty space are left on the board), “checkWin” will jump to this procedure. This procedure will print out message that there has been a tie, and jump to “restart” function – which questions players to restart.

▪ Restart:

Restart procedure will ask players to enter number 1 to replay, if not entered, game will jump to outro. In case players want to restart, game will jump to empty matrix procedure to empty the matrix and start a new game.

• Outro:

If players doesn't enter 1 to replay the game, an intro will be printed out by file handling and reading, printing using buffer\_read.

[illegible][illegible]

Computer Architecture - Semester 1 (2022 - 2023)



After entering 'y', game will print out the matrix before the move has been played. This happens so that when players choosing to undo, they can see clearly what happens before that move.

Computer Architecture - Semester 1 (2022 - 2023) 1.





```
*	*	*	*	*	*
*	*	*	*	*	*
*	*	*	*	*	*
Total undos left: 3					
Player 2 undo ? Press 'y' to repick or any other key to continue					
4					
Player 1's turn, choose a column(1-7): 5					
*	*	*	*	*	*
*	*	*	*	*	*
*	*	*	*	X	*
*	*	*	*	O	X
*	*	*	*	O	X
*	*	*	*	O	X
*	*	*	*	O	X
Total undos left: 3
Player 1 undo ? Press 'y' to repick or any other key to continue
5
Player 1 wins!(<^_>)
Press 1 to restart
0

#####
```

END.