CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #4

Spring 2019

Assignment Due: Friday, May 10, 2019 by 11:59 pm

Updates to the Document:

- 4/26/2019: Part 13: more details about the pseudocode have been provided.
- 4/24/2019: Part 9, Example 6 was corrected.

Learning Outcomes

After completion of this programming project you should be able to:

- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly register conventions.
- Implement algorithms that process structs and 2D arrays of values.
- Read files from disk and build data structures based on their contents.

Getting Started

Visit Piazza and download the files `proj4.asm` and `boards.zip`. Fill in the following information at the top of `proj4.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `proj4.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `proj4.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory

will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `proj4.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- **You must use the Stony Brook version of MARS posted on Piazza.** Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement homework assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Do not submit a file with the function/label `main` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- All test cases must execute in 500,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in Mars, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There really is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best a form of laziness, so don't do it. We might comment out any such code we find.

How to Test Your Functions

Testing mains are not being provided for this assignment because by this point in the semester you should be able to write your own. However, you will be provided with text files that correspond the the `.txt` files referenced in the examples below. Additionally, the testing mains should be pretty simple this time around. The general structure should look like this:

```
.data
filename: .asciiz "board1.txt"
board: .space 1000
row: .word 5
col: .word 8
```

```
player: .byte 'X'

.text
la $a0, board
la $a1, filename
jal load_board

la $a0, board
lw $a1, row
lw $a2, col
lbu $a3, player
jal check_horizontal_capture

# code here to print any return value(s) and the updated state
#    of the game board, if appropriate
```

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Pente

For this project you will be implementing functions that could be used to complete an implementation of the board game called Pente. Take a few minutes to review the rules of the game before proceeding.

In our functions, the two players will be denoted by the characters `'X'` and `'O'` (capital letter Oh). A blank spot ("slot") on the board will be denoted with a period character, `'.'`.

Preliminaries

In this assignment, you will continue to work with structs. In this case, a struct will represent the state of a game of Pente:

```
struct Board {
   int num_rows; // 4 bytes
   int num_cols; // 4 bytes
   char[] slots; // num_rows*num_cols bytes
}
```

We can see that the total memory consumed by the game board is `8 + num_rows*num_cols`.

Recall that 2D arrays can be stored in two ways: row-major order and column-major order. Our game board will be stored in row-major order. Because 2D arrays are implemented as an array of 1D arrays in memory, this means

that the right-most slot in a row is followed in memory by the left-most slot of the next row. The 2D array will also order the rows such that the top of the board comes first in the array. Thus, memory addresses will increase as we progress from the left-most slot to the right-most slot of a row, and as we progress from the top-most row to the bottom-most row of the game board.

The board will have `n` rows and `m` columns, which are inputs to the game.

Throughout this document we will refer to slots of the board in (row, col) format. Position `(0,0)` of the board is the top-left corner. Position `(n-1, m-1)` is the lower-right corner of a board.

For example, the 42 slots in a board with 6 rows and 7 columns would be indexed as in the figure below.

Top of Board

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)

Bottom of Board

It is entirely up to you if you want to create a driver file that will let you play the actual game. You can even choose which variant of Pente you want to implement. Note that any drivers or testers you create file will not be graded.

Functions to Implement

Part I: Load a Game Board from Disk

```
int load_board(Board board, string filename)
```

The `load_board` function attempts to read the contents of the file named `filename`, formatted as described below, and use the contents to initialize the given (uninitialized) `Board` struct:

```
# of rows (1 or 2 digits)
# of columns (1 or 2 digits)
contents of the board (X, O or . characters normally)
```

As an example, below is the contents of `diag_capture3.txt`:

```
9
11
..OX.......
X..OO..X.O.
..X.XO..X..
..O..XXXO..
..XXX.O....
```

```
..X..O.....
.O.O...O...
.X.O..X....
..O........
```

The function updates the fields `num_rows` and `num_cols` as given in the file. It also writes the contents of the game board as provided in the file into `board.slots`.

The function takes the following arguments, in this order:

- `board`: a pointer to an uninitialized `Board` struct that the function will initialize
- `filename`: the name of the file containing the state of a game of Pente

Returns in `$v0`:

- an integer that encodes the number of X's, O's and invalid characters read from the file. This encoding is explained below.
- `-1` if the file of the given name does not exist

Additional requirements:

- The function must not write any changes to main memory except as specified.

As the function reads the contents of the board provided in the file, it should expect to see only `'X'`, `'O'` and `'.'` characters. Each such valid character is written directly into the `slots` grid. Whenever the function encounters a character not equal to one of these, it writes a `'.'` into the corresponding slot instead of the invalid character. The example given below will help to clarify this requirement of the function.

Assuming that the input file exists, the function returns a value which encodes the number of X's, O's and invalid characters it read. For this assignment we will assume that each count is at most 255. These three counts are encoded each as an unsigned byte in the 32-bit return value. Specifically:

```
bits: 31-24    23-16      15-8       7-0
        0      # of X's   # of O's   # of invalid characters
```

For example, suppose the function read 9 X's, 13 O's and 3 invalid characters. In binary, the return value would be:

```
00000000 00001001 00001101 00000011
```

Example:

Suppose an input file contained the following:

```
7
10
.XX.X.XX..
..........
.....X....
```

```
..XXOXX...
...X.X....
..XX....X.
..ZQ.....X
```

`board.num_rows` would be assigned `7`, `board.num_cols` would be assigned `10`, and `board.slots` would be initialized as follows:

```
.XX.X.XX..
..........
.....X....
..XXOXX...
...X.X....
..XX....X.
.........X
```

Note how the invalid characters in the file have been replaced with periods in memory.

The function must assume that every line of the file ends only with a `'\n'` character, *not* the two-character combination `"\r\n"` employed in Microsoft Windows. If you create your own game board files for testing purposes, use MARS to edit the files. If you are developing on a Windows computer, do not use a regular text editor like Notepad. Such an editor will insert both endline characters. In contrast, MARS will insert only a `'\n'` at the end of each line, *so only use MARS to create custom game boards*.

Part II: Get the Character in a Particular Slot in a Game Board

```
char get_slot(Board board, int row, int col)
```

The `get_slot` function returns the character located in a particular slot in a game board.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct from which we will retrieve a character
- `row`: the row of the slot from which we want to retrieve the character
- `col`: the column of the slot from which we want to retrieve the character

Returns in `$v0`:

- the character found in `board.slots[row][col]` if `row` and `col` are valid indices, or
- `-1` if either `row` or `col` (or both) are invalid

Additional requirements:

- The function must not write any changes to main memory.

Examples:

Game board struct used in the below examples (rows = 7, columns = 10):

generic_board1.txt

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Arguments	Return Value
board, -2, 8	-1
board, 13, 5	-1
board, 3, 5	88
board, 3, 0	46
board, 0, 4	88
board, 2, 9	46
board, 6, 2	88

Part III: Set the Character in a Particular Slot in the Game Board

`char set_slot(Board board, int row, int col, char character)`

The `set_slot` function changes the character located in a particular slot in a game board.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct at which we will change a character
- `row`: the row of the slot at which we want to change the character
- `col`: the column of the slot at which we want to change the character
- `character`: the character to be written into the `slots` array

Returns in `$v0`:

- the character written into `board.slots[row][col]` if `row` and `col` are valid indices, or
- `-1` if either `row` or `col` (or both) are invalid

Additional requirements:

- The function must not write any changes to main memory except as specified.

Examples:

Game board struct used in the below examples (rows = 7, columns = 10):

generic_board1.txt

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

The updated state of the board is not shown for valid arguments, but it should be obvious what the new board status is for each such case.

Arguments	Return Value
`board, -2, 8, 'A'`	`-1`
`board, 13, 5, 'X'`	`-1`
`board, 3, 5, 'Q'`	`81`
`board, 3, 0, 'X'`	`88`
`board, 0, 4, 'E'`	`69`
`board, 2, 9, 'Z'`	`90`
`board, 6, 2, 'O'`	`79`

Part IV: Place a Piece into a Particular Slot in a Game Board

```
char place_piece(Board board, int row, int col, char player)
```

The `place_piece` function attempts to change the character located into a particular slot in a game board.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct in which we want to change a character
- `row`: the row of the slot at which we want to change the character
- `col`: the column of the slot at which we want to change the character
- `player`: the character we want to write into the given slot

Returns in `$v0`:

- the character written into `board.slots[row][col]` if `row` and `col` are valid indices and `player` is a valid character, or
- `-1` if either `row` or `col` (or both) are invalid, `player` is neither `'X'` nor `'O'`, or the slot at the given row or column already contain a player's piece

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `set_slot`.

Examples:

Game board struct used in the below examples (rows = 7, columns = 10):

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

The updated state of the board is not shown for valid arguments, but it should be obvious what the new board status is for each such case.

Arguments	Return Value
`board, -2, 8, 'X'`	`-1`
`board, 13, 5, 'X'`	`-1`
`board, 2, 3, 'O'`	`79`
`board, 4, 0, 'X'`	`88`
`board, 0, 5, 'O'`	`79`
`board, 2, 9, 'X'`	`88`
`board, 6, 7, 'O'`	`79`
`board, 3, 5, 'O'`	`-1`
`board, 1, 9, 'q'`	`-1`
`board, 1, 9, 'X'`	`-1`

Part V: Get the Status of the Game

`(int, int) game_status(Board board)`

The `game_status` function simply counts the number of X's and O's in the board, returning each count separately in `$v0` and `$v1`, respectively. The idea behind this function is that it could be used in the driver to provide statistics about the game and also to test when the game board is completely filled.

The function takes one argument:

- `board`: a pointer to a `Board` struct

Returns in `$v0`:

- the number of X's in the board

Returns in `$v1`:

- the number of O's in the board

Additional requirements:

- The function must not write any changes to main memory.

Example #1:

Game board struct (rows = 10, columns = 12):

`empty10x12.txt`

```
............
............
............
............
............
............
............
............
............
............
```

Return values: `0, 0`

Example #2:

Game board struct (rows = 7, columns = 8):

`game_status1.txt`

```
XXX...X.
..X...OO
.O.X..X.
......OO
X...XOX.
....X...
..XOO.X.
```

Return values: `13, 8`

Example #3:

Game board struct (rows = 7, columns = 8):

`game_status2.txt`

```
XXX...X.
........
...X..X.
........
X...XXX.
....X...
```

```
..X...X.
```

Return values: `13, 0`

Example #4:

Game board struct (rows = 7, columns = 8):

`game_status3.txt`

```
........
......OO
.O......
......OO
.....O..
........
...OO...
```

Return values: `0, 8`

Part VI: Check for Captures in a Row

`int check_horizontal_capture(Board board, int row, int col, char player)`

The `check_horizontal_capture` function checks whether placing a piece (`player`) at the given row and column would cause a capture horizontally and, if so, perform the capture by replacing the captured pieces of the `player`'s opponent with the character `'.'`. The general idea of this function is that, prior to this function call, a piece has already been placed at the given row and column, and this function checks if the placement of that piece causes captures. Note that placing a single piece can result in more than two captures. Examples below will help to clarify the capturing procedure further.

For example, suppose along a row we find the following configuration:

```
...XOOX...
      *
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` marked with a star. The row will be updated as follows:

```
...X..X...
```

It is possible that a player can capture more than two pieces along a single row. For example, suppose along a row we find the following configuration:

```
...XOOXOOX...
      *
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` marked with a star. The row will be updated as follows:

```
...X..X..X...
```

A player cannot capture his own pieces. For example, suppose along a row we find the following configuration:

```
...XOOX......
      *
```

and the function is called with `'O'` for `player` and with values for `row` and `col` that correspond with the `'O'` marked with a star. The row will be remain in its original state:

```
...XOOX......
```

No pieces are captured by this move. The same no-self-capture rules also applies along columns and diagonals.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct in which we want to check for a capture
- `row`: the row of the slot in which we want to check for a capture
- `col`: the column of the slot in which we want to check for a capture
- `player`: the character which we will check at the given row and column to see if its placement caused a capture

Returns in `$v0`:

- the number of pieces captured or
- `-1` for any of the following error cases:
  - `row` or `col` (or both) are invalid
  - `player` is neither `'X'` nor `'O'` (capital letter Oh)
  - the slot at the given row and column is not equal to `player`

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `get_slot` and `set_slot`.

Example #1:

```
row = 4
col = 5
player = 'Z'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
```

```
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #2:

```
row = 2
col = -5
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #3:

```
row = 22
col = 5
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #4:

```
row = 3
col = 4
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
```

```
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #5:

```
row = 4
col = 5
player = 'X'
```

Game board struct (rows = 9, columns = 9)

`horiz_capture1.txt`

```
..O......
X...O..X.
..X.XO...
..O..X..O
..XOOX...
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Updated board:

```
..O......
X...O..X.
..X.XO...
..O..X..O
..X..X...
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Return value: `2`

Example #6:

```
row = 4
col = 2
player = 'X'
```

Game board struct (rows = 9, columns = 9)

`horiz_capture1.txt`

```
..O......
X...O..X.
..X.XO...
..O..X..O
..XOOX...
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Updated board:

```
..O......
X...O..X.
..X.XO...
..O..X..O
..X..X...
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Return value: `2`

Example #7:

```
row = 3
col = 4
player = 'O'
```

Game board struct (rows = 7, columns = 10)

`horiz_capture2.txt`

```
..OXXO..X.
....X.O...
.OXO..XXX.
.OXXOXXO..
.........X
...XOOX.X.
....OO..X.
```

Updated board:

```
..OXXO..X.
```

```
....X.O...
.OXO..XXX.
.O..O..O..
.........X
...XOOX.X.
....OO..X.
```

Return value: `4`

Example #8:

```
row = 5
col = 4
player = 'O'
```

Game board struct (rows = 7, columns = 10)

`horiz_capture2.txt`

```
..OXXO..X.
....X.O...
.OXO..XXX.
.OXXOXXO..
.........X
...XOOX.X.
....OO..X.
```

Updated board:

```
..OXXO..X.
....X.O...
.OXO..XXX.
.OXXOXXO..
.........X
...XOOX.X.
....OO..X.
```

Return value: `0`

Example #9:

```
row = 0
col = 8
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`horiz_capture2.txt`

```
..OXXO..X.
....X.O...
.OXO..XXX.
.OXXOXXO..
.........X
...XOOX.X.
....OO..X.
```

Updated board:

```
..OXXO..X.
....X.O...
.OXO..XXX.
.OXXOXXO..
.........X
...XOOX.X.
....OO..X.
```

Return value: `0`

Part VII: Check for Captures in a Column

```
int check_vertical_capture(Board board, int row, int col, char player)
```

The `check_vertical_capture` function checks whether placing a piece (`player`) at the given row and column would cause a capture vertically and, if so, perform the capture by replacing the captured pieces of the `player`'s opponent with the character `'.'`. The general idea of this function is that, prior to this function call, a piece has already been placed at the given row and column, and this function checks if the placement of that piece causes captures. Note that placing a single piece can result in more than two captures. Examples below will help to clarify the capturing procedure further. As with captures in a row, a player cannot capture his own pieces along a column.

For example, suppose down a column we find the following configuration:

```
.
X
O
O
X*
.
.
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` marked with a star. The column will be updated as follows:

```
.
X
.
```

```
.
X
.
.
```

It is possible that a player can capture more than two pieces down a single column. For example, suppose along a column we find the following configuration:

```
.
X
O
O
X*
O
O
X
.
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` marked with a star. The column will be updated as follows:

```
.
X
.
.
X
.
.
X
.
```

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct in which we want to check for a capture
- `row`: the row of the slot in which we want to check for a capture
- `col`: the column of the slot in which we want to check for a capture
- `player`: the character which we will check at the given row and column to see if its placement caused a capture

Returns in `$v0`:

- the number of pieces captured or
- `-1` for any of the following error cases:
  - `row` or `col` (or both) are invalid
  - `player` is neither `'X'` nor `'O'` (capital letter Oh)
  - the slot at the given row and column is not equal to `player`

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `get_slot` and `set_slot`.

Example #1:

```
row = 4
col = 5
player = 'Q'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #2:

```
row = -2
col = 5
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
```

```
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #3:

```
row = 4
col = 10
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #4:

```
row = 6
```

```
col = 3
player = 'x'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #5:

```
row = 2
col = 5
player = 'O'
```

Game board struct (rows = 10, columns = 10)

`vert_capture1.txt`

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Updated board:

```
.....O....
.X.....XO.
...X.O....
..O....X..
.XX....O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Return value: `2`

Example #6:

```
row = 5
col = 5
player = 'O'
```

Game board struct (rows = 10, columns = 10)

`vert_capture1.txt`

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Updated board:

```
.....O....
.X.....XO.
...X.O....
..O....X..
.XX....O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Return value: `2`

Example #7:

```
row = 4
col = 5
player = 'O'
```

Game board struct (rows = 8, columns = 9)

vert_capture2.txt

```
.........
...XOO...
..X..X.O.
.....X...
.X..OO.O.
.....X...
.X...XOO.
..O.OO...
```

Updated board:

```
.........
...XOO...
..X....O.
.........
.X..OO.O.
.........
.X....OO.
..O.OO...
```

Return value: 4

Example #8:

```
row = 5
col = 2
player = 'O'
```

Game board struct (rows = 10, columns = 10)

vert_capture1.txt

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
```

```
..X..O....
...O.XOOXX
......XX..
```

Updated board:

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Return value: `0`

Example #9:

```
row = 7
col = 5
player = 'O'
```

Game board struct (rows = 10, columns = 10)

`vert_capture1.txt`

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
..X..O....
...O.XOOXX
......XX..
```

Updated board:

```
.....O....
.X.....XO.
...X.O....
..O..X.X..
.XX..X.O..
..O..O....
..OOO..XO.
```

```
..X..O....
...O.XOOXX
......XX..
```

Return value: `0`

Part VIII: Check for Captures in Diagonals

`int check_diagonal_capture(Board board, int row, int col, char player)`

The `check_diagonal_capture` function checks whether placing a piece (`player`) at the given row and column would cause a capture diagonally and, if so, perform the capture by replacing the captured pieces of the `player`'s opponent with the character `'.'`. The general idea of this function is that, prior to this function call, a piece has already been placed at the given row and column, and this function checks if the placement of that piece causes captures. Note that placing a single piece can result in more than two captures. Examples below will help to clarify the capturing procedure further. As with captures in a row or a column, a player cannot capture his own pieces along a diagonal.

For example, suppose in a subset of the grid we find the following configuration:

```
........
..X..O..   *
...O....
.X..O...
.....X..
..O.X...
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` whose row marked with a star. The grid will be updated as follows:

```
........
..X..O..   *
........
.X......
.....X..
..O.X...
```

It is possible that a player can capture multiple pieces along diagonals. For example, suppose along a column we find the following configuration:

```
...X...X..
..O...O...
.....O..X.
.O..X.....   *
.....O...X
..X...O..O
.......X..
...O......
```

and the function is called with `'X'` for `player` and with values for `row` and `col` that correspond with the `'X'` whose row is marked with a star. The grid will be updated as follows:

```
...X...X..
..O.......
........X.
.O..X.....   *
.........X
..X......O
.......X..
...O......
```

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct in which we want to check for a capture
- `row`: the row of the slot in which we want to check for a capture
- `col`: the column of the slot in which we want to check for a capture
- `player`: the character which we will check at the given row and column to see if its placement caused a capture

Returns in `$v0`:

- the number of pieces captured or
- `-1` for any of the following error cases:
  - `row` or `col` (or both) are invalid
  - `player` is neither `'X'` nor `'O'` (capital letter Oh)
  - the slot at the given row and column is not equal to `player`

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `get_slot` and `set_slot`.

Example #1:

```
row = 4
col = 5
player = 'Q'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
```

```
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #2:

```
row = -2
col = 5
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #3:

```
row = 4
col = 10
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #4:

```
row = 6
col = 2
player = 'O'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Updated board:

```
.XX.X.XX..
```

```
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return value: `-1`

Example #5:

```
row = 4
col = 6
player = 'O'
```

Game board struct (rows = 9, columns = 9)

`diag_capture1.txt`

```
..OX.....
X..OO..X.
..X.XO...
..O..XX.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Updated board:

```
..OX.....
X..OO..X.
..X..O...
..O...X.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Return value: `2`

Example #6:

```
row = 5
col = 6
player = 'X'
```

Game board struct (rows = 10, columns = 11)

`diag_capture2.txt`

```
...OOXXX...
.....OOO...
..X..O..OX.
...X....O..
..OOX..O.O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
..XXX..OO..
```

Updated board:

```
...OOXXX...
.....OOO...
..X..O..OX.
...X.......
..OOX....O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
..XXX..OO..
```

Return value: `2`

Example #7:

```
row = 2
col = 9
player = 'X'
```

Game board struct (rows = 10, columns = 11)

`diag_capture2.txt`

```
...OOXXX...
.....OOO...
..X..O..OX.
...X....O..
..OOX..O.O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
```

```
..XXX..OO..
```

Updated board:

```
...OOXXX...
.....OOO...
..X..O..OX.
...X.......
..OOX....O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
..XXX..OO..
```

Return value: `2`

Example #8:

```
row = 1
col = 3
player = 'O'
```

Game board struct (rows = 9, columns = 9)

`diag_capture1.txt`

```
..OX.....
X..OO..X.
..X.XO...
..O..XX.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Updated board:

```
..OX.....
X..OO..X.
..X..O...
..O...X.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Return value: `2`

Example #9:

```
row = 4
col = 6
player = 'O'
```

Game board struct (rows = 9, columns = 11)

`diag_capture3.txt`

```
..OX.......
X..OO..X.O.
..X.XO..X..
..O..XXXO..
..XXX.O....
..X..O.....
.O.O...O...
.X.O..X....
..O........
```

Updated board:

```
..OX.......
X..OO..X.O.
..X..O.....
..O...X.O..
..XXX.O....
..X..O.....
.O.O...O...
.X.O..X....
..O........
```

Return value: `4`

Example #10:

```
row = 5
col = 4
player = 'X'
```

Game board struct (rows = 9, columns = 11)

`diag_capture4.txt`

```
..OX.......
X...O..X.O.
..X.XO.X...
```

```
..O...OOO..
.....O.....
..X.X......
.O.O...O...
.XOX..X....
.XO........
```

Updated board:

```
..OX.......
X...O..X.O.
..X.XO.X...
..O....OO..
..........
..X.X......
.O.....O...
.X.X..X....
.XO........
```

Return value: `4`

Example #11:

```
row = 2
col = 5
player = 'O'
```

Game board struct (rows = 9, columns = 11)

`diag_capture3.txt`

```
..OX.......
X..OO..X.O.
..X.XO..X..
..O..XXXO..
..XXX.O....
..X..O.....
.O.O...O...
.X.O..X....
..O........
```

Updated board:

```
..OX.......
X..OO..X.O.
..X.XO..X..
..O..XXXO..
..XXX.O....
..X..O.....
```

```
.O.O...O...
.X.O..X....
..O........
```

Return value: `0`

Example #12:

```
row = 3
col = 2
player = 'O'
```

Game board struct (rows = 9, columns = 9)

`diag_capture1.txt`

```
..OX.....
X..OO..X.
..X.XO...
..O..XX.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Updated board:

```
..OX.....
X..OO..X.
..X.XO...
..O..XX.O
..XXX.O..
..X..O...
.O.O...O.
.X.O..X..
..O......
```

Return value: `0`

Example #13:

```
row = 6
col = 8
player = 'X'
```

Game board struct (rows = 10, columns = 11)

`diag_capture2.txt`

```
...OOXXX...
.....OOO...
..X..O..OX.
...X....O..
..OOX..O.O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
..XXX..OO..
```

Updated board:

```
...OOXXX...
.....OOO...
..X..O..OX.
...X....O..
..OOX..O.O.
......XOO..
..X...OXX..
...OOXOO...
X....OO....
..XXX..OO..
```

Return value: `0`

Part IX: Check for a Win Across the Rows

`(int, int) check_horizontal_winner(Board board, char player)`

The `check_horizontal_winner` function checks an entire game board for five or more sequentially positioned instances of the character `player` along any row. The function may assume that at most one such horizontal run appears in the game board.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct we want to check for a win along a row
- `player`: the character we want to check for

Returns in `$v0`:

- the row number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a row

Returns in `$v1`:

- the column number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a row

Additional requirements:

- The function must not write any changes to main memory.

Example #1:

```
player = 'Z'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #2:

```
player = 'X'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #3:

```
player = 'O'
```

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #4:

`player = 'X'`

Game board struct (rows = 7, columns = 10)

`horiz_win1.txt`

```
.XX.X.XX..
..OO.O....
.....X....
..XXOXXXXX
...X.X....
........X.
X.OO.O....
```

Return values: `3, 5`

Example #5:

`player = 'X'`

Game board struct (rows = 7, columns = 10)

`horiz_win2.txt`

```
.XX.X.XX..
..OO.O....
XXXXXO.X..
..XXOXXX.X
...X.X....
..OOOO..X.
..........
```

Return values: `2, 0`

Example #6:

`player = 'O'`

Game board struct (rows = 10, columns = 11)

`horiz_win3.txt`

```
...OOXXX...
.....OOO...
..X..O..O..
...X....O..
..OOX..O.O.
......OOO..
..X...OXX..
...OOOOO...
X....OO....
..XXX..OO..
```

Return values: `7, 3`

Example #7:

`player = 'X'`

Game board struct (rows = 10, columns = 11)

`horiz_win4.txt`

```
...OOXXX...
.....OOO...
..X..O..O..
...X....O..
..OOX..O.O.
......OOO..
..X...OXX..
.XXXXXXXXX.
X....OO....
..XXX..OO..
```

Return values: `7, 1`

Part X: Check for a Win Down the Columns

`(int, int) check_vertical_winner(Board board, char player)`

The `check_vertical_winner` function checks an entire game board for five or more sequentially positioned instances of the character `player` down any column. The function may assume that at most one such vertical run appears in the game board.

The function takes the ollowing arguments, in this order:

- `board`: a pointer to a `Board` struct we want to check for a win along a column
- `player`: the character we want to check for

Returns in `$v0`:

- the row number of the topmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a column

Returns in `$v1`:

- the column number of the topmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a column

Additional requirements:

- The function must not write any changes to main memory.

Example #1:

```
player = 'Z'
```

Game board struct (rows = 7, columns = 10)

```
generic_board1.txt
```

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #2:

```
player = 'X'
```

Game board struct (rows = 7, columns = 10)

```
generic_board1.txt
```

```
.XX.X.XX..
```

```
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #3:

`player = 'O'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #4:

`player = 'X'`

Game board struct (rows = 8, columns = 7)

`vert_win1.txt`

```
....O..
.O...O.
.......
.X.X.X.
.O.X.O.
...X..O
.O.X..O
...X...
```

Return values: `3, 3`

Example #5:

`player = 'O'`

Game board struct (rows = 8, columns = 7)

`vert_win2.txt`

```
..O.O..
.OO..O.
..O....
.XOX.X.
.OOX.O.
...O..O
.O.X..O
...X...
```

Return values: `0, 2`

Example #6:

`player = 'X'`

Game board struct (rows = 8, columns = 7)

`vert_win3.txt`

```
....O..
.O..XO.
....X..
.X.XXX.
.O.XXO.
...OX.O
.O.O..O
...X...
```

Return values: `1, 4`

Example #7:

`player = 'O'`

Game board struct (rows = 11, columns = 8)

`vert_win4.txt`

```
....O...
.O..XO..
..O.X...
.XOXOX..
.OOXXO..
..OO..O.
.OOO..O.
```

```
..OX....
..O.....
..O.....
........
```

Return values: `2, 2`

Part XI: Check for a Win Along the SW→NE Diagonals

`(int, int) check_sw_ne_diagonal_winner(Board board, char player)`

The `check_sw_ne_diagonal_winner` function checks an entire game board for five or more sequentially positioned instances of the character `player` along any diagonal that runs southwest/northeast (i.e., the leftmost piece of the run has a *higher* row number than the rightmost piece).

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct we want to check for a win along a southwest/northeast diagonal
- `player`: the character we want to check for

Returns in `$v0`:

- the row number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a diagonal

Returns in `$v1`:

- the column number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`
- `-1` if player `player` has not placed 5 or more characters in a run along a diagonal

Additional requirements:

- The function must not write any changes to main memory.

Example #1:

`player = 'Z'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
```

```
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #2:

`player = 'X'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #3:

`player = 'O'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #4:

`player = 'X'`

Game board struct (rows = 10, columns = 10)

`sw_ne_diag_win1.txt`

```
....OOXO..
.X.....O..
..O..X....
.XO.X..XOX
...X..X...
..X..O.X..
.X........
X...O..O..
..O..XX.O.
.....X..XX
```

Return values: `7, 0`

Example #5:

```
player = 'X'
```

Game board struct (rows = 9, columns = 8)

`sw_ne_diag_win2.txt`

```
..XO.OX.
.XOX..OX
..X..O.X
XOO...X.
.X...X..
.O..X...
.O.X.X..
...XO.O.
XO.O....
```

Return values: `6, 3`

Example #6:

```
player = 'O'
```

Game board struct (rows = 9, columns = 8)

`sw_ne_diag_win3.txt`

```
.X.XO.O.
O...O...
.X.O....
..O.XOX.
.O......
O..O....
.....XO.
```

```
.X..X...
O..O....
```

Return values: `5, 0`

Example #7:

```
player = 'X'
```

Game board struct (rows = 8, columns = 7)

`sw_ne_diag_win4.txt`

```
..XX..X
..O..X.
.O.OX..
.X...X.
.O..X..
...X.X.
..X.O..
.X..X..
```

Return values: `7, 1`

Example #8:

```
player = 'O'
```

Game board struct (rows = 10, columns = 10)

`sw_ne_diag_win5.txt`

```
.O.X...O.X
O...XO.O..
..X...O.XO
...O.O.O..
.OX.O...O.
O..O.....O
..O..XO...
.O.......X
...O..X...
....X.O.X.
```

Return values: `7, 1`

Example #9:

```
player = 'X'
```

Game board struct (rows = 12, columns = 9)

`sw_ne_diag_win6.txt`

```
...X...XO
..O.OXO.X
O......X.
.X.X..X..
.O..XX...
.X..XXX..
.X.X..O..
..X...O..
.X.OX.XO.
X.....O..
...O...XX
....O.O..
```

Return values: `9, 0`

Part XII: Check for a Win Along the NW→SE Diagonals

`int, int check_nw_se_diagonal_winner(Board board, char player)`

The `check_nw_se_diagonal_winner` function checks an entire game board for five or more sequentially positioned instances of the character `player` along any diagonal that runs nortwest/southeast (i.e., the leftmost piece of the run has a *lower* row number than the rightmost piece). Recall that the topmost row in the game board has the *lowest* row number.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct we want to check for a win along a southwest/northeast diagonal
- `player`: the character we want to check for

Returns in `$v0`:

- the row number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`

Returns in `$v1`:

- the column number of the leftmost character in the run
- `-1` for error if `player` is neither `'X'` nor `'O'`

Additional requirements:

- The function must not write any changes to main memory.

Example #1:

`player = 'Z'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #2:

`player = 'X'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #3:

`player = 'O'`

Game board struct (rows = 7, columns = 10)

`generic_board1.txt`

```
.XX.X.XX..
.........X
.....X....
..XXOXX...
...X.X....
..XX....X.
..XOOO...X
```

Return values: `-1, -1`

Example #4:

`player = 'X'`

Game board struct (rows = 9, columns = 9)

`nw_se_diag_win1.txt`

```
...XOXOX.
..O..O...
X..O..O..
X.....O..
.X..O....
.OX......
X..X.....
O...X.X..
O.X......
```

Return values: `3, 0`

Example #5:

`player = 'O'`

Game board struct (rows = 9, columns = 8)

`nw_se_diag_win2.txt`

```
..X.OXO.
..O..O..
.O.O..XO
..X.O..O
..X..O..
......O.
.XOXO..O
..O.....
...X.X..
```

Return values: `1, 2`

Example #6:

`player = 'O'`

Game board struct (rows = 10, columns = 10)

`nw_se_diag_win3.txt`

```
.O...O....
..O..XX.O.
.X.O..XO..
.X..O...O.
...O.O..X.
.X..O.O.X.
..X..O.O..
.X.X.O..X.
...O...O..
.OO..X....
```

Return values: `0, 1`

Example #7:

`player = 'X'`

Game board struct (rows = 8, columns = 9)

`nw_se_diag_win4.txt`

```
O..X..O..
...O..XX.
XO...X...
..X...OO.
...X..OX.
.O..X..O.
..X..X...
.OO.O.X..
```

Return values: `3, 2`

Example #8:

`player = 'X'`

Game board struct (rows = 12, columns = 14)

`nw_se_diag_win5.txt`

```
.....X......X.
.X..XO.O.XOXO.
..X...X.O...O.
...X...XX.O...
....X....O..O.
.X...X....O.O.
.O..X.X...O...
..OX...X..O...
...X.O....O...
```

```
.O..X...O.....
..O..X...O..O.
...O..X.......
```

Return values: `1, 1`

Example #9:

```
player = 'O'
```

Game board struct (rows = 11, columns = 10)

`nw_se_diag_win6.txt`

```
.X.OX..XO.
O.X..X..O.
.OX.X..O..
X.O.XOO..X
..XO....O.
.X..O.X.X.
.....O....
.....OOX..
...O.X.O..
....X...O.
..X.O..X..
```

Return values: `1, 0`

Part XIII: Simulate a Game of Pente

```
int simulate_game(Board board, string filename, string turns,
                  int num_turns_to_play)
```

The `simulate_game` function simulates part of a game of Pente.

The function takes the following arguments, in this order:

- `board`: a pointer to a `Board` struct we will use to simulate the game
- `filename`: the name of the file containing the starting state of a game of Pente
- `turns`: a null-terminated string that represents a series of turns in the game
- `num_turns_to_play`: the maximum number of turns to simulate

Returns in `$v0`:

- the number of valid turns simulated

Returns in `$v1`:

- `'X'` or `'O'` to indicate the winner of the game (if any), or
- `-1` if the game ended in a tie (board filled with no run of 5 or more identical player characters)

Each turn in `turns` is encoded as a 5-character substring:

```
[1-chararacter player][2-digit row][2-digit column]
```

For example, `X0213` indicates that player `X` wants to place a piece at row 2, column 13.

Each two-digit row and two-digit column is guaranteed to be a non-negative number. However, there could be other errors in the five-character substring:

- the character for the player is neither `'X'` nor `'O'`
- the row number is $\geq$ the number of rows in the board
- the column number is $\geq$ the number of columns in the board

When an invalid turn is detected in the `turns` string, that turn is simply ignored, and the function moves on to the next turn.

`num_turns_to_play` could be less than, equal to, or greater than the number of valid turns encoded in the `turns` string. The function should process as many turns as it can from the `turns` string. See the pseudocode below.

Implement the pseudocode given below. Do not deviate from this pseudocode, or make your own custom version of this function. Please ask on Piazza if you have questions about the pseudocode.

The following variables with similar names have the following meanings:

`turns_length` The number of turns encoded in the `turns` array. This number can be easily computed using the formula `len(turns) / 5`.

`valid_num_turns` The number of turns actually played so far.

`num_turns_to_play` The maximum number of valid turns that should be played. It is possible that there are fewer valid turns in the `turns` array than we actually can play.

`turn_number` The number of turns we have attempted to play so far. Think of this as an index into the `turns` array, where the `turns` array as consists of 5-character strings laid out contiguously in memory.

```
attempt to load the board game by calling load_board()
if the board cannot be loaded then
    return 0, -1
game_over = False
while not game_over and valid_num_turns < num_turns_to_play and
      turn_number < turns_length do
    extract the next player, row and column characters from turns[] string
    if the player char is invalid, or row or column number is invalid, then
        skip this turn

    r = place_piece(board, row, col, player)
    if a piece was successfully placed then
```

```
        check_horizontal_capture(board, row, col, player)
        check_vertical_capture(board, row, col, player)
        check_diagonal_capture(board, row, col, player)

        r1, r2 = check_horizontal_winner(board, player)
        if the player won the game then
            record the winner
            game_over = True

        r1, r2 = check_vertical_winner(board, player)
        if the player won the game then
            record the winner
            game_over = True

        r1, r2 = check_sw_ne_diagonal_winner(board, player)
        if the player won the game then
            record the winner
            game_over = True

        r1, r2 = check_nw_se_diagonal_winner(board, player)
        if the player won the game then
            record the winner
            game_over = True

    r1, r2 = game_status(board)
    if game is not over and r1 + r2 equals number of slots in the board then
        game_over = True  # tie game

    return number of valid turns played, winner of game
```

Additional requirements:

- The function must not write any changes to main memory itself. Any changes to memory will be written by called functions.
- The function must call the following functions:
  - `load_board`
  - `place_piece`
  - `check_horizontal_capture`
  - `check_vertical_capture`
  - `check_diagonal_capture`
  - `check_horizontal_winner`
  - `check_vertical_winner`
  - `check_sw_ne_diagonal_winner`

- check_nw_se_diagonal_winner
- game_status

Below is a simple example of `simulate_game`'s behavior for some sample input:

```
filename = "empty10x12.txt"
turns = "X0207O0712X0211O0611X0506O0812X0108O0101X1107O0400X1010O0305X0304
O0302X0203O0505X1006O0306X0708O0412"
num_turns_to_play = 20
```

Game board struct initially (rows = 10, columns = 12)

```
............
............
............
............
............
............
............
............
............
............
```

Updated board:

```
............
.O......X...
...X...X...X
..O.XOO.....
O...........
.....OX.....
..........O
........X...
............
............
```

Return value: `14, -1`

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or

hard-copy) is academic dishonesty and helps others plagiarize my work.

4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `proj4.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on "Assignments" in the left-hand menu and click the link for this assignment.
3. Click the "Browse My Computer" button and locate the `proj4.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.