Assignment 1: Fortran (25%)

Choose **one** of the following topics.

1. SEARCHING THE MAZE

A maze is an area surrounded by walls; in between you have a path from starting position to ending position. Maze searching is a classic data-structures type problem, which can be solved using elements such as stacks, and backtracking, or recursion. A maze can be represented as a 2D array of characters, each representing a cell. For example:

```
* * * * * * * * * * * * *
* o . . . . . . . . . . *
* . * . * * . * . * . * *
* * . . . * * . . * . * *
* . * . * . * * . * . * *
* * . * . * . * . * * * *
* . * * * * * . . . . * *
* . * * . . * . * . . * *
* * . * * * . * . . * * *
* . . . . . * * * * * * *
* . * . * . . . . . e * *
* * * * * * * * * * * * *
```

with the following character representations:

```
walls          * (asterisk)
open space     . (period)
start          o (lowercase 'O')
finish         e (lowercase 'E')
```

The maze is surrounds by walls, and as it is traversed, the periods are replaced with o's. Note that the maze above is shown with spaces in between for clarity. So the input format would be of the form:

```
12 12
************
*o.........*
*.*.**.*.*.*
**...**..*.*
*.*.*.**.*.*
**.*.*.*.***
*.*******....*
```

```
*.**..*.*..*
**.***.*..**
*.....******
*.*.*.....e*
************
```

Each cell in the maze is either open, or blocked by an internal wall. From any open cell, you may move left, right, up, or down to an adjacent empty cell. To solve a maze, you must find a path of open cells from a given start cell, 'o', to a specified end cell, 'e'.

The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and change the direction. The direction for traversing be North, East, West and South. We have to continue "move and backtrack" until we reach the ending point. Now from the starting position, the algorithm moves to the next free path. If a dead-end is encountered, the algorithm back-tracks and makes the cells in the path ones (wall). This continues until the end point is reached. There are two possible non-recursive approaches to solving the maze:

a depth-first approach using a stack
a breadth-first approach using a queue

For example, a solution using a stack might use an algorithm of the form:

```
create an empty stack named S;
push start onto S;
while S is not empty do
    current_cell ← pop from S;
    if current_cell is the finish-point then
            output "Maze traversed ok";
            S ← empty stack;
        else if current_cell is not a wall and
                current_cell is not marked as visited then
            mark current_cell as visited;
            push the cell to the east onto S;
            push the cell to the west onto S;
            push the cell to the north onto S;
            push the cell to the south onto S;
        end if
     end while
```

TASK

Design a Fortran program for traversing the maze using either a stack or queue, and using a Fortran *module* to contain the data structure. Test your program on a series of mazes, including easy and non-trivial. Note that the maze should be read from a file and the user should be prompted for the maze filename. The solution can be output to standard output.

TOPIC 1 INFORMATION

REFLECTION DOCUMENT

Discuss your program in 3 page (or more if you like) *reflection document* (single-spaced), explaining your algorithm and decisions you made in the process of designing your program. Consider the document a synopsis of your experience with Fortran - this should be at least 2 pages in length. The write-up should explain your approach to the problem, including any modifications made to the specifications. In addition you should include answers to the following questions:

- What were the greatest problems faced while designing the algorithm in Fortran?
- Was it easy to create a data structure in Fortran?
- What do you think about the concept of the Fortran *module*?
- What particular features made Fortran a good language?

In addition, one page should include answers to the following questions:

- Would it have been easier to write the program in a language such as C?
- Is there a better way of writing the program?
- Given your knowledge of programming, was Fortran easy to learn?
- What structures made Fortran usable? (In comparison to C for instance)

DELIVERABLES

Either submission should consist of the following items:

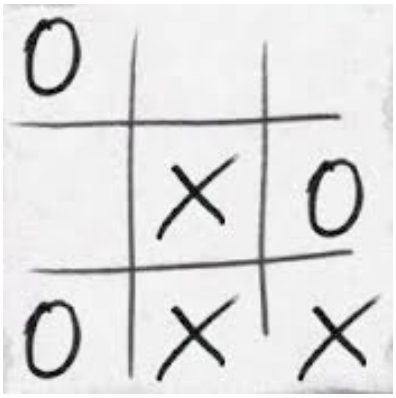
- The reflection document.
- The code (well documented and styled appropriately of course).

SKILLS

- Fortran programming, file I/O, recursion/data structures.

2. TIC TAC TOE

The game of *Tic-Tac-Toe*, known in Britain as *Nought's and Crosses*, is a game played with pencil and paper. Two players, X and O, take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical or diagonal row wins the game.



In this version of the game, a player (X) plays against a computer (O) using a very simple AI engine. The game starts with the user choosing a box to play their X in.

1	2	3
4	5	6
7	8	9

For example, if the user chooses 6, then an "X" is placed in square 6. The computer then plays its turn. After each turn, the board is checked to see if a winner has emerged. If there is a winner, the game ends, otherwise it continues.

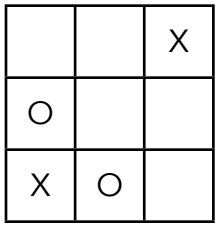
A piece of legacy Fortran code, is provided which has the following components:

- **TICTACTOE** - A main program which plays the game of Tic-Tac-Toe between a human player and a computer.
- **CHKOVR** - A subroutine to check to see if the game is over.
- **SAME** - A function to check to see if three elements in a row, column or diagonal are the same.
- **BOARDSETUP** - A subroutine to setup the TicTacToe board.
- **CHKPLAY** - A subroutine that checks to make sure the human player cannot make a play in a square that is already occupied.
- **COMPMOVE** - A subroutine that plays for the computers move.

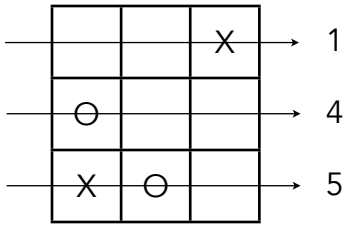
In the main program, there is a call to a function:

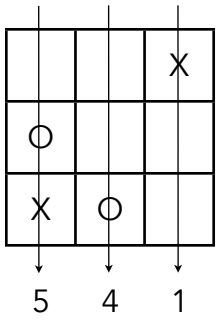
```
CALL COMPMOVE(TICTAC)
```

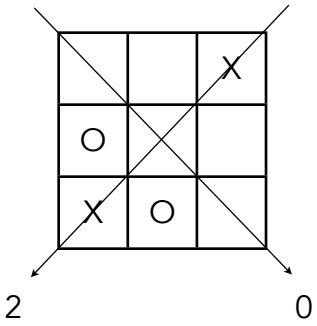
This piece of AI performs the computers move. It is a simple algorithm which involves checking the eight paths through the board and deriving a sum for each path based on a value for each of X (1), O (4), or blank (0). For example, the following board:



Would create the following sums:







The algorithm first searches for a sum of 8, which signifies that there are two O's in a path, and then proceeds to add the third O in order to win the game. If this scenario does not work, the algorithm then looks for a sum of 2, signifying a path with two X's and a potential win for the opponent. If one exists, a O is used to block that path. In neither of those situations exist, the algorithm chooses a random spot to place an O.

Note: Anther algorithm is the *minimax* algorithm [1,2]. You may choose to replace the simple AI engine with a more complex algorithm if you choose.
[1] http://neverstopbuilding.com/minimax
[2] http://beej.us/blog/data/minimax/

TASK

The requirement for this topic is to modernize the code, by replacing legacy code structures, with contemporary ones (from Fortran 95 or later). This means removing unstructured code such as **goto** statements, and arithmetic **if**s, and cleaning up loops. Configuring the code may require modularizing more of the code, e.g. the piece of code that prints out the board.

TESTING

Here is a sample run with a simple AI engine.

```
 PLAY TIC-TAC-TOE. ENTER 1-9 TO PLAY

         1 | 2 | 3
        ---+---+---
         4 | 5 | 6
        ---+---+---
         7 | 8 | 9

 Your move?
4
 After your move...
    |   |
 ---+---+---
  X |   |
 ---+---+---
    |   |
 After my move...
  O |   |
 ---+---+---
  X |   |
 ---+---+---
    |   |
 Your move?
9
 After your move...
  O |   |
 ---+---+---
  X |   |
 ---+---+---
    |   | X
 After my move...
  O | O |
 ---+---+---
  X |   |
 ---+---+---
    |   | X
```

```
 Your move?
3
 After your move...
  O | O | X
 ---+---+---
  X |   |
 ---+---+---
    |   | X
After my move...
  O | O | X
 ---+---+---
  X |   | O
 ---+---+---
    |   | X
Your move?
7
 After your move...
  O | O | X
 ---+---+---
  X |   | O
 ---+---+---
  X |   | X
 After my move...
  O | O | X
 ---+---+---
  X |   | O
 ---+---+---
  X | O | X
 Your move?
5
 After your move...
  O | O | X
 ---+---+---
  X | X | O
 ---+---+---
  X | O | X
 The game is over!
 The winner is: X
```

TOPIC 2 INFORMATION

REFLECTION DOCUMENT

Discuss your re-designed program in 3 page (or more if you like) *reflection document* (single-spaced), explaining decisions you made in the re-engineering process. Consider the document a synopsis of your experience with your re-engineering process - this should be at least 2 pages in length. This should include a synopsis of the approach you took to re-engineer the program (e.g. it could be a numbered list showing each step in the process). Identify the legacy structures/features and how you modernized them.

In addition, one page should include answers to the following questions:

- Would it have been easier to translate the program into a language such as C?
- Is there a better way of modernizing the program?
- Given your knowledge of programming, was Fortran easy to learn?
- What structures made Fortran usable? (In comparison to C for instance)

DELIVERABLES

Either submission should consist of the following items:

- The reflection document.
- The code (well documented and styled appropriately of course).

SKILLS

- Fortran programming, re-engineering,