Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

Exercise: 10	Working: Individual
Title: Maze Navigation	
Issue date: 19th January 2004	Due date: 26th January 2004
System: linux	Language: Java

Aim

- To write a simple Java program with a single main program class so that you become familiar with the Java syntax.
- To write a program using recursion in Java and to introduce you to *backtracking*, in this case by finding all the paths through a given *maze*.

The Problem

- Write a Java program **Maze.java** which finds all the paths that traverse a maze from a unique **entrance** to any **exit**, visiting each location in the maze at most once.
- Your program should read in the maze from a file and store it in an appropriate data structure. The name of the file containing the maze should be given as a *command line argument* when you run the **Maze** program.

UNASSESSED/ Submit by Monday 26th January 2004

What To Do

Before writing any code

- Use the command **exercise 10** to copy the skeleton of the file **Maze.java** and the test files **amaze-1.dat** .. **amaze-8.dat**. Note this skeleton contains methods for reading in a maze from a file and for writing out a maze to the screen. You should add no further I/O methods to those provided.

- A maze can be represented textually in a file, or on the screen, by a two dimensional array of characters, where paths are represented by spaces and walls by `'#'` characters. A valid maze contains at least one start location and finish location, denoted by a space characters. All mazes have an entrance on the left wall and one or more exits on the right wall.
- A sample maze with one source location and one target location is shown below.

  ```
  ###################
  #       # #   #   #
  ##### # # # # ### #
  #     # # # #      
  ### ### # # #######
  #     # #     #   #
  # ##### ##### # ###
  # #           #   #
  # ##### ##### # # #
  # #   # #   # # # #
  # # ### # # # # # #
  #   #   # # # # # #
  ### ### # ### # ###
          #         #
  ###################
  ```

  All the rows are the same width so that the whole maze is always rectangular.
- If your program contains assertions you will need to compile your program with the following flag **javac -source 1.4 FILENAME(S).java** and run it with the following flag **java -ea FILENAME**.

Writing your Maze.java program

- As explained above, a "solution" means a path that starts at the entrance, finishes at any exit, and never crosses itself (*i.e.* never repeats a position already used earlier on the path).

  As an example the program should output the following when given the above maze as input.

```
# # # #   # # # # # # # # # # # # # #
#         . . . #   # . . . #       #
# # # # # . # . #   # . # . # # #   #
#     . . . # . #   # . # . . . . . .
# # # . # # # . #   # . # # # # # # #
# . . .     # . #     . . . #       #
# . # # # # # . # # # # # . #   # # #
# . #         . . . . . . . #       #
# . # # # # #   # # # # #   #   #   #
# . #       #   #       #   #   #   #
# . #   # # #   #   #   #   #   #   #
# . . . #       #   #   #   #   #   #
# # # . # # #   #   # # #   #   # # #
. . . .         #                   #
# # # # # # # # # # # # # # # # # # #
```

```
# # # #   # # # # # # # # # # # # # #
#               #   # . . . #       #
# # # # #   #   #   # . # . # # #   #
#           #   #   # . # . . . . . .
# # #   # # #   #   # . # # # # # # #
#           #   #     . . . #       #
#   # # # # #   # # # # # . #   # # #
#   #         . . . . . . . #       #
#   # # # # # . # # # # #   #   #   #
#   #       # . #       #   #   #   #
#   #   # # # . #   #   #   #   #   #
#       #     . #   #   #   #   #   #
# # #   # # # . #   # # #   #   # # #
. . . . . . . . #                   #
# # # # # # # # # # # # # # # # # # #
```

Design Suggestions

- To help with autotesting, your program should use **readMaze** and **printMaze** for I/O. In addition:
  - when exploring the maze, you should search for a valid next move from your current position in the order **north, south, east, west** .
- We recommend you write a recursive *method*, called for example **exploreMaze**, which finds all routes from the *current* position (wherever that may be!) to any exit.

  The base case for **exploreMaze** will be when the current position is on the right-hand wall. In that case you have found a solution, and so you should draw the user's attention to the solution by displaying the path with **printMaze**.

If, on the other hand, you are not at an exit right now, **exploreMaze** will need to continue looking by calling *itself* once for each of the adjacent positions (north, south, east, west) it is legal to move to, trying them in the order of compass directions specified above.

- You should deposit a **mazeMarker** at the current position at the *start* of **exploreMaze**, and only rub it out (by overwriting it with an **mazePath**) at the *end* of **exploreMaze**.
- Note that you can only move *to* a position which has valid coordinates for the width and height of the maze *and* which currently contains an **mazePath**. Any other character must be either the maze wall, which is of course totally out of bounds, or else a **mazeMarker**, which means you've been there before and must not use it again in this path.
- Once you have written **exploreMaze**, you can complete the method **main** so that it looks down the left-hand edge for the entrance and calls **exploreMaze** with the current position set to the entrance.
- The above design suggestions are by no means the only approach and you are free to try other structures for your code if you like. But remember to stick to the search *order* north, south, east, west.

Unassessed

- Extend the program to use classes such as a Maze class to store the maze and a Position class to store each position. Provide the appropriate methods for each class.
- When you have found a solution print not only the maze but also the path through the maze from source to target, e.g. North, North, East, North, West, West etc. This requires maintaining a *stack* of directions (north, south, east or west) as you visit the various locations. For example, if you go north you push a representation of 'north' onto the stack. When you backtrack (return from exploring the north direction) you pop the stack. At the target location the stack contains the list of directions for getting *back* to the source. The required path is the reverse of this! You need to decide on a good way to represent (and print) the directions. The stack can be an array and an index referencing the stack top. You need to think about where best to place the 'push' and 'pop' code. Remark: remember the L-systems exercise last term? There again we used a stack to 'traverse' a multidimensional data structure. It's a similar story here.
- Separate the top-level (main method) code from the `Maze` class and place it in a separate class, e.g. `MazeSolve`.
- Modify the code to allow *non-terminal* target locations. This means that having reached a target location (and printed the solution) it may be possible to proceed *from* that location to some other exit location.

Submission

- Submit **Maze.java** in the usual way by typing **submit 10** at your Linux prompt.

Assessment

```
UNASSESSED
```