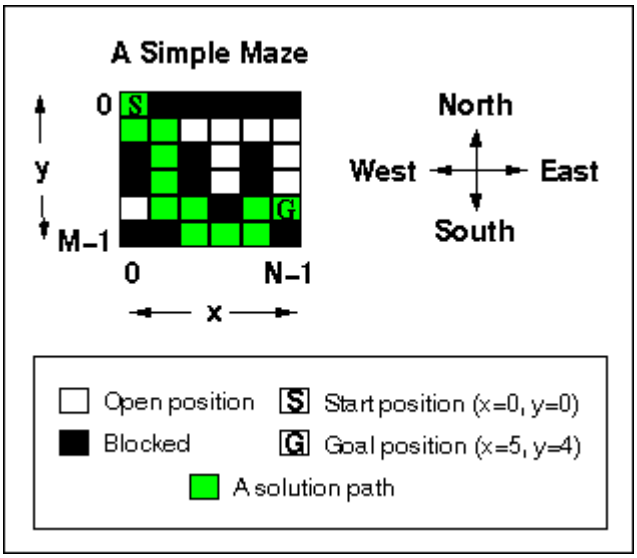
The Problem

A robot is asked to navigate a maze. It is placed at a certain position (the *starting* position) in the maze and is asked to try to reach another position (the *goal* position). Positions in the maze will either be open or blocked with an obstacle. Positions are identified by (x,y) coordinates.



At any given moment, the robot can only move 1 step in one of 4 directions. Valid moves are:

- Go North: (x,y) -> (x,y-1)
- Go East: (x,y) -> (x+1,y)
- Go South: (x,y) -> (x,y+1)
- Go West: (x,y) -> (x-1,y)

Note that positions are specified in zero-based coordinates (i.e., 0...size-1, where *size* is the size of the maze in the corresponding dimension).
The robot can only move to positions without obstacles and must stay within the maze.
The robot should search for a path from the starting position to the goal position (a *solution path*) until it finds one or until it exhausts all possibilities. In addition, it should mark the path it finds (if any) in the maze.

Representation

To make this problem more concrete, let's consider a maze represented by a matrix of characters. An example 7x7 input maze is:

S	O	O	O	O	X	O
X	X	O	X	O	O	X
O	X	O	O	O	X	X
X	X	X	O	O	X	O
X	X	X	X	O	X	X
G	O	O	O	O	O	X
X	X	X	X	X	X	X

‘O’ -- where the robot can move (Open Position)

‘X’ -- obstacles (blocked positions)

‘S’ -- Start Position

‘G” -- Goal Position

Aside: Remember that we are using *x* and *y* coordinates (that start at 0) for maze positions. A *y* coordinate therefore corresponds to a row in the matrix and an *x* coordinate corresponds to a column.

A path in the maze can be marked by the '*' symbol and ‘#’ if not part of the path

---

Algorithm

We'll solve the problem of finding and marking a solution path using *recursion*.
Remember that a recursive algorithm has at least 2 parts:

- *Base case(s)* that determine when to stop.
- *Recursive part(s)* that call the same algorithm (i.e., itself) to assist in solving the problem.

Recursive parts

Because our algorithm must be *recursive*, we need to view the problem in terms of similar *subproblems*. In this case, that means we need to "find a path" in terms of "finding paths."
Let's start by assuming there is already some algorithm that finds a path from some point in a maze to the goal, call it FIND-PATH(x, y).
Also suppose that we got from the start to position x=1, y=2 in the maze (by some method):
To find a path from position x=1, y=2 to the goal, we can just ask FIND-PATH to try to find a path from the North, East, South, and West of x=1, y=2:

- FIND-PATH(x=1, y=1) *North*
- FIND-PATH(x=2, y=2) *East*
- FIND-PATH(x=1, y=3) *South*
- FIND-PATH(x=0, y=2) *West*

Generalizing this, we can call FIND-PATH recursively to move from any location in the maze to adjacent locations. In this way, we move through the maze.

Base cases

It's not enough to know how to use FIND-PATH recursively to advance through the maze. We also need to determine when FIND-PATH must stop.
One such *base case* is to stop when it *reaches the goal*.
The other base cases have to do with moving to invalid positions. For example, we have mentioned how to search North of the current position, but disregarded whether that North position is legal. In order words, we must ask:

- *Is the position in the maze* (...or did we just go outside its bounds)?
- *Is the position open* (...or is it blocked with an obstacle)?

Now, to our base cases and recursive parts, we must add some steps to mark positions we are trying, and to unmark positions that we tried, but from which we failed to reach the goal:
FIND-PATH(x, y)

1. if (x,y outside maze) return false
2. if (x,y is goal) return true
3. if (x,y not open) return false
4. mark x,y as part of solution path
5. if (FIND-PATH(North of x,y) == true) return true
6. if (FIND-PATH(East of x,y) == true) return true
7. if (FIND-PATH(South of x,y) == true) return true
8. if (FIND-PATH(West of x,y) == true) return true
9. unmark x,y as part of solution path
10. return false

All these steps together complete a basic algorithm that finds and marks a path to the goal (if any exists) and tells us whether a path was found or not (i.e., returns true or false). This is just one such algorithm--other variations are possible.

Note: FIND-PATH will be called at least once for each position in the maze that is tried as part of a path.
Also, after going to another position (e.g., North):
if (FIND-PATH(North of x,y)[1] == true) return true[2]
if a path to the goal was found, it is important that the algorithm stops. I.e., if going North of x,y finds a path (i.e., returns **true**[1]), then from the current position (i.e., current call of FIND-PATH) there is no need to check East, South or West. Instead, FIND-PATH just need **return true**[2] to the previous call.
Path marking will be done with the '+' symbol and unmarking with the 'x' symbol.

Using Algorithm

To use FIND-PATH to find and mark a path from the start to the goal with our given representation of mazes, we just need to:

1. Locate the start position (call it *startx*, *starty*).

2. Call FIND-PATH(startx, starty).
3. Re-mark ***** the start position with 'S'.

*****In the algorithm, the start position (marked 'S') needs to be considered an *open* position and must be *marked* as part of the path for FIND-PATH to work correctly. That is why we re-mark it at the end.

Create a class called MazeSolver:

```
public class MazeSolver
{
   private char[][] maze;
   private int startx,
               starty;

  Public MazeSolver(String fileName) throws IOException
  {
    // create an object to read information from “fileName”
    // read the maze dimension (row  col) from the file
    // Allocate the space for maze
    // initialize array maze with contents of the file
    // find startx and starty

    printMaze();  // a method that prints the maze

    // solveMaze() is a recursive method to solve the maze
    if(solveMaze(maze,startx,starty)) {
      System.out.println(“Solution found”);
      printMaze();
   }
   else {
      System.out.println(“No solution found”);
   }
```