# MazeMaster

When our program only has to make one decision, we can loop through each of the options, evaluate each one, and pick the best. If we have two options, we can nest one loop inside the other so that we try each possible combination. However, if we have a lot of decisions to make (possibly not even knowing how many decisions we’ll need to make), this approach doesn’t work.

For example, in a maze we have multiple options for which way to go. Each of those options leads to new choices, which in turn leads to new choices as the path continues to branch. The path from start to finish requires making an unknown number of subsequent decisions on which way to turn. Instead of trying to make all of these decisions at once, we make just one decision at a time, in any of the four directions. With that new location as the center, then we make recursive calls to try each of the four directions. The recursive structure builds up and eventually unwinds, keeping track of all the previous trial-and-error decisions.

Let’s solve some mazes. We should encapsulate the maze and its methods in its own class. What does the Maze class need to do?

* Which data structure(s) do you need?

WWWWWWWW   
W....W.W   
WW.WW..W   
W....W.W   
W.W.WW.**E**   
**S**.W.WW.W   
WW.....W   
WWWWWWWW

* How will you represent the path, each step, the walls,

the start, and the exit? Would constants be helpful?

* How will you get the maze into the program? From a

file or hard-code each one? Let’s read a 2-D maze from a file.

* Let’s make a precondition that every maze has one ‘S’ and one ‘E’.

You need to start at ‘S’. How do you find the ‘S’?

* What constructor do you need?
* Do you need to display the maze?
* How will you solve the maze? We have given you a solve method which will call one of these methods below. These methods are all **recursive**.
  1. From ‘S’ to ‘E’, change every DOT to PATH. (This is a lot like AreaFill)
  2. From ‘S’ to ‘E’, change every DOT to PATH, and count the number of recursions.

WWWWWWWW  
W....W.W  
WW.WW..W  
W\*\*\*.W.W  
W\*W\*WW\*E  
S\*W\*WW\*W  
WW.\*\*\*\*W  
WWWWWWWW

3 & 4. If there is a path (how will it know?), solve the maze

and mark the path.

1. Solve the maze, mark the path, count the steps.
2. Solve the maze, mark the path, list the (r,c) steps.

What does the driver class need to do?

**Extension** Write a constructor that generates random mazes.

Help with #3 Marking the Path in MazeMaster

public boolean markTheCorrectPath(int r, int c)

WWWWWWWW  
W....W.W  
WW.WW..W  
W\*\*\*.W.W  
W\*W\*WW\*E  
S\*W\*WW\*W  
WW.\*\*\*\*W  
WWWWWWWW

1. The method returns a boolean.
2. It’s recursive.
3. It checks in all 4 directions. Each level generates 4 more recursive levels.
4. As it goes, it changes the "." to a temporary character, such as "o".
5. Most of the base cases return false, because the path in that direction ends.
6. One base case, reached at the very end of the recursive calls, will return true.
7. As the recursions return, they all need to be OR’d.
8. If all 4 OR’d recursions return false (and most are false), change the "o" back to a "."
9. If the OR’s recursions are true (there is only one true path), change the "o" back to a "\*".
10. The recursions all unwind. What’s left is a path of "\*".

Our code generates left, down, right, up, which each (eventually) returning a boolean. The tree looks something like this:

WWEWW  
W...W   
S.W.W   
WWWWW

what happens to the matrix when one is true?

true

WWEWW  
WoooW   
SoW.W   
WWWWW

WWEWW  
W\*\*.W   
S\*W.W   
WWWWW

what happens to the matrix when all are false?

WWEWW  
W\*\*.W   
SoW.W   
WWWWW

WWEWW  
Wo\*.W   
SoW.W   
WWWWW

false

WWEWW  
Woo.W   
SoW.W   
WWWWW

false

false

false

false

WWEWW  
WoooW   
SoWoW   
WWWWW

false

false

false

WWEWW  
WoooW   
SoW.W   
WWWWW

false

false

false

WWEWW  
Woo.W   
SoW.W   
WWWWW

false

false

false

WWEWW  
Wo..W   
SoW.W   
WWWWW

false

false

WWEWW  
W...W   
SoW.W   
WWWWW

false

false