Part 1 Disjoint Set Class (10 points)
Create a class to do union/find operations in an efficient manner. Demonstrate the class works.

For ease in grading, leave this test in your TestMaze.cpp file.

Part 2 Solve the Maze (10 points)
MazeStarter.zip contains some starter code that may be helpful. You do NOT need to use it if you don’t like it. The supplied code reads the maze from a file. Read the input maze at maze1.txt and solve the maze using recursive backtracking.

The number of rows and columns must be odd. For simplicity, we assume the size of the maze is a constant. The starting position will always be at position (1,1) The final position will be at (MAZE_ROW-2, MAZE_COL-2)

Recursive backtracking: This will find a solution, but it won't necessarily find the shortest solution. Let the curr location be Start. If every direction (from curr) is either a wall or someplace you’ve already visited, return failure. If you're at the finish, return success. Otherwise, recursively try moving in the four directions from your curr location. When you move north, south, east or west, mark the previous spot with ‘^’, ‘v’, ‘>’, or ‘<’ so we can tell which way you went. Erase the symbol when you return failure (from curr), and a single solution will be marked out when you hit the final location. When backtracking, it's best to mark the space with a special visited value, so you don't visit it again from a different direction. In Computer Science terms, this is basically a depth first search. This method will always find a solution if one exists, but it won't necessarily be the shortest solution.

For example, consider this as the input to your initial maze.

```
+---------+---------+
<p>| | |
|         |         |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>

+---------+---------+
<p>| | |
|         |         |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>

+---------+---------+
<p>| | |
|         |         |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

Hint: In the debug phase, I found it useful to take a step (in the maze) and then print the maze so I could see the progress. Because there is so much printing if I print out each step, I created a pause() routine which asked the user to input a character to continue. This is very helpful in seeing what is happening. What you don’t want to do is run the whole thing, get the wrong answer, and have no clue why. This 'step/print'
technique is absolutely essential in getting enough information to solve your own problems. Some of you are way too dependent on the tutors. You MUST learn to do your own debugging.

We use '.' to mean we backtracked at location. After solving recursively we see the following: **Your result may differ as the exact cells visited depends on the order each direction is tried.**

**Solved Maze**

```
012345678901234567890
0 +-+-+-+-+-+-+-+-+-+-+  0
1 |>>>>v| | | | | | | |  1
2 +-+-+v+-+-+-+-+-+-+-+  2
3 | | |v| | | | | | |  3
4 +-+-+v+-+-+-+-+-+-+-+  4
5 | | |v.......| | | |  5
6 +++++v+++-+-+.-+-++-+  6
7 | | |v| | | | | | |  7
8 +++++v+++-+-+.-+-++-+  8
9 | | |v| | | | | | |  9
10 +++++v+++-+-+.-+-++-+ 10
11 | | |v| | |.....| | | 11
12 +++++v+++-+-+.-+-++-+ 12
13 | | |v| | | |.....| | 13
14 +++++v+++-+-+.-+-++-+ 14
15 | | |v| | |.....| |.| 15
16 +++++v+++-+-+.-+-++-+ 16
17 | |v<<| | | | | | |.| 17
18 +++++v+++-+-+.-+-++-+ 18
19 | |>>>..<<<<<<..<<<<<<| 19
20 +++++v+++-+-+.-+-++-+ 20
012345678901234567890
```

Create another maze (maze2.txt) of the same size and test your solution with the maze you create.

For ease in grading, leave the code to solve both maze1 and maze2 in your TestMaze.cpp file.

**Part 3: Create a maze randomly (10 points)**

Use your disjoint set class to randomly throw out walls (in an initial maze) similar to what is described in your text. **Print out the maze you create and solve it with your recursive backtracker.**

The following setup will make it a bit easier. All cells have their own equivalence class. A wall with a "|" will join cells to the left and right of it. A wall with a "-" will join cells above and below it. A "+" wall CANNOT BE REMOVED.

The idea behind creating a maze is that we want to randomly remove walls such that we get something interesting. Our definition of interesting is that if there is already a path between two cells of the maze, we do NOT want to allow another path to be created between them.

To create a maze, all initially empty cells are given an identification number to be used by your disjoint set class. In my numbering cell[1,1] has ID 0, and cell[3,5] has ID 12 – but any unique mapping will work. I found it useful to be able to convert between the two aliases for the same location (the position in terms of row/column and the ID in the disjoint set).

```cpp
intgetID(Pos p) { return (p.row/2)*MAZE_COLS/2 +p.col/2; }
```
Note, if needed, you can figure out the position (row and col) of the cell from the ID by using integer division and mod.

So the algorithm is:
while (start is not connected to final)
  { randomly pick a cell containing a wall and consider removing the wall
    remove the wall if the neighboring cells (in two directions) are not connected via a path
    If the neighbors are already connected, do not remove the wall.
  }