Recursive Problem Analysis

From previous classes, you should already have some exposure to problem solving in programming. However, I know that students come from many different backgrounds. For this reason, I want give a short elaboration/review on basic problem solving techniques in terms of recursion. My expectation is that these are basic problem analysis skills you are already starting to learn – you will need to grow them to solve more complex problems. Or, more importantly, problems which you may not have a very similar example to review.

1 Understand the problem completely.

Ideally, we should be able to run an algorithm on paper before we try to implement it. There is no magic that happens when you start randomly typing at keyboard as opposed to working it out on paper. In fact, the symbols and other programming language constructs can detract from understanding the problem.

At a high level, our problem is to take some blank area and replace it by a maze-like structure. This will have to exist in an array – meaning a grid of elements. Reading the description of the algorithm tells us something about recursively dividing a space into smaller pieces and adding walls. The pictures should make sense – they look like a maze. The method, perhaps not... Let's try to understand it.

We need to know something about the *structure* of the problem. From the problem statement, we know that it is a recursive algorithm – this is what we have to start. One topic that the textbook doesn't discuss much is how to analyze recursive problems. Not in terms of Big-Oh, but in terms of understanding. To address this, we'll turn to a process called Fantastic Four (Chen and Tsai; 2015) analysis. The purpose is to have a scaffold, if you would, that tells us specific steps to follow when trying to devise a recursive algorithm for the description of a problem. Let's take a look at F4 for the simple example of a factorial before applying it to the maze problem. The four steps are as follows:

1. Formulate the size-n problem. Size-n is the name we give to the "complete" problem. Like a loop, recursion is necessary only if you want to solve a problem that needs to repeat the same operations for a number of times. We assume the total number of iterations is n. In most cases, n is obvious. For example, if we want to compute factorial *n!*, the size n is already given. Formulating size-n problem, in some cases, is merely choosing a method name and using n as the parameter of the function. Thus, the size-n problem for factorial is *int factorial (int n)*. The return value of the size-n method is what the method is supposed to compute, or the value we are looking for. In this step, we do not need to design the solution for size-n problem.

2. Find the base case and the corresponding return value. Like a while-loop, every recursive function starts with checking a conditional (base case). If the base case is true, we return the corresponding value and exit the method. Otherwise, we enter the body of the recursive method. In some cases, identifying the base case and corresponding value is trivial. For example, the base case of *factorial(n)* is $n = 0$ and the corresponding value is *1*. This gives us *if(n==0) return 0;*

3. Formulate the size-m problem and find m. The size-m problem is simply a "smaller" problem; where how much smaller m is than n is determined by how much we can reduce the problem size in one iteration. If we can only reduce the problem size by one, m is *n-1*. For example: *factorial(n-1)*. Note that the problem type is still the same as it was for the size-n problem (compute a factorial), but the problem is now smaller/simpler. Do not try to define a solution, or the return value in this step! All we need to do here is ASSUME the call to solve the size-m problem will return a value and then use the value, e.g., *factorial(n-1)*, to get us to the stopping condition. We need to make sure we reduce n every iteration so we can be assured that the base case will be triggered at some point.

4. Construct the solution of the size-n problem from the size-m problem. In this step, we will use the assumed solution or the statement for size-m problem (e.g., *factorial(n-1)*) to construct the solution of the size-n problem. This step is application-specific. In the case of factorial problem, the solution of the size-n problem is *n*factorial (n-1);* Putting everything together, we get:

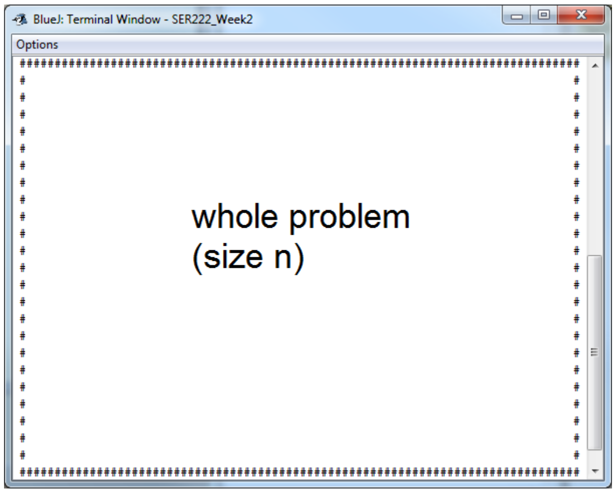
```
int factorial (int n) {
   if(n==0)
     return 1;

   return n*factorial(n-1);
}
```

Fantastic Four (F4) gives a general mechanism to understand recursive problems – however, we must move beyond its literal interpretations of n and m to apply it to less mathematical problems. Okay, so let's do a F4 analysis on our actual problem.

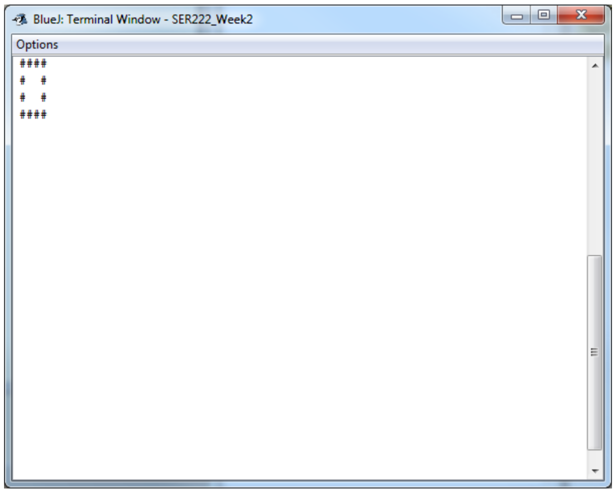
1. **Formulate the size-n problem.**

   We should think of the size-n problem as being the *whole problem* (for some size metric n). Even without reading the algorithm, we should be thinking that our overall problem is to generate a maze:



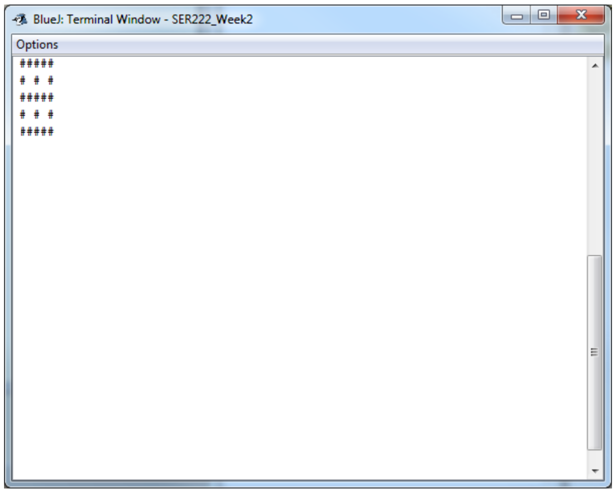
2. **Find the stopping condition and the corresponding return value.**

   Let's think just about mazes in general again. Surely a very small area cannot be a maze since there is no space to place walls. It seems likely that 1x1 will certainly be too small, but what about a 2x2 blank area?



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

This is simply too small to contain a maze! A 3x3 area means we would have space for walls:

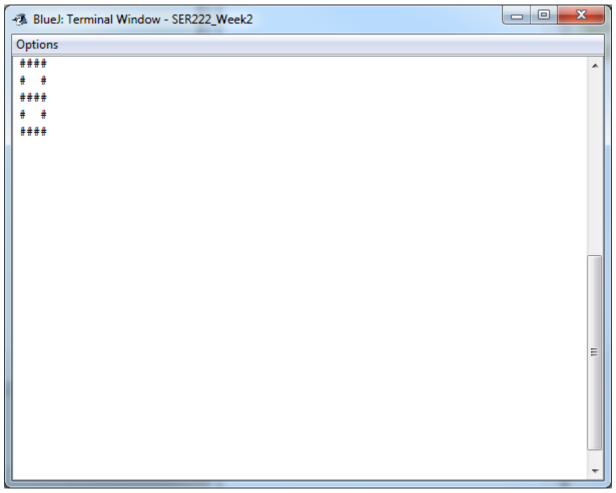


```
#####
# # #
#####
# # #
#####
```

Hence, *if we have a maze smaller than 3x3, we can stop our recursive algorithm.*

When you do the actual implementation, the function will be a void type so you don't need to worry about the return value. *The value that we are calculating is actually the layout of the maze.* The maze is already stored as an array parameter (remember pass-by-reference?) and doesn't need to be returned.

We are actually simplifying the stopping condition a bit – for now this is fine because we are only trying to understand the problem. The issue is that 2x3 mazes are possible too. However, we can't add both walls – only one:



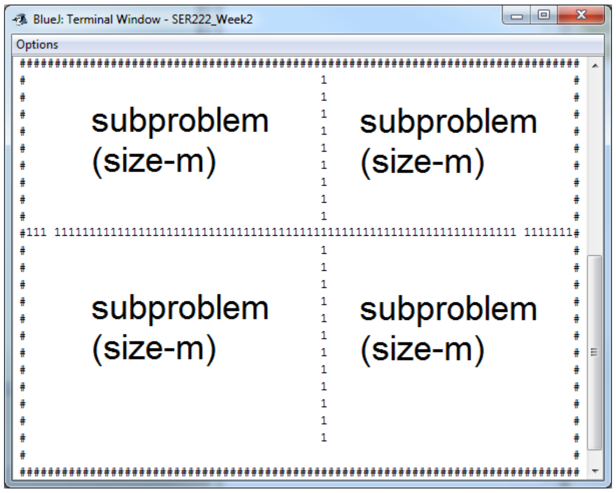
```
####
#  #
####
#  #
####
```

For now, we can neglect this. It is an implementation detail and doesn't relate to the concept. Same thing with doors – for sure we will be able to add them if we can add walls (why?).

3. **Formulate the size-m problem and find m. In many cases, m = n - 1;**

   We should think of the size-m problem as being the *sub-problem* or smaller problem. It's still the same kind of problem as the size-n one, but it's smaller and easier to solve. It must get us closer to the stopping condition.

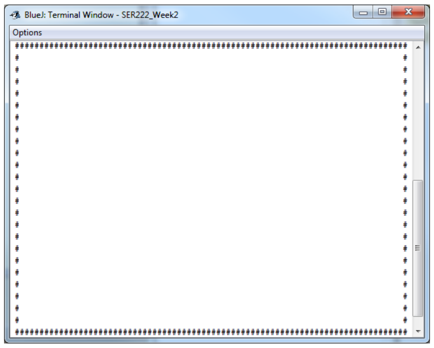
   Here, thinking about it leads us to see that *a maze can be thought of as multiple smaller mazes* that have been joined together in some way. This relates back to the idea of recursive division – divide the whole maze region into mini-maze quadrants:



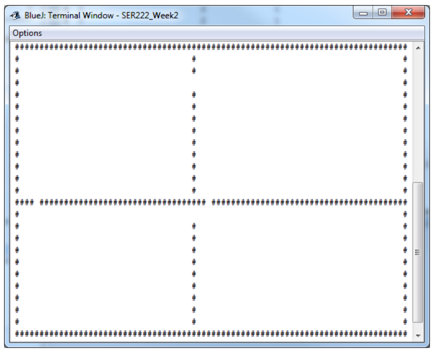
   Here, we end up with four sub-problems – not just one or two like in the examples. It doesn't matter. A sub-problem is a sub-problem - we can make as many recursive calls as needed to solve each of them. Also, remember that *we assume the sub-problem can be solved recursively.*

4. **Construct the solution of size-n problem from size-m problem.**

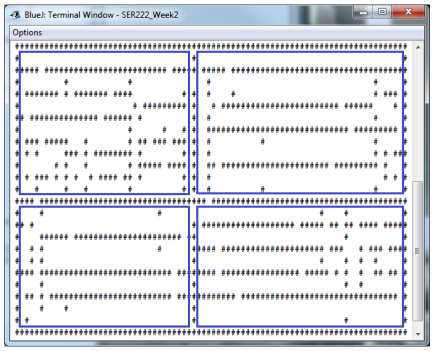
   This is the tricky bit. What we want to do is identify the part of the problem we will solve 'immediately' and the part we will be recursing on. Based on our previous analysis, we know we will be generating sub-mazes. Going back to the recursive division algorithm, we see that it asks us to divide the area. This is the progress we make towards solving the whole problem.



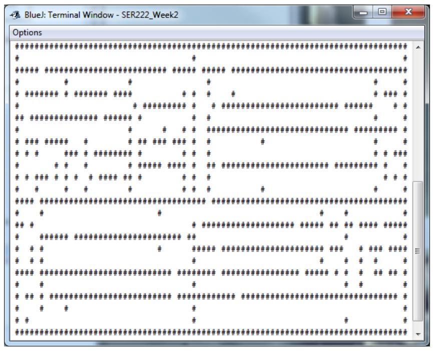
Becomes:



For the maze to be completable, we need to add holes in the walls. It should be the case that a player in any quadrant can reach any other quadrant. This allows us to combine smaller mazes and know that the result will (generally) be solvable. Below, the solutions to the first four size-m problems are highlighted in blue. The walls not in blue were added by the main call.



Thus, the four sub-mazes, in the context of the two walls, become a full size maze:



You should now have a much better idea of the structure of the problem and its solution. You should now be able to break down the basic steps and write a basic implementation outline:

- Create makeMazeRecursive(...) – comes from Fantastic Four (part 4)!
  - Add the two walls that divide the area. This is two for-loops preceded by the generation of two random numbers (wall positions).
  - Based on the position of the walls added, punch holes in them.
  - Call recursively on sub-mazes if needed (3x3 or larger).

2 Identify the rough topics that you need to know.

- Maze engine: Need to review the base source code and see how the static map is loaded and displayed.
- Recursion: Need to understand how recursion works. See slides (ctrl-f is your friend).
- 2D Arrays: Need to understand how to create and manipulate them (coordinate system). Reference CST200 textbook.
- Random numbers: Need to generate random integers for wall positions. Not in slides. Use Google! Look for something labeled as a reference or example, e.g. `www.javapractices.com/topic/TopicAction.do?Id=62`

3 Implementation. (Please don't skip ahead and read this.)

For more implementation hints, see the assignment. In addition to the hints, the assignment gives the method signature: makeMazeRecursive(char[][]level, int startX, int startY, int endX, int endY);

We should make an effort to understand this before proceeding with our implementation. Whatever hints or information shows up in a problem prompt should be considered to be at least one valid approach to solving the problem. You don't have to do it that way but it is an option.

As you read the function signature your thought should be: what do these parameters do? The first looks like a reference to the maze data (easily checked by browsing the source code). The others look like two points. Points in what? The maze. What is the coordinate system? It's one used by arrays – the positions have to be in terms of arrays since that is what is used to store levels.

- char[][] level: array for level data.
- int start_x, start_y: point for upper left corner of maze to add in level data.
- int end_x, end_y: point for lower right corner of maze to add in level data.

We're not quite done yet. We need a better understanding for the points... Are those parameters meant to be inclusive or exclusive? i.e., it could mean build a maze starting at startX (inclusive) or it could mean build a maze starting at startX+1 (exclusive). We need to have precisely defined bounds before the algorithm will work correctly. It's your choice – for now, I'll assume you've made them both inclusive.

Given this new understanding, what should the initial call to make_maze look like? See the base code for the answer.