Assignment #2
Due October 18th
60 points

You can write your programs in the environment of your preference (e.g. Visual Studio, or Unix). Turn in your source code files via Blackboard by submitting a single zip file `LASTNAME_assignment_2.zip`. This zip file should contain a folder containing the source code for each problem. The zip file should also contain a `LASTNAME_README.txt` file that describes how to run each program as well as what doesn't work.

1) System Crash – 10 pts

The following C++ code has a problem where the loop can exceed the size of the array if the user inputs too many numbers.

```
#include <iostream>
using namespace std;

int main()
{
	int nums[20] = { 0 };
	int a[10] = { 0 };

	cout << a << endl;
	cout << nums << endl;

	cout << "How many numbers? (max of 10)" << endl;
	cin >> nums[0];
	for (int i = 0; i < nums[0]; i++)
	{
		cout << "Enter number " << i << endl;
		cin >> a[i];
	}
	// Output the numbers entered
	for (int i = 0; i < 10; i++)
		cout << a[i] << endl;
	return 0;
}
```

If this program is run and we enter 255 for how many numbers, and 9 for every single number, then the program does something like this:

```
How many numbers? (max of 10)
255
Enter number 0
9
```

```
Enter number 1
9
Enter number 2
9
Enter number 3
9
Enter number 4
9
Enter number 5
9
Enter number 6
9
Enter number 7
9
Enter number 8
9
Enter number 9
9
Enter number 10
9
Enter number 11
9
Enter number 12
9
9
9
9
9
9
9
9
9
9
(Program may crash at this point, or possibly just exit)
```

Your results may vary, depending on what compiler, operating system, and CPU that you are using. If the program does not crash on your compiler, try it on transformer or the Windows machines in the lab and you should get the behavior shown above.

Why didn't the program loop 255 times? Your answer should be detailed and reference the way in which memory is organized and what is put into memory.

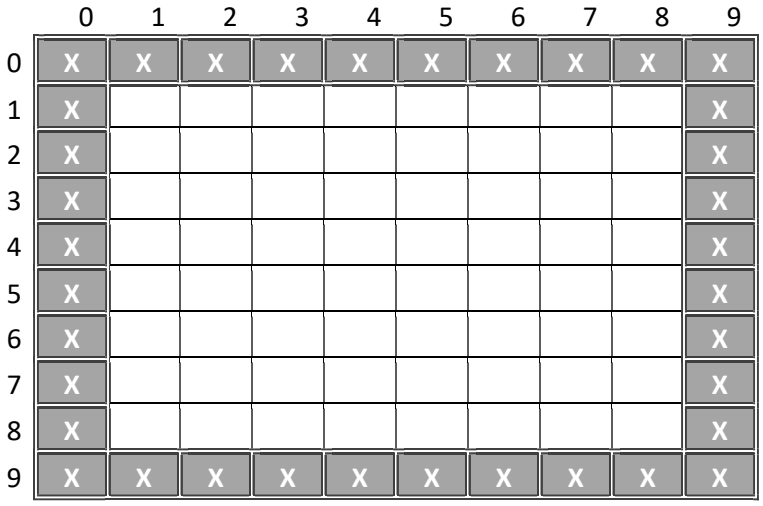
2) Maze Solving – 20 pts

Start with the computer maze-solving program covered in class. The source code is posted along with the homework.

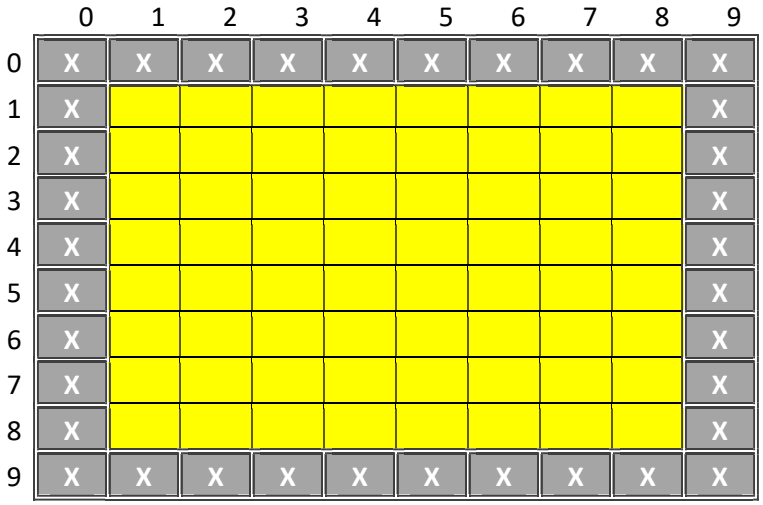
1. Change the dimensions of the maze to 20x20 with walls around the borders. Out of the empty space in the middle, randomly fill 25% of them with walls. Then pick a random start location (that is empty) and a random Exit location (that is empty). Since you are using random numbers, you should get a different maze with a different start and end. There is a small chance that it is impossible to get from the start to the exit, but don't worry about that possibility (when the program runs it should just not print any solution).
2. The algorithm we covered in class prints out the path from the start to the exit in reverse order. Modify the program so the path is output in forward order. One way to do this is to make an array (or arrays) that store the (x,y) coordinates of the current position as the recursive function exits. With all of the coordinates in an array, inside the main function you can now loop through the array in the proper order to output the moves from start to exit.
3. Your program should output a solution to the randomly generated maze, if one exists.

3) Recursive Maze Generation – 20 pts

Implement the following recursive algorithm to randomly generate a maze. In the example below I have made the maze 10x10 to save space but your maze should be 40x40. Start with a 2D array of char like we used previously and assign walls to the borders:

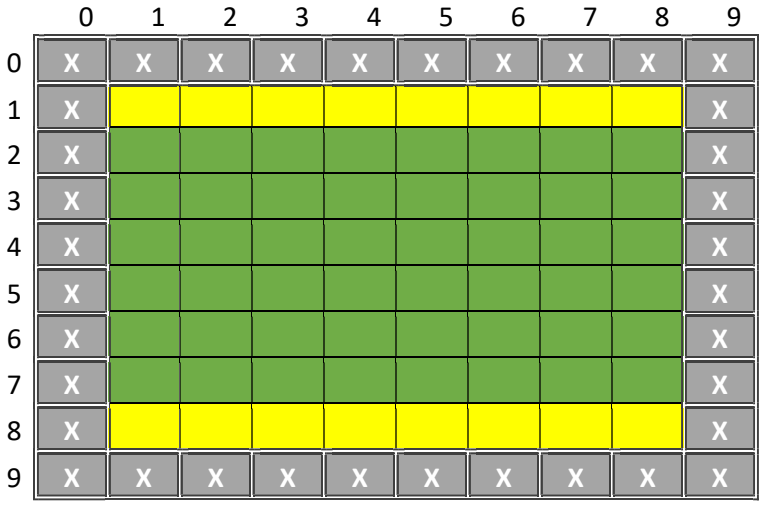


Call the recursive function with parameters that specify the blank area in the middle, in this case, highlighted by yellow below. You could do this a couple of ways, for example, pass in the coordinates of the upper left and lower right corner, or pass in the coordinate of the upper left corner plus the width and height of the area.



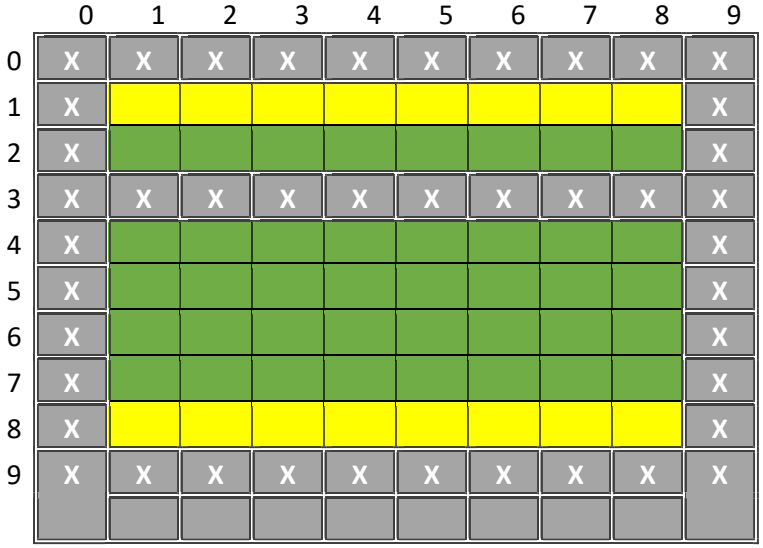
If the width of the yellow area is <=2 cells or the height of the yellow area is <=2 cells then the function should return, doing nothing. This is the base case that terminates recursion.

If the height of the yellow area is greater than or equal to the width of the yellow area, then the yellow area is either square or taller than it is wide. In this case, the idea is to draw a horizontal wall somewhere in the green area:

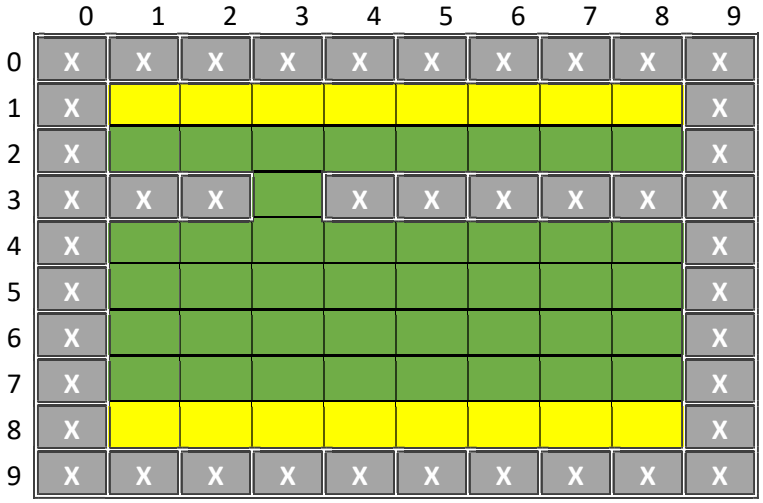


By avoiding the top and bottom row of the yellow area, we prevent ourselves from drawing a wall that might box ourselves in by covering up a passageway.

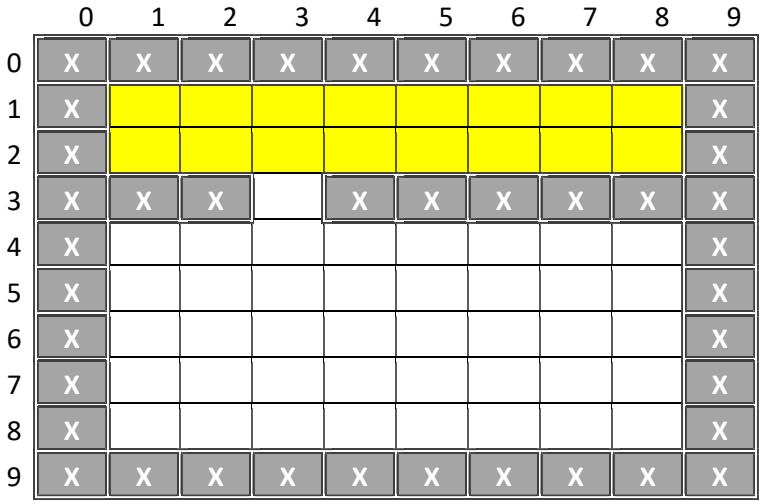
In the green area, randomly select a vertical position (in this case from y=2 to y=7) and draw a horizontal wall all the way across the green. In this example, I randomly selected y=3:



Next, check if the ends of the line we just drew are next to a blank. If so, make the end wall a blank. This is to allow passage through the blank instead of blocking it. In this example, check if (0,3) is a blank. If it is, then make (1,3) a blank. Also check if (9,3) is a blank. If it is then make (8,3) a blank. There is no blank to add in this case. Next, randomly select one of the cells in the wall that was added and turn it into a blank. This adds a hole to allow passage between the two halves of the maze separated by the new wall. In this example, I randomly picked x=3 and turned cell (3,3) into a blank:

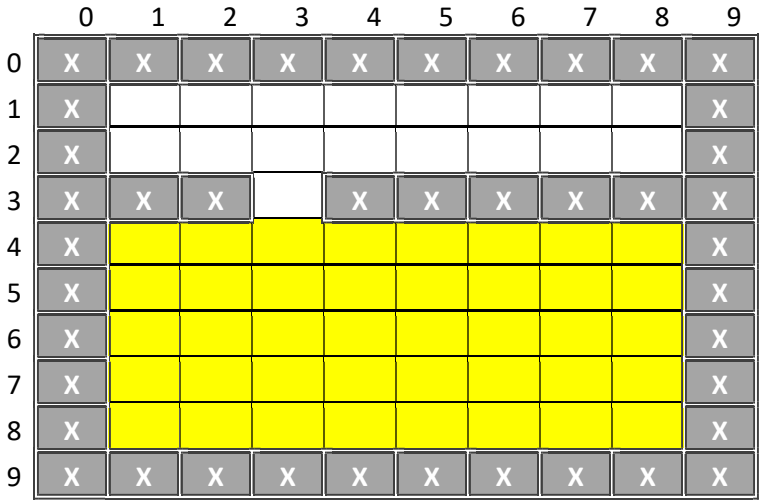


Next, recursively repeat the process in area above the wall we just drew, and below the wall we just drew. Here would be the recursive call for the area above the wall, highlighted in yellow:

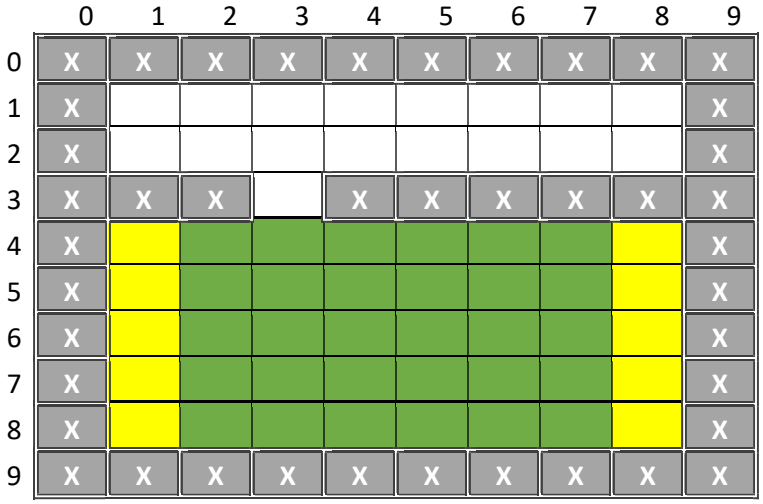


This area has a height that is <=2 so the recursive call will just exit.

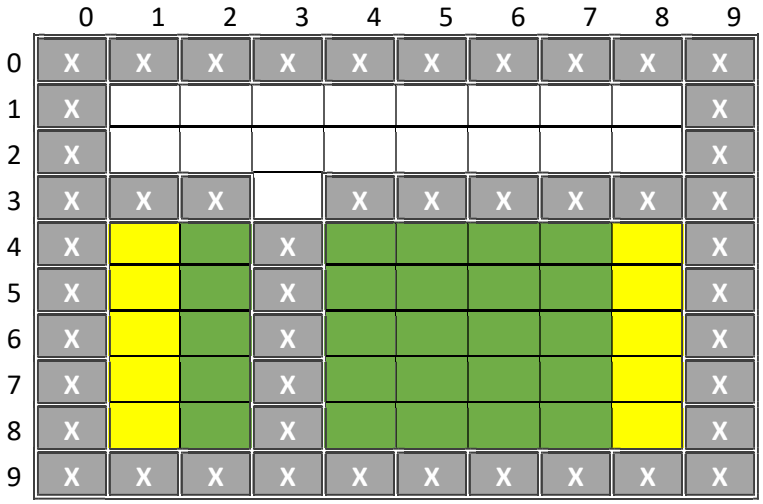
Next, here is the recursive call for the area below the wall, with the area highlighted in yellow:



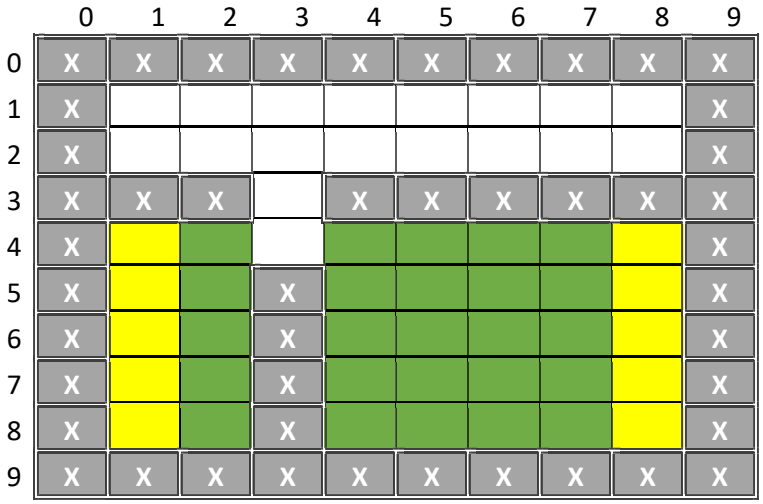
This area has a large enough width and height to continue. In this case the width is greater than the height so instead of a horizontal wall we draw a vertical wall, using the same idea as before. First we pick a random x coordinate in the green area to draw a vertical line. As before we skip the edges as candidates to draw a wall to avoid blocking passages:



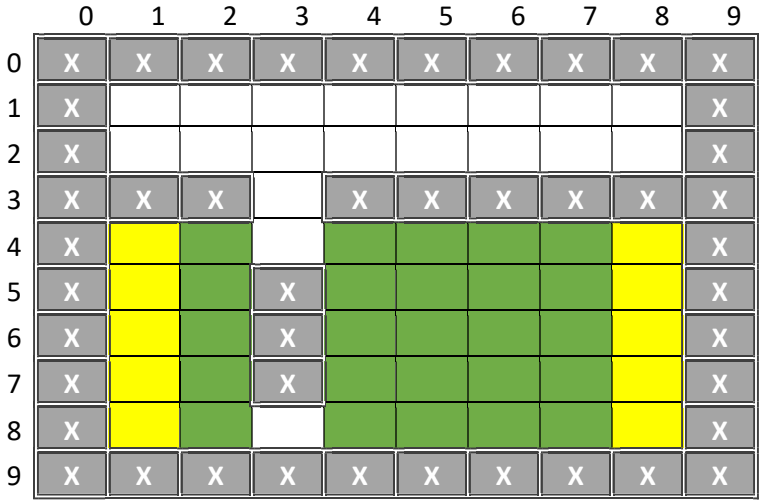
The random x coordinate for this wall could be anywhere from x=2 to x=7. Let's say we happen to pick x=3. This results in a vertical wall at x=3:



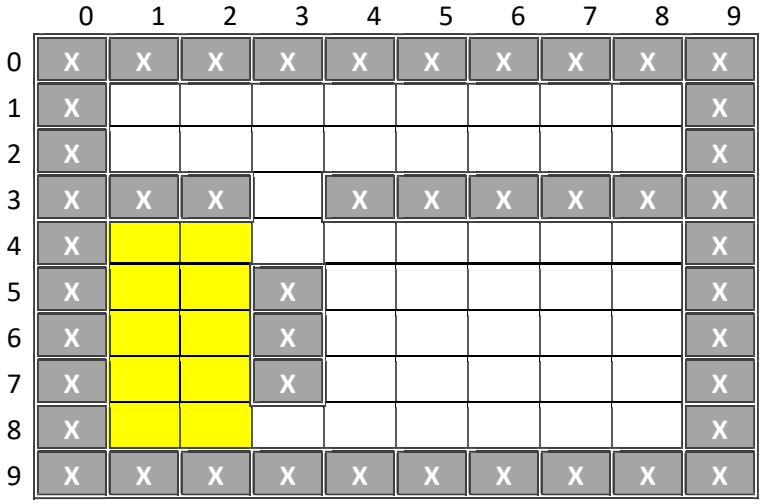
Check the ends of the wall we just made to make sure there aren't any blanks we are blocking. In this case we check (3,3) and it is a blank! So we turn (3,4) into a blank so we don't block the passage. We would also check (3,9) for a blank. It's not but if it was we would turn (3,8) into a blank. At this point we have:



Next we randomly select one of the cells we turned into a wall and make it into a blank, to poke a hole in the wall and allow passage between our wall. Let's say we randomly pick the cell at (3,8):

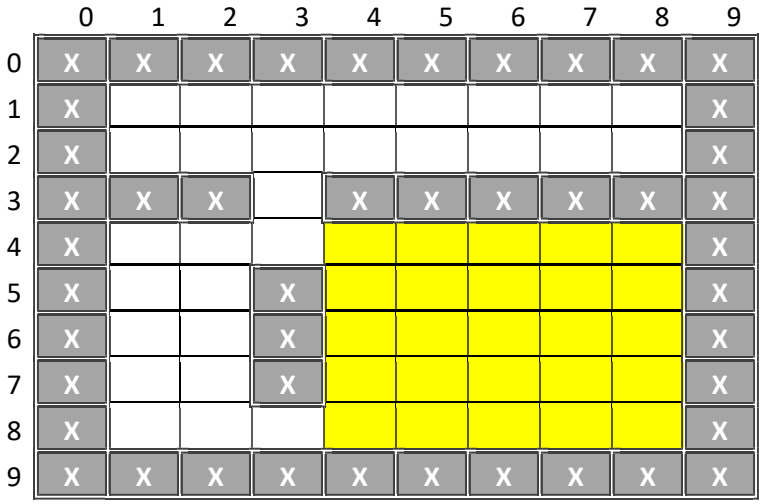


Now we make a recursive call to the area to the left of the wall we just made. In this case it would be a recursive call focusing on the yellow area:

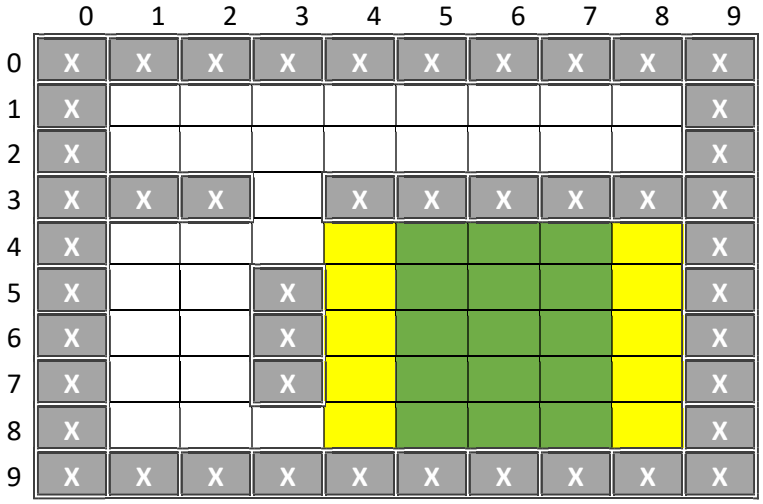


This area has a width that is too small, so it just exits.

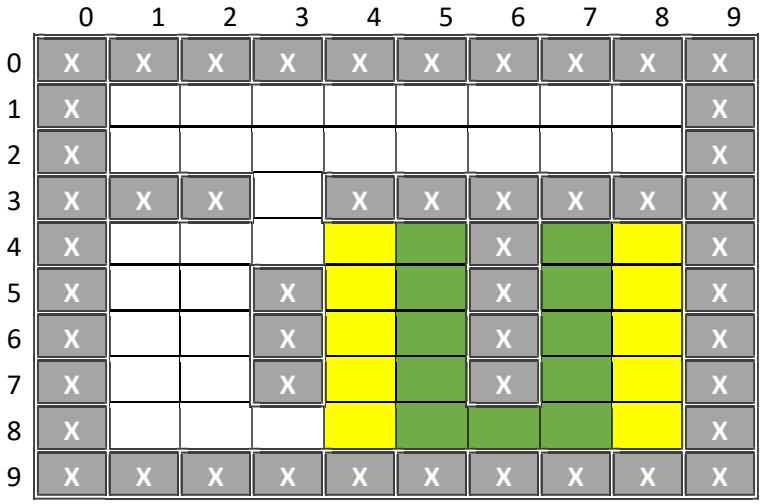
We would also make a recursive call to the area on the right of the previous wall we made. In this case it is this area in yellow:



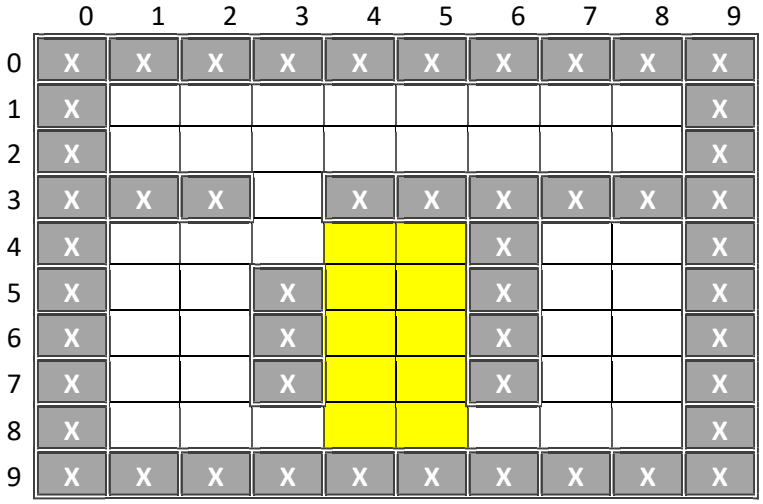
The width is greater than the height so we draw a vertical wall somewhere in this green area:



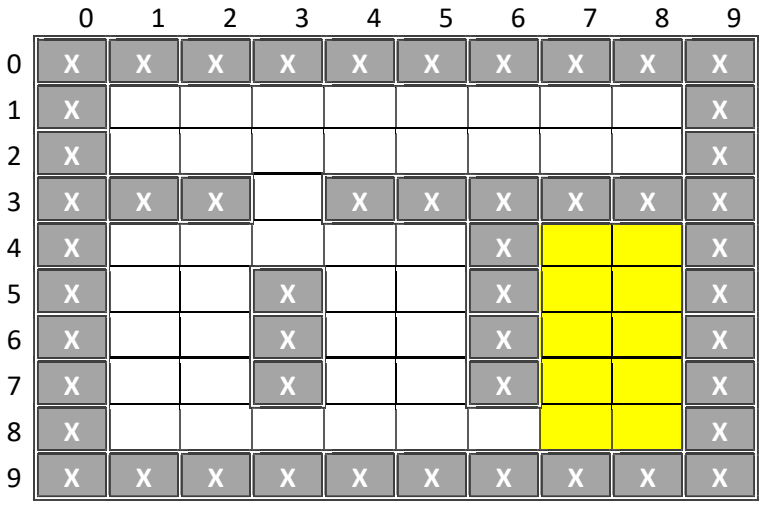
Say that we randomly pick x=6. We check the ends and poke a hole in a random spot in the wall, say at y=8:



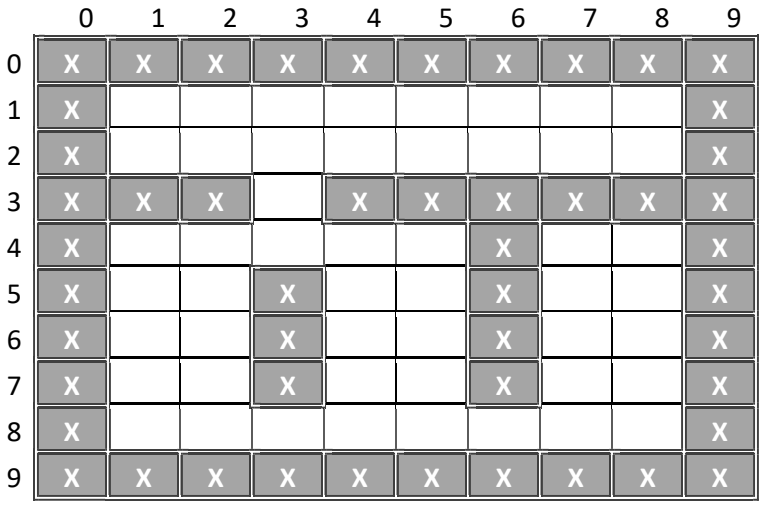
We make recursive calls on the area to the left of the wall we just made but it is too small so it exits:



Finally, we make a recursive call on the area to the right of the wall we made but it is also too small and exits:



This is the final maze! Optionally, you can add a start and end position. It will be possible to reach every empty space on the maze from any other empty space.



TO DO: Write a program that randomly generates 40x40 mazes using the algorithm described here. You don't need to solve the maze or add a start and exit, but you could do that if you wanted to.

If you want to run a sample solution you can do so on transformer by running the executable I put on transformer. The pathname to run the program is:

```
~sebastian/CSCE_A211/mazegen
```

4) Decrypt the Enemy Message – 10pts

Your country is at war and your enemies are using a secret code to communicate. You have managed to intercept a message that reads as follows:

```
,vtaNm a_"dabp!!
```

The message is 16 characters long. The message is encrypted using the enemy's secret code. You have just learned that the encryption algorithm is to take the original message, treat each group of 4 bytes like an integer, add a secret key to the integer, then copy the resulting number to the encrypted message treating it like four characters.

For example, if the original string is **"HI THERE"** and the secret key is the number **2**, then the algorithm would:

- Take the first four characters, which are the first four bytes, which are "HI T".
- If these four bytes are typecast to a 4 byte int (the size of an int on most machines) then it has the value 1411402056.
- Add the secret key of 2 to the value resulting in the value 1411402058
- Typecast the 1411402058 back as a 4 character string, resulting in "JI T" (basically it just increases the leftmost character by 2 in the ASCII code)

The process is repeated for the next group of 4 characters, "HERE":

- These four bytes are typecast to a 4 byte int which is the value 1163019592
- Add the secret key of 2 to the value resulting in the value 1163019594
- Typecast 1163019594 back as a 4 character string, resulting in "JERE"

The entire encrypted string would be "JI TJERE"

In the case of `,vtaNm a_"dabp!!` you have figured out that the secret key is a number between 1 and 500.

Write a function that decrypts an encrypted message using a key that input as a parameter. From main, call the function with numbers between 1 and 500 for the key, printing out the resulting decrypted text each time. When you hit the correct key you will get a message that makes sense and have cracked the code! You should implement your function/program with pointers that uses typecasting to map back and forth between `(char *)` and `(int *)` as appropriate.

What is the secret key and the decrypted message?

Note: This is not a very strong encryption system so you probably wouldn't want to use it in practice