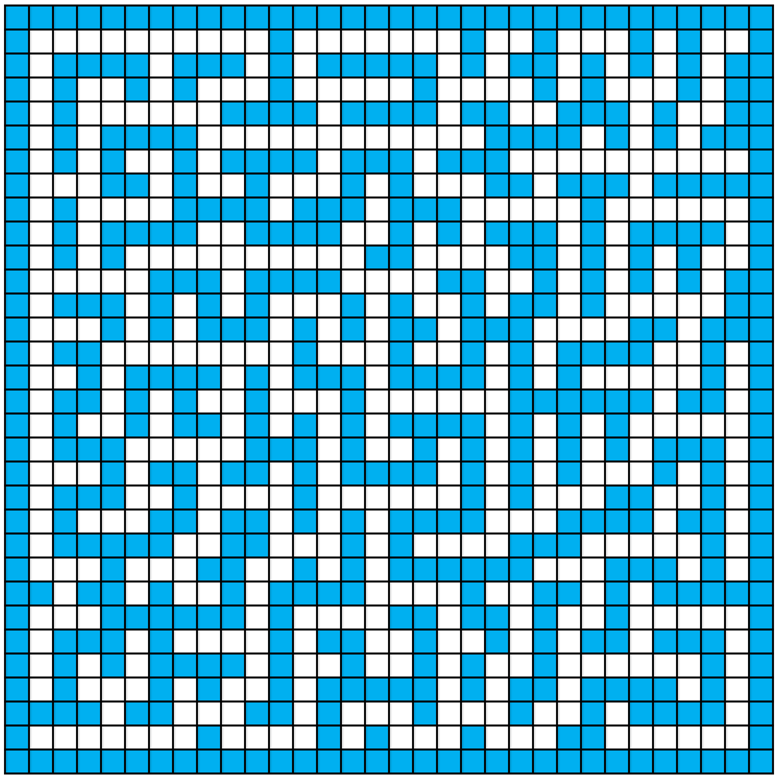
COMP2401 - Assignment #2

(Due: **Monday, October 12th, 2020 @ 6pm**)

In this assignment, you will practice using bit manipulation.

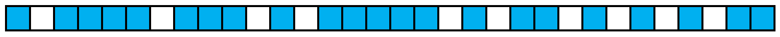
Consider a 32x32 maze as shown here.



This maze can easily be represented as a 2D array (i.e., char maze[32][32]) where each char is a 1 for a wall or 0 otherwise. This would require **1024** bytes to store. Instead, we will consider encoding a maze by making use of each bit in the byte. We will take each row of the maze, and encode it into a single 32-bit integer (i.e., an **int** in c).

Therefore, the encoded representation of the maze will only require 32 **ints** of storage (i.e., one per row) … which is just **128** bytes in total.

Looking, for example, at the third row of the maze … we can convert it to a 32-bit sequence of bits where a 1 indicates a wall and 0 indicates an open spot. Then, if stored in an int, when printed, it would show the two's compliment representation, which in this case is -1109928277 … as shown here:



10111101110101111101011010101011

= **-1109928277** in two's compliment notation

Write a program called **decodeMaze.c** which does the following:

1) The **main** function should define an array of 32 **int** values that represent the above maze. You should hard-code the values for this maze. The value for the third row has been given to you above. There are some on-line binary to two's compliment converters that can do this. You MUST NOT hard-code this maze anywhere in your code as a 2D array. It MUST be hard-coded as a single 32 **int** array.

2) Write a function that has this signature:

```
void printEncodedMaze(int [], unsigned char)
```

It should take in the above 32 **int** maze array and the dimension (which is 32) and then display the maze as shown on the next page as figure (a). Note that the character representing a wall is the @ symbol. The function must not create any other arrays, nor attempt to convert the maze to another format. You may not use helper functions. It must simply process the encoded maze and display it as shown. Call this function from your main function to make sure that it works.

3) Consider now a particular encoded path in the maze. The path is encoded as 32 integers as well, but this time a bit is set to 1 if the path is on that maze cell and 0 otherwise. Obviously, the path will have a zero for each bit that corresponds to a wall. In your main function, define the following array that represents the path:

```
{0, 0, 0, 0, 12, 8, 56, 32, 8032, 4416, 134115648, 67354944, 67109184,
 67109312, 133169152, 1048576, 1835008, 262144, 262144, 262144, 458752,
 65536, 65536, 65536, 65536, 983040, 67633152, 67633152, 201850880,
 164102144, 259522560, 0};
```

Write a function that has this signature:

```
void printEncodedMazeWithPath(int [], int [], unsigned char)
```

Here, the first parameter is the maze, the second is the encoded path, the third is dimension again. The function should display the maze and the path together as shown in (b) below. The code must be as efficient as possible. The highlighted yellow is just to make it clear where the path is … you will not be highlighting anything in your code. Make sure to call this function from your main function to ensure that it works properly.

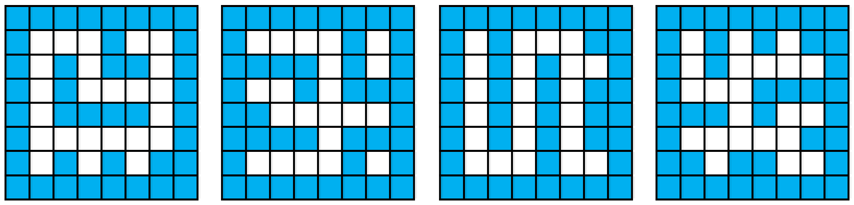
```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@          @       @  @   @ @  @
@ @@@@ @@@ @ @@@@@ @ @@ @ @ @ @@
@ @  @ @   @     @    @ @   @ @@
@ @      @@@@ @@@@ @@  @@@ @  @@
@ @ @@@@            @@@@ @ @ @@@
@ @ @  @ @@@@ @@@ @@@          @
@   @@ @  @   @ @   @@ @@@ @@@@@
@ @    @@@@ @@@ @@@     @      @
@ @ @@@@  @@@@  @ @ @@@ @ @@@@ @
@ @ @          @@    @@ @ @ @  @
@     @@@ @@@@    @@  @ @ @ @ @@
@ @@@ @ @ @   @ @  @ @@ @     @@
@   @ @ @@@ @ @ @@ @@@    @@ @@@
@ @@        @   @  @ @ @@@@  @ @
@  @ @@@@ @ @@@ @@@@ @ @     @ @
@ @@ @ @  @   @      @@@@@@ @@ @
@ @  @ @@ @ @ @ @@@@ @ @ @     @
@ @@@     @@@ @  @ @ @ @ @ @@@ @
@   @ @@ @@ @ @@@@ @ @ @   @ @ @
@ @@@  @    @      @ @   @@  @ @
@ @   @@ @@ @ @ @@@@   @@@@ @@ @
@ @@@@@  @@   @ @    @@@     @ @
@   @   @@  @ @ @@@@@@   @@@ @ @
@@ @@ @  @ @@@@    @  @@ @ @@@@@
@   @@@@@@ @    @@ @@ @  @     @
@ @@@ @    @ @@  @  @ @ @@ @@@ @
@ @ @ @@@@ @  @  @ @  @      @ @
@ @   @ @  @ @@@@@ @ @@ @@@@ @ @
@@@@ @@   @@ @   @   @  @ @@@@ @
@       @    @ @   @   @@      @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

(a)

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@          @       @  @   @ @  @
@ @@@@ @@@ @ @@@@@ @ @@ @ @ @ @@
@ @  @ @   @     @    @ @   @ @@
@ @      @@@@ @@@@ @@  @@@ @..@@
@ @ @@@@            @@@@ @ @.@@@
@ @ @  @ @@@@ @@@ @@@     ...  @
@   @@ @  @   @ @   @@ @@@.@@@@@
@ @    @@@@ @@@ @@@.....@..    @
@ @ @@@@  @@@@  @ @.@@@.@.@@@@ @
@ @ @..........@@... @@.@.@ @  @
@    .@@@ @@@@....@@  @.@.@ @ @@
@ @@@.@ @ @   @ @  @ @@.@.    @@
@   @.@ @@@ @ @ @@ @@@ ...@@ @@@
@ @@ .......@   @  @ @ @@@@  @ @
@  @ @@@@ @.@@@ @@@@ @ @     @ @
@ @@ @ @  @...@      @@@@@@ @@ @
@ @  @ @@ @ @.@ @@@@ @ @ @     @
@ @@@     @@@.@  @ @ @ @ @ @@@ @
@   @ @@ @@ @.@@@@ @ @ @   @ @ @
@ @@@  @    @...   @ @   @@  @ @
@ @   @@ @@ @ @.@@@@   @@@@ @@ @
@ @@@@@  @@   @.@    @@@     @ @
@   @   @@  @ @.@@@@@@   @@@ @ @
@@ @@ @  @ @@@@.   @  @@ @ @@@@@
@   @@@@@@ @....@@ @@ @  @     @
@ @@@.@    @.@@  @  @ @ @@ @@@ @
@ @ @.@@@@ @. @  @ @  @      @ @
@ @ ..@ @  @.@@@@@ @ @@ @@@@ @ @
@@@@.@@...@@.@   @   @  @ @@@@ @
@   ....@....@ @   @   @@      @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

(b)

4) Consider the following four 8x8 mazes:



In your main function, hardcode these mazes as char arrays of 1's and 0's

```
{{1,1,1,1,1,1,1,1},
 {1,0,0,0,1,0,0,1},
 {1,0,1,0,1,1,0,1},
 {1,0,1,0,0,0,0,1},
 {1,0,1,1,1,1,0,1},
 {1,0,0,0,0,0,0,1},
 {1,0,1,0,1,0,1,1},
 {1,1,1,1,1,1,1,1}}

{{1,1,1,1,1,1,1,1},
 {1,0,0,0,0,1,0,1},
 {1,1,1,1,0,1,0,1},
 {1,0,0,1,0,1,1,1},
 {1,1,0,0,0,0,0,1},
 {1,1,1,1,0,1,1,1},
 {1,0,0,0,0,1,0,1},
 {1,1,1,1,1,1,1,1}}

{{1,1,1,1,1,1,1,1},
 {1,0,1,0,0,0,1,1},
 {1,0,1,0,1,0,0,1},
 {1,0,1,0,1,0,1,1},
 {1,0,1,0,1,0,1,1},
 {1,0,1,0,1,0,1,1},
 {1,0,0,0,1,0,0,1},
 {1,1,1,1,1,1,1,1}}

{{1,1,1,1,1,1,1,1},
 {1,0,1,0,1,0,1,1},
 {1,0,1,0,0,0,0,1},
 {1,0,0,0,1,1,1,1},
 {1,1,1,0,1,0,0,1},
 {1,0,0,0,0,0,1,1},
 {1,1,0,1,1,0,0,1},
 {1,1,1,1,1,1,1,1}}
```

Write a function that has this signature:

```
void encode8by8Maze(char inputMaze[8][8], int encodedMaze[8]) {
```

The function should encode the given 8x8 maze into an array of 8 **ints**. Normally we would store it as an array of 8 **unsigned chars** to save more space, but we will be passing these encoded mazes into our print function, which requires an **int**. The structure of your code must be efficient.

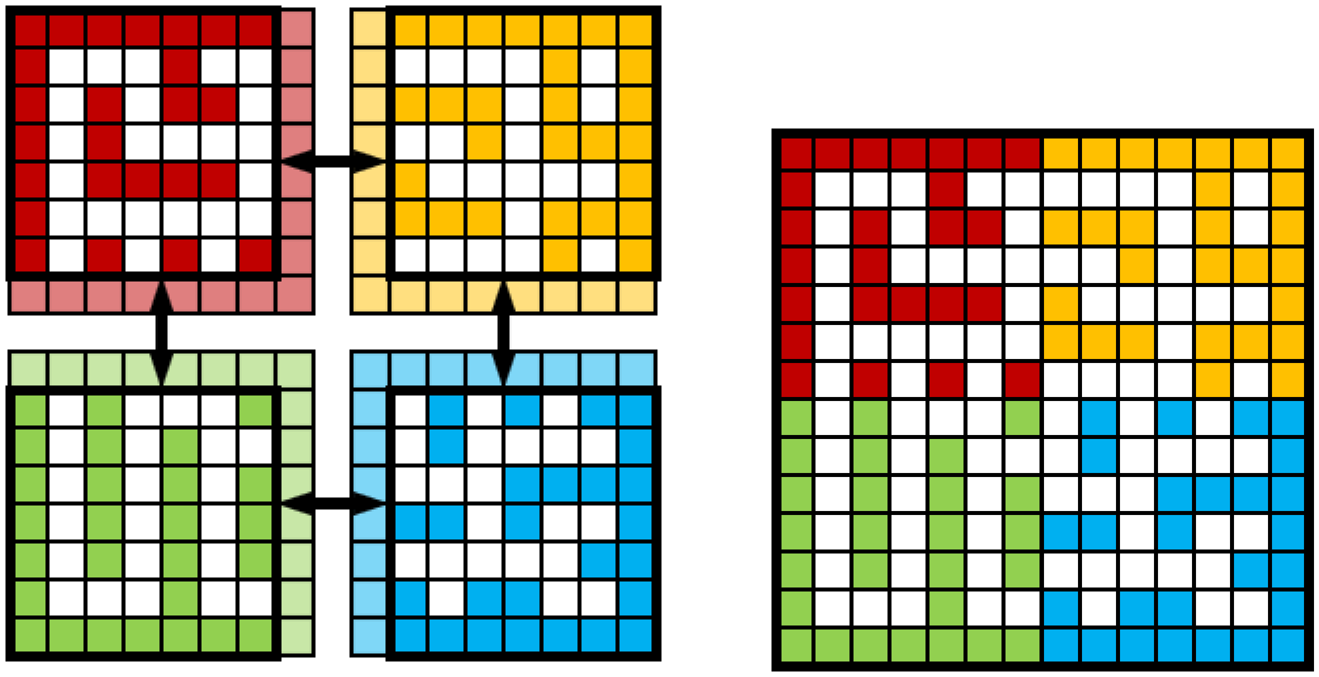
Make sure to call this function for each of the 8x8 mazes and then display each maze by calling the print function from step 2). You should be able to verify the output from the images of the mazes that I gave you in this step.

5) We will now merge mazes together. Write a function with this signature:

```
void merge8by8Mazes(int topLeft[8], int topRight[8],
                    int bottomLeft[8], int bottomRight[8],
                    int finalMaze[14]) {
```

It should take in four encoded 8x8 mazes and output a final encoded 14x14 maze. You MUST NOT convert the mazes into any other format and you cannot create any other arrays within the function. You may not use helper functions. Your code must be concise and efficient.

The function should combine the first 7 rows of the topLeft and topRight mazes such that the last column of the topLeft maze and the first column of the topRight maze is discarded. The result is that there will be 14 columns. Similarly, the last 7 rows of the bottomLeft and bottomRight mazes are combined. The result is a 14x14 maze as shown below. Essentially, the inside border cells have been removed. You must write this function as efficiently as possible.



6) In your **main** function, write some test code that will generate all 256 combinations of 14x14 mazes by combining all possible combinations of the given four 8x8 mazes. Your code should be efficient (i.e., with minimal duplication of code) and it must make use of your function in step 5) as well as the print function in step 2 to display the 14x14 maze each time. Also, just before printing the 14x14 maze, indicate which maze combination you used. For example, in the above example, it was mazes 1, 2, 3 and 4. Whereas if maze 1 (i.e., the red one) was used in all 4 corners, you would print out 1, 1, 1, 1 before the 14x14 maze.

7) Write a function with this signature:

```
char isPathInMaze(int[], int[], unsigned char)
```

Here, the first parameter is an encoded maze, the second is an encoded path and the third is the maze dimension. The code should return 1 if the given path fits properly in the maze and 0 otherwise. A path fits properly if it lies completely in the open areas of the maze. That is, the path should not cross onto a wall of the maze. You must write this code efficiently without creating any additional arrays nor helper functions and you must not try to convert the arrays into any other format.

In your main function, after displaying the large maze and the maze with the path, call this function to determine whether or not the path fits in the maze. It should. Display the result of the function call to be sure.

In your **main** function … when you display the 256 maze combinations … for each one … you should call this **isPathInMaze()** function with the following path (for a 14x14 maze):

{0, 4096, 4096, 4096, 4096, 7936, 256, 256, 448, 112, 16, 28, 6, 0}

You can hardcode this path in your main function. When you run the 256 combinations, you should notice that exactly 8 mazes will return a result of 1 (or true) for the function call. As you are printing the 256 mazes, make sure to only display (with a sentence of some sort) which mazes can fit the path. If a maze does not fit the path, do not print a sentence. You should be able to easily spot the 8 mazes that fit the path as you scroll through the results.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files (in this case there is just one) as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code

2. All of your **.c source** files and all other files needed for testing/running your programs.

3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).