When I initially outlined the basic architecture of the game, the first thing that needed to be implemented was the board of the game itself. I considered what was contained within the board and what information was needed from the board. This information was the length of the board, the number of different `colours`, and the array of `colours` itself. As each board is different and depending on the set up of the board it made sense that I made a structure of the board that contained these three pieces of information.

The first function that I built was ‘setUpBoardMem()’ which dynamically sets up the space in memory for a board according to the board length given. To test this I passed in a pointer to the board and a length and tested that the board colour array pointer was not null having set up the board. To prevent memory leaks, I created a simple function to `clean up` the board which frees the board pointer array once we are finished with the board.

\*\*\*A TEST YOU CAN TRY IS ASSERT THAT BOARD ARRAY IS INDEED NULL

Once I had tested that I could dynamically set up the board there were a few more things to consider. These considerations were:

the setting up of the `default` board of a length of 14x14.

Setting up a board by taking in an argument from the user

Using this argument or reading in a file and from there determining the dimensions of the board.

At this stage I was only concerned with the length of the board and not the `colour count` and therefore the next function that I built checked that the length given either by the user from the command line or the file, was a valid board length which I determined to be up to 20 as per the assessment guidance. There was no specified minimum length for the board, so for the purposes of this game it is possible to have a board of 2x2, but a board of 1x1 made little sense as it could only fit in one cell of an array and therefore would always result in a ‘win’.

I tested all possible valid lengths of the board and asserted that they are indeed valid board lengths. The function I used to test this was `assertBoardLenValid`() and to test that this function worked I looped through all possible valid board lengths and call the function for each length. A valid board length is at least the minimum side of the board, which is 2, and smaller than the maximum side of the board, which is 20. I then created a similar function that dealt with the colour count (the number of different `colours`) so I simply checked that no colour was less than the minimum number of colours which was one and no more than the maximum number of colours which is nine. Personally I would have chosen two as the minimum number of colours, however one was specific in the handout. Therefore any board set up with minimum number of colours in an immediate win in this version of the game.

To test that the colour count function was valid I ran a similar loop of the colour count of the board, checking each number of colours from minimum to maximum ( one to nine) passed through the function. It was difficult to test invalid colours ie above ten or more because I had created a void function For both asserBoardLenValid() and its counterpart asserColourCountValid() it was difficult to test for invalid arguments because the function would quit the programme. Therefore if these two tests pass, these arguments are valid and the function works as expected. To make it easier to test if each individual `colour` is valid, I created a boolean function which is much easier to test as I can loop through the valid number of colours – between minimum and maximum – and assert that each colour is valid. I then assert that any number above the maximum number of colours up to 255 which is the highest an unsigned char type can be and I assert that each of these colours between 21 and 255 are invalid for the purposes of this game. I also assert that 0 number of colours is invalid; I do not need to look below 0 because an unsigned char cannot be a negative number which is why I used an unsigned char typedefed as a colour type (`Colour\_t`) so when we are dealing with colours the code is more readable and it is clear that colours are being dealt with.

Having tested this function, I realised that – as I was dealing with unsigned chars, initially chosen because I knew I would not be dealing with negative numbers of numbers above 255 – I edited my code so that I did not confuse unsigned chars used for indexing with unsigned chars that referred to the colour and it was at this point I typedefed colour type for readability. Building my programme and testing it as I went helped to scale down each component of the game and encouraged me to check back to the house style and ensure I maintained good practice throughout my code.

I was noticing that I was indexing frequently and directly accessing the board pointer colour array which is bad practice. Therefore, I created three simple functions that I could use throughout the rest of the programme which indexed the colour array, set a colour at that index, and also allowed the programme to get a colour from that index. The trickier part of building getIndexFromRowCol() was getting my head around reading through a one-dimensional array but treating the array as if it was two-dimensional. In order to index a specific cell I used the formula *Length X Row + Column*, which took in a pointer to the board and could be dynamically used depending on the user’s specific length of the board. This made getting and setting colour at a specific cell much easier.

To test that I was indexing correctly I use a nested four loop of the rows and columns up to the length of the board and assert that each index starting at 0 equals the index we get from getIndexFromRowCol(), then we increment the index by 1, thus checking each cell in a test board. This small function saved me using this same nested four loop structure throughout my code, making more complicated functions smaller and easier to read. getColourAt() and setColourAt() uses getIndexFromRowCol() so it was important to test that getIndexFromRowCol() worked properly.

For instances whereby the board is not filled in from a file we need to randomly generate the colours that we put into the board array. This is quite simple to do as we can use the libraryrand() function setting the upper limit as the colour count which we get from the board, plus the minimum number of colours – which is 1 – to generate the colours between 1 and the upper limit as specified by the user. I test random colours by looping 100 times to ensure that each of the 100 colours generated by the function produces a valid colour every time, which I tested by asserting that each test colour passes the isValidColour() function. Finally with the board, I use the library function memset to fill the board array with a colour we know is invalid as I hard code in a large number. I then call the fillBoard function which should fill the board array with valid random numbers and then we test that the board has been filled with valid numbers by again using a nested four loop through the rows and columns, getting each colour and asserting that it is a valid colour.

getColourAt() gets the index using getIndexFromRowCol() and returns the colour at that index and setColourAt() is similar to getColourAt() but instead of returning the colour at the specific index, it puts the colour given in the parameter of the function into the board array at that index. In order to test that we are getting and setting colours at a specific index of the board, we loop through rows and columns, generate a random colour, set that random colour using setColourAt() and assert that the getColourAt() function returns the same colour that we generated.

The above functions are all related to the setup of the board, and the following are functions that deal with handling the arguments that are passed in from the command line or file when we create the first board. Up to this point I have just used a hard-coded value to do this.

There are four possible ways to set up the board. Bearing in mind the four different ways to set up the board I now had to handle these arguments. Because of the functions that I had built thus far, it was a relatively simple process to set up the default board, taking in a length of the board, and taking in a length and number of different `colours`. Once I had set these up for the default board, I asserted that the board length was 14 and the colour count was 6. Then for setting up a board with a length I assert that the length I gave the function is the length that is stored in the structure of the board and repeat this again for setting up a board with a length and colour and assert that the length and number of colours I put into the function are then stored in the board structure.

It was difficult to make an automated test to check argv to handle arguments from the command line and test that sscanf worked correctly, though I was able to manually test this I can only expect that a library function would work as it should. As I used sscanf to take in the board parameter and number of `colours` I was unable to automatically test this function, however I do test that the parameters given in to handling these arguments work properly because I tested the two different setupBoard functions. Upon reflection, using the error message and exiting (-1) makes this function more difficult to test because we cannot test for incorrect values because the programme quits. If I were to redo this coursework instead of using exit (-1) I would instead return 0 to indicate failure. By this point this modus operandi was too integrated with the way in which I had built the programme and whilst I initially liked this feature it did make testing either difficult or in some cases not possible and this is something that moving forward I would keep in mind and change how I report failures such as using errno instead. Unless I wrote a script that set argv it was not possible to automatically test. Functions that take in arguments from the command line with I-O (input output functions) are ‘impure’ and such functions can’t be easily automated. When handling the file the things that we need to consider are checking that the file contains only numbers between 1 and 9, that the ‘board’ in the file is a square, and to find the highest number (number of different colours).

Once we are satisfied, then we can read the contents of the file into the board array whilst also bearing in mind that we convert the characters in the file do not return ASCII value of these characters which we do by converting the digits from the board into `colour\_t`. The tests that I run for handling the file are checking that none of the lines are empty in the file, checking each line length is the same, checking that we have converted all the digits to `colours`, and that the colours from the text file are being correctly inserted into my board array, and that the colour count we get back from the board is correct. It was difficult to make an automated test for this for handling line length so I had to run a manual test for inconsistencies in line length and if the programme exited then it was working as I expected it to be. Again, this is another example of it being difficult to automate testing this function because exit (-1) renders it difficult to test for incorrect values.

Finally we test the game functions by asserting that the checkIfWon() function works and we do this by using memset to flood the board with a single colour so that every colour on the board is the same and therefore the checkIfWon() function is true. To check that the game logic works correctly we feed a test array into the updateBoard function and a test colour and then we assert the updated test array against a hard coded expected solution of the board after this ‘turn’. By testing updateBoard we also test updateBoardRecursive because it is called in updateBoard. A bug that I discovered whilst testing this function was that there was a segmentation fault if flood and target colour were the same. To get around this bug, I added a protective if statement for instances whereby target colour and flood colour were the same, and in those instances we do not do anything because the board is already in the state that we desire it to be in.

Another bug that I picked up on when manually testing the game was a fault in the game logic, whereby if a colour in the row above was the same as the colour in [0,0] it did not change on the next turn, meaning the game was not connecting colours above or colours in row [0] and down the column. This bug occurred because I was initially only looking along a row and down a column but not also within rows checking the column above or within a column checking the row behind. To resolve this, I added within the recursive updateBoard() function the capability to ‘look above’ and ‘look behind’ for matching `colours`. This is now fixed.

As the captureInputTurn() function is an ‘impure’ function because it uses scanf, we cannot test this but within this function we manually check that the input from the user is correct. Importantly we use fflush() because scanf does not clear out the buffer so for invalid input it keeps looping the invalid input infinitely which we do not want. The desired effect of clearing out the input buffer is why I used fflush because I want to clear out any input other than the first input and play each ‘turn’ separately.