**AC21009 Assignment 3 – Cellular Automaton**

**Group F**: David Topping, Caleb Harmon, Heather Currie

**Word Count**: 597

**Command to compile program** :g++ \*.cpp -Wall -Wextra -Werror -o cellularAutomaton

**Command to run program**: ./cellularAutomaton

# Our Approach to the cellular Automaton

Our initial approach was to make sure everyone understood the requirements of the assessment and to discuss potentials ways to create the program. We held a meeting where we looked at pre-existing examples of cellular automatons and discussed how we could implement our own. Initially we planned to program in C, later we decided to use C++ so that we could utilize the advantages of object oriented programming.

To hold the data of the rule being used, the ‘Rule’ class is used. The array holding each line’s data was compared to each individual rule in a ‘RuleSet’ class which was is a collection of 8 rules (each representing a binary bit). The Rule class contains an integer array of size 3 which contains either a 1 or a 0 depending if a cell is on or off. It also contains Boolean which defines if the pattern is found in the array will result in the cell below being on or off (1 or 0). The ‘RuleSet’ class takes a denary number and converts it into 8-bit binary to determine the rule being used.

The function used to for the automaton is the ‘newLine’. Initially we created this function to return a pointer to an array of the next line. This function also took two arguments, one being the Rule that was currently in use, the other being the previous line. However, this function didn’t work as well as was hoped, as the code was becoming cluttered when we had to call the newLine function again using the returned array pointer. We found more concise solution, which was to use recursion. Instead of returning an array pointer, we had the function call itself using this array pointer as one of its arguments. We also had to create a third argument for the function, an integer called ‘endCondition’. This argument let the user decide how many times they wanted the function to run for and gave the recursion an exit point.

# Our Solution

We have come up with a robust cellular automaton that has a menu system with multiple options. Our automaton can be run complete with user defined conditions; the user can enter the rule (in denary), the number of generations they want to complete and the initial starting point. The main menu also gives the option to run the automaton for 40 generations but on a random rule, this is interesting for seeing all the possible patterns. Lastly, every time the automaton is run the result is also saved to a text file so we have an option in our menu to see the previous cellular automaton result.

# Problems We Faced

While we were working on the assessment, we used GitHub to collaborate and organize our workflow. However, we initially had issues with configuring GitHub to work with Visual Studio Code and WSL Ubuntu. Pushes and fetches weren’t going through, but we managed to fix this by using the Git console commands instead of the built-in Git interface in Visual Studio Code. We also had a few Git flow problems, where we had merge conflicts between our branches on GitHub. These were caused by people working on the same file but on separate branches and had to be resolved by either manually fixing the conflicts within the file or manually merging the code from one branch into another.