Requirements

Your task is to create a simple 1-dimensional (1-D) cellular automaton (CA) which will display each generation of your automaton in the shell (as this exercise is to give you practice C programming in a Unix-like environment, it should be performed using these tools in the lab, or equivalent).

Our CA will be binary, so each cell will display either 1 or 0. In a 1-D CA, each generation is just a single line. The values of the cells in each generation are determined by applying the rules to the values of the cells in the previous generation.

For example, if we use rule set 30 (remembering that 30 = 00011110 in binary), gives:

Current state of 3 cells above 111 110 101 100 011 010 001 000

New state for centre cell 0 0 0 1 1 1 1 0 ( =30^10 )

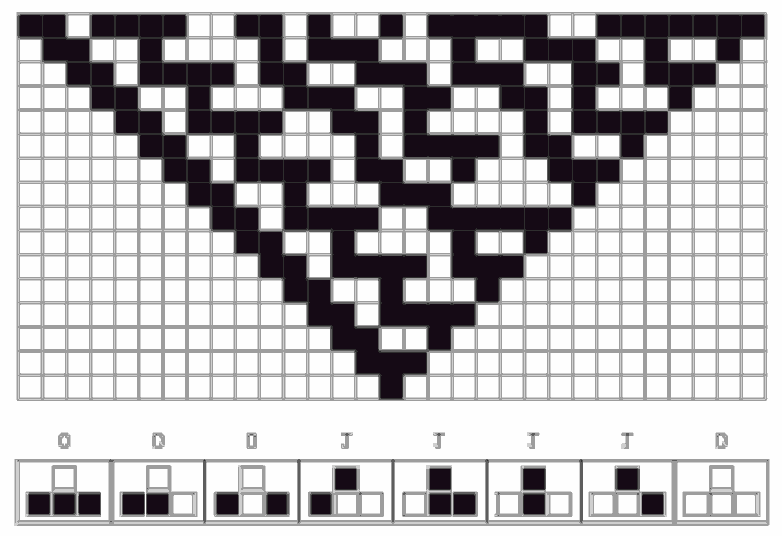
Convert to binary function

Use string for rule set conversions to binary because that way each character has a separate address which can be easily referred to for rule construction

if you want to advance to a 2D CA then you need 2D arrays for reference since each configuration will be 3\*3, and there will be 256 configurations for every ruleset and 65536 possible generation states

then we should get output similar to the following (although yours does not actually have to

look exactly like this):



Getting Started

You should probably start with storage for two generations of cells: a parent generation and a child (or current) generation. One way of implementing this storage is using arrays. Initially you will seed the parent generation at time t=0, i.e. put the value 1 or 0 into each cell of the parent array. You now need to work out what the child generation should contain given the contents of the parent.

Parent pointer as part of the “generation” struct?

You do this by applying the rules as discussed above, for example, if the parent cell and its neighbours to either side contain the values 0, 1, 1, then the child cell should contain the value 1. Once you have completed the child generation you can discard the parent, although you might want to consider writing it to a file so that you can compare the output between different runs.

echo append a dump file?

At this point the child generation becomes the new parent generation, and you should work out what the next child generation should be.

You might want to start with rule set 30, as shown above, for specifying how the child generation is derived from the parent. Alternatively, you might want to put together your own rule set (there are 256 possible different rule sets that you can generate from various combinations of the 3 ancestor cells, and this is only for the most simple type of automata).

Before you do this you should decide how big you want the storage for each of your generations to be, e.g. how many cells make up a generation? 10? 50? 100? Too few and you might have insufficient space for interesting patterns to emerge, too large and you will not be able to fit much of any given generation into your terminal window.

Rather than spending your time learning to do graphics in C/C++ (although this might be a good extension once you have the basics working, because graphics give you more resolution for the output) you should consider how you could indicate the different cells in the terminal, for example, using a 1 to indicate black and a 0 (or even possibly a space) to indicate white.

One other thing that you will have to consider is how to store your automata data whilst you are working on it (e.g. the previous generation and current generation). You might want to use an array of INTs to begin with, so that each integer in the array is a cell in the automata, but there are much more efficient ways to store the cells if you consider that each cell has only two states and is therefore very similar to a bit (binary integer).

You could store your data very efficiently as bit arrays but that would limit you to only two states for your cells and thus your code would be limited to the very simplest automata. Perhaps a happy medium might be to store each cell as a CHAR because a CHAR generally uses about half as much memory as an INT?

Use chars to store the actual value of the cell, or maybe char\* pointers to the value in the related Struct

Perhaps however there is more data that you would like to store with each cell and therefore some sort of simple STRUCT would be better?

This seems to be the right play because it allows for “easy” integrations of parent generations and thus parent neighbour cells via pointers as part of the STRUCT

To get the very highest marks possible for this project, you should consider implementing some of the extensions to the basic requirements. There are many extensions that you could implement if you have sufficient time. For example, you might consider the following:

Simple Extensions - RECOMMENDED

To perform well in this assignment, your solution might include the following features:

• conversion of binary strings e.g. “101001010101” into an array of variables (binary or

integers) and back again

• conversion of binary numbers into their decimal values and back again

perhaps by comparing decimal value to nearest power of 2 and then decreasing from there?

• saving the output of your program to a text file on disk (and loading saved files)

echo to a text file to save

load via time frame variable and generation code? (Would take more time to load the later the frame is)

Other Simple Extensions

• Investigate rules other than rule 30 given above and compare the output that you get

- maybe you could get your program to randomly generate the rule set at the start.

int rng between 0 & 255

• Allow the user to select different lengths of the first generation, and to select different numbers of generations to produce.

Function to set 2d array size?

• Allow the user to select, or create, the first generation

• Wrap the ends of your automata array around so that the right-hand end continues

onto the left-hand end and vice-versa.

• Write your automata out to a text file so that you can look at the results once the

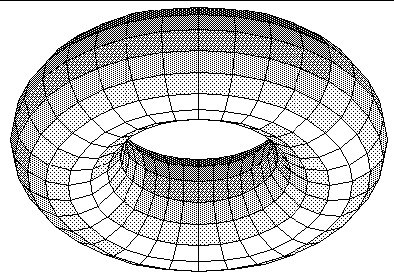
program has completed.

Further extensions

• Implement a version of Conway’s Game of Life (which is basically a 2-dimensional cellular automaton with a particular set of rules).

• Develop a different 2-dimensional automaton.

• Try wrapping around all of the sides so that it forms a 3-dimensional torus as shown

below: