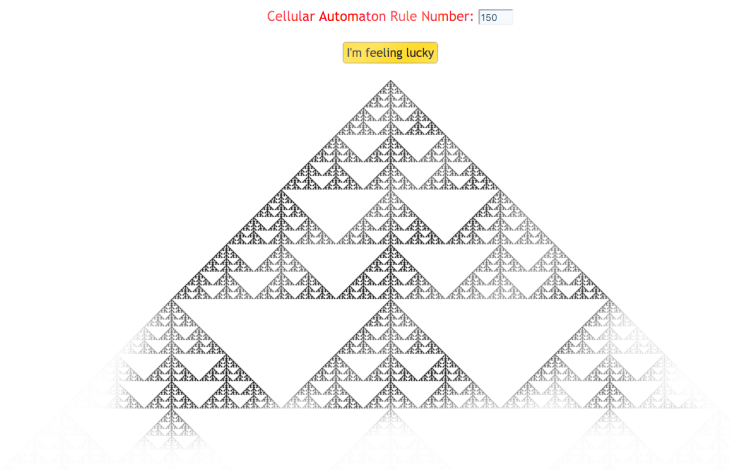
Ey! My et'unda mynah

Engadi 1-ode nainah

# Automata




If you haven't read it yet, read the text, check the internet or refer to the modules for discussions of *Cellular Automata*.

Provided you understand what Cellular Automata are, you'll be clear on what we're going to try and do in this quest.

To help you understand the topic better, here's a slightly different (but related) take on the subject from what you may find elsewhere. Read them all and one might make more sense than it did before. Or just read this spec (it's sufficient).

In this quest, you will implement your own copy of the one dimensional CA you can find at https://quests.nonlinearmedia.org/foothill/demos/ca1d

Let's define an *automaton* to be a deterministic (aka programmatic) process by which a given string of bits is transformed into another string of bits. So you can simply consider the automaton to be a collection of rewrite rules. It's like a mathematical function *y = Automaton(x)* where *Automaton(x)* is a function of the variable *x*, our bit string.

Let's say that a *generation* is an infinite bit string. If you feed it to the automaton, it will return to you another infinite bit string, your *next generation*.

In making the next generation string, the automaton will deterministically produce each bit by mapping a particular sequence of bits in the current generation into this single bit in the next generation.

Obviously, the possibilities here are endless - you can inject all kinds of variations into this mapping process - even external factors like time of day or cosmic ray intensity. But we want to keep it simple (almost trivial) to see if anything interesting emerges with the least amount of external input). That's why we're restricting our attention to automata that only consider a small set of *parent* bits when deciding what the next generation *child* bits should be.

In fact, we consider only possibilities where the value of a bit in the next generation (child) depends on some *contiguous* number of bits in the current generation (parents).

What this means is that *1* or more contiguous parents in an infinite string of bits *determine* each child in the next generation - another infinite string of bits.

You can imagine a sliding window of N bits that moves over the current generation bits to generate the next generation. See Figure 1.

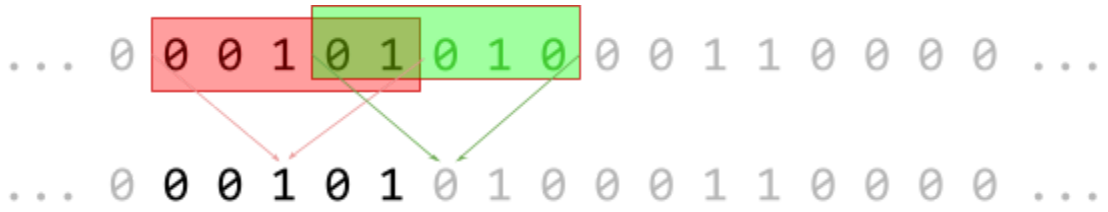



**Figure** 1:How a 5-parent Automaton stitches together the next generation (bottom) from the current (top)

Now, let's go one step further and exclude more automata. Let's ignore automata that have non-odd or negative numbers of parents[1]. That means we're saying "*Maybe there are interesting rule sets for those parent combinations too, but we'll save that for another day.*" Read Robert Frost, maybe.

This is the small subset of automata you'll be building in this quest. You will implement an `Automaton` class, which I can use to stamp out my own `Automaton` objects.

The way the automaton should work is by turning one *generation* of bits into the next. If I invoke it iteratively by feeding its own output back to it, I should be able to generate successive generations of bit strings.

But how do I get started? Let's assume that the very first generation, which I call the *seed generation*, is a single 1 (*the seed bit*) in an infinite sea of all 0s. By restricting our attention to this seed bit and a finite *interesting portion* around this seed that may grow with each successive generation, we can get both easier programming, and aesthetically pleasing outcomes.

Since every bit in the next generation will be based on an odd number of parent bits in the current generation, we'll say that it is the child of a sequence of parent bits which are centered around this child bit (if the two generations, like infinite ticker tapes, were lined up with corresponding bits under each other (as in Figure 1).

Now consider the interesting case when an automaton considers exactly one parent bit for each child bit. Then it needs two rules in its inventory:

1. value (1 or 0) to use if the parent bit is 1
2. value (1 or 0) to use if the parent bit is 0

[1] Actually the miniquests will only check 1 and 3 parent versions. Extra Credit rewards await insightful posts regarding 5-parent Automata.

Since the two values (read bottom to top in the above ordered sequence) themselves form a 2-bit number, you could kinda say that your 1-parent Automaton is characterized simply by one of the four possible 2-bit patterns: 00, 01, 10 or 11. See Table 1.

| parent | child |
|---|---|
| 0 | *p* |
| 1 | *q* |

Table 1. The generic 1-parent automaton. p and q are binary. Each "pq" defines a different 1-parent automaton

If p and q are both 0 (00), you get a particular kind of 1-parent automaton (one that always gives you all 0s no matter what the parent is). Similarly, if "pq" is "11", you get another automaton which always gives you all 1s no matter what. How many such distinct 1-parent automata can there be?

It is simply the number of distinct 2-bit strings, the values of p and q. Table 2 shows all four of them:

| Parent | Auto-A emits this child (nullifier?) | Auto-B emits this child (copier?) | Auto-C emits this child (inverter?) | Auto-D emits this child (fullifier?) |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |

Table 2. The 4 possible 1-parent automata

Obviously there can't be any other 1-parent automata than the above. But if you get 4 possible automata with just 1 parent, you can imagine how many automata you'd have with 3 or more parents. Try this (on paper) before reading on.

3 parents can come in $2^3 = 8$ combinations. (Convince yourself: 000, 001, 010, etc.) Each of those combinations can result in a child that can either be a 1 or a 0. This automaton will be a program that translates 3-bit binary strings into single bits. Its rules will consist of a table of 8 rows, which maps every possible 3-parent combination into a separate child. Reading off the child values in this table from bottom to top we'd get an 8-bit binary string. There are a total of 256 possible 8-bit binary strings (Convince yourself: 00000000, 00000001, etc.) Thus there are 256 possible 3-parent automata.

Oh, wait. We named our 1-parent automata A, B, C and D. That's not gonna work here 'cuz we have way more than Z automata. What might be a better naming strategy then? Here's one: We'll just call the automata by the n-bit number formed by reading off the child bits in reverse. That means we would have Automata 0, 1, 2, and 3 for the 1-parent case, and Automata 0 through 255 for the 3 parent case.

Care to extrapolate for more parent cases?

## Implementation detail

How do you print out infinite bit strings on finite media in finite time? Well, actually, you can't.

But we'll introduce a clever abstraction called an `_extreme_bit` which stands for an infinite sequence of the same bit. Initially, we'll say that _extreme_bit is 0, and our current_generation is simply all _extreme_bits. Then we'll drop a seed in this *infinite ocean of extreme bits*, a 1.[2]

Then at each generation, we'll find that we just have three chunks of the infinite current generation to process, and we can process them all in finite time.

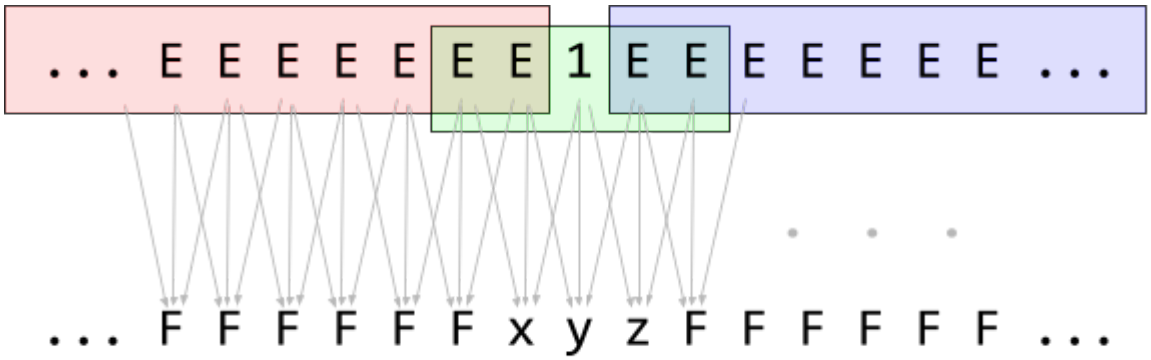



Figure 2: This 3-parent automaton is able to process an infinite bit stream in finite time by processing it in 3 chunks.

The first chunk processed by the automaton in Figure 2 (the reddish rectangle to the left) consists entirely of extreme bits (E) in the current generation (which is internally stored just as a single bit proxy in the boolean variable `_extreme_bit`). The value of the extreme bit for the next generation (F) is simply what the automaton maps 3 consecutive extreme bits (EEE) to.

The second chunk (the greenish rectangle in the middle) consists of a window that moves through the non-extreme bits. In this example, 1 is the only non-extreme bit, And so, the automaton produces values x, y and z, which are values mapped from `EE1`, `E1E` and `1EE` by the automaton.

Then the next generation can now be represented by just a 3 character string "xyz" and a single value for the new extreme bit, F.

---

[2] Here is an interesting forum discussion starter: What kind of infinite strings can you NOT represent on finite media using the extreme-bit abstraction?

To find the generation after that, use this next generation as the new current one. This is the way to propagate the seed through multiple generations. At each generation, the *interesting part* of the bit string, the part around the seed trimmed of extreme bits, grows just by a few bits on each side (1 for the 3-parent automaton, 2 for the 5 parent automaton, and so on. How many padding bits should I use for an N-parent automaton?)

When it comes time to print a generation, we will use a similar approach. That is, we will simply print its *interesting portion* padded by an equal number of extreme bits on either side for a given display width.

Hmm… I wonder which rule this snail used…

# Overview

I managed to get a sneak peek at the header file and capture a snap. Unfortunately parts of it got obscured. But I figured something is better than nothing. And you guys can easily recreate the missing bits.

```
 Automaton {
:e:
:atic const size_t MAX_PARENTS = 5;

ool _is_valid;
ize_t _num_parents; // 2^{_num_parents} = possible parent combos

ector<bool> _rules; // size() = 2^{_num_parents} rules. One for each parent combo
nt _extreme_bit;

:
utomaton(size_t num_parents, size_t rule);
ool set_rule(size_t rule);
ool equals(const Automaton& that);

ool make_next_gen(const vector<int>& current_gen, vector<int>& next_gen);
:ring get_first_n_generations(size_t n = 100, size_t width = 101);

' instance utility
:ring generation_to_string(const vector<int>& gen, size_t width);

' static utilities
:atic size_t pow_2(size_t n) { return 1 << n; }
:atic size_t translate_n_bits_starting_at(const vector<int>& bits, size_t pos, size_t

:iend class Tests; // Don't remove this line
```

Figure 1: A fuzzy photo of the Automaton class def

Here is some helpful information to go with it:

- Save yourself by setting a hard-limit. `MAX_PARENTS` is your friend. I'll only test cases for 1 and 3 parents, though you're free to try 5 on your own.
- `_is_valid` is a private internal boolean that you use to tell if your current automaton is valid. The idea is that you would set this flag on Automata that were created using rules or parent counts your other methods choose not to handle. They can simply check the value of this flag instead of querying a bunch of variables to see if they'll succeed.
- `_num_parents` is the number of parents this Automaton considers (1, 3 or 5)
- `_rules` is a vector of booleans. It should have as many elements as the number of possible parent combinations (e.g. 8 for 3-parent automata). This is where you will look up what the child should be given the values of its N parent bits.

Also, note that the *better thing to do* is to package the extreme bit with a generation, not the automaton. However, it got done the other way here for historical reasons and I figured it's better to leave it like it is as an example of *improvable structuring*. Check our sub.

And here is more detail in the following mini-quest specs.

## Your first miniquest - Utilities

Just like in Monopoly.

Get these right and you'll have an ongoing sense of security that will let you focus on the things that matter in this quest.

Implement `pow_2` inline within the header file to return a power of two as the result of a bit shift.

Note that $2^0 = 1$. If you left-shift a 1 one time, you get 2 (in binary) and so on. Return the value got by left-shifting `n` times, where `n` is the exponent. Don't invoke the Math function `pow()` to calculate small integer powers of 2 if it's a calculation that will be done many times (as in our case).

Implement (complete the code between the braces in the header file, in one line):

```
size_t Automaton::pow_2(size_t n);
```

You can expect `n` to be small (< 64) in my tests.

Now implement:

```
size_t Automaton::translate_n_bits_starting_at(
    const vector<int>& bits, size_t pos, size_t n
);
```

It should interpret the n-bit sub-vector of `bits`, specifically `bits[pos]` to `bits[pos + n - 1]` as the digits of a binary number, and return the corresponding numerical quantity. `pos` starts at 0 for the first element.

For example, if I invoke it on the bit string "101010110101" to translate the 5 bits from pos 2, it should return the quantity 0b10101, which is 21 in decimal.

Make sure you check this method for correct behavior at the edges: It must return 0 if `n` bits starting at pos extend past the end of bits.

## Your second miniquest - Generation to string

There will be a time when you want to print out a generation to the console.

Implement:

```
string generation_to_string(const vector<int>& gen,
                            size_t width);
```

It should convert the contents of the given vector into a sequence of binary values, and stitch together a string of these bits, with 0 represented by space and 1 represented by an asterisk (`*`).

Further, this string should be centered within a larger string of the given width. Since the string's length will always be odd (why?) it can only be centered within the given width if the width is also odd. Therefore this method should refuse to stringify a generation into a field of even width. If width is even, or the automaton is invalid, simply return the empty string.

If it is odd, then you can easily calculate by how much to pad (or clip) the generation's bits on either side to get a string of the required width. The question now becomes: "*With what value should we pad the generation? 0 or 1?*"

Recall that this is precisely where the extreme bit comes in. In fact, it is also the reason I had to make this method a stateful instance method rather than a stateless static helper, which I would have preferred. Can you think of a good way for me to have my cake and eat it too?

## Your third miniquest - Set Rule

Implement:

```
bool Automaton::set_rule(size_t rule);
```

You can assume that when this method is invoked, the `_num_parents` member of the current object will be set. However, you need to check its value. If it is greater than permitted by the MAX_PARENTS value in the spec, you should return `false` immediately.

Furthermore, you should also return `false` if the value is greater than the maximum permitted by the `_num_parents` value (what might that be?). These cases result in invalid automata.

Otherwise, you must fill in the `_rules` vector of the current object with the appropriate values for the given rule and return `true`. You can also assume that the vector would already have been sized correctly in the constructor (when the number of parents was set).

Before returning, reset the `_is_valid` flag to true if you have succeeded in setting this rule correctly.

## Your fourth miniquest - Constructor

This one is going to be easy if you've got `set_rule()` right.

Implement:

```
Automaton::Automaton(size_t num_parents, size_t rule);
```

Since we aren't throwing exceptions when the constructor fails, we'll do the next best thing. Remember how every Automaton has a boolean member called `_is_valid`? If either the requested number of parents is illegal or the rule failed to set (you can invoke `set_rule()` from your constructor), you must set `_is_valid` to `false` and return immediately.

## Your fifth miniquest - Equals

Implement:

```
bool Automaton::equals(const Automaton& that);
```

It should return true if the `*this` Automaton is equal to `that`. Equality is defined thus: Two Automata are equal if either of the following conditions is true. They are inequal otherwise.

1. They are both invalid
2. They are both valid AND have the same number of parents, the same extreme bit and the same rules.

# Your sixth miniquest - Make next gen

This is the biggie. Implement:

```
bool Automaton::make_next_gen(const vector<int> &current_gen,
                              vector<int> &next_gen);
```

This method should accept a const vector of ints (by reference), which it will use as the current generation. Then, using the rules in the current automaton (`this`), it should proceed to stitch together the next generation in place (in `next_gen`).

Interpret the vector as the *interesting portion* of the infinite string of bits of the current generation. Likewise, your returned next_gen should also be just the interesting portion. Make sure that you update the value of the extreme bit in the automaton after processing a generation.

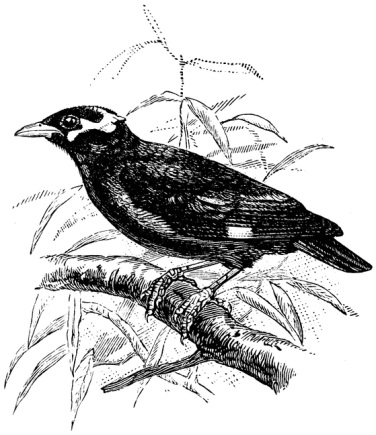
Importantly, this method must return false if either the automaton it is invoked on is marked invalid, or if the length of the current_gen is an even number, with the exception of 0.

If the length of the current generation is 0, make the next generation the *seed*, which is a vector with a single element = 1. That is, `{1}`.

Here is an interesting forum discussion starter for you:

It is obviously possible to maintain state within the Automaton by storing the *current generation* as a member. In this implementation, I decided to reduce the *statefulness* of this class and move the generation into the hands of the client (the code that constructs an Automaton).

Did I achieve my goal of statelessness? If I didn't, discuss what more is left to do and some elegant ways to do it.



# Your seventh miniquest - Get first n gens

Implement

```
string Automaton::get_first_n_generations(size_t n, size_t width);
```

It should return a string representation of the automaton. `n` is the number of generations to draw and `width` is the number of characters to use for each line of output. Return the empty string for even `width` (including 0) or invalid automatons.

Use newlines in this string to break it up into a number of separate equal width lines. For example, the string

```
"    *    \n   ***   \n  * * *  \n ** * ** \n*   *   *\n"
```

represents the first 5 generations (0-4) of an Automaton(150), drawn on a canvas 9 characters wide:

```
    *
   ***
  * * *
 ** * **
*   *   *
```

## How long is a well-written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 250 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

# Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your source files (excluding your main function) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

## Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

Happy hacking,

&