To understand Axios, one must first understand its core data structure. Axios operates on a list of cells that grow over time. Each cell can have two values, zero or one. (The implementation in this repository uses bits to represent cells, but Boolean values also suffice.)

At the beginning of an Axios program, the list contains one cell with a value of zero:

```
[0]
```

The brackets indicate the pointer lies at this point in the list.

Axios also divides code into states, much like a Turing machine. At the beginning of each state, the value of the cell under the pointer is changed (unless user input is accepted). In fact, unless the very first state asks for user input, the first cell is always changed to one at the beginning of the program.

In addition to always changing the value at the current cell, the program contains instructions for whether each state moves the pointer down the list or leaves it where it is. If the pointer is already located at the end of the list, a new cell of value zero is appended to the end of the list, and the pointer returns to the beginning.

Finally, the program marks the location of a possible next state. This custom next state is only accepted if the value of the cell is changed to one. If the value of the cell is changed to zero, the program instead shifts to the next state as written in code (which will make more sense shortly).

In essence, each state performs the following functions:

- To always change the value at the current cell (unless user input is accepted)
- To move the pointer down the list OR leave it where it is (depending on the programmer's instructions)
- To go to another state:
    - If zero, go to the next state as organized in the program
    - If one, go to a specific custom state noted by the programmer

As an example, consider a program with seven states in the following order. This example explains every step in detail and is quite verbose. Feel free to skim past it according to your understanding of the concepts.

State A	
Move pointer	If cell is changed to one, go to State B

State B	
Move pointer	If cell is changed to one, go to State C

State C	
Move pointer	If cell is changed to one, go to State D

State D	
Leave pointer	If cell is changed to one, repeat State D

State E	
Move pointer	If cell is changed to one, go to State F

State F	
Move pointer	If cell is changed to one, go to State G

State G	
Leave pointer	If cell is changed to one, go to State F

The list starts with one cell whose value is zero:

```
[0]
```

State A changes the current value of the cell and moves the pointer. Since the list only has one element, the pointer is already at the end of the list. Thus, a new cell is added to the end, and the pointer remains at the beginning of the list. Since the cell has been changed to one, the next state is State B. Here is our list below:

```
[1] 0
```

State B also changes the current value of the cell and moves the pointer. The value of the cell (before moving) becomes zero, so the next state is actually the next state in order, State C. (In this case, State B transitions into State C regardless of the value of the cell.) Here is how our list looks now:

```
 0 [0]
```

State C changes the current cell and moves the pointer. As with State A, the pointer is already at the end of the list, so a new cell is added to the end, and the pointer returns to the beginning of the list. Here is our list now:

```
[0] 1  0
```

State D is different from the other states so far. It does not move the pointer, and calls itself whenever the current cell is changed to one. Thus, it changes the current cell to one, and the list appears as follows:

```
[1] 1  0
```

Back at State D, the cell then changes to zero. Thus, instead of repeating State D again, we go to the next state in order, State E. Here is our list:

```
[0] 1  0
```

State E is identical to A, B, and C. The current value of the cell changes to one and the pointer moves down the list. State E always transitions into State F. Here is our list:

```
 1 [1] 0
```

Likewise, State F changes the current value of the cell and moves the pointer down the list. State F always transitions into State G. Here is our list:

```
 1  0 [0]
```

State G changes the current value of the cell and does not move the pointer. If the cell is changed to one (as it is here), we go back to State F.

```
 1  0 [1]
```

State F changes the current cell and moves the pointer. It lies at the end of the list, so the pointer returns to the beginning and a new cell is added.

```
[1] 0  0  0
```

State G changes the current cell to 0. In this case, it does not go back to State F. In Axios, the last state is the termination state. Therefore, continuing in order from State G (the last *written* state), the program terminates. Here is our list at the program's end.

```
[0] 0  0  0
```

This example illustrates how Axios fundamentally operates.

As described in the README file, Axios has two core operations, the 0 and 1 operators. (Again, these are not to be confused with the cells whose values can be zero or one.) Their functions are as follows:

1 operators divide the program into states. So before and after each 1 is a separate state. There is also another, unwritten state after all the others that terminates the program.

0 operators perform two functions:

- The presence of any 0 operators in a state indicates the pointer does not move. The absence of a 0 operator indicates the pointer does move.
- The number of 0 operators indicates the next state. By default (i.e. no 0 operators), the next state is the next one written in code. Otherwise:
    - If the current cell is one (after changing it), each 0 operator indicates that the next state moves back one. So one 0 repeats the current state written in code, two 0s goes to the state immediately before the current state as written in code, etc.
    - If the current cell is zero (after changing it), continues as normal to the next state written in code.
    - If the number of 0 operators go past the beginning of the code, this process wraps around the end of the program. Thus, it is possible to access ANY state (including the termination state) with 0 operators, assuming the current cell is one.

Note that none of these operators need to be Western Arabic, ASCII characters. The implementation in this repository supports Eastern Arabic, Devanagari, Persian, Bengali, Tamil, Thai, Lao, Tibetan, Burmese, Khmer, and fullwidth numerals as well.

Let’s go back to the example with the seven states (plus the unwritten termination state). In Axios, it would be represented with the following code:

```
111011100
```

A state exists to the right and left of each 1 operator, totalling up to seven. Here, each state is separated by a space:

```
 1 1 1 01 1 1 00
```

Here is how this translates:

- The absence of a zero for the first three states indicates the pointer moves.
- In the fourth state, the pointer does not move, and the 0 operator indicates that, if the current cell changes to one, the next state is one behind what it would otherwise be. In other words, the fourth state repeats until the cell’s value is zero.
- The fifth and sixth state move the pointer forward and continue to the next state written down.
- The two 0 operators in the seventh state indicate the next state is two behind what it would otherwise be. In other words, if the current cell is one, we return to the sixth state. Since this is the last written state, if the current cell is zero, the program terminates.

In addition, because 0 operators wrap around, one should note that the following programs are equivalent:

```
111011100
```

```
1110000000001110000000000
```

```
11100000000000000000111000000000000000000
```

Keep in mind, it takes eight 0 operators to wrap all the way around, because it is also possible to access the termination state (and thus terminate the program) with 0 operators.

Also note that whenever the pointer moves, the next state is always the next one written in code, due to how 0 operators work.

Finally, here is how I/O works in Axios, so you can see actual results from your program.

Output relies on the use of the 2 operator. Each 2 operator within a state outputs a single bit of a UTF-32 character, based on the value of the current cell. Although UTF-32 contains thirty-two bits, Axios only requires twenty-one bits be output, because the rest must always be zero. Thus, every twenty-one 2 operators outputs a UTF-32 character.

There is one exception. Outputting twenty-one bits with the value one does NOT output a character (no valid Unicode character is associated). Instead, it clears the input queue, which has more to do with how the 3 operator works.

There is no visually pleasing way to show how the 2 operator works, so if you would like to see for yourself, check out the “Hello, world!” program in the Example Programs folder.

Each 3 operator changes the current cell according to a single bit from the input queue, and then it removes the front of the queue. When the queue is empty (as it is at the beginning of the program) the program requests the user input more characters into the queue. This includes the new-line character associated with each user input.

Characters are added to the queue according to their UTF-32 encodings. (If necessary, any implementation should convert to UTF-32.) Only the first twenty-one bits are used per character (in little-endian order), as the rest are trivially zero.

It is possible to empty the input queue by attempting to output twenty-one one values via the 2 operator.

As a technical detail, there is one more important fact to know about operators. Every state performs the following functions in order:

- Change the current bit OR input a bit (depending on the presence of a 3 operator)
- Output the current bit (associated with the 2 operator)
- Decide whether to move the pointer
- Go to the next state, often depending on the value of the current cell

And that is everything there is to know about Axios! Feel free to let me know if you have any questions.