(Th)ink Machine

Notes

- ✉ You don't have to do the projects in order! Do whatever seems fun at the moment.
- ✉ You don't have to do the projects at all! They're just here to inspire you. If you think of something cool, run with it—and please tell me about it, because I love cool things.

Key

- ✉ Here's a thing you should know.
- ☞ Write a program to do this.
- ❒ Can you write a program to do this?
- ☕ Think about this.

Arithmetic

- ✉ Representing a number by a tally means representing it like this:

1	`o`
2	`oo`
3	`ooo`
4	`oooo`

Of course, any symbol will work just as well as `o`.

Products

☞ IN Two numbers (as tallies)
OUT The product of the numbers (as a tally)

☞ IN A number (as a tally)
OUT The square of the number (as a tally)

☞ IN A number n (as a tally)
OUT The sum of the numbers from 1 to n (as a tally)

Remainders

☞ IN A number (as a tally)
OUT The number's remainder when divided by five (as a tally)

☞ IN Two numbers (as tallies)
OUT The first number's remainder when divided by the second (as a tally)

The *Euclidean algorithm* is a method for finding the greatest common divisor of two numbers. It works like this.

Start with a pair of numbers. Subtract the smaller number from the larger number (if the numbers are equal, subtract either one from the other). Now you have a new pair of numbers. Keep subtracting the smaller from the larger until one of the numbers is gone. The remaining number is the greatest common divisor of the numbers you started with.

Write a program that finds the greatest common divisor of two numbers (represented as tallies).

Write a program that takes in a number n (as a tally) and a string of bars, and turns every nth bar into a dot.

IN	A number (as a tally)
OUT	The empty string if the number is prime
	A non-empty string otherwise

Sequences

... of symbols

The *Fibonacci strings* are the strings $F_0, F_1, F_2, \dots$ in the alphabet -, |, built up recursively according to the following rules:

$$F_0 = \texttt{-}$$
$$F_1 = \texttt{|-}$$
$$F_{n+1} = F_n F_{n-1}$$

The first few look like this:

n	F_n
0	-
1	-|
2	-|-
3	-|--|
4	-|--|-|-
5	-|--|-|--|--|

Just as this table might lead you to suspect, each Fibonacci string is the beginning of the next one. That means there's an infinite string F_∞ that each Fibonacci string is the beginning of.

You can turn F_n into F_{n+1} using the substitution

$$\texttt{-} \mapsto \texttt{-|}$$
$$\texttt{|} \mapsto \texttt{-}$$

Doing this substitution on F_∞ gives you back F_∞.

Write a program that shows you longer and longer pieces of the infinite Fibonacci string as it runs.

Take a strip of paper and fold it in half over and over, always folding in the same direction. Then, open each crease out into a 90° corner, keeping the natural direction of the crease. The shape you end up with is called a *dragon curve*.

Folding paper over and over gets difficult really quickly. It would be much easier to fold a dragon curve if you knew the sequence of left and right folds ahead of time—then you'd only have to fold one layer of paper.

Write a computer that will show you all the folding sequences for dragon curves.

… of numbers

- Write a program that will show you all the numbers (as tallies), one by one.
- Write a program that will show you all the Fibonacci numbers (as tallies), one by one.
- Write a program that will show you all the prime numbers (as tallies), one by one.
- Write a computer that churns out the never-ending string

```
o_oo_ooo_oooo_ooooo_oooooo_  ...
```

(the ellipsis isn't part of the string).

Coding

Digits

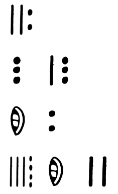
- IN A number (in binary)
  OUT The number (as a tally)
- IN A number (as a tally)
  OUT The number (in binary)
- IN A number (in binary)
  OUT The number (in trinary)
- In the Maya number system, numbers up to nineteen are written as tallies, like this:

0	𝋠	1	𝋡	2	𝋢	3	𝋣	4	𝋤
5	𝋥	6	𝋦	7	𝋧	8	𝋨	9	𝋩
10	𝋪	11	𝋫	12	𝋬	13	𝋭	14	𝋮
15	𝋯	16	𝋰	17	𝋱	18	𝋲	19	𝋳

Larger numbers are written in base twenty, using the tallies as digits:

20×2	𝋢	20×2	𝋢	20×8	𝋨
1×0	𝋠	1×5	𝋥	1×3	𝋣
40		**45**		**163**	
$20^2 \times 10$	𝋪	$20^2 \times 16$	𝋰	$20^2 \times 4$	𝋤
20×0	𝋠	20×8	𝋨	20×6	𝋦
1×19	𝋳	1×1	𝋡	1×9	𝋩
4019		**6561**		**1729**	

Tipping Maya numerals on their sides gives a nice way to write them as strings:



```
||..
...~|...
@~..
|||....~@~||
```

Write a program that turns numbers (in whatever form you want) into Maya numerals.

Parsing

IN	A string of parentheses
OUT	The empty string if the parentheses are balanced
	A non-empty string otherwise

IN	A boolean expression—a tree with "and" or "or" at each node, and "true" or "false" at each leaf
OUT	The value of the expression

Computation

In this section, we'll be talking about lots of different kinds of programs, so I'll refer to the ones we've been writing as "Markov algorithms" instead of just "programs."

Computer architecture

Given a Markov algorithm that does computation A, and another Markov algorithm that does computation B, can you systematically build a Markov algorithm that does computation A and uses the output as the input for computation B?

Changing the rules

Given any Markov algorithm, can you systematically build a Markov algorithm that does the same computation using only the symbols `0` and `1`?[1]

What does "the same computation" mean, anyway?

For that matter, what does it mean to "systematically build" something?

A "vertically nondeterministic" Markov algorithm doesn't have to go straight down the list of rules, applying the first one that can be applied. It applies the rules in any order it wants to!

Given any ordinary Markov algorithm, can you systematically build a vertically nondeterministic Markov algorithm that does the same computation?

Given any vertically nondeterministic Markov algorithm, can you systematically build an ordinary Markov algorithm that does the same computation?

A "horizontally nondeterministic" Markov algorithm doesn't have to go through the string it's working on from left to right, applying each rule in the first place it can be applied. It applies the rules anywhere it wants to!

Languages

Given a regular language, can you write a Markov algorithm that recognizes the language?

IN	A string
OUT	The empty string if the string is in the language
	A non-empty string otherwise

If the answer to the previous question is "yes," can you automate the process?

IN	A description of a regular language
OUT	A description of a Markov algorithm that recognizes the language

Can you come up with a language that can't be recognized by any Markov algorithm?

[1] What if you could use the symbol `2` as well? If your answer looks the same, you may be missing something important...

Halting

- Pick a number. If it's even, divide it by two. If it's odd, multiply it by three and add one. Now you have a new number. Keep doing this until you reach one.

  Pretty tedious, isn't it? Write a program to do it for you!

- Can you write a program that looks at the rules and input of any program and tells you whether the program will eventually stop running when it's given the specified input?

Emulation

- FRACTRAN is a programming language invented by John Conway. Instead of working on strings of symbols, a FRACTRAN program works on positive integers.

  A FRACTRAN program is described by a list of positive rational numbers. To run a step of the program, go through the list in order, looking at the product of each rational on the list with the integer you're working on. If you get an integer product, replace the integer you're working on with that, and go back to the beginning of the list. If you never get an integer product, you're done!

  Given any FRACTRAN program, can you systematically build a Markov algorithm that does the same computation?

  Given any Markov algorithm, can you systematically build a FRACTRAN program that does the same computation?

- A *finite state machine* is a simple creature that eats up a string of symbols and then tells you something about it. The machine has a finite set of moods, or "states," that it can be in. When you feed the machine a symbol, its mood can change—it moves to a new state, which is determined by its current state and the symbol you gave it. If you find the machine in a certain state, and then you feed it a string of symbols, the state the machine ends up in after it's eaten the whole string tells you something about the string.

  Given a finite state machine, can you systematically build a Markov algorithm that does the same computation? What about vice versa?

- A finite state machine doesn't have a very good memory; its current state is all it knows about the past. Let's help it out by giving it a place to write things down. We'll make it a notebook with an infinite number of pages, so it can turn them to the right or to the left as many times as it wants without running out.

  Each page of the notebook can hold one symbol; for simplicity, we'll say a blank page holds "the blank symbol." Instead of having us feed it symbols, the machine eats symbols off the pages of the notebook. When the finite state machine eats a symbol, it doesn't just go to a new state; it can also write a symbol on the page it's currently looking at, and then it can turn the page in either direction. Like before, what the machine does is determied by its current state and the symbol it just ate.

  A finite state machine with a notebook like this is called a *Turing machine*. Given a Turing machine, can you systematically build a Markov algorithm that does the same computation? What about vice versa?

- Can you write a Markov algorithm that looks at the rules and input of a finite state machine and tells you what the machine will do when given the specified input?

- Can you write a Markov algorithm that looks at the rules and input of a Turing machine and tells you what the machine will do when given the specified input?

- Can you write a Markov algorithm that looks at the rules and input of a Markov algorithm and tells you what the algorithm will do when given the specified input?

Group theory

✉ Let's say you have a generators-and-relations presentation for a group. A string of generators and inverse generators is called a "word." It's often very hard to tell whether two words represent the same group element just by looking at them. (For example, can you prove that the word

$$aba^{-1}b$$

in the group

$$\langle a, b \mid ab^2a^{-1}b^{-3}, ba^2b^{-1}a^{-3} \rangle$$

represents the identity?)

To figure out whether two words represent the same group element, it's enough to be able to figure out whether a given word represents the identity. I'll call a program that does this an "identity recognizer." To keep things focused, let's say an identity recognizer has to act like this:

IN	A word
OUT	The empty string if the word represents the identity
	A non-empty string otherwise

➪ The *dihedral group* D_{2n} is the symmetry group of an n-sided polygon. It can be presented with generators and relations as

$$\langle r, s \mid r^n, s^2, (rs)^2 \rangle,$$

where s flips the polygon and r rotates it by one step.

Pick an n and write an identity recognizer for D_{2n}.

➪ The *Heisenberg group* $H_1(\mathbb{Z}/N\mathbb{Z})$ shows up in physics as the symmetry group of a "quantum frog" hopping around on a circle of lily pads. It can be presented with generators and relations as

$$\langle u, v, q \mid u^N, v^N, q^N, uqu^{-1}q^{-1}, vqv^{-1}q^{-1}, uvu^{-1}v^{-1}q^{-1} \rangle,$$

where u changes the frog's position by one step, v changes the frog's momentum by one step, and q is there for, uh... technical reasons.[2]

Pick an N and write an identity recognizer for $H_1(\mathbb{Z}/N\mathbb{Z})$.

➪ The *triangle group* $D(p, q, r)$ is the orientation-preserving symmetry group of a sphere, plane, or hyperbolic plane tiled by triangles with angles $\frac{\tau}{2p}$, $\frac{\tau}{2q}$, and $\frac{\tau}{2r}$. It can be presented with generators and relations as

$$\langle x, y \mid x^p, y^q, (xy)^r \rangle,$$

where x rotates by $\frac{\tau}{p}$ around the $\frac{\tau}{2p}$ vertex, y rotates by $\frac{\tau}{q}$ around the $\frac{\tau}{2q}$ vertex, and xy rotates by $\frac{\tau}{r}$ around the $\frac{\tau}{2r}$ vertex.

Pick some p, q, and r and write an identity recognizer for $D(p, q, r)$. Don't set q and r to two—that's cheating! Also, don't spend more than ten minutes on this project. Ask me for the answer and you'll find out why...

☕ Using the relations of a group to rewrite a word is a lot like using the relations of a Markov algorithm to rewrite a string, isn't it? Can you make this analogy more precise?

🕮 Can you write a computer that looks at a group (presented with generators and relations) and a word (in the given generators) and tells you whether the word represents the identity?

[2]If you really want to know, see "Central Extensions in Physics," by G.M. Tuynman and W.A.J.J. Wiegerinck.