CDM

Turing Machines

Klaus Sutner

Carnegie Mellon University

1 **Turing Machines**

2 **Universality**

3 **Wolfram Prize**

So far, we have one rather powerful notion of computability: primitive recursive functions–obtained essentially by recursion from successor.

But are all intuitively computable functions already primitive recursive?

We need a more careful analysis of what "intuitively computable" could possibly mean in a more technical sense.

Alan Turing (1912–1954)

There are two requirements:

- Our definition should not go beyond what a human computor could do (in principle), and
- it should capture everything a human computor can do.

Note that there won't be a theorem: the abilities of a human computor are certainly subject to discussion–though, at this point, only cranks will raise objections to the standard model.

Also note that we want to ignore "merely physical" constraints such as compute time and memory consumption, we are interested in logical constraints only at this point.

Brilliant Idea: Observe a human computor, then abstract away all the merely biological stuff and formalize what is left.

Everyone agrees that mathematicians compute (among other things such as drinking coffee or proving theorems). So we could try to define an abstract machine that can perform any calculation whatsoever that could be performed in principle by a mathematician, and only those. To this end we need to formalize what a computor is doing.

Mathematicians tend to write down their calculations on paper. This amounts simply to bookkeeping of all intermediate results.

We need to formalize what can be written down and the rules that control the process of writing things down, as well as reading off already existing information.

Aside: Turing had terrible handwriting as a child, he was always interested in typewriters. Detractors would say that his machines are just glorified typewriters.

- Mathematicians use two-dimensional notation, but it could all be flattened out into one-dimensional notation.
- Only finitely many symbols are allowed.
- Can think of having a strip of paper subdivided into cells, each cell containing exactly one symbol (possibly the blank symbol).

This is entirely obvious to anyone who has every used a keyboard. Instead of

$$\int_0^1 \frac{1}{2+x^2}\,dx = \frac{\arctan(\frac{1}{\sqrt{2}})}{\sqrt{2}}$$

one types something like

```
Integrate[1/(2 + x^2), {x,0,1}] ==
   ArcTan[1/Sqrt[2]]/Sqrt[2]
```

- The finite input is written on an otherwise empty tape.
- The computor's mind has finitely many possible states and is always in one particular of these states (a slight stretch, but not unreasonable).
- At each step, the computor focuses on one particular cell, the current cell.
- The computor reads the symbol in the current cell (one symbol from a fixed, finite collection of possible symbols) and then takes action depending on her own state of mind and the symbol just read.
  - She may overwrite the symbol in the current cell.
  - She may shift attention to the cell on the left or right.
  - She may switch into a new state of mind.

- Initially, the computor is in a particular state “start” and looks at the first empty cell to the left of the input.
- The computor repeats the basic read-write-shift-change operation until a particular state of mind “finished” is reached.
- What is written on the tape at this point constitutes the result of the computation.

We assume that only finitely many steps lead from “start” to “finished”, so the output is always finite.

These assumptions are quite robust.

- Infinitely many symbols make no sense, since it would take the computor infinitely long to learn the meaning of all these symbols.
- Quantum physics suggests that the computor's mind can assume only finitely many internal configurations (though that number is probably quite large in a human being).
- A computor cannot simultaneously pay attention to infinitely many symbols, only a finite number (that number is apparently quite large in a human being, in the single digits). This is essentially the same as a single symbol.

- The computer can only shift at a bounded distance, which is equivalent to just shifting by a single place.
- All but finitely many cells will always contain a special blank symbol. Otherwise someone would already have had to spend an infinite amount of time writing down the initial tape contents.

The last condition is a bit harsh, a real number input for example would generally require an infinite input.

There is a major conceptual problem hiding here, for the time being we will insist on simple, finite initial tape inscription,

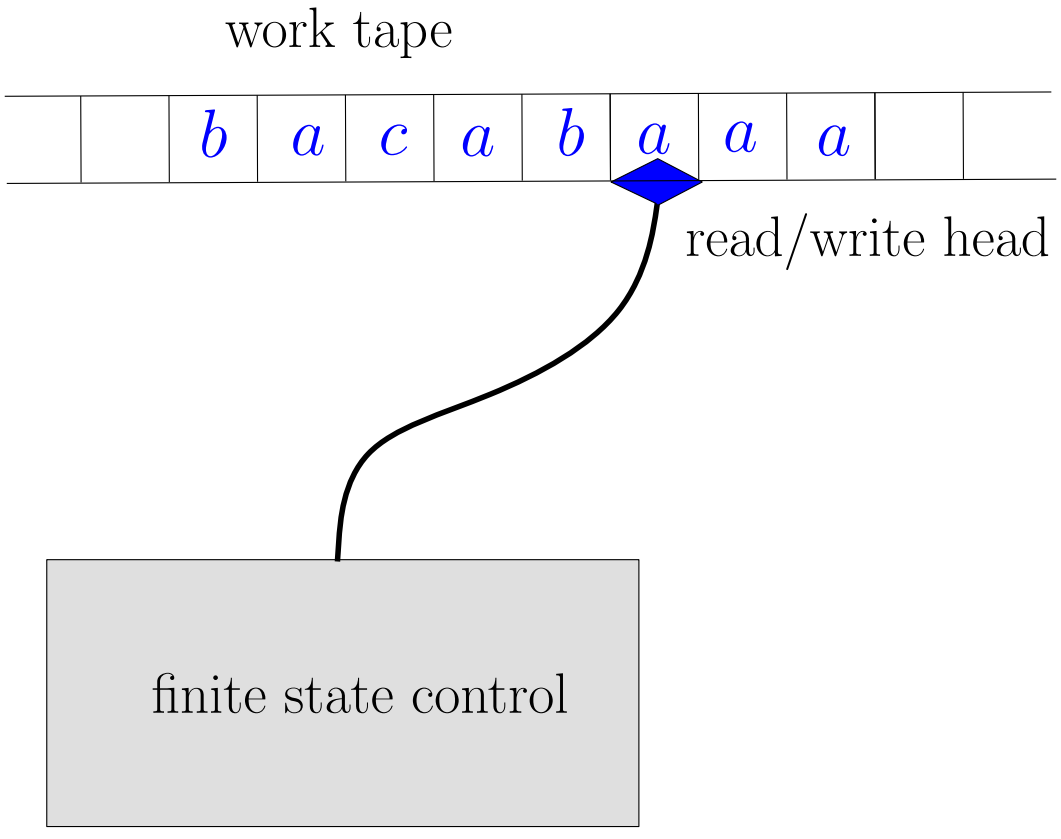
Also note that insisting on blanks-only surrounding the input is just a convenience. For example, an inscription of the form

$$\ldots 0000100001000001x_1x_2\ldots x_{n-1}x_n1000010000100001\ldots$$

is surely harmless.

For Turing machines, this makes no difference whatsoever, but there are other models of computation such as cellular automata where the more general starting configurations are of great interest.

A Turing Machine



- A tape: a bi-infinite strip of paper, subdivided into cells. Each cell contains a single letter; all but finitely many contain just a blank. So we have a tape inscription.
- A read/write head that is positioned at a particular cell. That head can move left and right.
- A finite state control that directs the head: symbols are read and written, the head is moved around and the internal state of the FSC changes.

- alphabet Σ: finite set of allowed symbols, special blank symbol $\textvisiblespace \in \Sigma$
- state set Q: finite set of possible mind configurations
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$: transition function
- a special initial state $q_{\text{init}} \in Q$
- a special halting state $q_{\text{halt}} \in Q$.

Definition

A Turing machine is a structure $M = \langle Q, \Sigma, \delta, q_{\text{init}}, q_{\text{halt}} \rangle$.

Lame Example: Successor Function

Tape alphabet $\Sigma = \{␣, 1\}$

States $Q = \{0, 1, 2, 3\}$

Initial state $q_{\text{init}} = 0$, final state $q_{\text{halt}} = 3$.

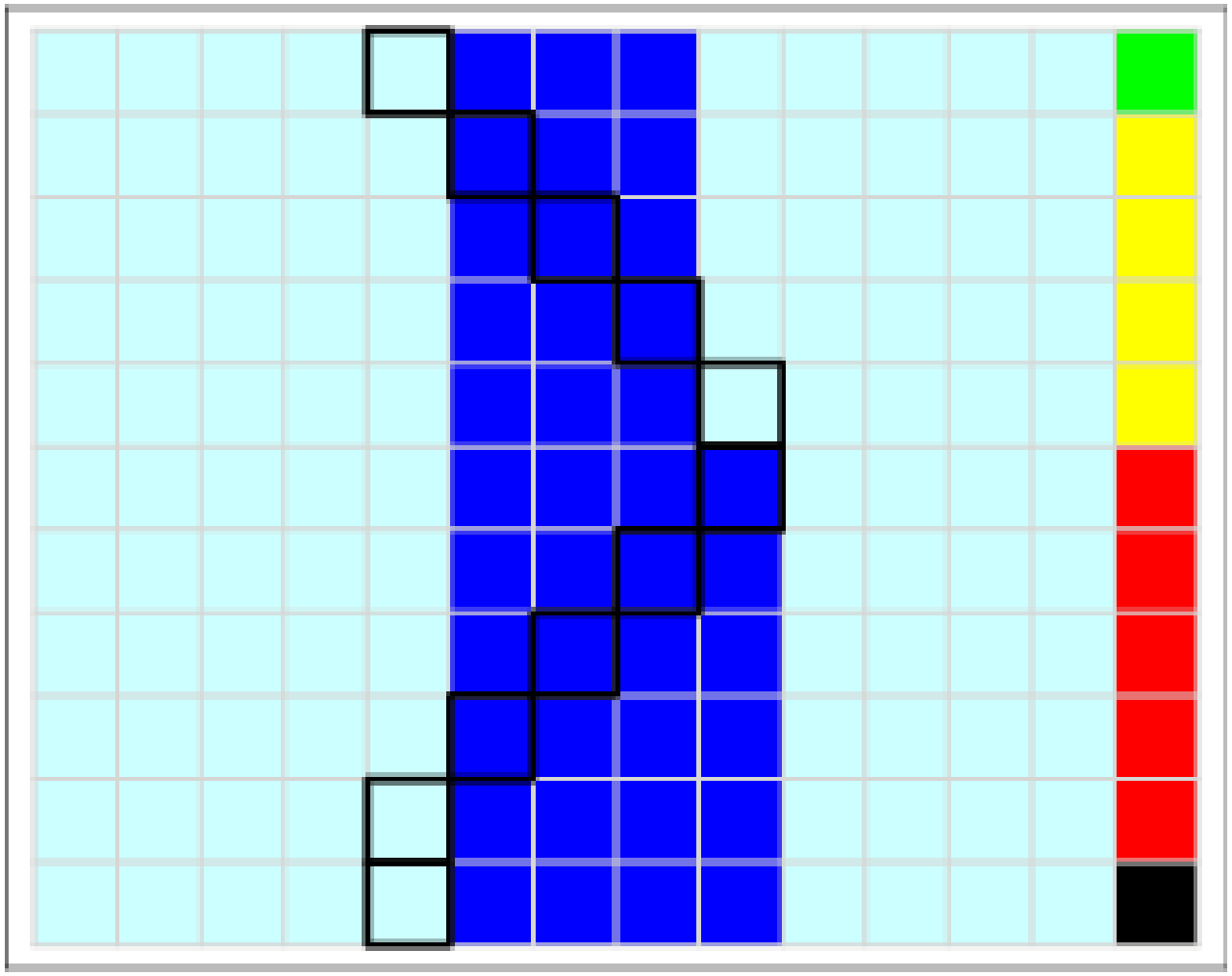
The transition function δ is given be the following table:

p	σ	$\delta(p, \sigma)$		
0	␣	1	␣	$+1$
1	␣	2	1	0
1	1	1	1	1
2	␣	3	␣	0
2	1	2	1	-1

No other transitions are needed, so we cheat a little and use a partial transition function.

It is entirely straightforward to write a program that simulates a Turing machine (at least if the machine is small and the computation short). Hence, we can perform computational experiments on these machines. The following visualization method is sometimes useful to understand behavior of small Turing machines. In the pictures:

- The tape is represented by a row of colored squares.
- Time flows from top to bottom (usually; sometimes from left to right).
- The position of the tape head is indicated by a black frame around a square.
- The rightmost column indicates the state of the machine.



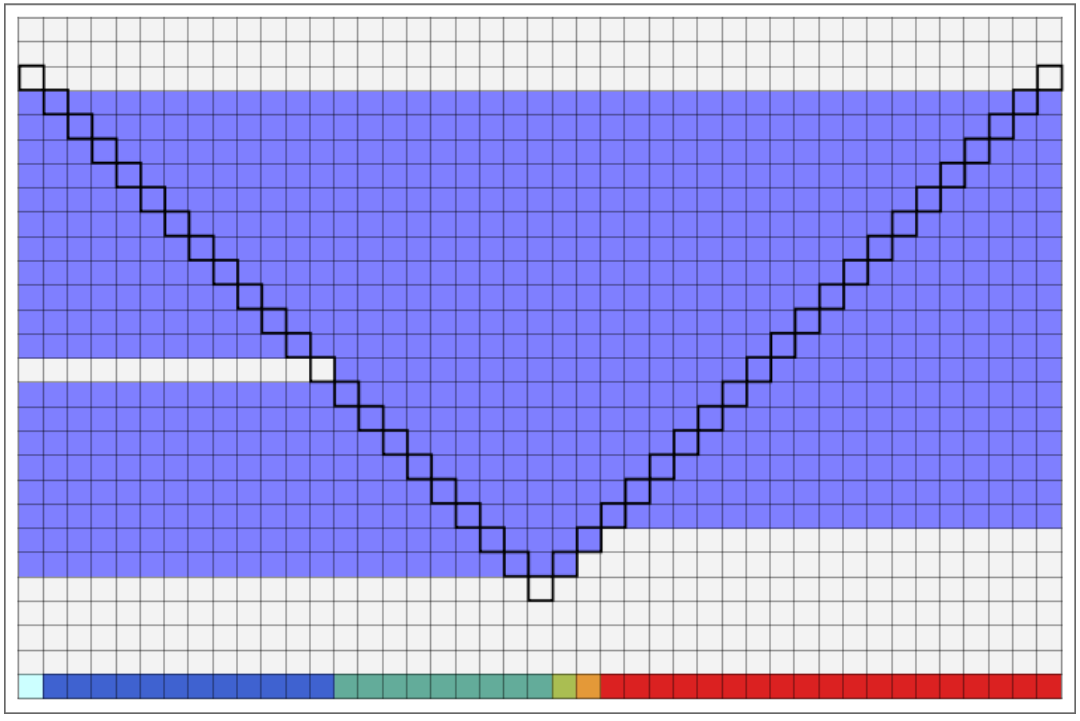
We are using unary notation, n is represented by

$$n \mapsto \underbrace{111\ldots11}_{n+1}$$

Hence we have to erase two 1's at the end:

0	$\sqcup$	1	$\sqcup$	1
1	$\sqcup$	2	1	1
1	1	1	1	1
2	$\sqcup$	3	$\sqcup$	-1
2	1	2	1	1
3	1	4	$\sqcup$	-1
4	1	5	$\sqcup$	-1
5	$\sqcup$	6	$\sqcup$	0
5	1	5	1	-1

0 is the initial state, 6 is the final state



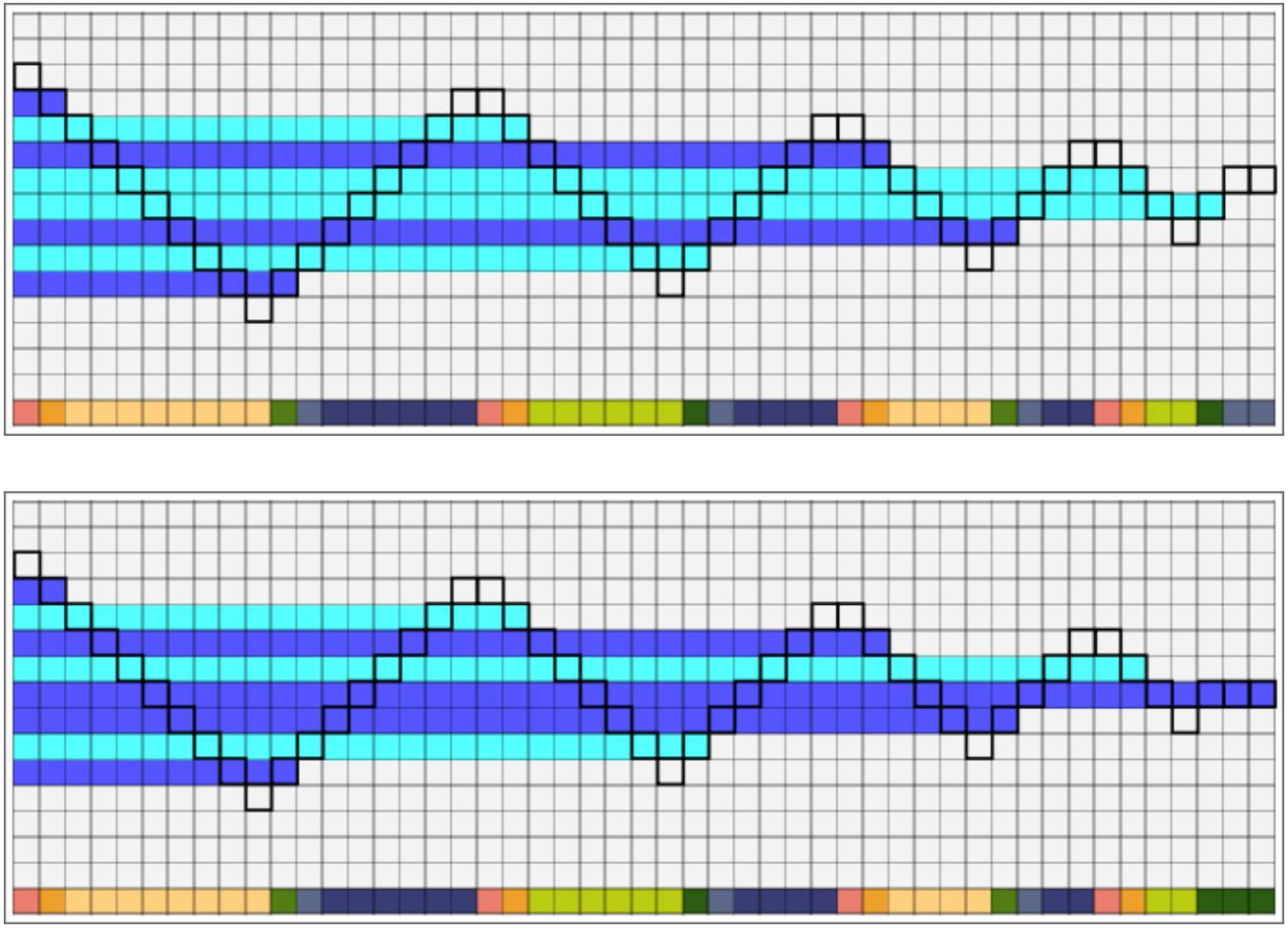
Note that for very simple tasks such as addition in unary one can almost read off a correctness proof from these pictures.

Exercise

What would a complete correctness proof for the Turing machine that performs unary addition look like? What is difficult about the proof?

Exercise

Construct a Turing machine that performs addition when the input is given in binary. What should be pictures look like in this case? How hard is a correctness proof?



The first picture represents a successful computation, and the second an unsuccessful one.

Exercise

Figure out how this machine works and prove that it is correct (you need a convention for accepting versus rejecting computations).

Exercise (Hard)

Show that any one-tape Turing machine requires quadratic time to recognize palindromes.

We need a real definition of what it means for a TM to compute. First we describe a snapshot during a computation consisting of:

- the current tape inscription (non-blank part only)
- the current head position
- the current state

For the following, assume for simplicity that $Q \cap \Sigma = \emptyset$.

Definition

A configuration or instantaneous description (ID) is a word $b\,p\,a$ where $a, b \in \Sigma^\star$ and $p \in Q$:

$$b_m b_{m-1} \ldots b_1\, p\, a_1 a_2 \ldots a_n$$

means that the read/write head is positioned at a_1 and the whole tape inscription is $b_m \ldots b_1 a_1 \ldots a_n$.

Next we need to explain a single step in a computation:

$$bpa \;\left|\frac{1}{M}\right.\; b'qa'$$

Let $\delta(p, a_1) = (q, c, \Delta)$ Then the next configuration is defined by

$$\begin{aligned}
b_m \ldots b_1\, p\, a_1 a_2 \ldots a_n \quad &\rightsquigarrow \\
b_m \ldots b_2\, q\, b_1 c a_2 \ldots a_n \quad &\Delta = -1 \\
b_m \ldots b_1\, q\, c a_2 \ldots a_n \quad &\Delta = 0 \\
b_m \ldots b_1 c\, q\, a_2 \ldots a_n \quad &\Delta = +1
\end{aligned}$$

Here we assume that a and b are non-empty, otherwise we can set $b_1 = a_1 = \sqcup$.

Now we extend the "one-step" relation to multiple steps by iteration:

- $C \left|\frac{1}{M}\right. C'$: one step
- $C \left|\frac{t}{M}\right. C'$: exactly t steps
- $C \left|\frac{}{M}\right. C'$: any finite number of steps

The last notion is problematic: there is no bound on the number of steps. Note, though, that for feasible computations this is never an issue: we know ahead of time how long a computation can possibly take.

Given any input $x = x_1 x_2 \ldots x_n \in \Sigma^\star$, the initial configuration for x is

$$C_x = q_{\text{init}} \sqcup x_1 x_2 \ldots x_n$$

A final configuration is of the form

$$C_y^H = q_{\text{halt}} \sqcup y_1 y_2 \ldots y_m$$

For the initial configuration, we have chosen to place the head at the last blank to the left of the input symbol, there are lots of other possibilities (e.g, $q_{\text{init}} x_1 \dots x_n$).

Also, we require our machines to erase the tape (except for the output) and return the tape head to the initial position before halting; nothing important changes if we drop this condition (and it kills any chance of reversible computation).

If $C_x \vdash_M C_y^H$ then $y = y_1 y_2 \dots y_m \in \Sigma^\star$ is the output of the computation of machine M on input x.

M computes the partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ if, for all $\boldsymbol{x} \in \mathbb{N}^k$,

- If f is defined on $\boldsymbol{x}$ then $C_x \vdash_M C_y^H$ and $f(\boldsymbol{x}) = y$.
- If f is undefined on $\boldsymbol{x}$ then the computation of M on x does not halt.

Here $\boldsymbol{x}$ is a k-tuple of natural numbers and $x \in \Sigma^\star$ is the corresponding tape-inscription (say, write the numbers in binary and separate them by blanks).

Note that a Turing machine may very well fail to reach the halting state q_{halt} during a computation.

For example, the machine could ignore its input and just move the head to right at each step.

Or it could go into an "infinite loop" where the same sequence of configurations repeats over and over again.

As it turns out, this is not a bug, but a feature: we will see shortly that failure to halt is a problem that cannot be eliminated.

There are restricted types of Turing machines that a guaranteed to halt on all inputs, but unfortunately they do not capture the full notion of computability.

Definition

A partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ is (Turing) computable if there is a Turing machine M that computes f.

Note that we are dealing with partial rather than total functions: on any particular input the TM may fail to halt. In fact, the machine may fail to halt on all inputs.

Historically these functions are also called partial recursive functions. The ones that are in addition total are called (total) recursive functions.

For those worried about coding issues, let's assume numbers are written binary.

We can use the standard characteristic function of a relation $R \subseteq \mathbb{N}^k$

$$\text{char}_R(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{x} \in R \\ 0 & \text{otherwise.} \end{cases}$$

to translate relations into functions.

Definition

A relation $R \subseteq \mathbb{N}^k$ is (Turing) decidable if its characteristic function is Turing computable.

Historically these relations are also called recursive relations.

Since Turing machines may fail to halt, one can use halting to characterize a class of relations.

Definition

A relation $R \subseteq \mathbb{N}^k$ is (Turing) semidecidable if there is a Turing machine M that halts precisely on all $\boldsymbol{x}$ such that $R(\boldsymbol{x})$ holds.

But from a CS perspective it is quite weird: we have a "semi-decision procedure" for R, a broken decision algorithm that

- correctly returns the answer Yes, if the answer is indeed Yes, but
- diverges and runs forever if the answer is No.

Semidecidability may appear to be a rather useless idea in the world of real algorithms. Alas, it is arguably more fundamental than decidability, trust me.

For example, we will see shortly that a relation R is decidable iff R and its complement are both semidecidable.

Semidecidable sets are also called recursively enumerable (r.e.) because they can be generated by algorithms that run forever, see below.

OK, so we have a notion of a function being computable and a relation being decidable.

These notions do not change if we modify our definitions slightly:

- one-way infinite tapes
- multiple tapes
- multiple heads
- different input/output conventions
- different coding conventions

Note that without this kind of robustness our model would be of rather dubious value: each variant would produce a different notion of computability.

Turing called these machines a-machines, for automatic machines. He was not interested in the kinds of computational problems associated nowadays with Turing machines, but wanted to describe computable reals, construed as sequences $(a_i) \in \mathbf{2}^\omega$.

The tape of an a-machine is one-way infinite and alternates between write-only output cells and work cells:

w_0	a_0	w_1	a_1	w_2	a_2	$\ldots$	w_n	a_n	$\ldots$

During the computation, the output cells are filled left-to-right, with no gaps.

For such a machine to do its job it needs to run for ω many steps: it has to fill all the output cells with a bit. Halting here is most undesirable.

Doing Circles?

Turing calls a machine circular if it fails to produce an infinite output sequence, and thus does not determine a real number.

By contrast, an a-machine is circle-free if it writes out a full real, in ω many steps.

Turing showed that it is undecidable whether a machine is circle-free.

Turing also introduced c-machines (choice machines) and later o-machines (oracle machines). A choice machine stops every once in a while, and asks for external advice what to do next, essentially asking for a single bit. He is also aware of deterministic simulation of these machines.

An oracle machine, on the other hand, has access to a mysterious source of information than can compute some function: given some argument x, the oracle will produce $f(x)$. For sufficiently complicated f such a machine cannot be simulated.

In 1954, Hao Wang described a version of Turing machine that is equivalent to the original ones, but, at first glance, seems a lot weaker. The tape alphabet is $\mathbf{2}$, think of 0 as the blank symbol. There are five types of instructions in a Wang machine:

- `move left`
- `move right`
- `print` 1
- `goto` k
- `halt`

A Wang program is a list of numbered instructions I_1, I_2, ..., I_n that are executed sequentially.

The semantics of all the instructions is obvious, except for the goto: if the current symbol is 1, goto line k, otherwise continue with the next instruction.

We may safely assume that all goto labels are in $[n]$. Also, we may assume that I_n is the only halt instruction.

It is easy to see that we may guard the print instructions, so that only a cell containing a 0 can be changed into a 1. Thus, a Wang machine cannot erase a 1; once it's written, it will stay forever.

So these machines are non-erasing, which may seem like a serious constraint; it is not at all clear how to simulate an ordinary TM with a WM. Just try to implement a simple loop.

Also note that one has to adjust the input/output conventions, we cannot clean up the tape before the computation terminates.

They Are Equivalent

Wang was able to establish the following surprising result:

Theorem (Wang 1954)

Every partial recursive function can be computed by a Wang machine.

The proof is quite difficult and messy.

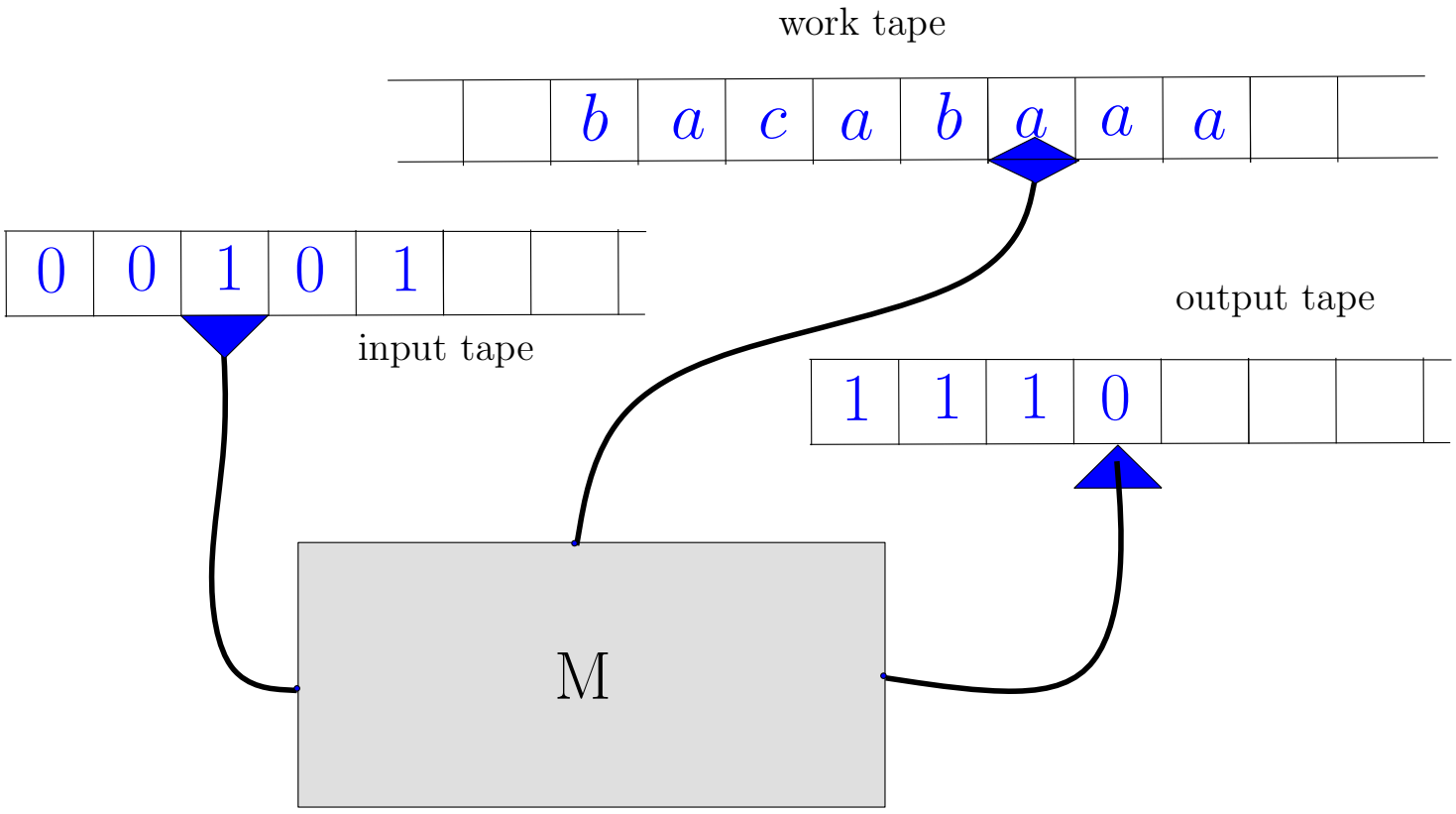
Exercise

Come up with a reasonable input/output convention for Wang machines.

Exercise (Hard)

Prove the theorem. Good luck.

Input/Output Tapes



In this model, the

- input tape is read-only, one-way infinite
- output tape is write-only, one-way infinite
- work tape is read/write, two-way infinite

As we will, see this is just the right model for space complexity.

It should be clear that Turing machines are really quite unwieldy: it would be a huge and mostly pointless effort to show that, say, the nth prime function is indeed Turing computable, a fact that is fairly obvious for primitive recursive functions. Truth in advertising: in the λ-calculus things are even more tedious.

Always remember: the critical point of Turing machines is that they arguably provide exactly the "right" class of functions. And, they are clearly physically realizable (if we ignore size/energy issues). Ease of use is a non-issue.

As we will see, in complexity theory one actually sometimes has to argue about constructing specific Turing machines, but that's a different story.

We need to adjust our definitions of clones to use partial functions rather than total ones–a routine generalization.

Theorem

Turing computable functions form a clone, that contains the clone of primitive recursive functions.

This is just the tip of an iceberg, there are several other natural models that all produce the exact same clone. More on this when we talk about the Church-Turing thesis.

Exercise

Explain how a multi-tape TM can be simulated by a single-tape TM.

Exercise

Determine how long it takes to recognize palindromes on two tapes versus one tape.

Exercise

Determine the general slow-down caused by switching from two tapes to one.

Exercise

Show that it does not matter whether the tape head is required to return to the standard left position at the end of a computation.

Our definition of a Turing machine as a structure $\langle Q, \Sigma, \delta, q_{\text{init}}, q_{\text{halt}} \rangle$ is fairly general, but we might as well assume that

- $Q = [n]$
- $\Sigma = [k]$
- $q_{\text{init}} = 1$, $q_{\text{halt}} = n$

The transition function now has the form

$$\delta : [n] \times [k] \to [n] \times [k] \times \Delta$$

and can easily be coded as list of length nk of triples of integers (code Δ as $\{0, 1, 2\}$).

But then we can express δ and hence all of M by just a single sequence number, the index for M.

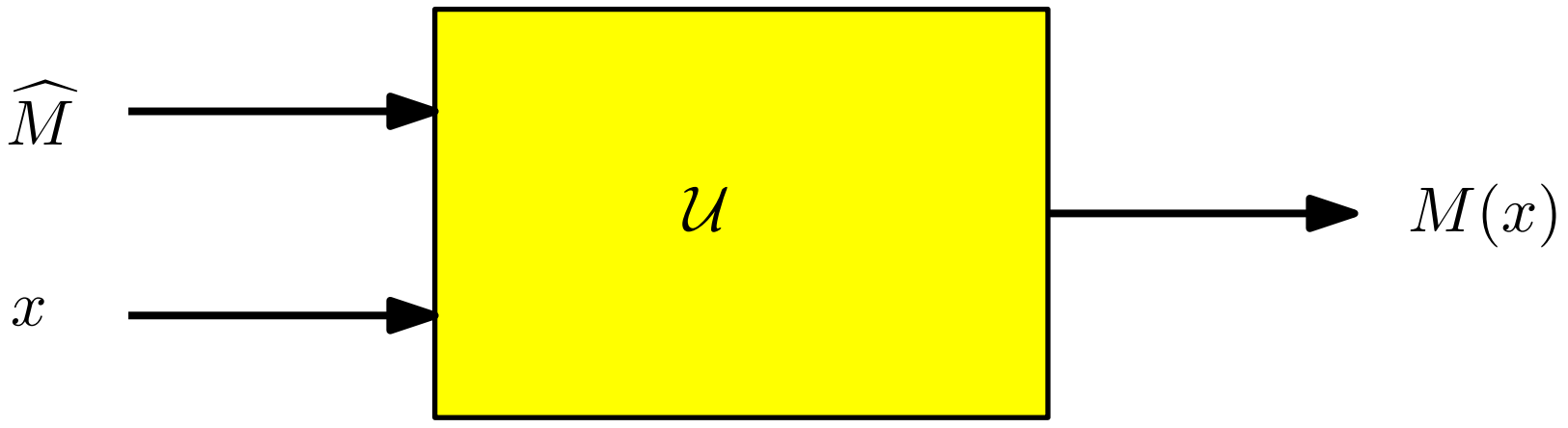
We will write $\widehat{M}$ for the index associated with TM M.

The fact that we can code a Turing machine as an integer, combined with the observation that Turing machine compute on integers, has an interesting consequence:

> We can build a Turing machine $\mathcal{U}$ that takes as input 2 numbers e and x, interprets e as the index $\widehat{M}$ of a Turing machine M, and then simulates M on input x.
>
> $\mathcal{U}$ will halt on e and x iff M halts on x; moreover, $\mathcal{U}$ will return the same output in this case.

Any such machine $\mathcal{U}$ is called a universal Turing machine.

The details of the construction of $\mathcal{U}$ are very tedious, but it's "clear" that this can be done.


$\widehat{M}$
x
$\mathcal{U}$
$M(x)$

If you really want to see the details of the construction of a universal Turing machine, read Turing's original paper:

> A. Turing
>
> On Computable Numbers, with an Application to the Entscheidungsproblem
>
> Proc. London Math. Soc., Series 2, 42 (1936-7), pp. 230–265.

Turing actually dealt with a-machines (a for automatic) whose purpose it is to write the infinite binary expansion of a real number on the tape.

Our description of a TM is the modern one due to Davis, Kleene. As we will see, TMs are also a perfect setting for complexity theory.

- If $\mathcal{U}$ is given a number e as input that fails to code a Turing machine we assume that $\mathcal{U}$ fails to halt.
- Clearly the argument also works for Turing machines with multiple inputs (which we can either keep separate or code into a single input).
- Note that $\mathcal{U}$ is by no means uniquely determined, there are lots and lots of choices.
- The simulation by a universal TM is even efficient in the sense that the computation of $\mathcal{U}$ won't take much longer than the computation of M.
- A very interesting question is how large a universal Turing machine needs to be. Amazingly, there is a 2-state, 5-symbol UTM.

Fix some universal Turing machine $\mathcal{U}$. Each TM M translates into an integer $e = \langle M \rangle$, the index for M.

Similarly we can think of e as describing a particular computable function, often written

$$\varphi_e \qquad \text{or} \qquad \{e\}$$

Hence, $(\{e\})_e$ is a complete listing of all computable functions.

We won't deal with arity issues here, they are purely technical. So we'll happily write things like $\{e\}(x, y)$ and so on.

Keeping notation simple is better than going overboard on precision. Read Hartley Rogers's book if you think otherwise.

When dealing with partial recursive functions it is convenient to have some shorthand notation for convergence and divergence:

$$\{e\}(x)\downarrow \qquad \text{versus} \qquad \{e\}(x)\uparrow$$

Similarly one writes

$$\{e\}(x) \simeq y$$

to mean that the left hand side converges with output y, or both sides are undefined.

Note that $\{e\}(x)\downarrow$ is semidecidable: we just run the computation.

Suppose $\mathcal{U}$ is a universal TM. Note that it has some index $u \in \mathbb{N}$. The critical property of $\mathcal{U}$ can then be expressed like so:

$$\{u\}(e, x) \simeq \{e\}(x)$$

For a CS person, this should be entirely unsurprising: think of $\mathcal{U}$ as an interpreter that takes as input a program e and an input x, and runs e on x.

In the 1930s this was quite amazing.

The universal Turing machine is just our old evaluation function eval, but this time in the realm of computable, partial functions.

As a consequence, the diagonalization trick that showed that there cannot be a universal primitive recursive function fails in this context:

$$f(x) := \mathsf{eval}(x, x) + 1$$

is certainly computable and must have an index e. Hence $f(e) \simeq \mathsf{eval}(e, e) + 1 \simeq f(e) + 1$, but there is no a contradiction: both sides diverge.

It might be tempting to think of partial recursive functions as total recursive functions restricted to some smaller domain of definition. Tempting, but very wrong.

Proposition

There is a partial recursive function g that is not the restriction of any total recursive function.

Proof.

Let

$$g(e) \simeq \begin{cases} \{e\}(e) + 1 & \text{if } \{e\}(e)\downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

It is easy to see that g cannot be of the form $g \simeq f \restriction D$ for any total recursive function f.

□

It was well-known in the 19th century that some problems have no solutions. E.g., a third-degree equation $x^3 + ax^2 + bx + c = 0$ has a root

$$-\frac{a}{3} - \frac{\sqrt[3]{2}\left(-a^2+3b\right)}{3\sqrt[3]{-2a^3+9ab-27c+3\sqrt{3}\sqrt{-a^2b^2+4b^3+4a^3c-18abc+27c^2}}} + \frac{\sqrt[3]{-2a^3+9ab-27c+3\sqrt{3}\sqrt{-a^2b^2+4b^3+4a^3c-18abc+27c^2}}}{3\sqrt[3]{2}}$$

But Ruffini and Abel showed that there is no analogous solution in radicals for equations of degree 5 or higher. Example: $x^5 - x + 1 = 0$.

And Galois was able to classify the polynomial equations that do admit solutions in radicals.

Problem:	**Halting Problem**
Instance:	An index e, an input x.
Question:	Does $\{e\}(x)$ converge?

Theorem

The Halting Problem is undecidable.

Proof. Suppose otherwise and define a function g by

$$g(e) \simeq \begin{cases} \{e\}(e) + 1 & \text{if } \{e\}(e)\downarrow, \\ 0 & \text{otherwise.} \end{cases}$$

Then g is computable, so $g \simeq \{e\}$ for some e. Contradiction via input e. □

A definition of the form

$$g(x) \simeq \begin{cases} f(x) & x \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

where f is computable and A is semidecidable produces a computable function. But

$$g(x) = \begin{cases} f_1(x) & x \in A, \\ f_2(x) & \text{otherwise.} \end{cases}$$

does not work in general. It is OK when A is decidable, though: in that case the argument is exactly the same as for primitive recursive functions.

In Pseudo-Code

```
// contradictory program

    if( halt(e,e) )
        return  eval(e,e) + 17;
    else
        return 0;
```

Here `halt(e,x)` is the halting tester that exists by assumption, and `eval(e,x)` is just the UTM: run M_e on input x.

As a practical matter, anyone who has ever written a complicated piece of code in a real language will have experienced the dreaded effect: you press "Return," the program starts to run, and runs, and runs, but does not seem inclined to ever terminate.

In the RealWorld, this happens because of an error, a bug, but it turns out that there is little one can do about this type of mistake: given a sufficiently powerful language, Halting becomes a real problem.

Some unsolved problems of math can be expressed nicely in terms of halting Turing machines.

For example, the Riemann Hypothesis in its original formulation says the following. Let

$$\zeta(s) = \sum_{n \geq 1} n^{-s}$$

where $\mathrm{Re}(s) > 1$.

By analytic continuation, one extend this to a function defined on all of $\mathbb{C}$.

Riemann Hypothesis: All non-real roots s of this function have $\mathrm{Re}(s) = 1/2$.

As written, this looks like a very complicated assertion. But, RH is equivalent to the following inequality, for all $n \geq 1$:

$$\sigma(n) \leq H_n + e^{H_n} \ln H_n$$

where σ is the divisor-sum function, and $H_n = \sum_{i \leq n} 1/i$ is the nth harmonic number.

One can construct a Turing machine that attempts to verify this inequality for all n, and halts only if it finds a counterexample.

Thus, this machine $\mathcal{M}$ halts iff RH is false.

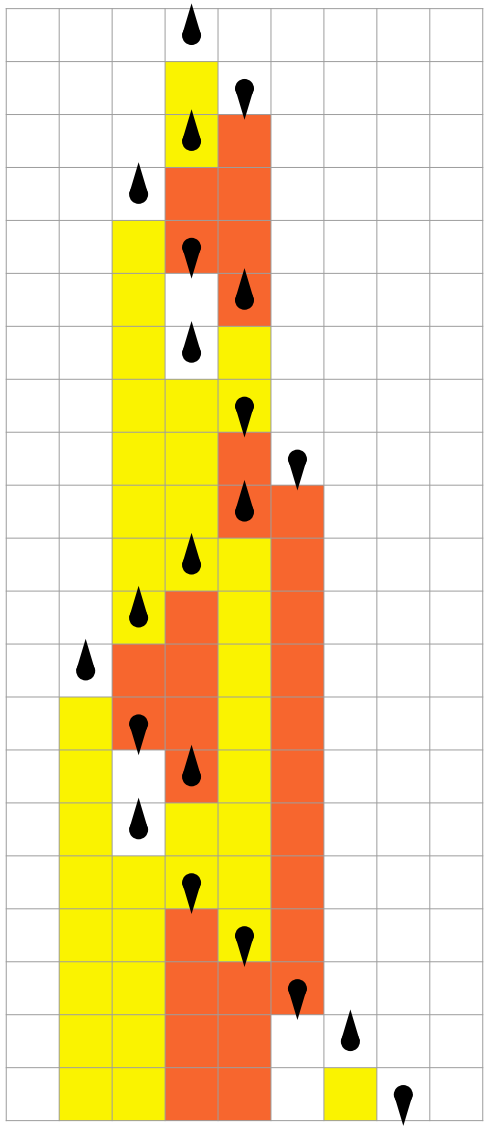
A Prize Question

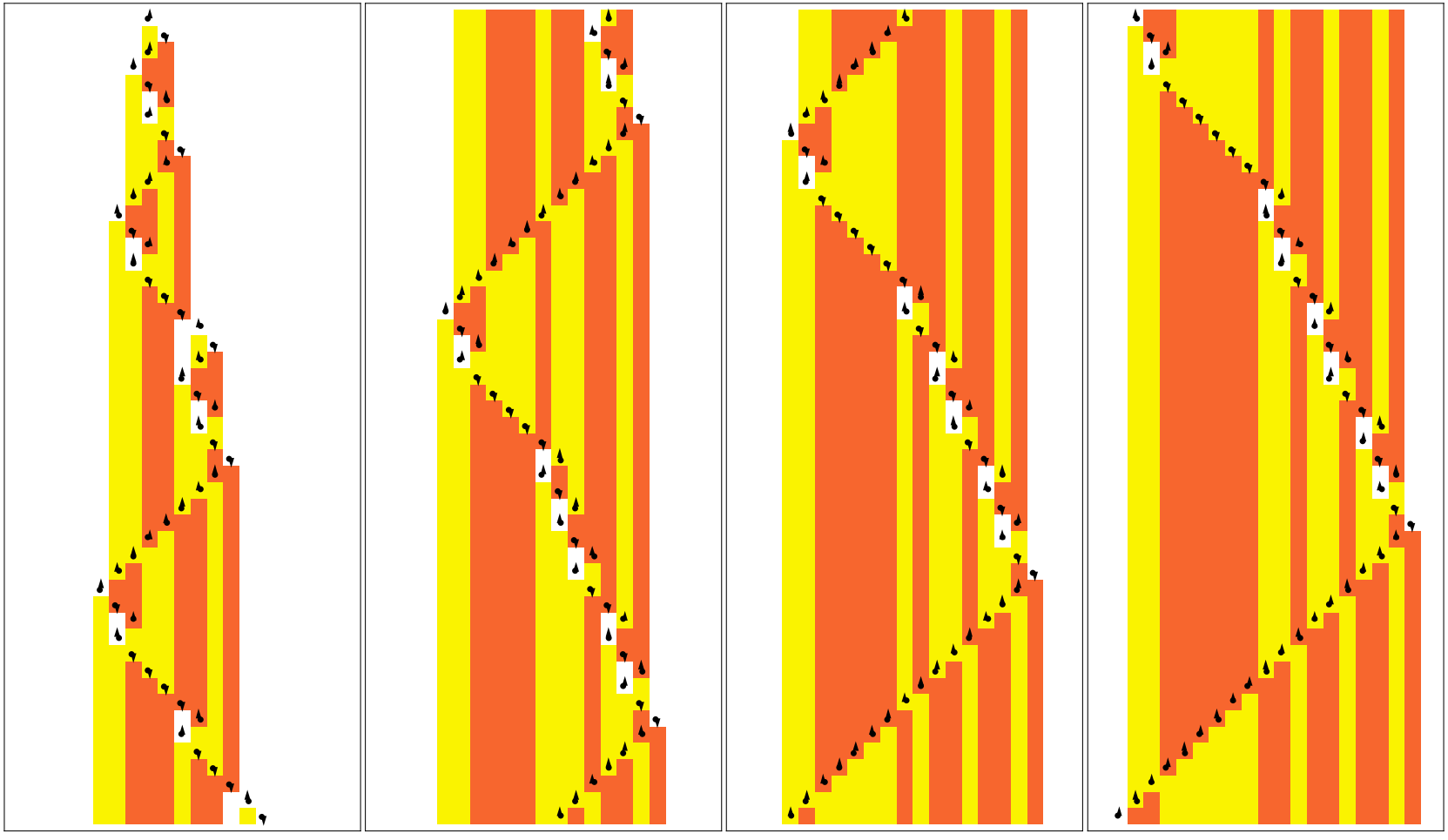
In May 2007, Stephen Wolfram posed the following challenge question:

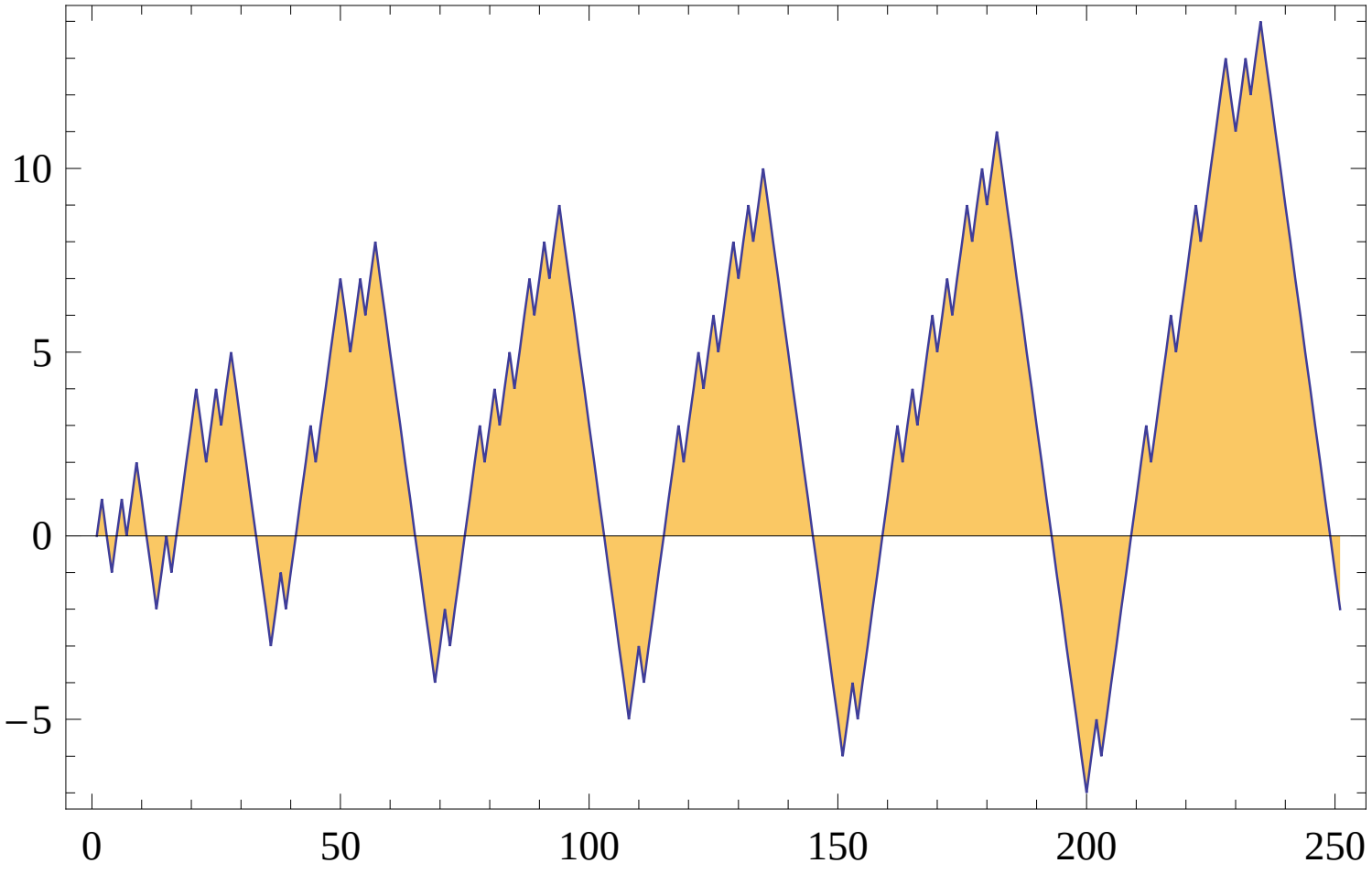
> Is the following (2,3)-Turing machine universal?

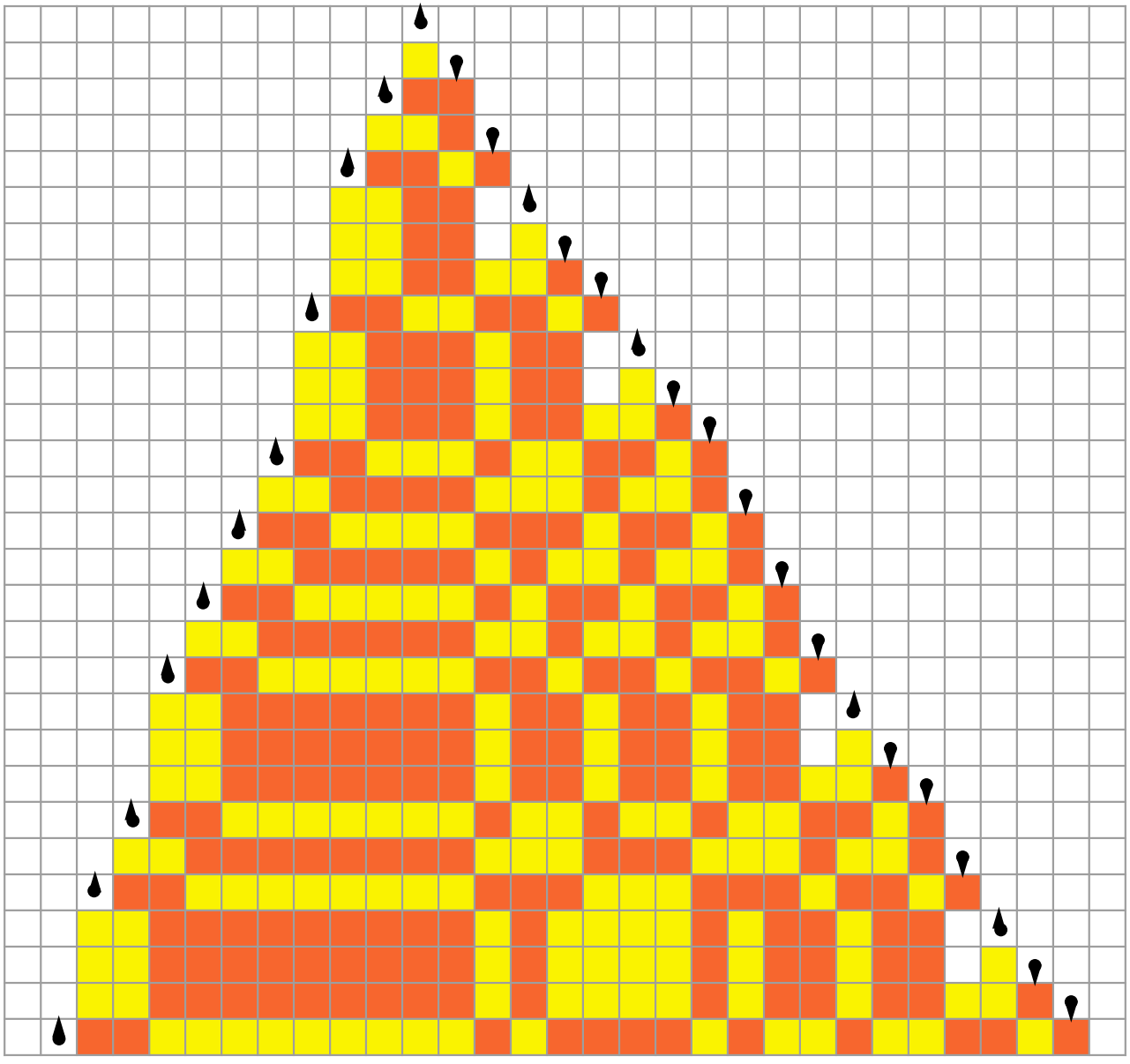
	0	1	2
p	(p,1,L)	(p,0,L)	(q,1,R)
q	(p,2,R)	(q,0,R)	(p,0,L)

Prize money: $25,000.

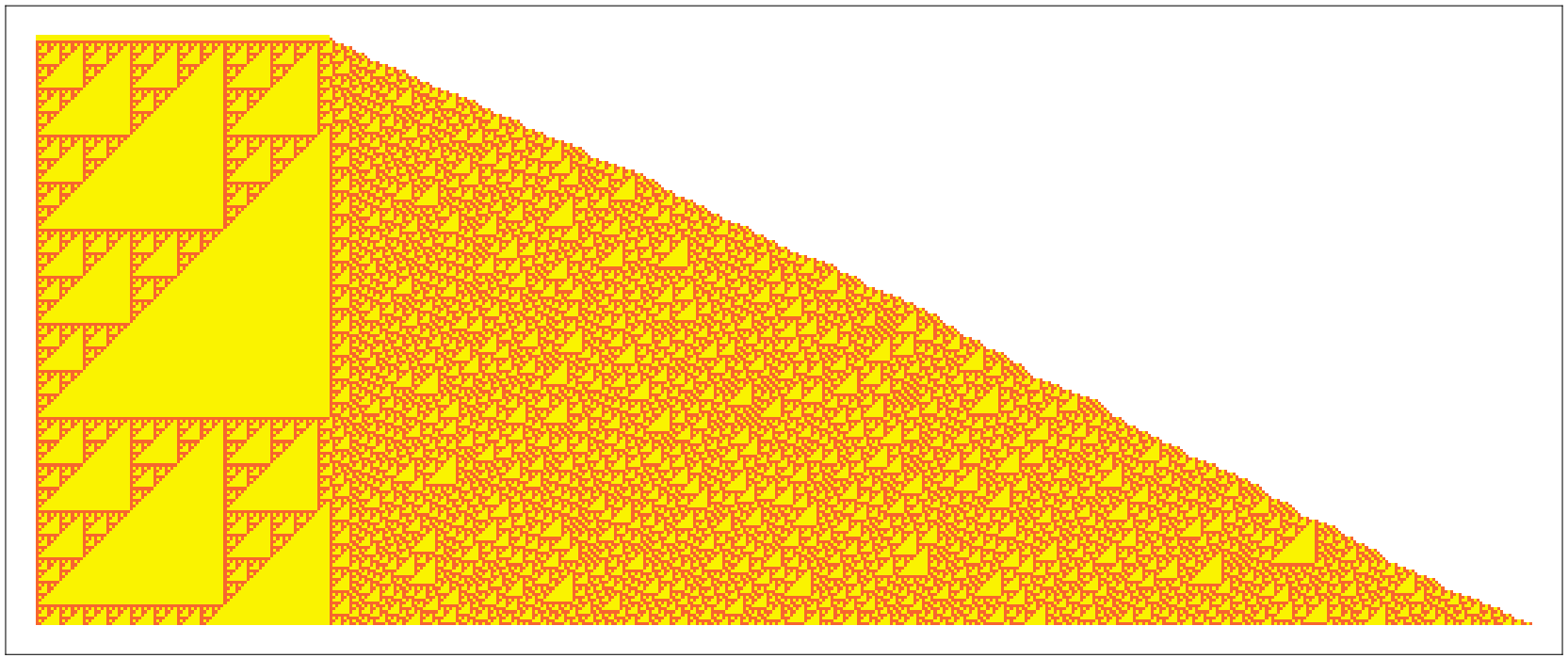





10
5
0
−5
0
50
100
150
200
250



Compressed Computation with Different Initial Condition



- We saw how to construct a universal universal Turing machine.
- But the prize machine is not "designed" to do any particular computation, much less to be universal.
- The problem here is to show that this tiny little machine can simulate arbitrary computations – given the right initial configuration (presumably a rather complicated initial configuration).
- Alas, that's not so easy.

- In the Fall of 2007, Alex Smith, an undergraduate at Birmingham at the time, submitted a "proof" that the machine is indeed universal.
- The proof is painfully informal, fails to define crucial notions and drifts into chaos in several places.
- A particularly annoying feature is that it uses infinite configurations: the tape inscription is not just a finite word surrounded by blanks.
- At this point, it is not clear what exactly Smith's argument shows.