COMP90057 Advanced Theoretical Computer Science

Lectures 2-6

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 2-6

Second Semester, 2017

COMP90057 Advanced Theoretical Computer Science

Lec 2: Turing Machine Variants

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 2

Second Semester, 2017

Recap

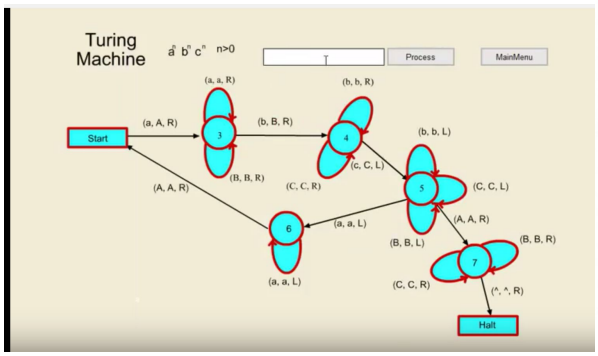
Last lecture

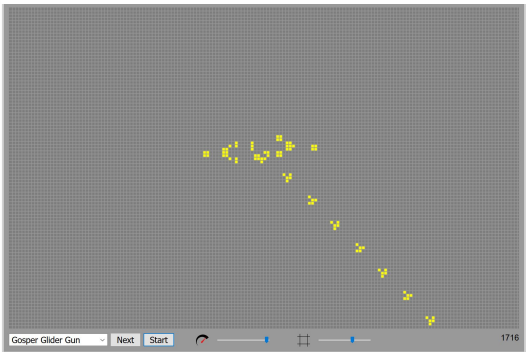
- ...

Strictly for entertainment

- `http://aturingmachine.com/index.php` - a physical rendering of an abstract machine

Looking Ahead: What's the Difference?





`https://www.youtube.com/watch?v=wZZo9VXTznY`

`https://bitstorm.org/gameoflife/`

Turing Machines and Languages

(Sipser, page 170)

The set of strings accepted by M is the language of M, $L(M)$.

Call a language A Turing recognisable (or recursively enumerable, often abbreviated as r. e.) if $A = L(M)$ for some Turing machine M.

Three behaviours are possible for M on input w: M may accept w, reject w, or fail to halt.

We say a language A is Turing decidable (or recursive, or decidable) if there is some M that recognises A and that halts on all inputs.

Ways of describing Turing Machines

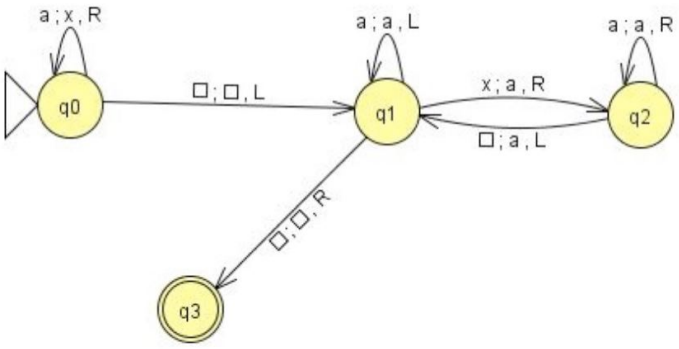
- formal description
- implementation description
- high-level description

Sipser, p 184-185

Exercise

(i) Give a high-level description of a TM with input alphabet $\{0,1\}$ that recognises the set of strings in 0^*10^*

(ii) Consider the TM, M, with the following state transition diagram. Draw a state transition table for M.



(iii) What is the language accepted by M?

(iv) Give a high-level description of a TM with input alphabet $\{0,1\}$ that recognises the set of strings with an equal number of 0s and 1s

Exercise

Robustness of Turing Machines

Other textbooks have definitions of Turing machines that differ slightly from Sipser's.

Some differences are technical and aim at making the machines easier to program (for example, we may insist that machines start with a tape that has the first cell blank, and they try to leave that cell blank—to make it easier to compose machines).

Turing machines are robust in the sense that such changes to the machinery do not effect what the machines are capable of computing.

Variants of Turing Machines

In an attempt to make the Turing machine more powerful we could alter its features, for example:

1. Let the tape stay-put after it reads a tape square (Sipser, p 176)
2. Let the tape extend indefinitely in both directions (Sipser Ex 3.11)
3. Let there be several tapes, each with an independent tape head (Sipser, p 176-7)
4. Add nondeterminism to the transition function (Sipser, p 178-9)

But it can be formally proved (with varying degrees of difficulty) that none of these increases the power of Turing machines capabilities as recognisers.

Q. How would you informally justify #1 on this list?

Multitape Machines (Sipser, pg 176)

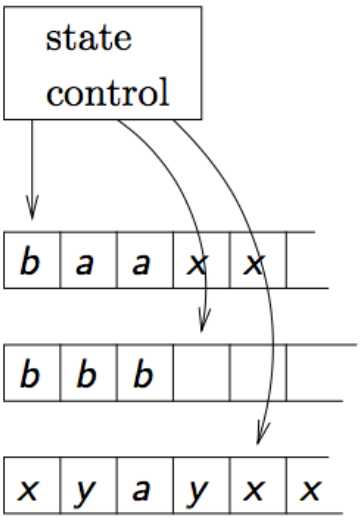
A multitape Turing machine has k tapes. It takes its input on tape 1, other tapes are initially blank.

The transition function has type

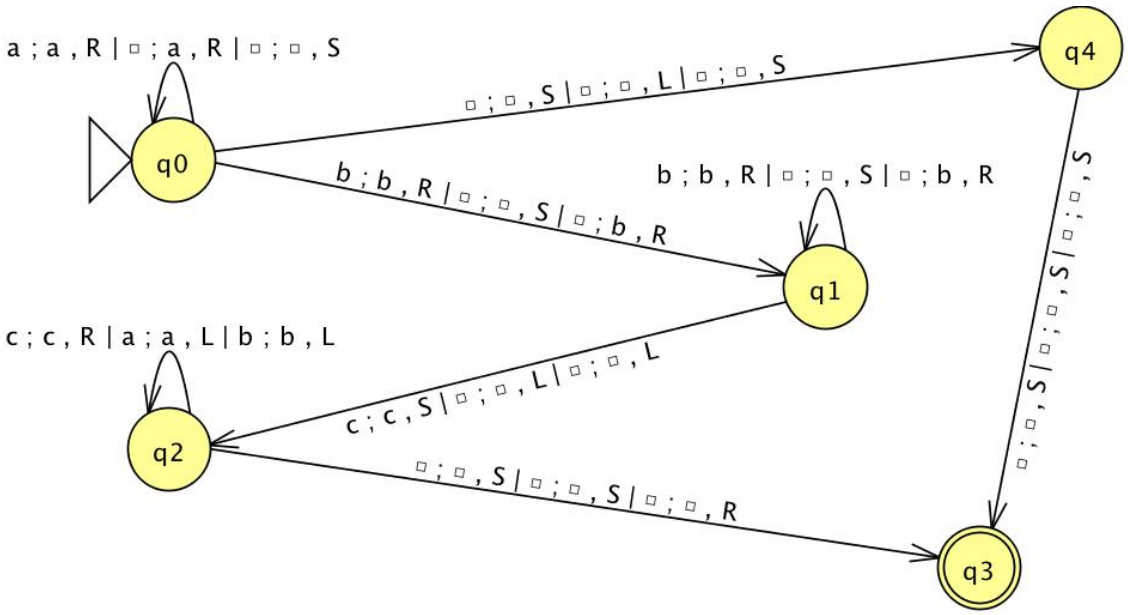
$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

and specifies how the k tape heads behave when the machine is in state q_i, reading $a_1, \ldots a_k$:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, (b_1, \ldots, b_k), (d_1, \ldots, d_k))$$



Multitape Machines with JFLAP



JFLAPlec2-eg1Multi.jff on LMS > Subject Resources > JFLAP example files

Simulating a Multitape Machine (Sipser Thm 3.13, Cor 3.15)

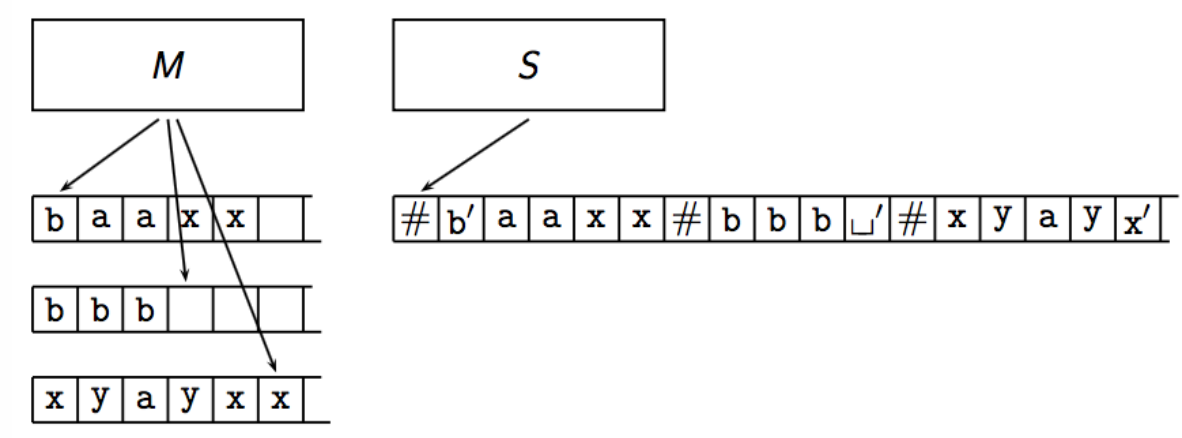
Theorem: Every multitape TM has an equivalent single-tape TM

Corollary: A language is Turing recognisable iff some multitape Turing machine recognises it.

Proof sketch for the Theorem: We show how to simulate a multitape machine M by a standard Turing machine S.

Consider an alphabet Γ' whose elements have a one-to-one matching with elements in Γ.

Machine S has tape alphabet $\{\#\} \cup \Gamma \cup \Gamma'$, where $\#$ is a separator, not in $\Gamma \cup \Gamma'$.



S reorganises input $x_1 x_2 \cdots x_n$ into

$$\#x_1' x_2 \cdots x_n \underbrace{\#\sqcup'\# \cdots \#\sqcup'\#}_{k-1 \text{ times}}$$

The elements of Γ' represent *marked* elements in Γ.

Simulating a Multitape Machine

Simulating an M move, S scans its tape to determine the marked symbols. On a second scan, it updates the tape according to M's transition function.

If one of the simulated tapes runs out of space, because the "simulated tape head" moves onto # cell, S S shifts that symbol, and every symbol after it, one cell to the right. In the vacant cell it writes $\sqcup$.

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

PROOF We show how to convert a multitape TM M to an equivalent single-tape TM S. The key idea is to show how to simulate M with S.

Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the new symbol # as a delimiter to separate the contents of the different tapes. In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be. Think of these as "virtual" tapes and heads. As before, the "dotted" tape symbols are simply new symbols that have been added to the tape alphabet. The following figure illustrates how one tape can be used to represent three tapes.

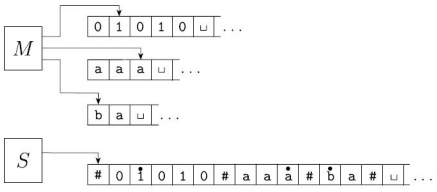


FIGURE 3.14
Representing three tapes with one

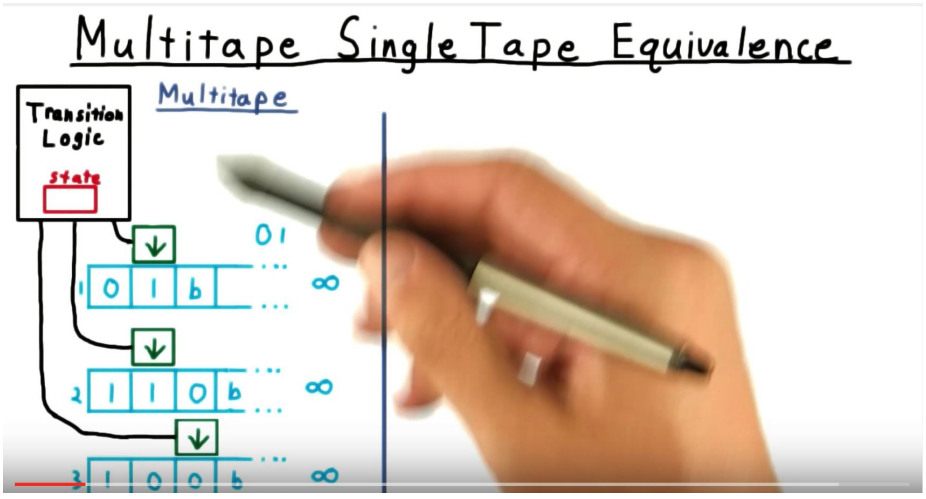
S = "On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains

$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#.$$

2. To simulate a single move, S scans its tape from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

Simulating a Multitape Machine with a Single Tape Machine



https://www.youtube.com/watch?v=otW5KDW1IJA (3 minutes)

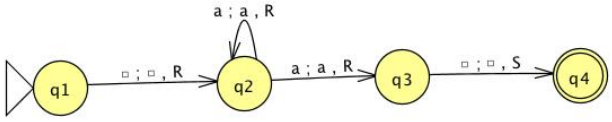
Nondeterministic Turing Machines

A nondeterministic Turing machine has a transition function of type

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

A deterministic computation is a (fixed) sequence. A NDTM computation is a tree, where each branch looks like a computation of a deterministic TM. If any single branch reaches an accepting state, the NDTM accepts - even if other branches reach the rejecting state; otherwise it rejects

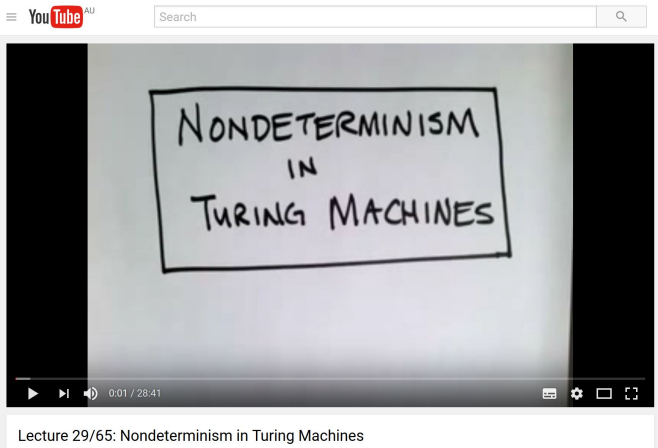
(Extremely) simple example (can draw with JFLAP, but not execute with JFLAP)



Nondeterministic Turing Machines

Gentle (15 minute) introduction

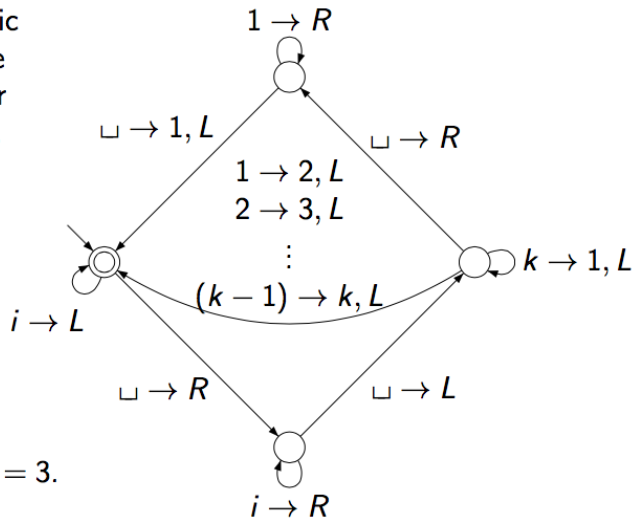
`https://www.youtube.com/watch?v=eMchSJaJJVU`



`http://web.cecs.pdx.edu/~harry/`

Simulating a Nondeterministic Turing Machine

First, a deterministic machine to generate $\{1, \ldots, k\}^*$, in order of increasing length.



Try running it for $k = 3$.

Simulating a Nondeterministic Turing Machine

Theorem: (Sipser 3.16) Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Corollary: (Sipser 3.18) A language is Turing recognisable iff some nondeterministic Turing machine recognises it.

Proof idea: (Sipser p 178)

PROOF IDEA We can simulate any nondeterministic TM N with a deterministic TM D. The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate.

We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N. The root of the tree is the start configuration. The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree. A tempting, though bad, idea is to have D explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design D to explore the tree by using breadth-first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

Simulating a Nondeterministic Turing Machine

Proof sketch: We need to show that every nondeterministic Turing machine N can be simulated by a deterministic Turing machine D.

We show how it can be simulated by a 3-tape machine.

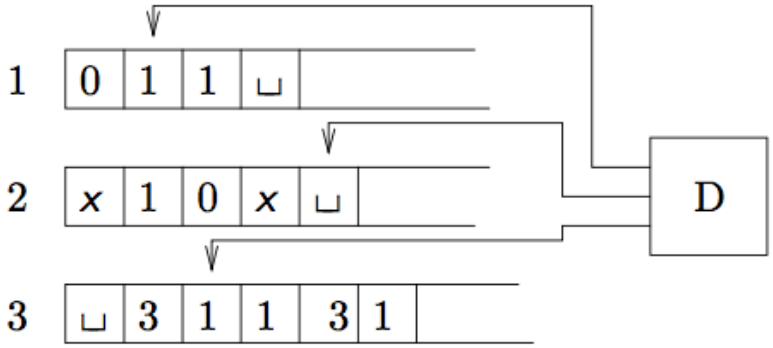
Let k be the largest number of choices, according to N's transition function, of all state-symbol combinations.

Tape 1 (always) contains the input.

Tape 3 holds longer and longer sequences from $\{1, \ldots, k\}^*$.

Tape 2 is used to simulate N's behaviour for each fixed sequence of choices given by tape 3.

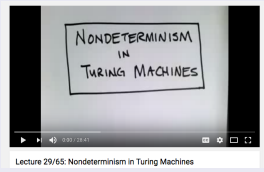
Simulating a Nondeterministic Turing Machine



1. Initially, tape 1 contains input w, and the other two tapes are empty.
2. Overwrite tape 2 by w.
3. Use tape 2 to simulate N. Tape 3 dictates how N should make its choices. If tape 3 gets exhausted, go to step 4. If N says accept, accept.
4. Generate the next "choice" sequence on tape 3. Go to step 2.

Review

- Sipser pp 173 -175 - Examples 3.9, 3.11 and 3.12, to become familiar with more complex TMs
- Sipser pp 176-177 (part of Section 3.2 *Variants of Turing machines*)
- `https://www.youtube.com/watch?v=eMchSJaJJVU`



Source: Portland State Univ: Prof. Harry Porter; `www.cs.pdx/~harry`

Preparation

- Read Sipser pp 182-184 - Section 3.3 - introducing the *Church-Turing thesis*
- Skim the Wikipedia entry `https://en.wikipedia.org/wiki/Church-Turing_thesis` and/or the AlanTuring.net entry `http://www.alanturing.net/turing_archive/pages/reference%20articles/The%20Turing-Church%20Thesis.html` to get a feel for both some of the history, and some of the contemporary philosophical discussions. *(Gain some awareness of the issues addressed, but the history and philosophy are not examinable)*

COMP90057 Advanced Theoretical Computer Science

Lecture 3: Church-Turing thesis; Computable numbers; Finite Automata

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 3

Second Semester, 2017

Recap

Last lecture...

- ???

Requested preparation for today

- Read Sipser pp 182-184 - Section 3.3 - introducing the *Church-Turing thesis*
- Skim the Wikipedia entry `https://en.wikipedia.org/wiki/Church-Turing_thesis` and/or the AlanTuring.net entry `http://www.alanturing.net/turing_archive/pages/reference%20articles/The%20Turing-Church%20Thesis.html` to get a feel for both some of the history, and some of the contemporary philosophical discussions. *(Gain some awareness of the issues addressed, but the history and philosophy are not examinable)*

The Church-Turing Thesis

"Every effective computation can be carried out by a Turing machine"

Intuitive notion of algorithms	=	Algorithms implemented via Turing machines

Note that we cannot hope to prove the Church-Turing thesis, only accumulate lots of evidence for its validity.

Deciders, Recognisers and Enumerators

Recall: A language A is called Turing recognisable if $A = L(M)$ for some Turing machine M.

Recall: A language A is called decidable if A is recognised by some Turing machine that halts on all inputs.

The Turing machine we built last week to generate all strings in $\{1, \ldots, k\}^*$ is an example of an enumerator.

We could imagine it being attached to a printer, and it would print all the strings in $\{1, \ldots, k\}^*$, one after the other, never terminating.

For an enumerator to enumerate a language L, for each $w \in L$, it must eventually print w. It is allowed to print w as often as it wants, and it can print the strings in L in an arbitrary order.

The following theorem explains why we also call Turing recognisable languages recursively enumerable.

Recognisers and Enumerators

Thm: L is Turing recognisable iff some enumerator enumerates L.

Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E: if $s = w$, accept.

Conversely, let M recognise L. Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate the strings in Σ^*: $s_1, s_2, \ldots$ Here is what E does:

1. Let $i = 1$.
2. Simulate M for i steps on each of $s_1, \ldots, s_i$.
3. For each accepting computation, print that string.
4. Increment i and go to step 2.

Hilbert's tenth problem (Sipser pp 182-184)

Find an algorithm that determines whether a polynomial equation with integer coefficients has an solution comprising only integer values.

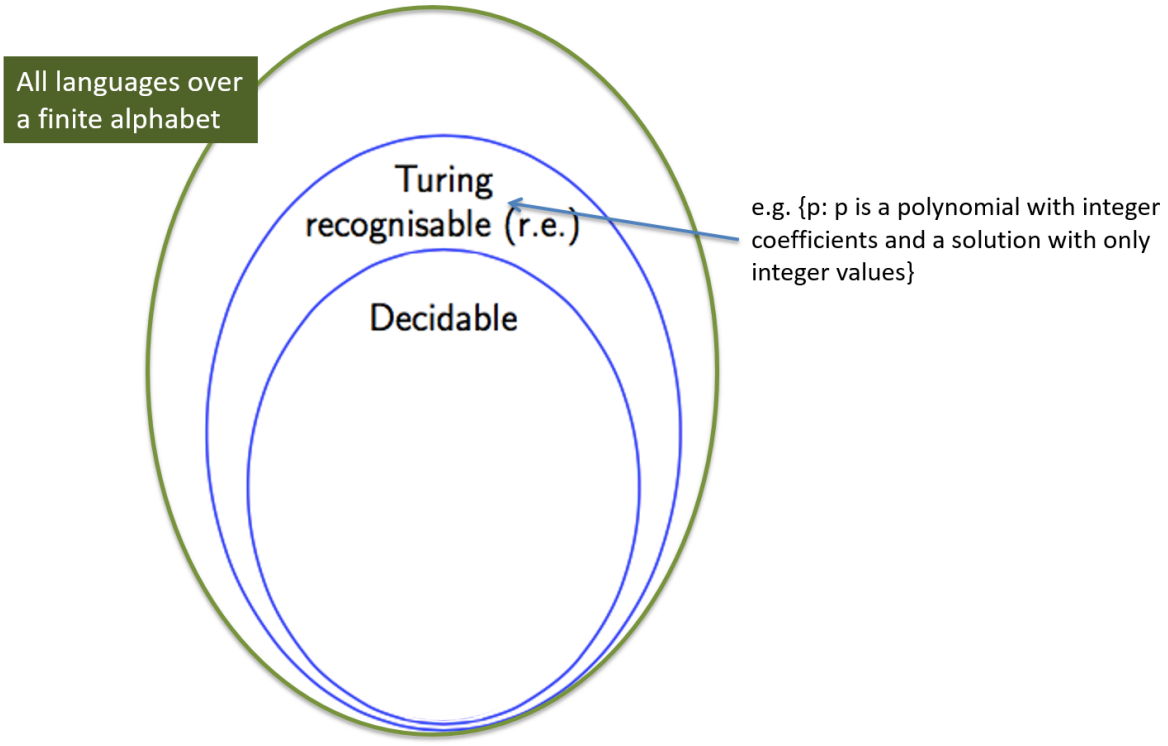
Theorem $\{p \mid p$ is a polynomial with integer coefficients and a solution with only integer values$\}$ is Turing recognisable.

Proof idea: Start thinking about a specific polynomial p with three variables: x, y, and z. Here is one way to build a Turing machine M to recognise if p has a solution with only integer values. M can enumerate all integer triples (i, j, k). So M can evaluate p on each value triple (i, j, k) in turn. If p has an integer solution, M will eventually find it and accept.

So for a specific p with three variables one can build a recogniser.
How to build a recogniser that can handle all polynomials with three variables?
How to build a recogniser that can handle all polyniomials with any number of variables?

Matijasevič in 1970, building on work by many others, showed that although Hilbert's tenth problem is Turing recognisable, it is not decidable.

Language Hierarchy



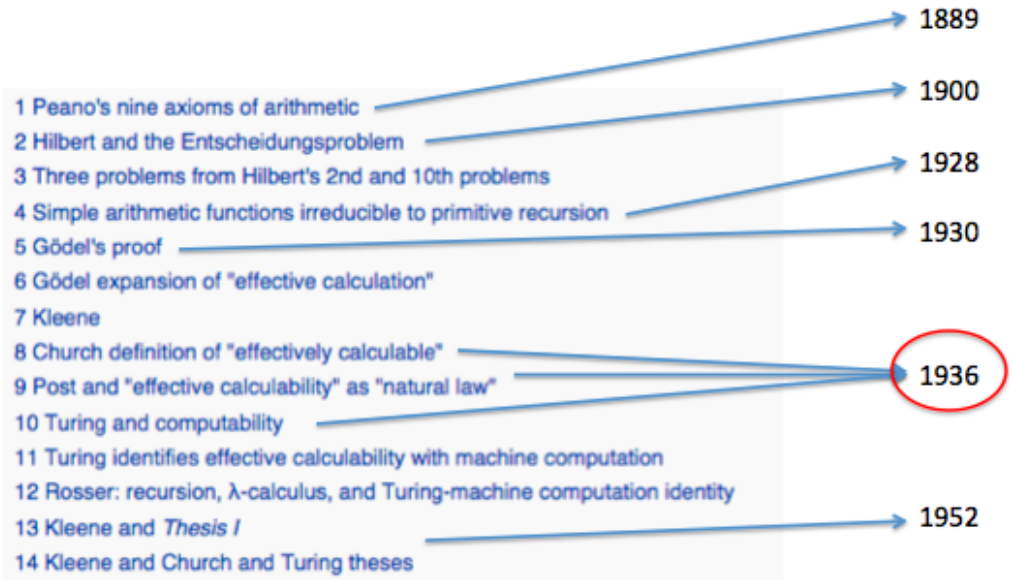
Pause...

The history[a] of the study of notions of computability, in particular the flow of mathematical, computational and philosophical ideas in the period 1900-1940 that culminated in what we now call the Church-Turing Thesis, and the more recent history[b] of cellular automata and associated discussions around such mechanisms that touch on topics as emergence and free will, are a reminder that the core ideas of computer science are more than algorithmic, and that the major figures contribute through their personality as well as their intellect.

[a] Copeland, B. Jack, "The Church-Turing Thesis", The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/archives/sum2015/entries/church-turing [b] Berto, Francesco and Tagliabue, Jacopo, "Cellular Automata", The Stanford Encyclopedia of Philosophy http://plato.stanford.edu/archives/sum2012/entries/cellular-automata/
[c] Bailey, David. H., (2002) "A Reclusive Kind of Science: Review of A New Kind of Science by Stephen Wolfram" http://www.davidhbailey.com/dhbpapers/dhb-wolfram.pdf
[d] Kurzweil, Ray (2002) "Reflections on Stephen Wolfram's "A New Kind of Science" http://cogweb.ucla.edu/crp/Media/2002-05-15_Kurzweil.html

Image source: https://urbangiraffe.com/2011/01/19/frozen-chatsworth/looking-back/

History of the Church-Turing Thesis



Copeland, B. Jack, "The Church-Turing Thesis", The Stanford Encyclopedia of Philosophy http://plato.stanford.edu/archives/sum2015/entries/church-turing/
en.wikipedia.org/wiki/History_of_the_Church_Turing_thesis

Turing's 1936 paper

1. Computing Machines
2. Definitions
3. Examples of computing machines
4. Abbreviated tables
5. Enumeration of computable sequences
6. The universal computing machine
7. Detailed description of the universal machine
8. Application of the diagonal process
9. The extent of the computable numbers
10. Examples of large classes of numbers that are computable
11. Application to the Entscheidungsproblem

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

FIGURE 7.8: Computers in the Hamburg observatory in the 1920s.

Image source: Moore and Mertens, *The Nature of Computation*, Fig 7.8

Computable numbers

- Turing's motivation for the 1936 paper was to solve a problem posed by mathematician David Hilbert in 1900 known as the *Entscheidungsproblem* - the existence of a formalised procedure to determine the provability of statements about mathematics
- The bulk of the paper is about computable numbers, and the distinction between real numbers and computable numbers is crucial to Turing's argument
- A computable number **r** is one for which there is a TM M_r that, given any input n on its initial tape, successively prints the first n digits of the decimal representation of **r** on its tape, then halts.
- The key notions in the definition are:
  - any n can be provided as input, i.e. in advance of the computation starting; and
  - for any provided n, the computation only takes a finite number of steps.

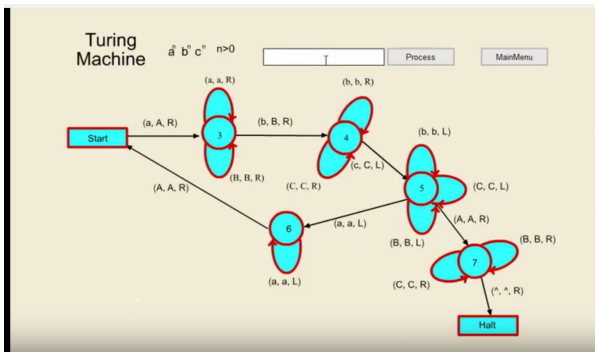
Computable numbers

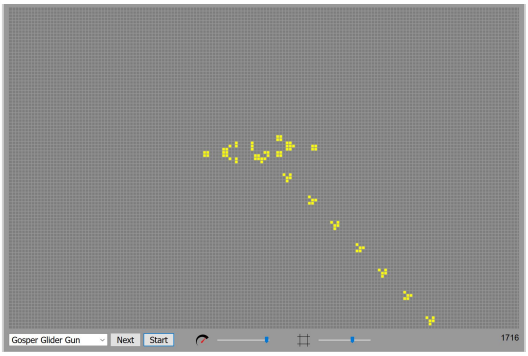
- Each computable number, even if it has an infinite decimal digit representation, has a finite definition (the state table of the relevant TM)
- Are these computable numbers?
  - $1/2$?
  - $1/9$?
  - $\sqrt{2}$?
  - π ?
  - e ?

  Answer: Yes, but $\sqrt{2}$, π, e require a lot of work to show this.

See pg 256 of Turing's 1936 paper for his argument that all algebraic numbers are computable, as well as some popular transcendental numbers. *(Algebraic numbers are the solutions of polynomial equations. Transcendental numbers are numbers that are not algebraic.)*

Reflection from Lecture 2: What's the Difference?

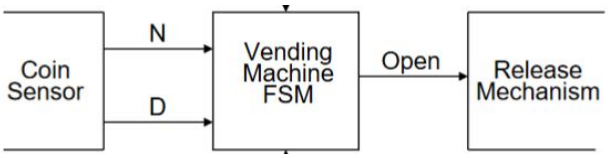




https://www.youtube.com/watch?v=wZZo9VXTznY

https://bitstorm.org/gameoflife/

Vending Machine as a finite state machine



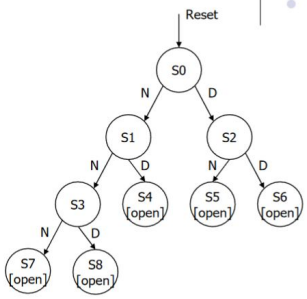
Suppose the machine is (oversimplistically) designed to release a single item after 15 cents deposited, and with no item choice by the customer

Single coin slot, no change issued

Accepts Dimes (10c) and Nickels (5c)

Vending Machine (Cont.)

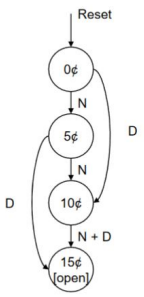
- Suitable abstract representation
  - typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



Initial vending-machine state diagram

Minimized Vending Machine's States

- Minimize number of states - reuse states whenever possible
- S4,S5….,S8 have identical behavior=> combine into a single state



present state	inputs D	N	next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Minimized symbolic state transition table

Finite Automaton: Formal Definition

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ are the accept states.

δ must be a total function i.e. defined on all of the domain.

Sipser pp 34-38 (*definition and examples of deterministic finite automata*)

Finite Automata – informally

Intuitively, TMs have a "scratch" memory in the form of tape; the configuration of a TM consists both of its current state and the current contents of the tape, which the TM may change as it executes.

Intuitively, Finite Automata have no "scratch" memory at all; the configuration of such an automaton is entirely accounted for by the state in which it currently finds itself, and its current progress in reading the input.

Although they are (provably) less powerful than TMs, Finite Automata are useful in pattern matching tasks, and in modelling certain types of control mechanisms.

Finite Automata - Acceptance and Recognition

What does it mean for a finite automaton to accept a string?

Let $M = (Q, \Sigma, \delta, q_0, F)$ and let $w = v_1 v_2 \cdots v_n$ be a string from Σ^*.

We say M accepts string w if there is a sequence of states $r_0, r_1, \ldots, r_n$, with each $r_i \in Q$, such that

1. $r_0 = q_0$
2. $\delta(r_i, v_{i+1}) = r_{i+1}$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

We say M recognises language L if $L = \{w \mid M \text{ accepts } w\}$.

The language recognised by machine M is written $L(M)$

Finite Automata, Regular Languages & Regular Expressions

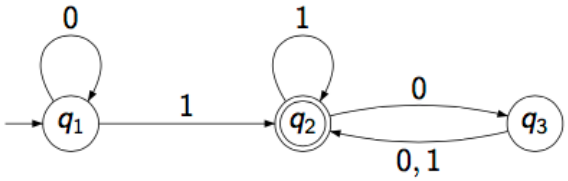
Definition: A language is called regular if there is a finite automaton that recognises it.

Theorem: A language is regular iff there is a regular expression that describes it.

Regular expressions - see Sipser section 1.3‘

Not part of this subject, but you may be interested to know that deterministic finite automata and non-deterministic finite automata recognise the same languages (see Sipser Theorem 1.39), but a similar statement is not true for the machines that recognise context-free languages (pushdown automata) - see the opening paragraph of Sipser section 2.4.

Example 1



The automaton M_1 (above) can be described as

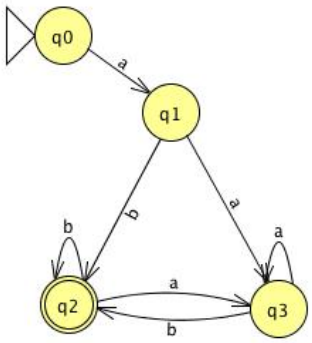
$$M_1 = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_2\}) \qquad \text{with}$$

$$\begin{aligned} \delta(q_1, 0) &= q_1, \delta(q_1, 1) = q_2, \\ \delta(q_2, 0) &= q_3, \delta(q_2, 1) = q_2, \\ \delta(q_3, 0) &= q_2, \delta(q_3, 1) = q_2 \end{aligned}$$

With some thought, you should be able to see that

$$L(M_1) = \left\{ w \;\middle|\; \begin{array}{l} w \text{ contains at least one 1, and an} \\ \text{even number of 0s follow the last 1} \end{array} \right\}$$

Example 2



Describe the language recognised by this machine.

Note: JFLAP supports a lot of manipulations with finite automata that go beyond what is examinable in this subject: for example, displaying traces of runs, (algorithmically) converting a regular expression to an non-deterministic finite automaton (Sipser pp 66-76), converting a non-deterministic finite automaton to a deterministic finite automaton (eg Sipser pp 55-58), minimising a DFA (Sipser p 327), and testing equivalence of DFAs. Some of these capabilities may be interesting for you to explore, but will not be assessed.

Example 3

Give state diagrams of DFAs recognising the following languages over the alphabet {0,1}:

- {w : w contains at least three 1s}
- {w : the length of w is at most 5}
- {w : w is any string except 11}

For more examples to try - see Sipser exercise 1.6, page 84

Limitations of DFAs

Is the language $\{0^n1^n \mid 0 \leq n \leq 4\}$ regular?

Is the language $\{0^n1^n \mid 0 \leq n \leq 999999999\}$ regular?

Now consider the language

$$\{0^n1^n \mid n \geq 0\} = \{\epsilon, 01, 0011, 000111, \ldots\}$$

Intuitively we cannot build a DFA to recognise this language, because a DFA has no memory of its actions so far.

Next lecture: a useful technique for showing (some) languages are not regular

Link between DFAs and TMs?

3.12 A ***Turing machine with left reset*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R}, \mathrm{RESET}\}.$$

If $\delta(q, a) = (r, b, \mathrm{RESET})$, when the machine is in state q reading an a, the machine's head jumps to the left-hand end of the tape after it writes b on the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

3.13 A ***Turing machine with stay put instead of left*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R}, \mathrm{S}\}.$$

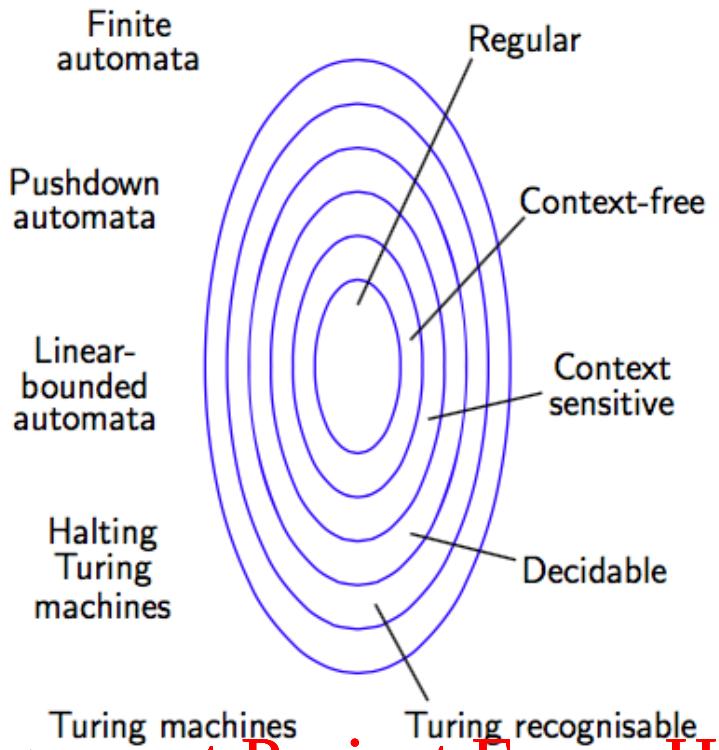
At each point, the machine can move its head right or let it stay in the same position. Show that this Turing machine variant is *not* equivalent to the usual version. What class of languages do these machines recognize?

Sipser, p 189

Reflection

- Convince yourself that the Turing machines described in Exercise 3.13 (on the previous slide) are just deterministic finite automata in 'disguise' - ie the language recognised by every such machine is regular, and every regular language can be recognised by such a machine.

Hierarchy of machines and languages



COMP90057 Advanced Theoretical Computer Science

Lecture 4: More computational models; Counter Machines; Conway's Game of Life

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 4

Second Semester, 2017

Recap

So far..

- ???

Regular Languages

Recall: A language is called regular if there is a finite automaton that recognises it.

Theorem: A language is regular iff there is a regular expression that describes it.

Pumping Lemma for Regular Languages

A great tool for proving languages non-regular.

Informally, it says that if we have a regular language A and consider a sufficiently long string $s \in A$, then a recogniser for A must traverse some loop to accept s. So A must contain infinitely many strings exhibiting repetition of some substring in s.

Pumping Lemma:

If A is regular then there is a number p such that for every string $s \in A$ with $|s| \geq p$, s can be written as $s = xyz$, satisfying

1. $xy^i z \in A$ for all $i \geq 0$
2. $y \neq \epsilon$
3. $|xy| \leq p$

p is called the pumping length.

– Proof of the Pumping Lemma (Sipser Section 1.4) is not examinable –

Pumping Lemma can be used to show...

- $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.
- $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$ is not regular.
- $D = \{ww \mid w \in \{0,1\}^*\}$ is not regular.
- $E = \{0^i 1^j \mid i > j\}$ is not regular.

Using the Pumping Lemma

Notation: $\exists$ "there exists"; $\forall$ "for all".

The pumping lemma says:

$$A \text{ regular} \Rightarrow \exists p \forall s \in A : s \text{ can be written } xyz \text{ such that } \ldots$$

We can use proof by contradiction* to show that a language is non-regular:

We suppose A is regular, but that "$\exists p \forall s \in A : s$ can be written xyz such that ..." is false

The falseness of the blue text is expressed $\forall p \exists s \in A : s$ cannot be written xyz such that ...

Given a p, coming up with such an s is sometimes easy, sometimes difficult.

* *Sipser Section 0.4*

Pumping Lemma Example 1

We prove that $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.

Suppose it is, and let p be the pumping length.

Consider $s = 0^p 1^p \in B$. Clearly s has length greater than p.

By the pumping lemma, $0^p 1^p = xyz$, with $xy^i z$ in B for all $i \geq 0$.

But y cannot consist of all 0s, since $xyyz$ then has more 0s than 1s.

Similarly y cannot consist of all 1s.

And if y has at least one 0 and one 1, then some 1 comes before some 0 in $xyyz$.

So if we assume that B is regular, we arrive at a contradiction.

Pumping Lemma Example 2

$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$ is not regular.

Suppose it is, and let p be the pumping length.

Consider $s = 0^p1^p \in C$. Clearly s has length greater than p.

Then, by the pumping lemma .....

Pumping Example 3

Consider $D = \{ww \mid w \in \{0,1\}^*\}$

Counter machines - description

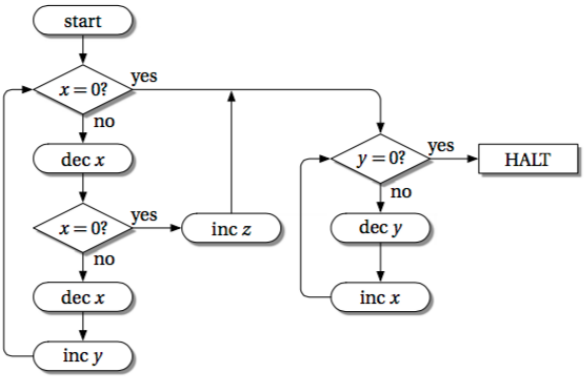
A counter machine has a finite number of internal states, and access to a finite number of integer counters. The only thing it can do to these counters is increment or decrement them, and the only question it can ask about them is whether or not they are zero.

The machine receives its input through the initial state of one of these counters, and like the Turing machine, it enters a HALT state when it is done.

Think of a counter machine as a flowchart or a program in a miniature programming language. Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are **inc** (increment), **dec** (decrement), and conditionals **if** $x = 0$

'Turing Machine Universality of the Game of Life,' Paul Rendell
Section 2.3, (copy available on the LMS Readings Online page)
also 'The Nature of Computation', C. Moore and S. Mertens, Oxford Press 2011, Section 7.6.1

Counter machines - example



This machine uses three counters, x , y , and z.
If their initial values are (n , 0, 0), it transforms them to ($\lfloor n/2 \rfloor$, 0, n mod 2).

It uses y as temporary storage for $\lfloor x/2 \rfloor$, first incrementing y for every two times it decrements x, and then incrementing x and decrementing y at the same rate.

It computes $2x$ by incrementing y twice each time it decrements x, and then feeding y back into x.

Moore and Mertens, Figure 7.10

Counter machines - results

Theorem

Any Turing machine can be simulated by a three-counter machine.

Theorem (Minsky, 1967)

Any Turing machine can be simulated by a two-counter machine.

Moore and Mertens, Chapter 7

Counter machines - observation

If a machine only has a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of steps.

But if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

Turing showed that this is the case even if this memory has a simple structure, and the machine can only observe and modify it through a very narrow window – namely, the symbol at its current location on the tape.

The counter machine shows we can make the memory even simpler.

But counter machines are **extremely** inefficient. Using the simulations from the undecidability proofs of Moore and Mertens Chapter 7, if there are n symbols on the TM tape, it takes the three-counter machine about 2^n steps to move the head left or right. The two-counter machine is even worse, taking as many as 5^{2^n} steps. Thus an algorithm that a Turing machine can run in polynomial time could take doubly-exponential time.

Moore and Mertens, p 265-266

Conway's Game of Life - Rules

The universe is an infinite two-dimensional grid of square cells, each of which is in one of two possible states, live or dead.

Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent.

At each step in time:

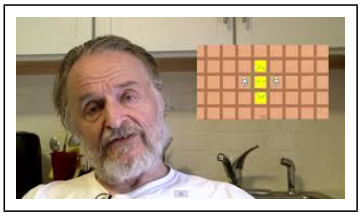
- Any live cell with fewer than two live neighbours dies, as if by needs caused by underpopulation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours cells will come to life.

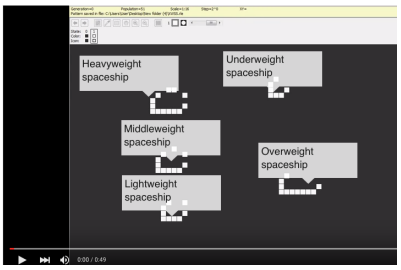
Initial pattern – the 'seed'. Each generation created by applying the rules simultaneously to every cell - i.e. births and deaths happen simultaneously.

e.g. http://www.bitstorm.org/gameoflife/

e.g. http://www.emergentuniverse.org/#/life

The Game of Life 'community'





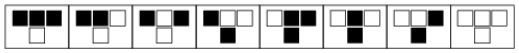
www.youtube.com/watch?v=sQDH-4XB-Ow
accessed Sept 16, 2015

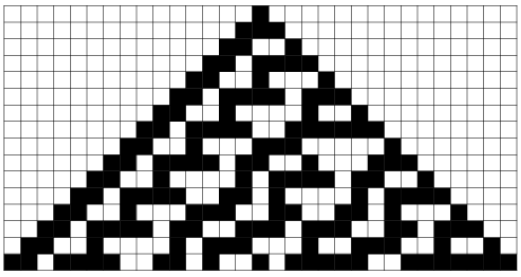
Did you know...

- ...that the pentadecathlon and the blinker are the only known oscillators that are polyominos in more than one phase?
- ...that it is impossible for a period 3 oscillator to be a phoenix?
- ...that the methuselah with the longest known lifespan, 40514M, lasts for over 40,000 generations before stabilizing? The second-place holder, Fred, runs for over 35,000 ticks.
- ...that replicators with quadratic population growth are known to exist in Conway's Game of Life, but none have yet been found?
- ...that the first known period 37 and 51 oscillators were found in 2009?
- ...that a pattern whose population grows without bound but does not tend to infinity is known as a sawtooth?
- ...that there are over 6.5 million distinct strict still lifes with 24 or fewer cells?
- ...that some infinitely-growing patterns can be constructed with as few as three gliders?
- ...that quadratically-growing patterns have been found with as few as 23 initial cells?
- ...that the blinker is the only known oscillator that is one cell thick?

www.conwaylife.com/wiki/Main_Page
accessed Sept 14, 2015

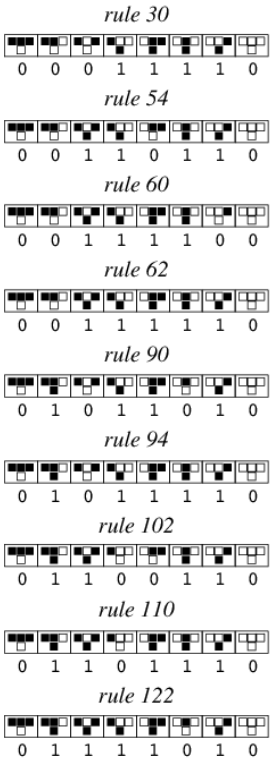
One-dimensional cellular automata

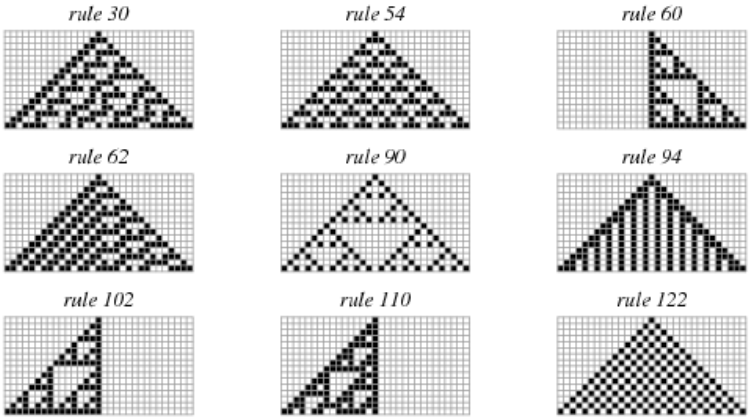
- Evolution completely described by a table specifying the state a given cell will have in the next generation based on the value of the cell to its left, the value the cell itself, and the value of the cell to its right
- 256 such automata
- rule 30 
- Note that decimal 30, in binary is 11110 . How is this relevant?
- after applying rule 30 for 15 generations, starting with a single black cell



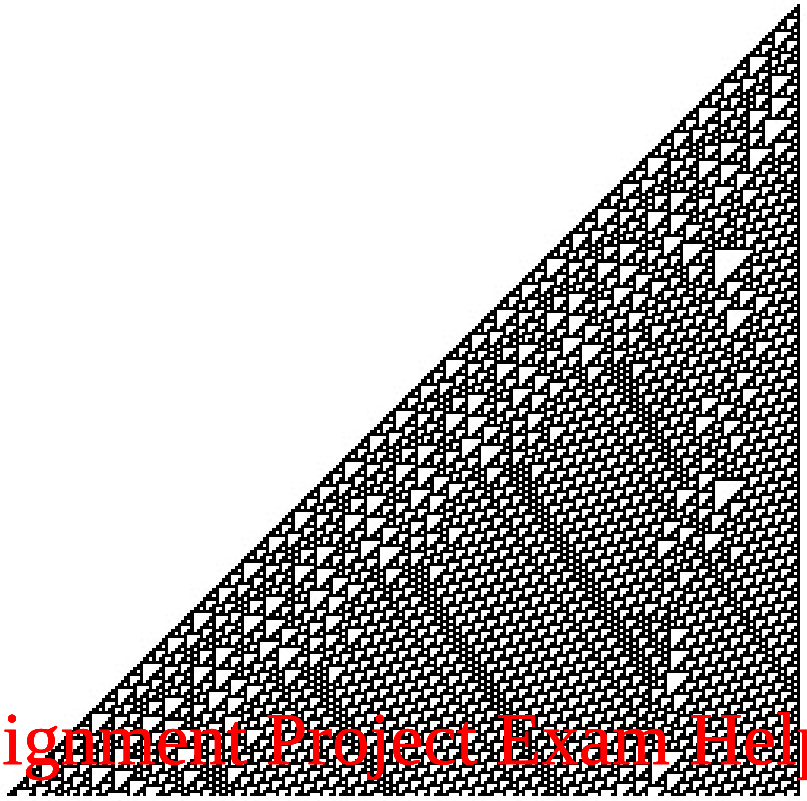
http://mathworld.wolfram.com/ElementaryCellularAutomaton.html
https://plato.stanford.edu/entries/cellular-automata/

Several rules - and 15 generations from a single 'on' cell





Rule 110 - 250 generations, starting with a single 'on' cell



The Game of Life - computational power

Conway established the Game of Life is computationally as powerful as Turing Machines computability, in his 1982 book *Winning Ways for Your Mathematical Plays: Volume 2*

- Conway's method uses counter machines (and we know they have infinite storage to work with, and that they have the same computational power as TMs)
  - The value of a counter can be represented by distance
  - A *block* pattern could be shifted along a diagonal with *gliders* (simulating increment and decrement), and that it was possible to detect when a block was in its base position (simulating a test for $\mathbf{0}$)
- He shows any counter machine can be simulated by a Game of Life automaton

– The structure and details of the proof are not examinable –

Rendell, Section 2.4 (pp 15-16)

For next lecture

- ???

COMP90057 Advanced Theoretical Computer Science

Lecture 5: Universal Turing Machines

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 5

Second Semester, 2017

Recap

Last week...

- ???

A Universal Turing Machine - (very) high-level description

It is possible to construct a universal Turing machine U that is able to simulate a (given) Turing machine.

On input $\langle M, w \rangle$, U simulates M on input w.

If M enters its accept state, U accepts.

If M enters its reject state, U rejects.

If M never halts, neither does U.

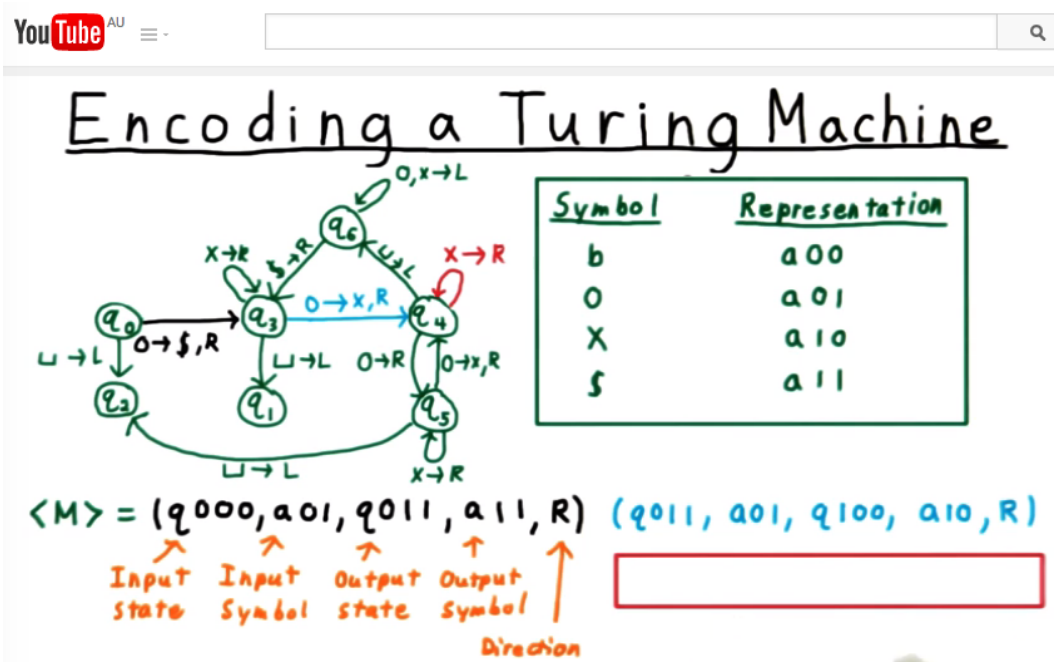
The existence of such a machine is critical to Turing's approach to proving the undecidability of certain problems - see his description of a UTM on pp 241-246 of his 1936 paper (available LMS-Subject Resources).

Encoding TMs as strings

https://www.youtube.com/watch?v=ZSqsXGjMUd8 (4 minutes)

Source: Encoding a Turing Machine - Georgia Tech - Computability, Complexity, Theory: Computability

Examples



Source: https://www.youtube.com/watch?v=znmk2a3pnQc

Universal Turing Machines - high-level descriptions

Sipser, page 202

`https://www.youtube.com/watch?v=C9T6iQYCW10`

Source: Building a Universal Turing Machine - Georgia Tech - Computability, Complexity, Theory: Computability

`https://www.youtube.com/watch?v=uM_Q-btwGms`

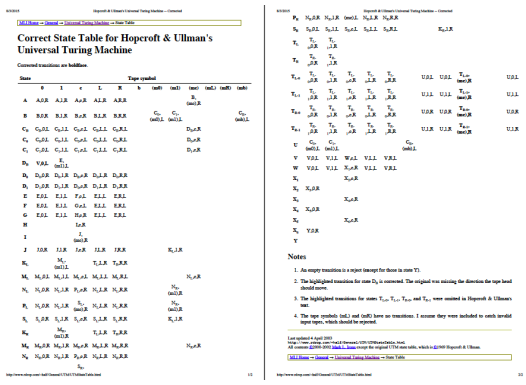
Source: Lecture 35/65 - The Universal Turing Machine - Georgia Tech - Computability, Complexity, Theory: Computability

pp 241-246 of Alan Turing's 1936 paper

Available: LMS-Subject Resources

A Universal Turing Machine - an implementation description

An oft-cited UTM description appears in the 1969 edition of the textbook Formal Languages and Their Relation to Automata by Hopcroft and Ullman. This UTM has 40 states and 12 tape symbols.

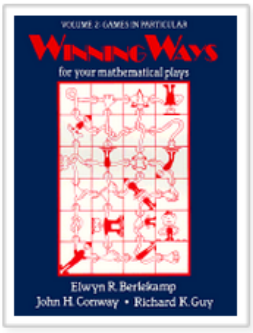


Source:`http://www.rdrop.com/~half/General/UTM/index.html`
State table at `http://www.rdrop.com/~half/General/UTM/UTMStateTable.html`
Table encoding at `http://www.rdrop.com/~half/General/UTM/ImplementationDetails.html`

A Universal Turing Machine - a 50 year quest for simplicity

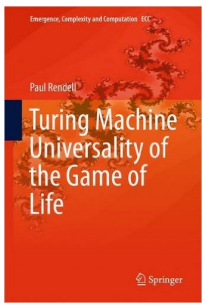
- Finding small UTMs has been actively pursued for some time. In 1962, Marvin Minsky found a small 7-state, 4-symbol universal machine; the record stood for 40 years until Stephen Wolfram in 2002 gave a 2-state, 5-symbol universal machine, and in 2007 a 2-state, 3 symbol machine was found: `http://blog.wolfram.com/2007/10/24/the-prize-is-won-the-simplest-universal-turing-machine-is-proved/`
- `http://arxiv.org/abs/1110.2230` - a survey paper on Universal Turing Machines published in 2011 *"The complexity of small universal Turing machines: a survey"*, by Turlough Neary and Damien Woods.
  - This paper is only to *skim* not read - but if you are interested, the *Introduction* gives the flavour of work that has been done in the last 50 years, and pointers to other sources.

The Game of Life - Universality



Berlekamp, Conway & Guy, 1982

(vol 2, chap 25)



Rendell

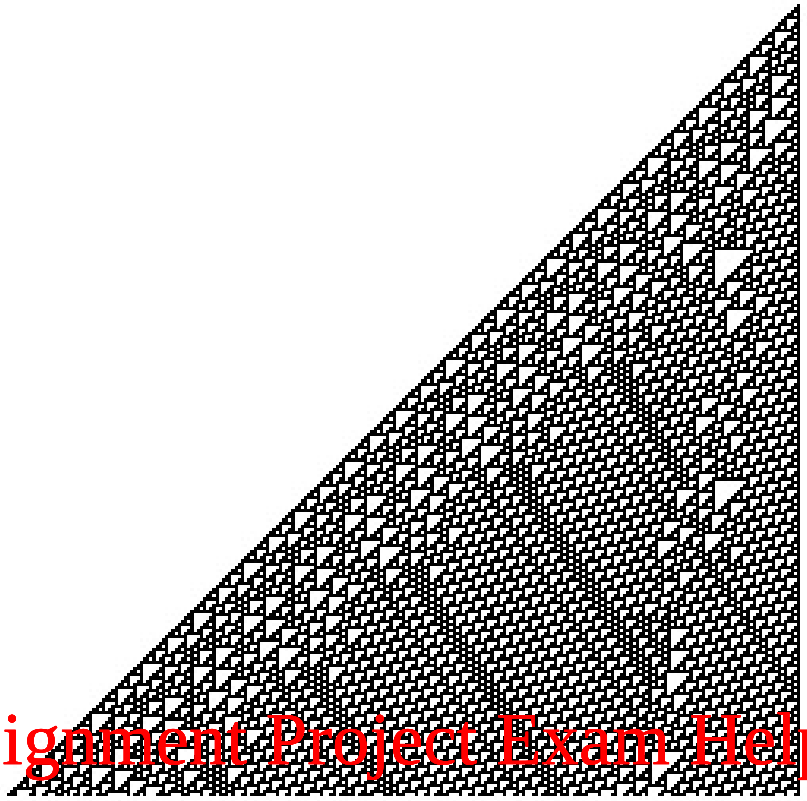
July 2015

`http://rendell-attic.org/gol/`

`https://www.youtube.com/watch?v=My8AsV7bA94`

Rendell built and ran UTMs illustrating their impracticality. For example, with TM M_1 for unary multiplication, on input (4, 4), i.e. computing $4*4$, M_1 took 443 Turing machine cycles, U_1 simulating M_1 took 57,108 cycles, and the Game of Life simulation of U_1 simulating M_1 took just over 1,700 million Life generations. (Rendell's thesis, sec 11.2.2.3)

Rule 110 - 250 generations, starting with a single 'on' cell



Rule 110 - a universal cellular automaton

Conjectured by Stephen Wolfram in 1986, and published in 2004 by Matthew Cook `http://www.complex-systems.com/pdf/15-1-1.pdf`

Cook showed that the Rule 110 cellular automaton can run a simulation of a cyclic tag system* that is running a simulation of a conventional tag system* that is running a simulation of a universal Turing Machine.

Not an efficient way to achieve universal computation, but a remarkable property for such a simple-looking cellular automaton. (See also `mathworld.wolfram.com/UniversalCellularAutomaton.html`)

* Tag systems are not covered in this subject - they are (yet) another model of computation - `https://en.wikipedia.org/wiki/Tag_system` - machines with finitely many states, an unbounded one-dimensional tape - and a mode of operating that involves deletions (from the start) and additions (at the end) of the symbols on the tape. Invented by Emil Post in around 1920, but not published until 1943; in 1961 Marvin Minsky proved that any Turing machine may be represented as a tag system - establishing their universality, and undecidability of the halting problem for tag systems.

- For next lecture
  - Sipser pages 21-22 - Proof by Contradiction
  - Sipser pages 203-205 - Counting and diagonalisation

COMP90057 Advanced Theoretical Computer Science

Lecture 6: Using counting arguments

Liz Sonenberg & Elena Kelareva

Drawing on material prepared by Harald Søndergaard & Tony Wirth

Lecture 6

Second Semester, 2017

Preparation suggested for this lecture

- Sipser pages 21-22 - Proof by Contradiction
- Sipser pages 203-205 - Counting and diagonalisation

This week

Today: establishing the existence of undecidable problems
Next week: in depth with undecidable problems

Notation and Terminology

$X \times Y$	Cartesian product of X and Y (the set of all pairs (x, y), $x \in X, y \in Y$)
X^k	Set of all k-tuples $(x_1, \ldots, x_k)$, $x_i \in X$
X^*	Set of all finite sequences of elements of X
$\mathcal{F}(X)$	Set of all finite subsets of X
$\mathcal{P}(X)$	Powerset: Set of all subsets of X
$X \to Y$	Set of all total functions from X to Y

If f is a function from X to Y: f is called *one-to-one* if it never maps two different elements to the same value, ie if whenever $x \neq y$ then $f(x) \neq f(y)$; f is called *onto* if it 'hits' every element of Y, ie if for every $y \in Y$ there is some $x \in X$ such that $f(x) = y$; f is called *bijective* if it is both one-to-one and onto.

The cardinality of a set is the number of elements in that set

Aside: 'one-to-one' is sometimes referred to as injective, and 'onto' is sometimes referred to as surjective.

Finite Cardinality

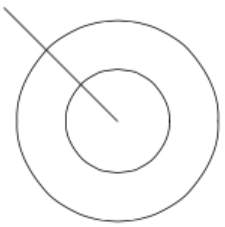
Let $B = \{false, true, maybe\}$.

Let $D = \{north, east, south, west\}$.

	With card(B)=3 and card(D) = 4	With card(B)=3 and card(D) = n
$card(B \times D)$	12	$3n$
$card(D^k)$	4^k	n^k
$card(\mathcal{P}(D))$	$2^4 = 16$	2^n
$card(D \to B)$	$3^4 = 81$	3^n
$card(D \to D)$	$4^4 = 256$	n^n

Trouble with Infinity

Galileo (1564-1642): The outer circle has twice as many points as the inner circle, as the ratio between the circumferences is 2. Yet considering radial lines suggests a one-to-one relationship.



Galileo's 'paradox': $\mathbb{N}$ and the set $\mathbb{SQ}$ of perfect squares are in a one-to-one relation:

$$\begin{array}{ccccccccc} \{ & 1 & 2 & 3 & 4 & 5 & \ldots & \} \\ & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & & \\ \{ & 1 & 4 & 9 & 16 & 25 & \ldots & \} \end{array}$$

Paradox? Some numbers are squares, while others are not; therefore, all the numbers, including both squares and non-squares, must be more numerous than just the squares. And yet, for every square there is exactly one positive number that is its square root, and for every number there is exactly one square; hence, there cannot be more of one than of the other.

Cantor's Criterion

So what do 'equals' and 'less' mean for infinite cardinality?

Cantor (1845-1918):

- $card(X) \leq card(Y)$ if there is a total, injective $f : X \to Y$.
- $card(X) = card(Y)$ if

$$card(X) \leq card(Y) \text{ and } card(Y) \leq card(X).$$

As a consequence, there are (infinitely) many levels of infinity.

Countably infinite

$\mathbb{N}$ denotes the set $\{1, 2, 3, ...\}$ - sometimes termed the *natural numbers*.

$\mathbb{Z}$ denotes the set $\{... -3, -2, -1, 0, 1, 2, 3, ...\}$ of all integers.

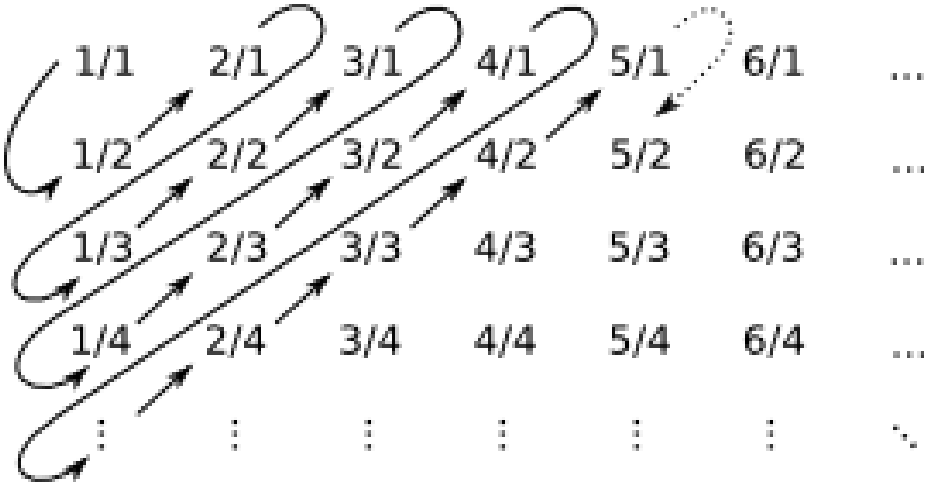
$\mathbb{Q}$ denotes the set of positive rational numbers $\{m/n\colon\ m, n \in \mathbb{N}\}$? .

We say X is countable if $card(X) \leq card(\mathbb{N})$.

We say X is countably infinite if $card(X) = card(\mathbb{N})$.

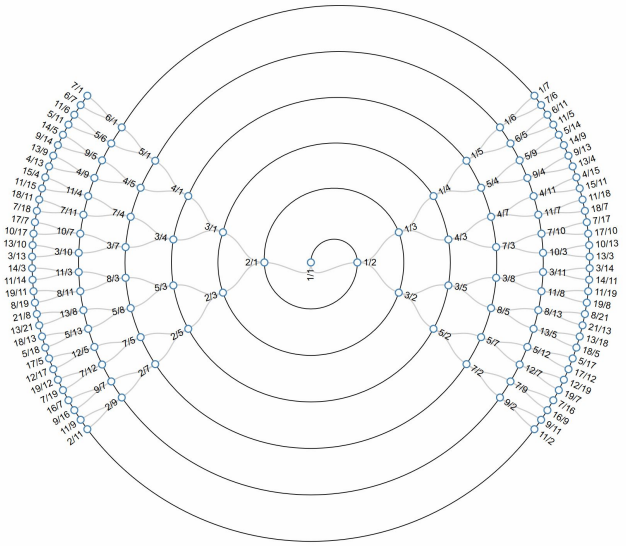
Examples: $\mathbb{Z}$ is countably infinite because ...

What about $\mathbb{Q}$?



This is a somewhat unsatisfying treatment: (a) it is not an explicit sequence - does it have an algebraic specification? and (b) numbers appear more than once.

Another view of $\mathbb{Q}$?



Not examinable

An enumeration in which every rational number appears exactly once, e.g. `https://www.jasondavies.com/calkin-wilf-tree/`.
Each vertex a/b has two children: $(a+b)/b$ and $a/(a+b)$. Traversing the tree breadth-first results in the CalkinWilf sequence, depicted by the spiral above.

Bigger than countably infinite - a diagonalisation argument

$\mathbb{R}$ denotes the set of all numbers that have decimal representations - generally termed the *real numbers*

Theorem: There is no bijection $h : \mathbb{N} \to \mathbb{R}$. (i.e. $\mathbb{R}$ is uncountable)

Proof: Assume h exists. Then $h(1), h(2), \ldots, h(n), \ldots$ contains every $r \in \mathbb{R}$, without duplicates. Construct $r' \in (0, 1)$ as follows: Its k'th digit is

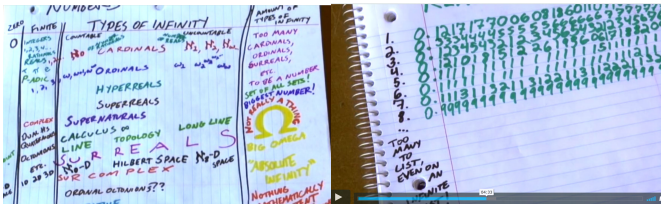
'1' if the k'th digit (after the decimal point) of $h(k)$ is not 1
'2' if the k'th digit (after the decimal point) of $h(k)$ is 1

Then $r' \neq h(k)$ for all k, a contradiction.

$h(1)$	14.	4	4	3	7	9	$\ldots$
$h(2)$	2.	6	1	6	9	3	$\ldots$
$h(3)$	87.	0	0	0	0	0	$\ldots$
$h(4)$	7.	7	7	7	7	6	$\ldots$
$\vdots$							

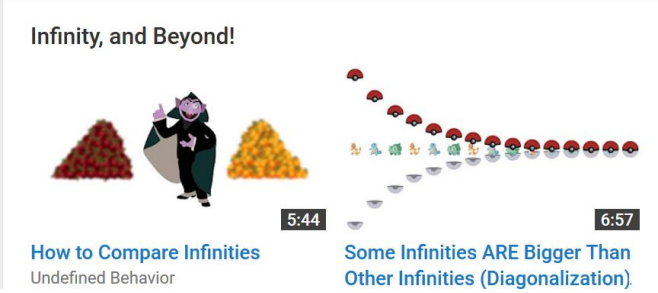
Recap ...

from `vihart.com`



`https://vimeo.com/147535698`, `https://vimeo.com/147535705`

from *Undefined Behavior*



`https://www.youtube.com/channel/UCZ4oRX0e157gHeJHp0XOULA/videos`

How many Turing Machines?

The number of possible Turing machines is countably infinite.

A Turing machine involves a finite set of states and a finite set of transitions between states over a finite alphabet.

We have seen how to encode (or 'flatten') each machine into one finite-length string that describes it.

We can place these strings into a one-to-one association with the positive integers, just as we can with rational numbers.

How many Languages?

The number of possible languages is uncountable.

Consider a finite alphabet Σ. A language L is a subset of Σ^*.

What is the size of Σ^*?

How many subsets are there of Σ^*? i.e. what is the cardinality of $\mathcal{P}(\Sigma^*)$

We will show $\mathcal{P}(\Sigma^*)$ is uncountably infinite by showing that $\mathcal{P}(\mathbb{N})$ is uncountably infinite.

(Given the bijection that exists between $\mathcal{P}(\mathbb{N})$ and $\mathcal{P}(\Sigma^*)$ because of the bijection that exists from $\mathbb{N}$ to Σ^*, it is sufficient to show that $\mathcal{P}(\mathbb{N})$ is uncountably infinite.)

There are uncountably many languages over a finite alphabet

Proof: Argue by contradiction:

Suppose $\mathcal{P}(\mathbb{N})$ is countably infinite. Then all the subsets of $\mathbb{N}$ can be listed $A_0, A_1, A_2, ..$, i.e. every subset of $\mathbb{N}$ is A_i for some i.

Now consider the set $B = \{i \geq 0 \text{ and } i \notin A_i\}$ which contains those integers i that are not members of their namesake set A_i.

But B is a subset of $\mathbb{N}$, so $B = A_j$ for some specific j.

Is that specific j in B?

By the definition of B, if $j \in B$, then $j \notin A_j$. But as $A_j = B$, that means $j \notin B$.

Similarly, one can argue that if if $j \notin B$, then $j \in A_j$. But as $A_j = B$, that means $j \in B$.

So we have $j \in B$ iff $j \notin B$, i.e. a contradiction. so $\mathcal{P}(\mathbb{N})$ is uncountable.

How many languages?

In an earlier lecture you were 'told' (i.e. did not see the proof) that a specific problem, Hilbert's Tenth Problem, is undecidable.

We can also use the existence of multiple infinities to prove that there are undecidable languages - without that proof providing any specific examples.

The number of possible Turing machines is countable (i.e. the smallest infinity).

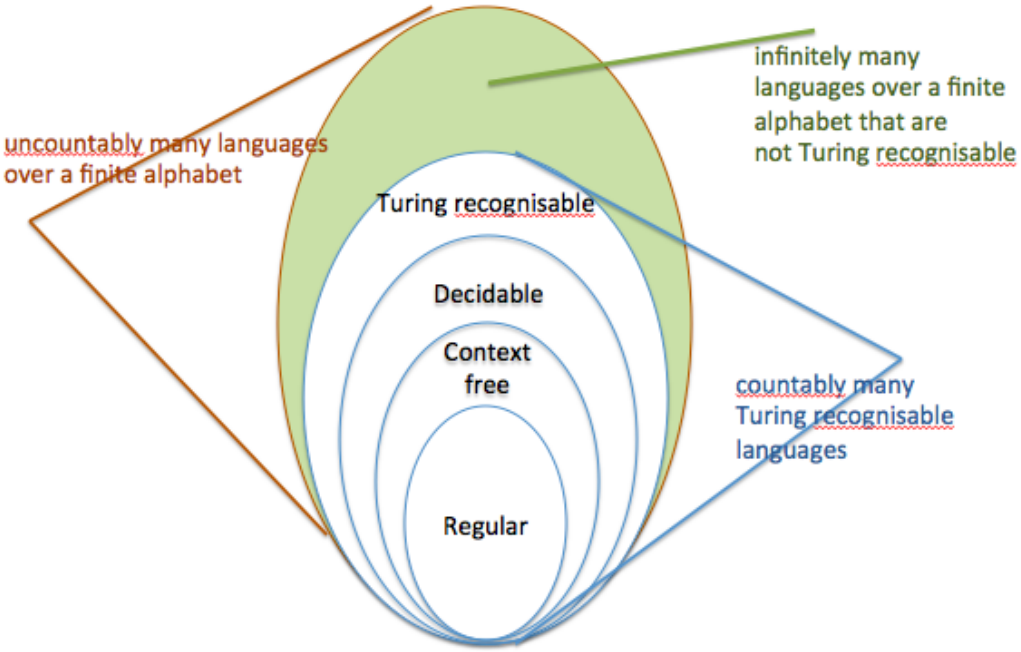
The number of languages, on the other hand, is a greater infinity: namely, the infinity of $\mathbb{R}$.

Therefore there are (infinitely many) more languages than there are Turing machines to solve them.

And, as we'll see shortly, some of these languages are ones we care about.

Sipser, pp 202-206

Infinitely many languages are not Turing-recognisable



Recall: Computable numbers

From Lecture 3

Computable numbers

- Turing's motivation for the 1936 paper was to solve a problem posed by mathematician David Hilbert in 1900 known as the *Entscheidungsproblem* - the existence of a formalised procedure to determine the provability of statements about mathematics
- The bulk of the paper is about computable numbers, and the distinction between real numbers and computable numbers is crucial to Turing's argument
- A computable number **r** is one for which there is a TM that, given any input n on its initial tape, successively prints the first n digits of the decimal representation of **r** on its tape, then halts.
- The key notions in the definition are:
  - any n can be provided as input; and
  - for any provided n, the computation only takes a finite number of steps.

How many computable numbers?

- The set of computable numbers has the same cardinality as the set of TMs...
- ... and the cardinality of the set of computable numbers is strictly smaller than the cardinality of the set of real numbers
- i.e. there are infinitely many computable numbers and there are infinitely many uncomputable numbers

Uncomputable numbers

So we know there are infinitely many uncomputable numbers.

But this is an existence proof, not (of course!) a simple construction of an uncomputable number...

The question remains - are there 'interesting' non-computable numbers?


Gregory Chaitin

www-2.dc.uba.ar/profesores/becher/ns.html 'The Omega Man' *N*ew Scientist, 2001
https://plus.maths.org/content/omega-and-why-maths-has-no-toes
Gregory Chaitin, "Omega and why maths has no TOEs"

Chaitin's Ω - provably uncomputable

Consider a Universal TM, U, define $\Omega_U = \sum_{U(p)\ halts} 2^{-|p|}$ where $|p|$ is the size in bits of TM program p

Each Ω_U is a real number between 0 and 1 (some delicate maths is used to show the sum converges).

Informally, Ω_U is the probability that a randomly selected TM halts.

Ω_U has a rigorous definition, yet is uncomputable: we can compute (only) finitely many of its digits with certainty - for the rest we can do no better than random

Watch `https://www.youtube.com/watch?v=NOJmzE539F0` to (at least) the first 5mins 30 sec

`http://hdl.handle.net/2292/3654`
Cris Calude et al, "Computing 80 initial bits of a Chaitin Omega Number"

Read for next week

- Sipser Section 4.2 pages 201-202, 207-209 - with the focus being the proof of Theorem 4.11