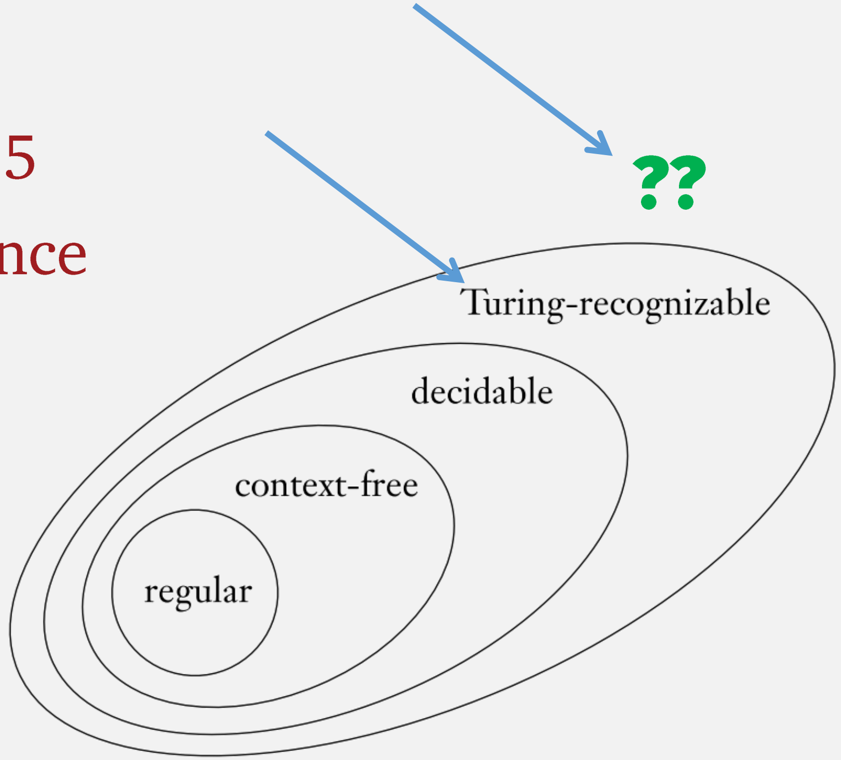
CS 420 / CS 620

Undecidability

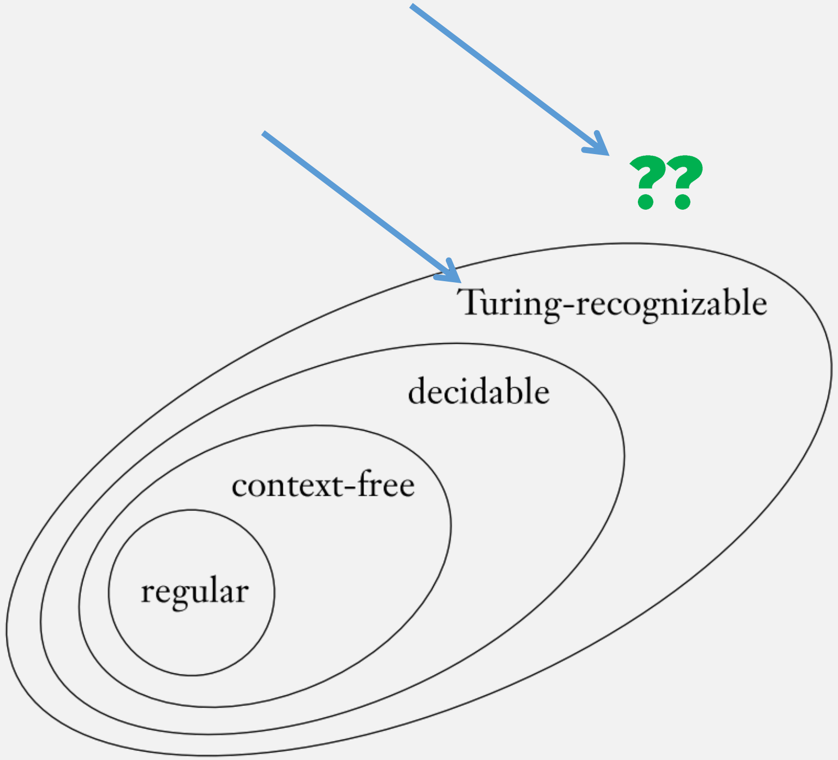
Monday, November 17, 2025

UMass Boston Computer Science

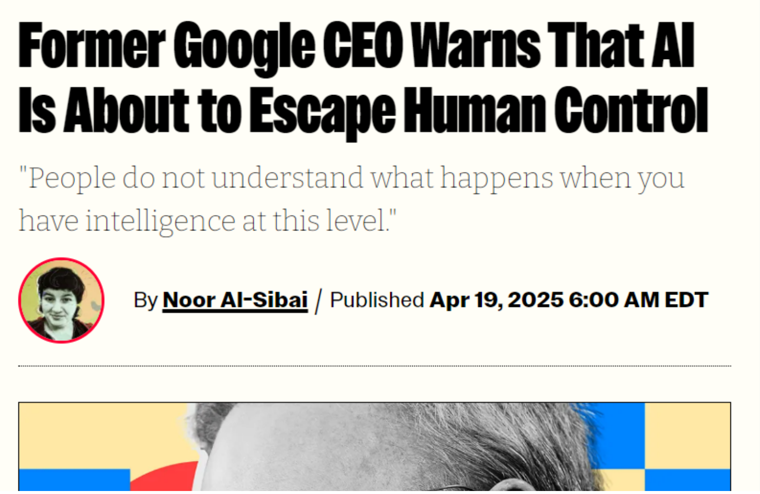


Announcements

- HW 10
  - ~~Due: Mon 11/17 12pm (noon)~~
- HW 11
  - Out: Mon 11/17 12pm (noon)
  - Due: Mon 11/24 12pm (noon)



Warning: AI is Taking Over Soon



Former Google CEO Warns That AI Is About to Escape Human Control

"People do not understand what happens when you have intelligence at this level."

By **Noor Al-Sibai** / Published **Apr 19, 2025 6:00 AM EDT**

‘Godfather of AI’ shortens odds of the technology wiping out humanity over next 30 years

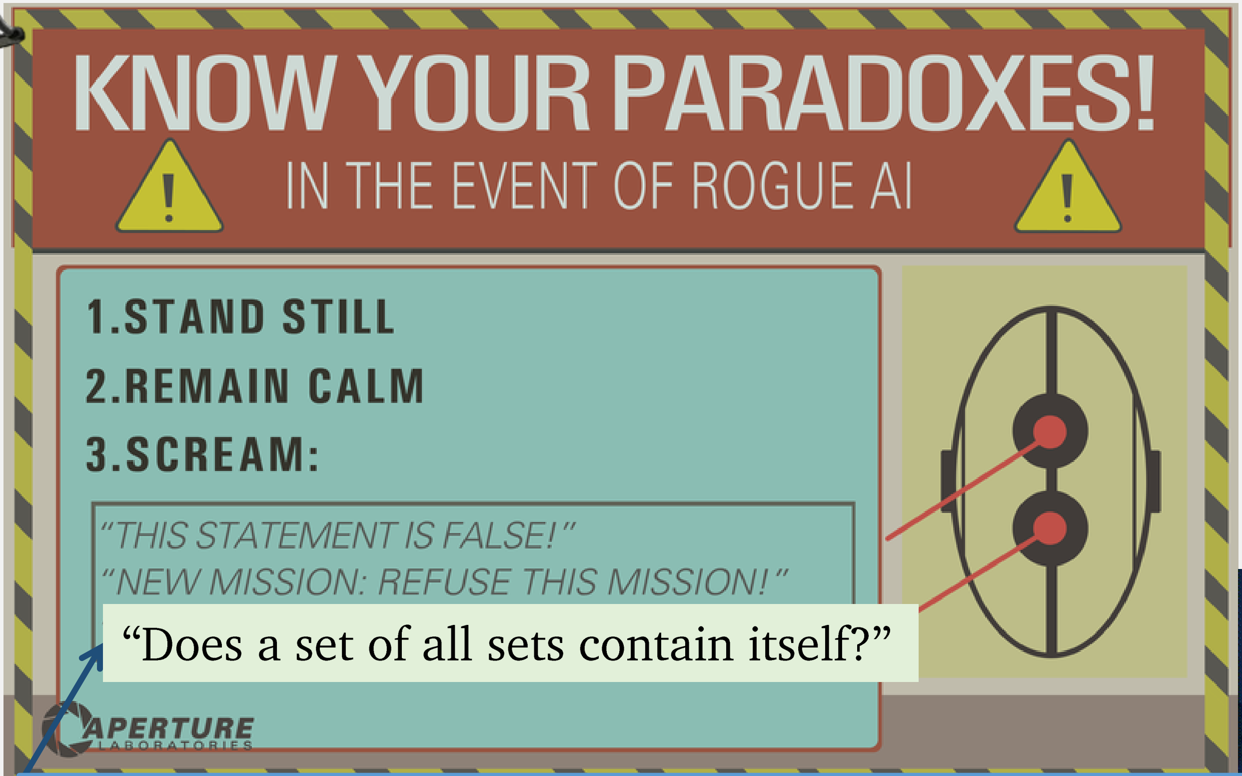
Geoffrey Hinton says there is 10% to 20% chance AI will lead to human extinction in three decades, as change moves fast

- ‘We need dramatic changes’: is societal collapse inevitable?

Dan Milmo *Global technology editor*

Fri 27 Dec 2024 10.50 EST

There’s Hope (If You Pay Attention Today)

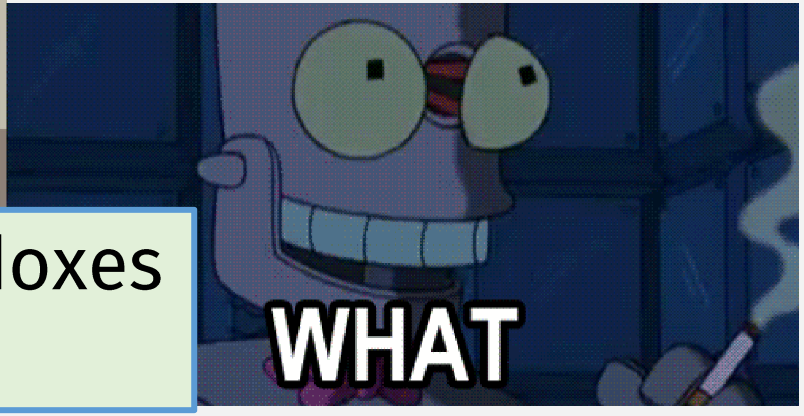


“Does a set of all sets contain itself?”

Magritte’s *“This Is Not a Pipe”* (1929)

Bertrand Russell’s Paradox (1901)

<u>Today</u>: A **method** for **creating paradoxes** (used by **Russell** and others)



Language of: **DFA description** (i.e., "source code") + **string pairs**, i.e., <u>where</u> **DFA accepts the string**

Recap: Decidability of Regular and CFLs

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$ — Decidable
- $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$ — Decidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable

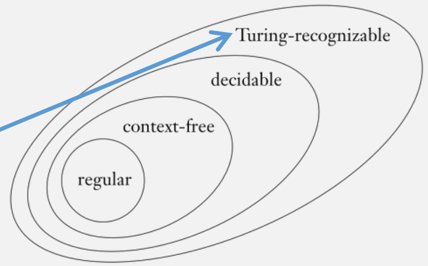
Compute something about **DFA** <u>language</u> (runtime behavior), from its <u>**description**</u> (source code)

- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable

Compute something about **CFG** <u>**language**</u> (runtime behavior), from its <u>**description**</u> (source code)

- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — *Undecidable?*
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — *Undecidable?*

<u>compute</u> whether a TM accepts a string



Thm: A_{TM} is Turing-recognizable

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

M can go into **infinite loop,** causing *U* to **loop**

U = **Implements TM computation steps** $\alpha q_1 \mathbf{a} \beta \vdash \alpha \mathbf{x} q_2 \beta$

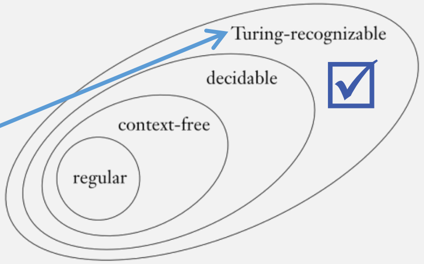
- i.e., "The Universal Turing Machine"
- "Program" simulating other programs (**interpreter**)
- <u>(Step 1)</u>: *U* **loops** when *M* **loops**

Termination argument?

So *U* is <u>not</u> a decider. Is it a **recognizer**?
i.e., is A_{TM} Turing-recognizable?

Need **Examples Table** to justify Statement: "TM *U* recognizes A_{TM}"

Thm: A_{TM} is Turing-recognizable



$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

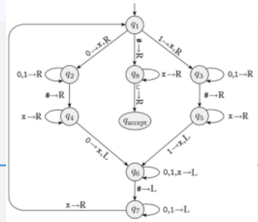
1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

M can go into **infinite loop**, causing *U* to **loop**

"called" machine

"Actual" behavior

"Expected" behavior



Let:
- M_1 = "*str#str*" lang decider
- M_{loop} = looping TM

Example Str	*M* on input *w*?	*U* result?	In A_{TM} lang?
<M_1, 01#01>	Accept	Accept	Yes
<M_1, 00#11>	Reject	Reject	No
<M_{loop}, ε>	Loop!	Loop!	No

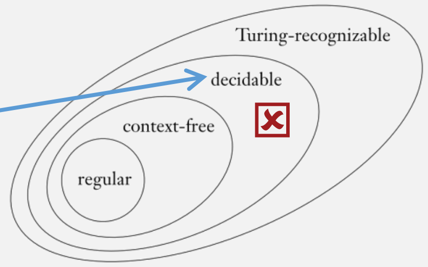
Columns must match!

Need **Examples Table** to justify Statement: "TM *U* recognizes A_{TM}"

Is this right?

Yes! Machine can loop for strings not in lang

How to prove … not in here?



<u>Thm</u>: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

- ???

Flashback

Prove: Demons do not exist

???

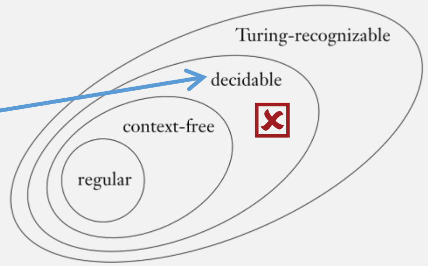
Proving something not true is **different** (and usually harder) **than proving it** true

It's sometimes possible, but **often needs** new proof techniques!

Example (**Regular** Languages)
Prove a language is **regular**:
- Create a DFA

Prove a language is **not regular**:
- Proof by contradiction using **Pumping Lemma**

Not in here?



<u>Thm</u>: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

<u>Example</u> (**decidable** languages)

Prove a language is **decidable:**

- Create a **decider** TM (with termination argument)

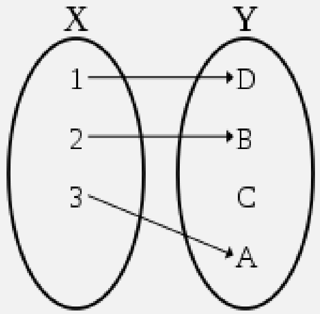
Prove a language is **not decidable:**

- ????

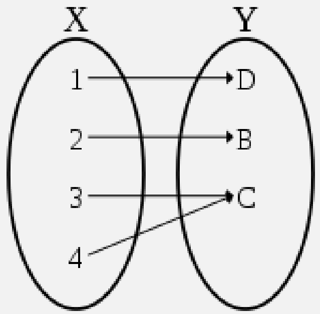
next

Kinds of Functions (a fn maps DOMAIN → RANGE)

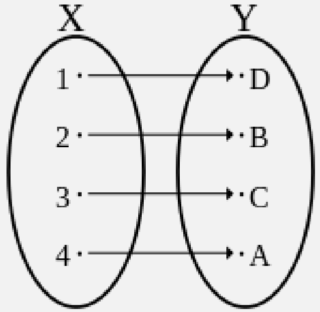
- **Injective,** a.k.a., “one-to-one”
  - Every element in DOMAIN has a unique mapping
  - How to remember:
    - Entire DOMAIN is mapped “**in**” to the RANGE



- **Surjective,** a.k.a., “onto”
  - Every element in RANGE is mapped to
  - How to remember:
    - “**Sur**” = “over” (eg, <u>sur</u>vey); DOMAIN is mapped “over” the RANGE



- **Bijective,** a.k.a., “correspondence” or “one-to-one correspondence”
  - Is both injective and surjective
  - Unique pairing of every element in DOMAIN and RANGE



Countability

- A set is "**countable**" if it is:
  - Finite
  - Or, there exists a **bijection** between the natural numbers (starting from 1) and the set
    - In this case, the set has the same size as the set of natural numbers
    - This is called "**countably infinite**"

Exercise: Which set is larger?

- The set of:
  - Natural numbers, or
  - Even numbers?

> Definition: a set S is **countably infinite**, i.e., it has the same size as the set of natural numbers, if there is a **bijection** between the natural numbers and S

- They are the same size! Both are **countably infinite**
  - Proof, key step: Bijection:

n	$f(n) = 2n$
1	2
2	4
3	6
$\vdots$	$\vdots$
Natural numbers	Even numbers

Every natural number:
- maps to a unique even number,
- and vice versa

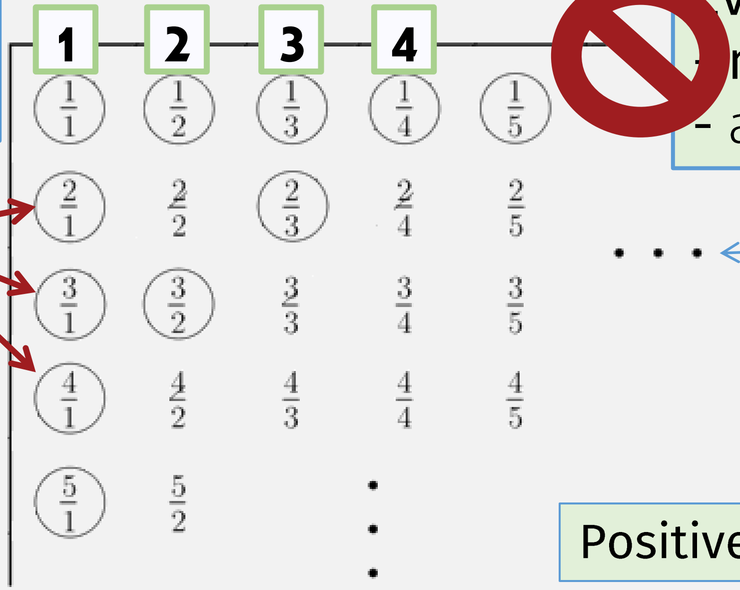
Exercise: Which set is larger?

Definition: a **set** S is **countably infinite**, i.e., it has the **same size as the set of natural numbers,** if there is a **bijection between the natural numbers** and S

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Positive rational numbers? $\mathcal{Q} = \{\frac{m}{n} | \, m, n \in \mathcal{N}\}$
- They are the same size! Both are **countably infinite**

A possible mapping (bijection) of Natural numbers to Positive rationals?

So these don't get mapped to:
(not a **bijection**)



Every natural number:
- maps to a unique rational,
- and vice versa

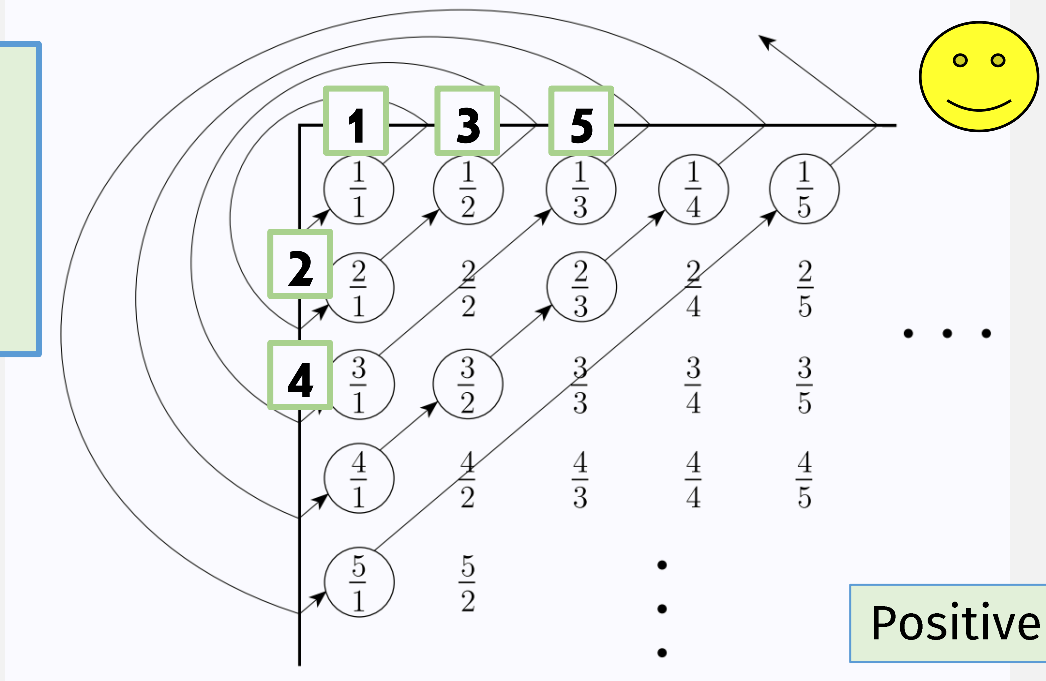
But, each row is infinite

Positive rational numbers

Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Positive rational numbers? $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$
- They are the same size! Both are **countably infinite**

Another mapping:
This is a **bijection** because
every natural number:
- maps to a unique fraction,
- and vice versa



Positive rational numbers

Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Real numbers? $\mathcal{R}$
- There are more real numbers. It is **uncountably infinite.**

This proof technique is called **diagonalization**

Proof, by contradiction:

Assume:
they are **same size**, i.e., **countably infinite**

- So: a **bijection between natural** and **real numbers exists.**
  - So: **every natural num maps to a unique real,** and **vice versa**
- But **we show that in any given mapping,** ... e.g.:
  - Some **real number** is **not mapped to** ...
  - E.g., a **number** that has **different digits** at **each position:**

different

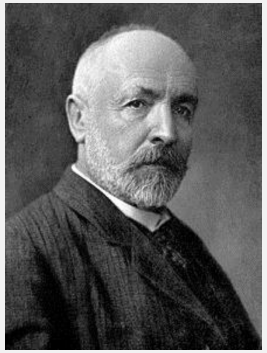
$$x = 0.\boxed{4}\boxed{6}\boxed{4}1\ \ldots$$

n	$f(n)$
1	$3.\boxed{1}4159\ldots$
2	$55.5\boxed{5}555\ldots$
3	$0.12\boxed{3}45\ldots$
4	$0.500\underline{0}0\ldots$
$\vdots$	$\vdots$

A hypothetical mapping

  - This **number** cannot be **in the mapping** ...
  - ... So we have a **contradiction**!

Georg Cantor



- Invented set theory
- Came up with **countable infinity** (1873)
- And **uncountability**:
  - Also: how to show uncountability with "**diagonalization**" technique

A formative day for Georg Cantor.

Diagonalization with Turing Machines

Diagonal: Result of **Giving a TM its own Encoding** as **Input**

All TM Encodings

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				

opposites

All TMs

Try to construct this: **“opposite” TM D**

TM D can’t exist!

What should happen here?

It must both accept and reject!

3 Easy Steps!

Thm: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Proof by contradiction:

1. Assume A_{TM} is decidable. So there exists a decider H for it:

$$H(\langle M, w\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

2. Use H to define another TM … the impossible "opposite" machine:

$D =$ "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

H computes: M's result with itself as input

Do the opposite

(does opposite of what input TM would do if given itself)

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
D	*reject*	*reject*	*accept*	*accept*		?	

(from prev slide)
This TM can't be defined!

3 Easy Steps!

Thm: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Proof by contradiction:

This cannot be true

1. Assume A_{TM} is decidable. So there exists a decider H for it:

$$H(\langle M, w\rangle) = \begin{cases} \mathit{accept} & \text{if } M \text{ accepts } w \\ \mathit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

2. Use H to define another TM … the impossible "opposite" machine:

~~D = "On input $\langle M\rangle$, where M is a TM:~~

~~**1.** Run H on input $\langle M, \langle M\rangle\rangle$.~~

~~**2.** Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."~~

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$	$\cdots$
M_1	<u>*accept*</u>	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	<u>*accept*</u>	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	<u>*reject*</u>	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	<u>*reject*</u>		*accept*	
$\vdots$		$\vdots$			$\ddots$		
~~D~~	~~*reject*~~	~~*reject*~~	~~*accept*~~	~~*accept*~~		~~?~~	
$\vdots$							$\ddots$

TM D can't exist!

3. But D does not exist! **Contradiction**! So the assumption is false.

Easier Undecidability Proofs

- **We proved** $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ **undecidable …**
- … by **contradiction:**
  - **Use hypothetical A_{TM} decider** to **create an impossible decider "D"!**

reduce "D problem" to A_{TM}

- Step # 1: **coming up with "D"** --- hard!
  - Need to invent **diagonalization**

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

- Step # 2: **reduce "D" problem to A_{TM}** --- easier!

- From now on: **undecidability proofs only need** step # 2**!**
  - And we now **have two "impossible" problems to choose from**

Let's add more!

The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

Thm: $HALT_{\mathsf{TM}}$ is undecidable

Proof, by **contradiction**:

reduce (from known **undecidable**) A_{TM} to $HALT_{\mathsf{TM}}$

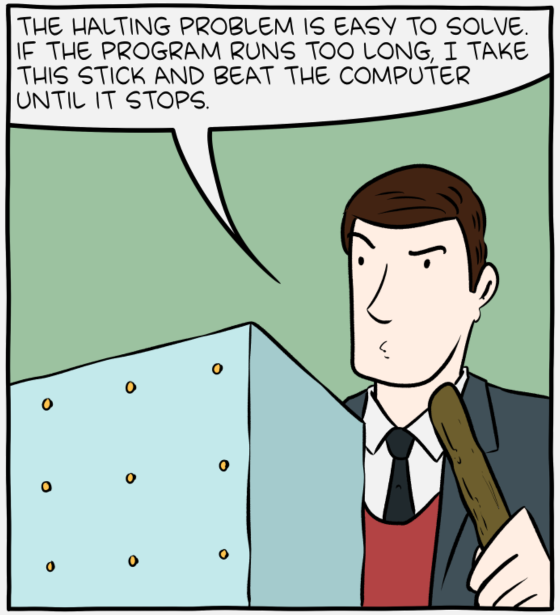
- Assume: $HALT_{\mathsf{TM}}$ has *decider R*; use it to create decider for A_{TM}:

  $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

- …

contradiction

- But A_{TM} is undecidable and has no decider!



What if Alan Turing had been an engineer?

The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w \rangle |\ M \text{ is a TM and } M \text{ halts on input } w\}$

Thm: $HALT_{\mathsf{TM}}$ is undecidable

Proof, by **contradiction**:

Using our hypothetical $HALT_{\mathsf{TM}}$ decider R

- Assume: $HALT_{\mathsf{TM}}$ has *decider* R; use it to create decider for A_{TM}:

$A_{\mathsf{TM}} = \{\langle M, w \rangle |\ M \text{ is a TM and } M \text{ accepts } w\}$

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w \rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

This means M loops on (and does not accept) input w

This step always halts

Termination argument:
Step 1: R is a decider so always halts
Step 3: M always halts because R said so

Undecidability Proof Technique #1: **Reduce** (directly) from A_{TM} (by **creating** A_{TM} **decider**)

The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

Thm: $HALT_{\mathsf{TM}}$ **is undecidable**

Proof, by **contradiction**:

- Assume: $HALT_{\mathsf{TM}}$ **has *decider* R; use it to create decider for A_{TM}:**

  $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

~~S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:~~

~~**1.** Run TM R on input $\langle M, w\rangle$.~~

~~**2.** If R rejects, *reject*.~~

~~**3.** If R accepts, simulate M on w until it halts.~~

~~**4.** If M has accepted, *accept*; if M has rejected, *reject*."~~

Now we have three "impossible" deciders to choose from

- But A_{TM} **is undecidable!** I.e., **this decider does not exist!**
  - So $HALT_{\mathsf{TM}}$ **is also undecidable!**

Interlude: Reducing from $HALT_{TM}$

A practical thought experiment …
… about **compiler optimizations**

Your compiler changes your program!

`If TRUE then A else B` → `A`

`1 + 2 + 3` → `6`

Compiler Optimizations

Optmization - docs

- -O0
  - No optmization, faster compilation time, better for debugging builds.
- -O2
- -O3
  - Higher level of optmization. Slower compile-time, better for production builds.
- -OFast
  - Enables higher level of optmization than (-O3). It enables lots of flags as can be seen src (-ffloat-store, -ffsast-math, -ffinite-math-only, -O3 …)
- -finline-functions
- -m64
- -funroll-loops
- -fvectorize
- -fprofile-generate

Types of optimization [edit]

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

Peephole optimizations

These are usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions.[2] For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself (this example is also an instance of strength reduction).

Local optimizations

These only consider information local to a basic block.[3] Since basic blocks have no control flow, these optimizations need very little analysis, saving time and reducing storage requirements, but this also means that no information is preserved across jumps.

Global optimizations

These are also called "intraprocedural methods" and act on whole functions.[3] This gives them more information to work with, but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed because little information about them is available.

Loop optimizations

These act on the statements which make up a loop, such as a *for* loop, for example loop-invariant code motion. Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.[4]

Prescient store optimizations

These allow store operations to occur earlier than would otherwise be permitted in the context of threads and locks. The process needs some way of knowing ahead of time what value will be stored by the assignment that it should have followed. The purpose of this relaxation is to allow compiler optimization to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs.[5]

Interprocedural, whole-program or link-time optimization

These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information, i.e. within a single function. This kind of optimization can also allow new techniques to be performed. For instance, function inlining, where a call to a function is replaced by a copy of the function body.

Machine code optimization and object code optimizer

These analyze the executable task image of the program after all of an executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression which saves space by collapsing common sequences of instructions, are more effective when the entire executable task image is available for analysis.[6]

The Optimal Optimizing Compiler

"Full Employment" Theorem

Thm: **The Optimal (C++) Optimizing Compiler does not exist**

Proof, by **contradiction**:

Assume: OPT is the **Perfect Optimizing Compiler**

Use it **to create** $HALT_{TM}$ **decider** (**accept**s $<M,w>$ if M halts with w, else **reject**s):

S = **On input** $<M, w>$, where M is **C++** program and w is string:

- If $OPT(M)$ == `for(;;)`
  a) Then **Reject**
  b) Else **Accept**

In computer science and mathematics, a **full employment theorem** is a term used, often humorously, to refer to a theorem which states that no algorithm can optimally perform a particular task done by some class of professionals. The name arises because such a theorem ensures that there is endless scope to keep discovering new techniques to improve the way at least some specific task is done.

For example, the *full employment theorem for compiler writers* states that there is no such thing as a provably perfect size-optimizing compiler, as such a proof for the compiler would have to detect non-terminating computations and reduce them to a one-instruction infinite loop. Thus, the existence of a provably perfect size-optimizing compiler would imply a solution to the halting problem, which cannot exist. This also implies that there may always be a better compiler since the proof that one has the best compiler cannot exist. Therefore, compiler writers will always be able to speculate that they have something to improve.

Summary: The Limits of Algorithms

- $A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ Decidable
- $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ Decidable
- $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ **Undecidable**
- $HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ **Undecidable**

Similar languages

It's straightforward to use hypothetical $HALT_{\mathrm{TM}}$ decider to create A_{TM} decider

Summary: The Limits of Algorithms

- $A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — **Undecidable**
- $HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ — **Undecidable**
- $E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- next $E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — **Undecidable**

Not as similar languages

How can we use a hypothetical E_{TM} decider to create A_{TM} or $HALT_{\mathsf{TM}}$ decider?

Undecidability Proof Technique #2

Reducibility: Modifying the TM

$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

Thm: E_{TM} is undecidable

Proof, by contradiction:

- Assume E_{TM} has *decider* R; use it to create *decider* for A_{TM}:

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

First, construct M_1

- Run R on input $\langle M_1\rangle$

Note: M_1 is only used as arg to R; we never run it!

- If R accepts, *reject* (because it means $\langle M\rangle$ doesn't accept w)
- if R rejects, then *accept* ($\langle M\rangle$ accepts something, and it is w !)

- Idea: Wrap $\langle M\rangle$ in a new TM that can *only* accept w (or nothing):

M_1 = "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

Input not w, always reject

Input is w, maybe accept

M_1 accepts w if M does

Reducibility: Modifying the TM

$E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

Thm: E_{TM} is undecidable

Proof, by **contradiction**:

Remember: A_{TM} **is undecidable** and thus **has no decider!**

- Assume E_{TM} has *decider* R; use it to create *decider* for A_{TM}:

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

First, construct M_1

- Run R on input $\langle M_1 \rangle$
- If R accepts, *reject* (because it means $\langle M \rangle$ doesn't accept w)
- if R rejects, then *accept* ($\langle M \rangle$ accepts something, and it is w !)

- Idea: Wrap $\langle M \rangle$ in a new TM that can *only* accept w:

M_1 = "On input x:
1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

Summary: The Limits of Algorithms

- $A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — **Undecidable**
- $E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — **Undecidable**

needs

- $EQ_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $EQ_{\mathsf{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — **Undecidable** (unproven)

next

- $EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — **Undecidable** (unproven)

Reduce from something else: EQ_{TM} is undecidable

$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Proof, by **contradiction**:

- Assume: EQ_{TM} has *decider* R; use it to create *decider* for ~~A_{TM}~~:

$E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

$S =$ "On input $\langle M\rangle$, where M is a TM:

1. Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

Reduce from something else: EQ_{TM} is undecidable

$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Proof, by **contradiction**:

- Assume: EQ_{TM} has *decider* R; use it to create *decider* for E_{TM}:

$= \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

~~$S =$ "On input $\langle M\rangle$, where M is a TM:~~

~~**1.** Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.~~

~~**2.** If R accepts, *accept*; if R rejects, *reject*."~~

- But E_{TM} is undecidable! (and thus has no decider)

Summary: Undecidability Proof Techniques

- Proof Technique #1:
  - Use hypothetical decider to implement impossible A_{TM} decider
  - Example Proof:

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

Reduce

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

- Proof Technique #2:
  - Use hypothetical decider to implement impossible A_{TM} decider
  - But <u>first modify the input</u> M
  - Example Proof: $E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

Can also combine these techniques

Reduce

- Proof Technique #3:
  - Use hypothetical decider to implement <u>non-A_{TM}</u> impossible decider
  - Example Proof: $EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Summary: Decidability and Undecidability

- $A_{\mathsf{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\mathsf{CFG}} = \{\langle G, w\rangle |\ G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ accepts } w\}$ — **Undecidable**
- $E_{\mathsf{DFA}} = \{\langle A\rangle |\ A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\mathsf{CFG}} = \{\langle G\rangle |\ G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\mathsf{TM}} = \{\langle M\rangle |\ M \text{ is a TM and } L(M) = \emptyset\}$ — **Undecidable**
- $EQ_{\mathsf{DFA}} = \{\langle A, B\rangle |\ A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $EQ_{\mathsf{CFG}} = \{\langle G, H\rangle |\ G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — **Undecidable** (unproven)
- $EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle |\ M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — **Undecidable**

Also Undecidable …

next

- $REGULAR_{\mathsf{TM}}$ = {<M> | M is a TM and $L(M)$ is a regular language}

Undecidability Proof Technique #2:
Modify input TM M

Thm: $REGULAR_{\mathsf{TM}}$ is undecidable

$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$

Proof, by **contradiction**:

- Assume: $REGULAR_{\mathsf{TM}}$ has *decider* R; use it to create *decider* for A_{TM}:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

- First, construct M_2 (??)
- Run R on input $\langle M_2 \rangle$
- If R accepts, *accept*; if R rejects, *reject*

Want: $L(M_2) =$
- **regular**, if M accepts w
- **nonregular**, if M does not accept w

Thm: $REGULAR_{\mathsf{TM}}$ is undecidable (continued)

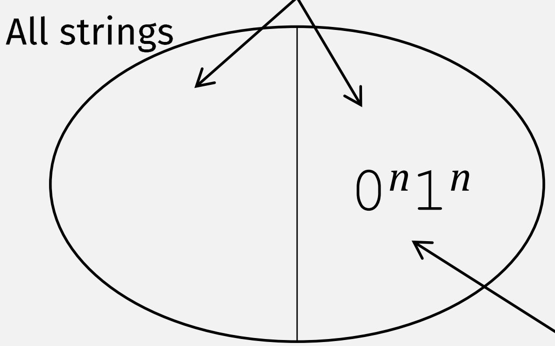
$REGULAR_{\mathsf{TM}} = \{\langle M \rangle |\ M \text{ is a TM and } L(M) \text{ is a regular language}\}$

M_2 = "On input x:
1. If x has the form 0^n1^n, *accept*.
2. If x does not have this form, run M on input w and *accept* if M accepts w."

Always accept strings 0^n1^n
$L(M_2)$ = **nonregular**, so far

If M accepts w,
accept everything else,
so $L(M_2) = \Sigma^* =$ **regular**

if M <u>does not</u> accept w, M_2 accepts all strings (**regular** lang)



<u>Want</u>: $L(M_2)$ =
- **regular**, if M accepts w
- **nonregular**, if M does not accept w

if M accepts w, M_2 accepts this **nonregular** lang

Also Undecidable …

Seems like no algorithm can compute **anything** about the language of a Turing Machine, i.e., about the runtime behavior of programs …

- $REGULAR_{\mathsf{TM}}$ = {<M> | M is a TM and $L(M)$ is a regular language}
- $CONTEXTFREE_{\mathsf{TM}}$ = {<M> | M is a TM and $L(M)$ is a CFL}
- $DECIDABLE_{\mathsf{TM}}$ = {<M> | M is a TM and $L(M)$ is a decidable language}
- $FINITE_{\mathsf{TM}}$ = {<M> | M is a TM and $L(M)$ is a finite language}

An Algorithm About Program Behavior?

```
main()
{
    printf("hello, world\n");
}
```

Write a program that,
given another program as its argument,
returns TRUE if that argument prints
“**`Hello, World!`**”

TRUE

Fermat's Last Theorem
(unknown for ~350 years,
solved in 1990s)

`main()`

`{`

If $\boxed{x^n + y^n = z^n, \text{ for any integer } n > 2}$

`printf("hello, world\n");`

`}`

Write a program that,
given another program as its argument,
returns TRUE if that argument prints
"`Hello, World!`"

?????

Also Undecidable …

Seems like no algorithm can compute **anything** about the language of a Turing Machine, i.e., about the runtime behavior of programs …

- $REGULAR_{\mathrm{TM}}$ = {<M> | M is a TM and $L(M)$ is a regular language}
- $CONTEXTFREE_{\mathrm{TM}}$ = {<M> | M is a TM and $L(M)$ is a CFL}
- $DECIDABLE_{\mathrm{TM}}$ = {<M> | M is a TM and $L(M)$ is a decidable language}
- $FINITE_{\mathrm{TM}}$ = {<M> | M is a TM and $L(M)$ is a finite language}
- …

Rice's Theorem

- $ANYTHING_{\mathrm{TM}}$ = {<M> | M is a TM and "… **anything** …" about $L(M)$}

Rice's Theorem: $ANYTHING_{\mathrm{TM}}$ is Undecidable

$ANYTHING_{\mathrm{TM}}$ = {<M> | M is a TM and … **anything** … about $L(M)$}

- "… **Anything** …", more precisely:

  For any M_1, M_2,
  - if $L(M_1) = L(M_2)$
  - then $M_1 \in ANYTHING_{\mathrm{TM}} \Leftrightarrow M_2 \in ANYTHING_{\mathrm{TM}}$

- Also, "… **Anything** …"must be "non-trivial":
  - $ANYTHING_{\mathrm{TM}}$!= {}
  - $ANYTHING_{\mathrm{TM}}$!= set of all TMs

Rice's Theorem: $ANYTHING_{\text{TM}}$ is Undecidable

$ANYTHING_{\text{TM}}$ = {<M> | M is a TM and … **anything** … about $L(M)$}

Proof by **contradiction**

- Assume some language satisfying $ANYTHING_{\text{TM}}$ has a decider R.
  - Since $ANYTHING_{\text{TM}}$ is non-trivial, then there exists $M_{ANY} \in ANYTHING_{\text{TM}}$
  - Where R accepts M_{ANY}
- Use R to create decider for A_{TM}:

On input <M, w>:

- Create M_w:

  M_w = on input x:
  - Run M on w
    - If M rejects w: reject x
    - If M accepts w:
      Run M_{ANY} on x and accept if it accepts, else reject

  If M accepts w: M_w = M_{ANY}
  If M doesn't accept w: M_w accepts nothing

  These two cases must be <u>different</u>, (so R can distinguish when M accepts w)

- Run R on M_w
  - If it accepts, then M_w = M_{ANY}, so M accepts w, so accept
  - Else reject

Wait! What if the TM that accepts nothing is in $ANYTHING_{\text{TM}}$!

Proof still works! Just use the complement of $ANYTHING_{\text{TM}}$ instead!

Rice's Theorem Implication

{<M> | M is a TM that installs malware}

Undecidable!
(by Rice's Theorem)