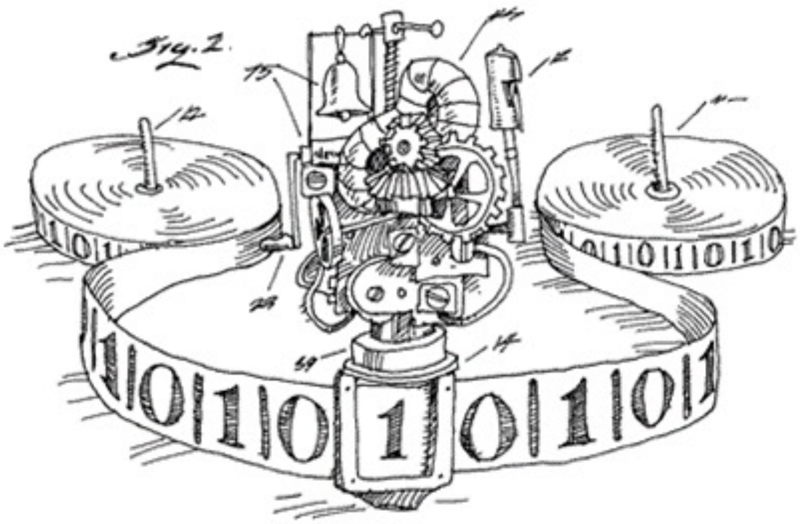
CSE 311: Foundations of Computing

Lecture 28: Turing machines



Recap: Proving irregularity

- Let $L \subseteq \Sigma^*$ be a language.
- Let $x \sim_L y$ iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$ (intuitively: $x \nsim_L y$ means DFA needs to distinguish x from y)

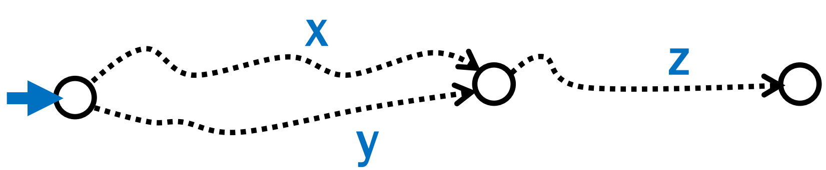
Theorem. Let $L \subseteq \Sigma^*$. If there is an infinite set $S \subseteq \Sigma^*$ with $x \nsim_L y$ for all distinct $x, y \in S$, then L is irregular.

Recap: Proving irregularity

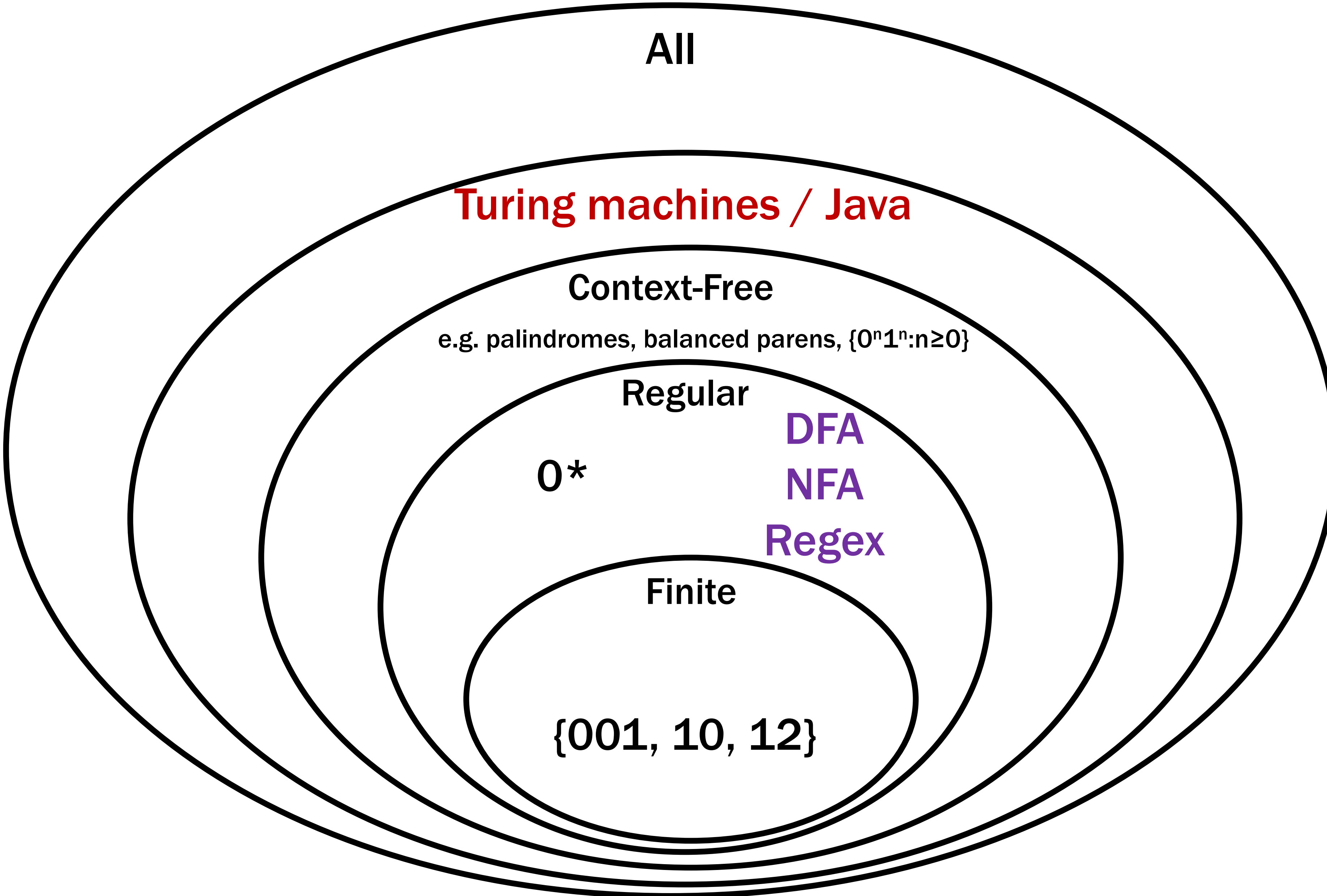
- **Let $L \subseteq \Sigma^*$ be a language.**
- **Let $x \sim_L y$ iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$ (intuitively: $x \nsim_L y$ means DFA needs to distinguish x from y)**

Theorem. Let $L \subseteq \Sigma^*$. If there is an infinite set $S \subseteq \Sigma^*$ with $x \nsim_L y$ for all distinct $x, y \in S$, then L is irregular.

Idea: If DFA is in same state after reading x and y then it is making a mistake.



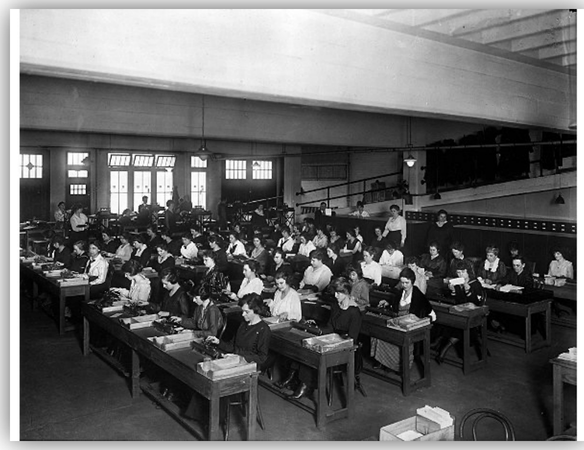
Last time: Languages and Representations



Computers and algorithms

- Does Java (or any programming language) cover all possible computation? Every possible algorithm?

- There was a time when computers were people who did calculations on sheets paper to solve computational problems



- Computers as we known them arose from trying to understand everything these people could do.

Before Java

1930's:

How can we formalize what algorithms are possible?

- **Turing machines** (Turing, Post)
  - basis of modern computers
- **Lambda Calculus** (Church)
  - basis for functional programming, LISP
- **μ-recursive functions** (Kleene)
  - alternative functional programming basis

Turing machines

Church-Turing Thesis:

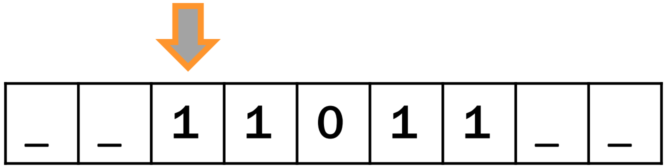
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

Evidence

- Intuitive justification
- Huge numbers of models based on radically different ideas turned out to be equivalent to TMs

Turing machines

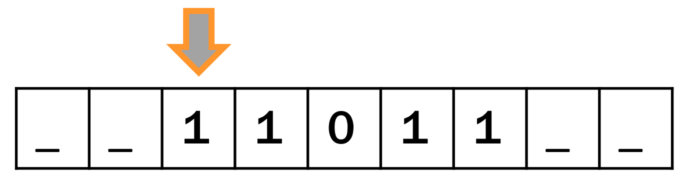
- **Finite Control**
  - **Brain/CPU** that has only a **finite # of possible "states of mind"**
- **Recording medium**
  - **An unlimited supply of** blank **"scratch paper"** on which to **write & read symbols**, each chosen from a **finite set of possibilities**
  - **Input** also supplied **on** the **scratch paper**



- **Focus of attention**
  - **Finite control** can only **focus on** a **small portion of the recording medium at once**
  - **Focus of attention** can only **shift a small amount at a time**

Turing machines

- **Recording medium**
  - An **infinite** read/write "**tape**" marked off into **cells**
  - Each **cell** can store **one symbol or** be "**blank**"
  - **Tape** is **initially** all **blank except** a few **cells** of the tape **containing the input string**
  - **Read/write head** can **scan one cell** of the tape - **starts on input**

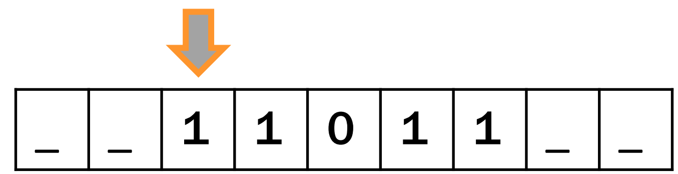


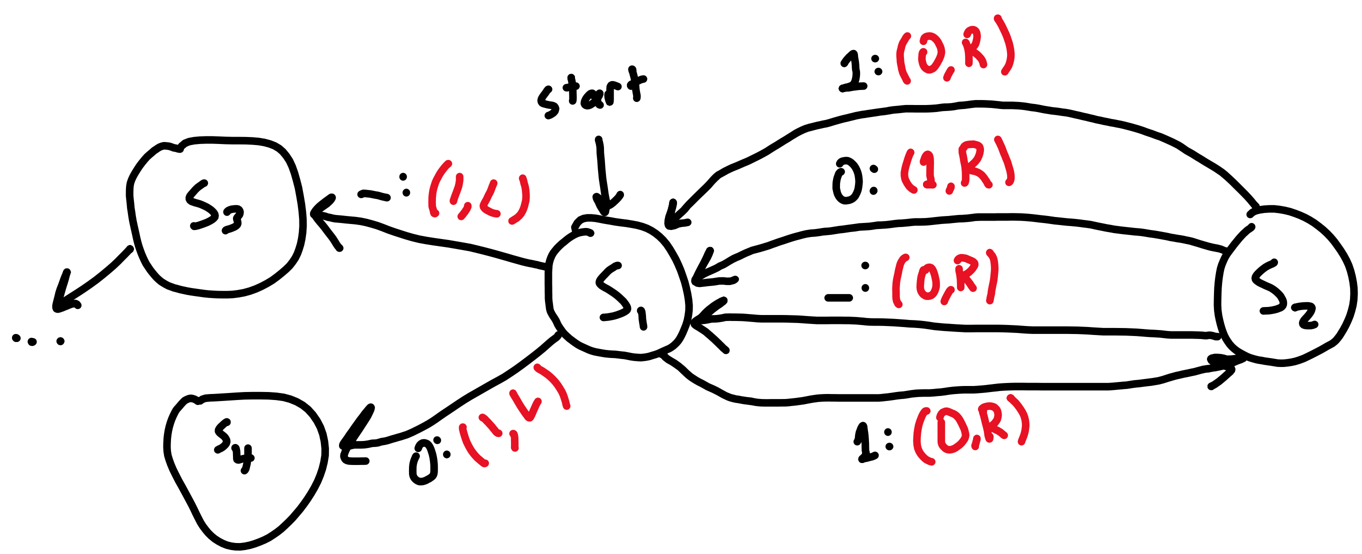
- **In each step, a Turing machine**
  1. **Reads** the **currently scanned cell**
  2. **Based on current state and scanned symbol**
     i. **Overwrites symbol in scanned cell**
     ii. **Moves read/write head left or right one cell**
     iii. **Changes to a new state**

- **Each Turing Machine is specified by its finite set of rules**
- **At any point TM can decide to either terminate & accept or terminate & reject**

Turing machines

	_	0	1
s_1	$(1, L, s_3)$	$(1, L, s_4)$	$(0, R, s_2)$
s_2	$(0, R, s_1)$	$(1, R, s_1)$	$(0, R, s_1)$
s_3			
s_4			





UW CSE's Steam-Powered Turing Machine

Original in Sieg Hall stairwell

Lecture 28 Activity

- You will be assigned to **breakout rooms**. Please:
- Introduce yourself
- Choose someone to share screen, showing this PDF
- Construct a Turing machine that recognizes the language $L = \{0^n 1^n : n \geq 0\}$ (high level description suffices).
- We recommend structuring the TM in 2 phases:
  - **Phase 1:** Verify that the input string is of the form 0^*1^*
    Hint: This is a regular task — no need to even write on the tape
  - **Phase 2:** Check that #0's=#1's
    Hint: You can overwrite characters on the tape by new symbols

Start position

Tape:

_	_	0	0	0	1	1	1	_

Fill out a poll everywhere for Activity Credit!
Go to pollev.com/thomas311 and login
with your UW identity

Turing machines

Ideal Java/C programs:

- Just like the Java/C you're used to programming with, except you never run out of memory
  - Constructor methods always succeed
  - **malloc** in C never fails

Equivalent to Turing machines except a lot easier to program:

- Turing machine definition is useful for breaking computation down into simplest steps
- We only care about high level so we use programs

Turing's big idea part 1: Machines as data

Original Turing machine definition:

– A different "machine" **M** for each task

– Each machine **M** is defined by a finite set of possible operations on finite set of symbols

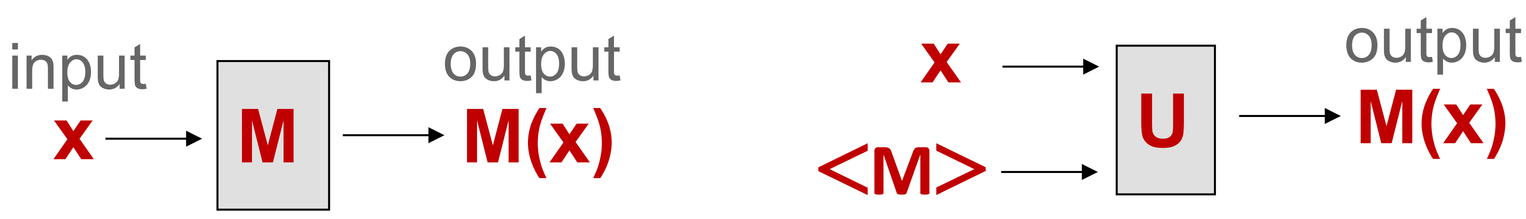
– So... **M** has a finite description as a sequence of symbols, its "code", which we denote **<M>**

You already are used to this idea with the notion of the program code or text but this was a new idea in Turing's time.

Turing’s big idea part 2: A Universal TM

- **A Turing machine interpreter U**
  - On input <M> and its input x,
    U outputs the same thing as M does on input x
  - At each step it decodes which operation M would have performed and simulates it.
- **One Turing machine is enough**
  - Basis for modern stored-program computer

    Von Neumann studied Turing’s UTM design



Computers from Thought

Computers as we know them grew out of a desire to avoid bugs in mathematical reasoning.

Hilbert in a famous speech at the International Congress of Mathematicians in 1900 set out the goal to mechanize all of mathematics.

In the 1930s, work of Gödel and Turing showed that Hilbert's program is impossible.

Gödel's Incompleteness Theorem
Undecidability of the Halting Problem

Both of these employ an idea we will see called diagonalization.

The ideas are simple but so revolutionary that their inventor Georg Cantor was shunned by the mathematical leaders of the time:

Poincaré referred to them as a "grave disease infecting mathematics."

Kronecker fought to keep Cantor's papers out of his journals.

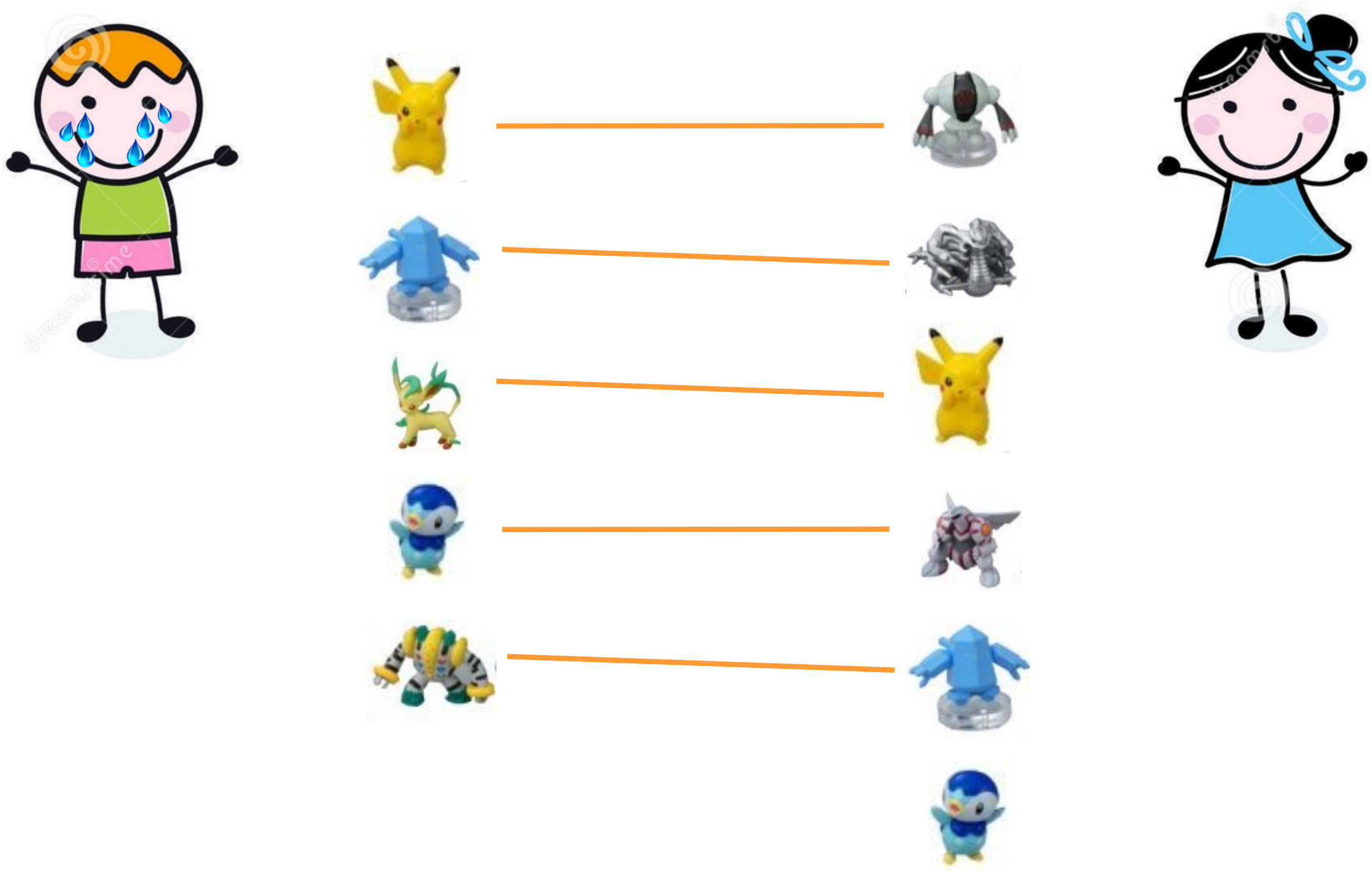
Cantor spent the last 30 years of his life battling depression, living often in "sanatoriums" (psychiatric hospitals).

Cardinality

What does it mean that two sets have the same size?

Cardinality

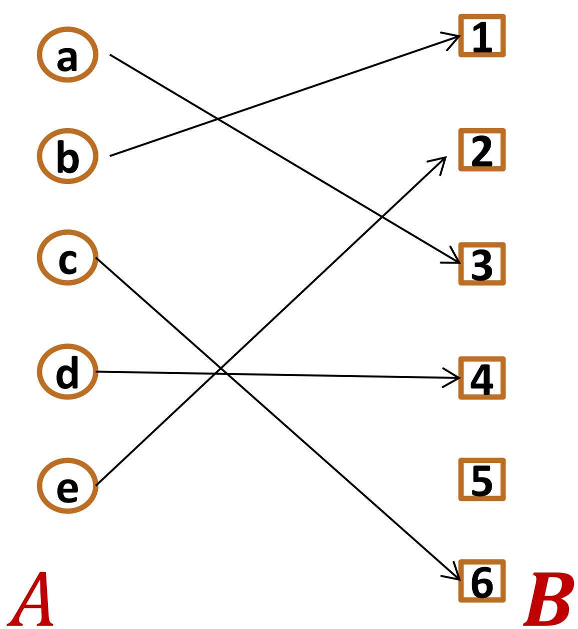
What does it mean that two sets have the same size?



Injective and surjective

A **function** $f : A \to B$ is **injective** (= **one-to-one**) if every output corresponds to at most one input;
i.e. $f(x) = f(x') \Rightarrow x = x'$ for all $x, x' \in A$.

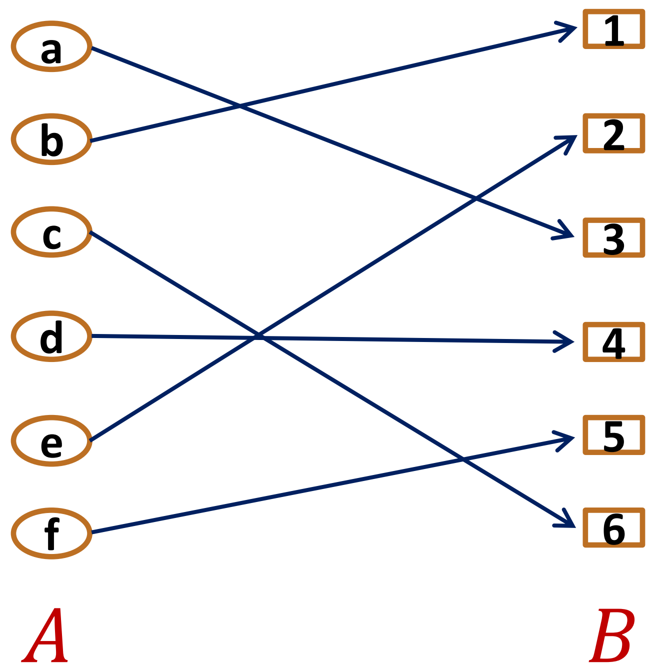
A **function** $f : A \to B$ is **surjective** (=**onto**) if every output gets hit; i.e. for every $y \in B$, there exists $x \in A$ such that $f(x) = y$.



Injective but not surjective

Cardinality

Definition: Two sets A and B have the same **cardinality** if there is a one-to-one correspondence between the elements of A and those of B.
More precisely, if there is an **injective and surjective (=bijective)** function $f : A \to B$.



The definition also makes sense for infinite sets!

Cardinality

Do the natural numbers and the even natural numbers have the same cardinality?

Yes!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	...

What's the map $\boldsymbol{f} : \mathbb{N} \to \mathbf{2}\mathbb{N}$? $\boldsymbol{f(n) = 2n}$

Countable sets

Definition: A set is **countable** iff it has the same cardinality as some subset of $\mathbb{N}$.

Equivalent: A set $\boldsymbol{S}$ is countable iff there is a *surjective* function $\boldsymbol{g} : \mathbb{N} \to \boldsymbol{S}$

Equivalent: A set $\boldsymbol{S}$ is countable iff we can order the elements $\boldsymbol{S} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\}$

Example: $\mathbb{Z}$ is countable

Claim: Σ^* is countable for every finite Σ

Idea: For $k = 0,1,2, \ldots$ list all the $|\Sigma|^k$ many strings of length k. Then each string in Σ^* appears in that list.

e.g. {0,1}* **is countable:**

{ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, … }

Countable sets

A set $\boldsymbol{S}$ is **countable** iff we can order the elements of $\boldsymbol{S}$ as

$$\boldsymbol{S} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \dots\}$$

Countable sets:

$\mathbb{N}$ - the natural numbers

$\mathbb{Z}$ - the integers

$\mathbb{Q}$ - the rationals

Σ^* - the strings over any finite Σ

The set of all Java programs

The set of all Turing machines

Enumerate in increasing length

Uncountable sets: ???

Are the real numbers countable?

Theorem [Cantor]:
The set of real numbers between 0 and 1 is **not** countable.

Proof will be by contradiction.
Uses a new method called diagonalization.

Real numbers between 0 and 1: $[0,1)$

Every number between 0 and 1 has an infinite decimal expansion:

1/2	=	0.50000000000000000000000...
1/3	=	0.33333333333333333333333...
1/7	=	0.14285714285714285714285...
π-3	=	0.14159265358979323846264...
1/5	=	0.19999999999999999999999...
	=	0.20000000000000000000000...

Representation is unique except for the cases that the decimal expansion ends in all 0's or all 9's.
We will never use the all 9's representation.

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

r_1	**0.50000000…**
r_2	**0.33333333…**
r_3	**0.14285714…**
r_4	**0.14159265…**
r_5	**0.12122122…**
r_6	**0.25000000…**
r_7	**0.71828182…**
r_8	**0.61803394…**
…	…

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4	5	6	7	8	9	...
r_1	0.	5	0	0	0	0	0	0	0	...	...
r_2	0.	3	3	3	3	3	3	3	3	...	...
r_3	0.	1	4	2	8	5	7	1	4	...	...
r_4	0.	1	4	1	5	9	2	6	5	...	...
r_5	0.	1	2	1	2	2	1	2	2	...	...
r_6	0.	2	5	0	0	0	0	0	0	...	...
r_7	0.	7	1	8	2	8	1	8	2	...	...
r_8	0.	6	1	8	0	3	3	9	4	...	...
...	....	...	....	....	...	...	...	...	...	...	

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4	5	6	7	8	9	...
r_1	0.	5	0	0	0	0	0	0	0	...	...
r_2	0.	3	3	3	3	3	3	3	3	...	...
r_3	0.	1	4	2	8	5	7	1	4	...	...
r_4	0.	1	4	1	5	9	2	6	5	...	...
r_5	0.	1	2	1	2	2	1	2	2	...	...
r_6	0.	2	5	0	0	0	0	0	0	...	...
r_7	0.	7	1	8	2	8	1	8	2	...	...
r_8	0.	6	1	8	0	3	3	9	4	...	...
...	....	...	....	....	...	...	...	...	...	...	

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4						
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	...	

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4						
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

If diagonal element is $\mathbf{0.}\,x_{11}x_{22}x_{33}x_{44}x_{55}\cdots$ then let's call the flipped number $\mathbf{0.}\,\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

It cannot appear anywhere on the list!

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4						
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
						2^{5}	1	2	2	...	...
						0	0^{5}	0	0	...	...
						8	1	8^{5}	2	...	...

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

For every $n \geq 1$:

$r_n \neq 0.\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

because the numbers differ on the n-th digit!

If diagonal element is $0.x_{11}x_{22}x_{33}x_{44}x_{55}\cdots$ then let's call the flipped number $0.\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

It cannot appear anywhere on the list!

Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4						
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
						2^{5}	1	2	2	...	...
						0	0^{5}	0	0	...	...
						8	1	8^{5}	2	...	...

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

For every $n \geq 1$:

$r_n \neq 0.\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

because the numbers differ on the n-th digit!

So the list is incomplete, which is a contradiction.

Thus the real numbers between 0 and 1 are **not countable**: “uncountable”

Last time: Countable sets

Countable sets:

$\mathbb{N}$ - the natural numbers

$\mathbb{Z}$ - the integers

$\mathbb{Q}$ - the rationals

Σ^* - the strings over any finite Σ

The set of all Java programs

The set of all Turing machines

Enumerate in increasing length

Uncountable sets:

$\mathbb{R}$ - the natural numbers

$\mathrm{P}(\mathbb{N})$ - power set of $\mathbb{N}$

Set of functions $f : \mathbb{N} \to \{0,1\}$

Uncomputable functions

We have seen that:

- The set of all (Java) programs is countable
- The set of all functions $f : \mathbb{N} \to \{0,1\}$ is not countable

So: There must be some function $f : \mathbb{N} \to \{0,1\}$ that is not computable by any program!

Interesting… maybe.

Can we come up with an explicit function that is uncomputable?

Some Notation

We're going to be talking about *Java code*.

CODE(**P**) will mean "the code of the program **P**"

So, consider the following function:

```
public String P(String x) {
    return new String(Arrays.sort(x.toCharArray());
}
```

What is **P**(CODE(**P**))?

"(((())))..;AACPSSaaabceegghiiiilnnnnnooprrrrrrrrrrsssttttttuuwxxyy{}"

The Halting Problem

CODE(**P**) means "the code of the program **P**"

The Halting Problem

Given: - CODE(**P**) for any program **P**
- input **x**

Output: **true** if **P** halts on input **x**
false if **P** does not halt on input **x**

Undecidability of the Halting Problem

CODE(**P**) means "the code of the program **P**"

The Halting Problem

Given: - CODE(**P**) for any program **P**
- input **x**

Output: **true** if **P** halts on input **x**
false if **P** does not halt on input **x**

Theorem [Turing]: **There is no program that solves the Halting Problem**

Proof by contradiction

Suppose that H is a Java program that solves the Halting problem.

Proof by contradiction

Suppose that H is a Java program that solves the Halting problem.

Then we can write this program:

```
public static void D(String s) {
    if (H(s,s) == true) {
        ...
    } else {
        ...
    }
}

public static bool H(String s, String x) { ... }
```

Does D(CODE(D)) halt?

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s) == true) {
        ...
    }
    else {
        ...
    }
}
```

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s) == true) {
        ...
    }
    else {
        ...
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        ...
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Suppose that **D**(CODE(**D**)) **halts**.

Then, by definition of **H** it must be that

H(CODE(**D**), CODE(**D**)) is **true**

Which by the definition of **D** means **D**(CODE(**D**)) **doesn't halt**

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        return;         /*   halt   */
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Suppose that **D**(CODE(**D**)) **halts.**

Then, by definition of **H** it must be that

H(CODE(**D**), CODE(**D**)) is **true**

Which by the definition of **D** means **D**(CODE(**D**)) **doesn't halt**

Suppose that **D**(CODE(**D**)) **doesn't halt.**

Then, by definition of **H** it must be that

H(CODE(**D**), CODE(**D**)) is **false**

Which by the definition of **D** means **D**(CODE(**D**)) **halts**

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        return;          /*    halt    */
    }
}
```

Does **D**(CODE(**D**)) halt?

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**) ... ot

Suppose that **D**(CODE(**D**)) **halts.**

Then, by definition of **H** it mus...

H(CODE(**D**), CO...

Which by the defi... **D**(CODE(**D**)) **doesn't halt**

Suppose th... **doesn't halt.**

Th... of **H** it must be that

...ODE(**D**), CODE(**D**)) is **false**

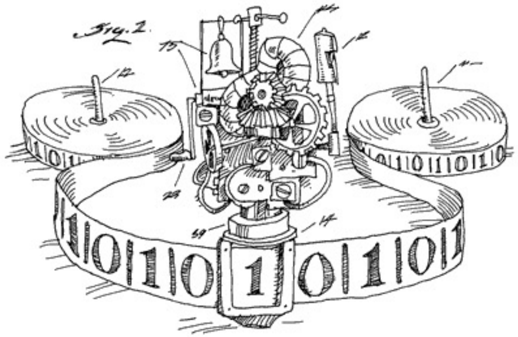
Whi... by the definition of **D** means **D**(CODE(**D**)) **halts**

Contradiction!

The ONLY assumption was that the program H exists so that assumption must have been false.

Done

- We proved that there is no computer program that can solve the Halting Problem.
  - There was nothing special about Java*
    [Church-Turing thesis]



- This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have.

Where did the idea for creating D come from?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        return;        /*    halt    */
    }
}
```

D halts on input code(P) iff **H**(code(P),code(P)) outputs false
iff P doesn't halt on input code(P)

Connection to diagonalization

Write <P> for CODE(P)

Some possible inputs **x**

$<P_1>$ $<P_2>$ $<P_3>$ $<P_4>$ $<P_5>$ $<P_6>$ ....

All programs **P**

P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8
P_9
.
.

This listing of all programs really does exist since the set of all Java programs is countable

The goal of this "diagonal" argument is not to show that the listing is incomplete but rather to show that a "flipped" diagonal element is not in the listing

Connection to diagonalization

Write <P> for CODE(P)

Some possible inputs **x**

All programs **P**

	$\langle P_1 \rangle$	$\langle P_2 \rangle$	$\langle P_3 \rangle$	$\langle P_4 \rangle$	$\langle P_5 \rangle$	$\langle P_6 \rangle$	….					
P_1	0	1	1	0	1	1	1	0	0	0	1	…
P_2	1	1	0	1	0	1	1	0	1	1	1	…
P_3	1	0	1	0	0	0	0	0	0	0	1	…
P_4	0	1	1	0	1	0	1	1	0	1	0	…
P_5	0	1	1	1	1	1	1	0	0	0	1	…
P_6	1	1	0	0	0	1	1	0	1	1	1	…
P_7	1	0	1	1	0	0	0	0	0	0	1	…
P_8	0	1	1	1	1	0	1	1	0	1	0	…
P_9	. . . .											
.	. . . .											
.												

(P,x) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

Connection to diagonalization

Write <**P**> for CODE(**P**)

Some possible inputs **x**

All programs **P**	<P_1>	<P_2>	<P_3>	<P_4>	<P_5>	<P_6>	….					
P_1	0^{1}	1	1	0	1							
P_2	1	1^{0}	0	1	0							
P_3	1	0	1^{0}	0	0							
P_4	0	1	1	0^{1}	1	0	1	1	0	1	0	…
P_5	0	1	1	1	1^{0}	1	1	0	0	0	1	…
P_6	1	1	0	0	0	1^{0}	1	0	1	1	1	…
P_7	1	0	1	1	0	0	0^{1}	0	0	0	1	…
P_8	0	1	1	1	1	0	1	1^{0}	0	1	0	…
P_9	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.												

Want behavior of program ***D*** to be like the flipped diagonal, so it can’t be in the list of all programs.

(P,x) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

Where did the idea for creating D come from?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        return;         /*    halt    */
    }
}
```

D halts on input code(P) iff **H**(code(P),code(P)) outputs false
iff P doesn't halt on input code(P)

Therefore for any program P, **D** differs from P on input code(P)