Understanding Computation

Mathematics & Computation

-Mathematics has been around for a long time as a method of computing.

-Efforts to find "canonical" way of computations.

-"Machines" have helped with calculations.

Can we use machines to reason too?

Formal Logic: Reasoning made into a calculation

Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians

Foundational problem:

Formalism or Intuitionalism? Are mathematics a formal language or a "fairy tale" with numbers that exist in human mind only?

Formalists: How to choose one's axioms?

They should not give rise to contradictions!

Early 1900s: Crisis in mathematical foundations

Contradictions discovered while attempting to formalize notions involving infinite sets

David Hilbert

- Proposed solution to the foundational crisis of mathematics, when attempts to clarify the foundations of mathematics were found to suffer from paradoxes and inconsistencies.

- As a solution, Hilbert proposed to ground all existing theories to a finite, complete set of axioms, and provide a proof that these axioms were consistent.

- He proposed that the consistency of more complicated systems, such as real analysis, could be proven in terms of simpler systems.
  Ultimately, consistency of all mathematics could be reduced to basic arithmetic.

Hilbert's Program

- A formalization of all mathematics; in other words all mathematical statements should be written in a precise formal language, and manipulated according to well defined rules.
- Completeness: a proof that all true mathematical statements can be proved in the formalism.
- Consistency: a proof that no contradiction can be obtained in the formalism of mathematics. This consistency proof should preferably use only reasoning about finite mathematical objects.

Mathematics & Computation

Mechanized math

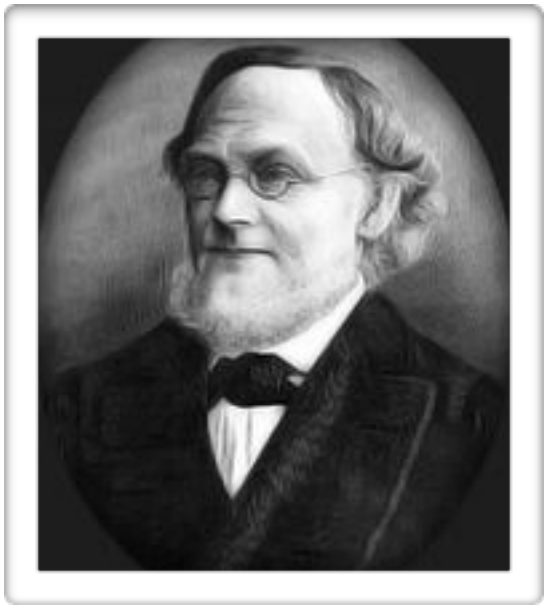
Beyond just philosophical interest!

Can resolve stubborn open problems

Replace mathematicians with mathe-machines!

Goldbach's Conjecture

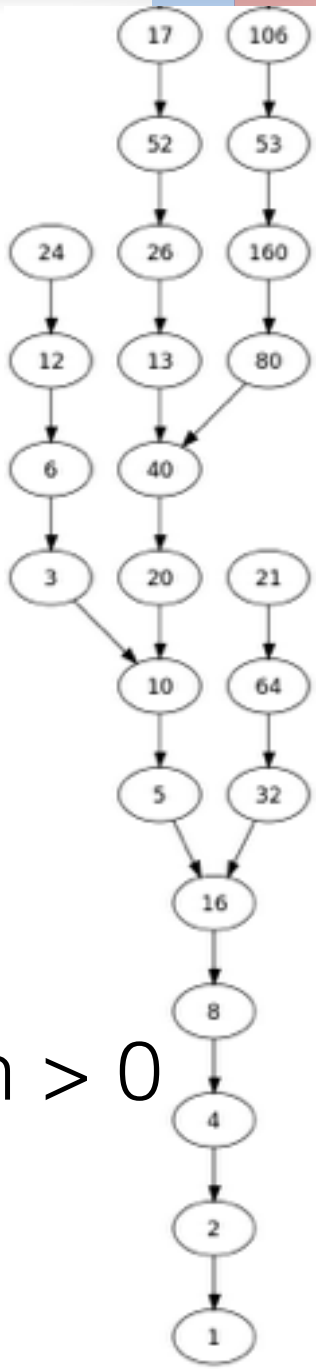
Every even number > 2 is the sum of two primes



Letter from Goldbach to Euler
dated 7 June 1742

Collatz Conjecture

```
Program Collatz (n:integer)
   while n > 1 {
     if Even(n) then n := n/2
     else  n := 3n+1
   }
```



Conjecture: Collatz(n) halts for every n > 0

Kurt Gödel

- German logician, at age 25 (1931) proved:

1) "No matter what (consistent) set of axioms are used, a rich system will have true statements that can't be proved"

2) "A system powerful enough to encode addition and multiplication of integers can not prove its own consistency, so it certainly cannot be used to prove the consistency of anything stronger with certainty"

Hilbert's Program can't work!

- Shook the foundations of
  - mathematics
  - philosophy
  - science
  - everything

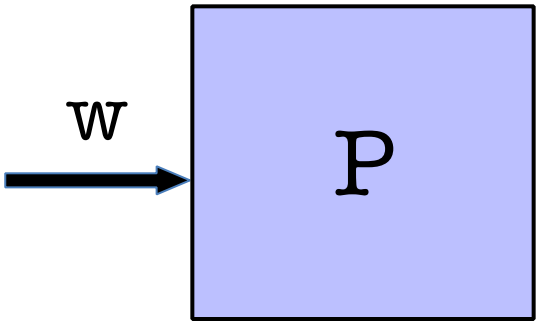
Alan Turing

- British mathematician
  - cryptanalysis during WWII
  - arguably, father of AI, CS Theory
  - several books, movies

- Mathematically defined computation
  - Invented Turing Machines at 23 (1936). Turing machines can compute everything that is computable. He proved that **The Halting Problem** has no general algorithm (it is not possible to decide whether a turing machine will ever halt)

Halting Problem

- Given program `P`, input `w`:



Will `P(w)` halt?

Why would we care about the Halting Problem?

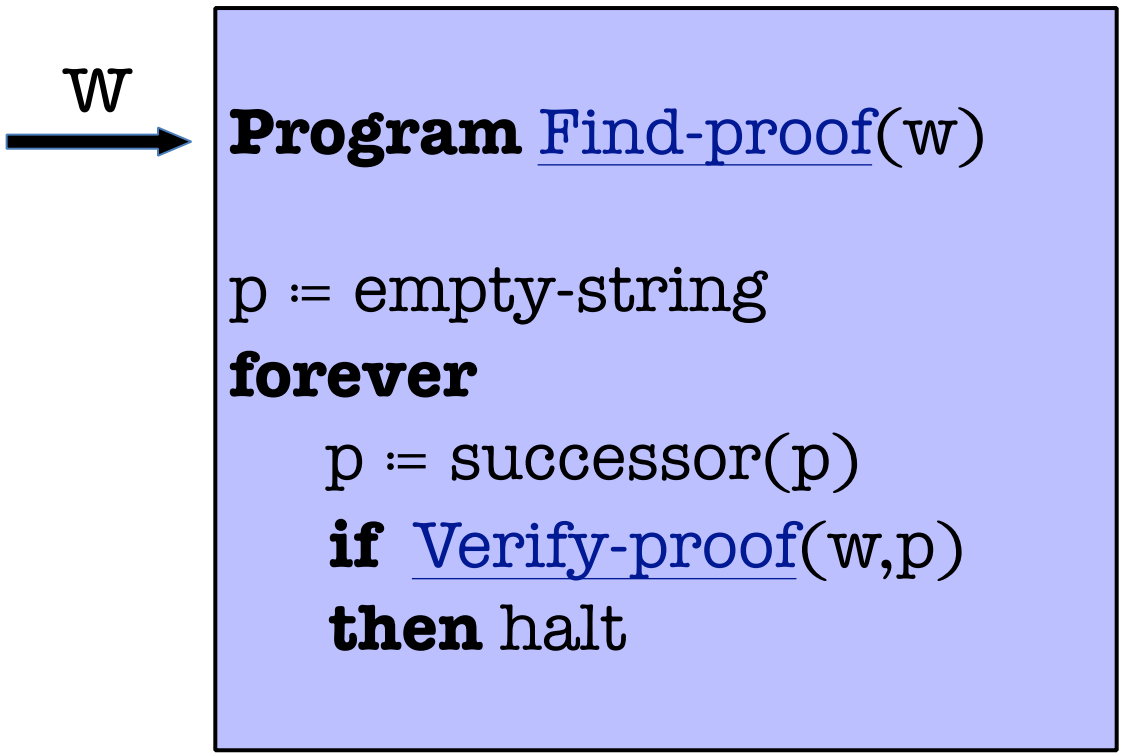
- Suppose halting problem had an algorithm...

```
Program P()
    n := 4
    forever:
        if found-two-primes-that-sum-to(n)
        then n := n + 2
        else   halt
```

Does P halt ? ⟵ **Solves Goldbach conjecture!**

Why would we care about the Halting Problem?

Does Find-proof halt on w? ≡ Is w a provable theorem?



Alas!

There is no program that solves the Halting Problem!

No use trying to find one!

How can there be problems that can't be solved?

What is a problem? What is a program?

Computation

Problem:	**Program**:
To compute a function F that maps each input (a string) to an output bit	A finitely described process taking a string as input, and outputting a bit (or not halting)

`P` computes F if for every x, `P(x)` outputs F(x) and halts

Too restrictive?

Enough to compute functions with longer outputs too:
`P(x,i)` outputs the i^{th} bit of F(x)

Enough to model *interactive* computation too:
`P*(x,state)` outputs `(y,new_state)`

Computation

Problem:
To compute a function F that maps each input (a string) to an output bit

Program:
A finitely described process taking a string as input, and outputting a bit (or not halting)

`P` computes F if for every x, `P(x)` outputs F(x) and halts

- A program is a finite bit string
- Programs can be *enumerated* — listed sequentially — (say, lexicographically) so that every program appears somewhere in the list
- The set of all programs is countable.

1	ε
2	0
3	1
4	00
5	01
6	10
7	11
8	000
9	001
10	010
11	011
12	100

Computation

Problem:
To compute a function F that maps each input (a string) to an output bit

Program:
A finitely described process taking a string as input, and outputting a bit (or not halting)

`P` computes F if for every x, `P(x)` outputs F(x) and halts

- A function assigns a bit to each finite string
- Corresponds to an infinite bit string
- The set of all functions is uncountable!
  - As numerous as, say, real numbers in [0,1]

1	ε	*0*
2	0	*0*
3	1	*1*
4	00	*0*
5	01	*1*
6	10	*1*
7	11	*0*
8	000	*0*
9	001	*1*
10	010	*1*
11	011	*0*
12	100	*1*

Computation

Problem:
To compute a function F that maps each input (a string) to an output bit

Program:
A finitely described process taking a string as input, and outputting a bit (or not halting)

`P` computes F if for every x, `P(x)` outputs F(x) and halts

There are uncountably many functions!

But only countably many programs

Almost every function is uncomputable!

Uncomputable Problems

But that doesn't tell us why some *interesting* problems are uncomputable

If *interesting* ≡ *has a finite description in English*, then only countably many interesting problems!

Proving that there are uncountably many real numbers: "**Diagonalization**" argument by Cantor

Showing Halting Problem to be uncomputable: a similar argument (*later*)

Strings

Why Strings?

- Algorithms manipulate sequences : arrays, lists, text, graphs..
- All of the inputs and outputs represented by finite string of symbols.

Definitions for strings

e.g., $\Sigma = \{0,1\}$, $\Sigma = \{\alpha, \beta, \ldots, \omega\}$, Σ = set of ascii characters

- **alphabet** Σ = finite set of symbols
- **string** = finite sequence of symbols of Σ
- **length** of a string w is denoted $|w|$.
- **empty string** is denoted "ε".

$|\varepsilon| = 0$

$|\text{cat}|=3$

Variable conventions (for this lecture)

$a, b, c, \ldots$ elements of Σ (i.e., strings of length 1)

$w, x, y, z, \ldots$ strings of length 0 or more

$A, B, C, \ldots$ sets of strings

What is a string?

Formally,

- string = ε
- string = *ax, where a is an element of* Σ *and x is a string*.

Length of string w is

- |w| = 0 if w= ε
- |w| =1+|x|, if w= *ax, where a is an element of* Σ *and x is a string*.

Much ado about nothing

- ε is a **string** containing no symbols. It is not a set.
- $\{\varepsilon\}$ is a **set** containing one string: the empty string ε. It is a set, not a string.
- $\emptyset$ is the **empty set**. It contains no strings.

Concatenation & its properties

- xy denotes the **concatenation** of strings x and y (sometimes written $x \cdot y$)
  - $w \cdot z = z$ *if* $w = \varepsilon$
  - $w \cdot z = a \cdot (x \cdot z)$ *if* $w = ax$

We can derive two properties:

- Associative: $(uv)w = u(vw)$ and we write uvw.
- Identity element ε : $\varepsilon w = w\varepsilon = w$ *(proof follows)*

Inductive proof

- **Theorem**: $w\varepsilon = w$
- **Proof**: By induction

Let w be an arbitrary string.

Assume for all strings x such that $|x|<|w|$ that $x\varepsilon=x$ *(Inductive Hypothesis).*

There are two cases:

-Base case: $|w| = 0$: i.e., $w= \varepsilon$.
Then: $w\varepsilon = \varepsilon\varepsilon = \varepsilon = w$ ✅

-*Inductive step* $w=ax$

Then: $w\varepsilon = (ax)\varepsilon = a(x\varepsilon) = ax = w$ ✅

Thus, string-induction proofs have the following boilerplate structure. Suppose we want to prove that every string is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent".

Proof: Let w be an arbitrary string.

Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.

There are two cases to consider.

- Suppose $w = \varepsilon$.

  Therefore, w is perfectly cromulent.

- Suppose $w = ax$ for some symbol a and string x.

  The induction hypothesis implies that x is perfectly cromulent.

  Therefore, w is perfectly cromulent.

In both cases, we conclude that w is perfectly cromulent. □

First, mindlessly write the green text.

Then fill in the red text. And then start thinking

Inductive Definitions

- Often operations on strings are formally defined inductively

$$\varepsilon^R = \varepsilon$$
$$(au)^R = u^R a$$

  - w^R (w reversed) ind[illegible]ngth
    - If $|w| = 0$, $w^R = \varepsilon$
    - If $|w| \geq 1$, $w^R = u^R a$ where $w = au$

Well-defined: $|u|<|w|$

$a \in \Sigma, u \in \Sigma^*$

      - e.g. $(cat)^R = (c \cdot at)^R = (at)^R \cdot c = (a \cdot t)^R \cdot c$
        $= (t)^R \cdot a \cdot c = (t \cdot \varepsilon)^R \cdot ac = \varepsilon^R \cdot tac = tac$