

Computability

CS70: Discrete Mathematics and Probability Theory

UC Berkeley – Summer 2026

Lecture 15

Ref: Note 12

Uncomputable Functions

From last time - are these countable or uncountable?

(A) Set of all predicates? **Uncountable**

(B) Set of all programs? **Countable**

So: No bijection possible between programs and predicates

⇒ Some predicates have no program to compute them

(*In fact: Uncountably many predicates have no program!*)

Last Lecture: Uncomputable predicates (functions) exist

This Lecture: Do *interesting/practical* functions exist with no program that can compute them?

We'll see the answer is **Yes!**

A Deep Problem: Self Reference

From Lecture 1!



Is this a proposition? “This statement is false.”

If it's false then it's true...

if it's true it's false...

Liar's paradox

Russell's Paradox:

Does the set of all sets which do not contain themselves contain itself?

R = set of all sets that are not members of themselves

Math-y: $R = \{x \mid x \notin x\}$

$R \in R$? Then $R \notin R$

$R \notin R$? Then $R \in R$

Important question: Were mind-altering substances involved?

Issue confounding both Liar's Paradox and Russell's Paradox:

Self-reference...

An Interesting/Practical Problem

Problem: Write a program that analyzes an input program P to determine if it halts on some program input x (or if it goes into an infinite loop).

Wait... a program that takes a program as input? Is that a thing?

Of course: That's what a compiler is!

That's what a static analysis tool is!

Specifically, I want the following function (which *always* gives an answer):

$HALT(P, x)$: P is a program, x is an input to P

Returns "Yes" if P halts when run on x

"No" if P loops forever when run on x

How do we provide a program as input to a program?

Seems weird at first, but it's really not.... a program is just a string of bits

Bits are bits – they have no "meaning" until we give them meaning

One byte: 01000001

Is it a number? Could be... could represent 65

Is it a character? Could be... could be ASCII character 'A'

Is it code? could be... Intel 8080 instruction "MOV B, C"

Implementing HALT

$HALT(P, x)$: P is a program, x is an input to P

Returns “Yes” if P halts when run on x

“No” if P loops forever when run on x

Solution: Just run the program! – in pseudo-Python....

Use Python `eval` function to evaluate $P(x)$

```
if  $P(x)$  halts:  
    return "Yes"  
else:  
    return "No"
```

Correct? Incorrect? Maybe?

Incorrect! If $P(x)$ loops forever, never get to the `if...`

Run for a million steps? Maybe it would halt in 2 million....

Run for a day? Maybe it would halt in 2 days....

Bottom line: Can't ever stop and say “that was enough”

No Program Exists for HALT – Part 1

Theorem: There is no program that correctly implements *HALT*.

Proof: Assume for the sake of contradiction that we have a program $HALT(P, x)$ that always gives the right answer (*HALT* always halts!).

Use *HALT* to define a new function/program:

Turing(P):

1. If $HALT(P, P)$ = “Yes”: Go into an infinite loop
2. Else: halt immediately

Possible? Yes: *HALT* exists by assumption – string of bits defines *HALT*
Turing exists? It's above – just more bits

Recap to now:

Assume *HALT* function/program exists – it's a string of bits (a file?)

Can add a new function to program, Turing – just more bits

Have bits – data – that is full implementation of Turing

Can do anything with those bits that we can do with any other data
... including passing those bits as data to a program!

No Program Exists for HALT – Part 2

Turing(P):

1. If $\text{HALT}(P,P) = \text{"Yes"}$: Go into an infinite loop
2. Else: halt immediately

Question: Does Turing(Turing) halt?

Yes?

Turing(Turing) calls $\text{HALT}(\text{Turing}, \text{Turing})$ which returns “Yes”
 $\implies$ Turing(Turing) goes into an infinite loop and doesn’t halt.

No?

Turing(Turing) calls $\text{HALT}(\text{Turing}, \text{Turing})$ which returns “No”
 $\implies$ Turing(Turing) halts

So if HALT works, then ...

Turing(Turing) halts $\implies$ Turing(Turing) doesn’t halt

Turing(Turing) doesn’t halt $\implies$ Turing(Turing) halts

Impossible! **Contradiction!** No program can correctly implement HALT . □

Confused? That’s normal. Questions?

Another View: Diagonalization

Fact 1: Any program is a fixed length string.

Fact 2: Fixed length strings are enumerable.

Fact 3: Program inputs are finite length strings (so enumerable).

Fact 4: A program halts or doesn't for any input (even if we don't know which)

Enumeration of programs (in rows below) – inputs as columns – Halt or Loop:

	P_1	P_2	P_3	...
P_1	H	H	L	...
P_2	L	L	H	...
P_3	L	H	H	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Important: We're not saying we can construct this table. Only that it exists!

Create “diagonal program” – flips H and L

Turing is the diagonal program

Turing is a program, so must be in the enumeration of all programs

... but it halts/loops differently than any program for at least 1 input

... so it's different from any program on the list

Contradiction!

So Turing can't be a program....

Take a Breather...

Question: What are programs?

- (A) Instructions
- (B) Text
- (C) Binary String
- (D) They run on computers

Answer: All are correct.

The *only* that makes programs “special” is something that interprets instructions as actions to perform.

Hardware (CPU) or software (interpreter)

A Technical Break For Some History

Question: “Turing”??? What’s that? **Who** is that?

This proof was invented by Alan Turing in 1937

“On Computable Numbers, with an Application to the Entscheidungsproblem”
The *what*?

David Hilbert (1862–1943): Mathematician and Philosopher

Famous for posing challenge problems – 23 famous problems in 1900

Similar: Seven “millennium problems” from the Clay Institute in 2000

Hilbert posed the Entscheidungsproblem in 1928

Asks if a process/algorithm exists to say whether a statement is valid

1936: Turing (and Alonzo Church): **No**

What’s an algorithm? What’s a computer? Can computation be mechanized?

All non-obvious in 1928

Alan Turing: Father of Computer Science

Defined abstract computing machine – now called a Turing Machine

Designed machines that broke codes in World War II

Talked about artificial intelligence and a test now called the “Turing Test”

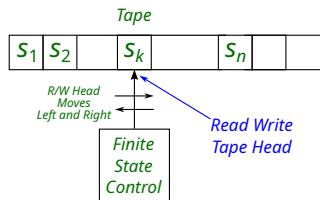
Namesake of the top prize in Computer Science: The Turing Award

Tragic life and victim of the time in which he lived... Movie: *The Imitation Game*

Turing Machines

How does someone envision a programmable “computing machine”?

A “Turing Machine” (TM):



The machine definition (Finite State Control) is an algorithm

A Universal Turing Machine’s algorithm simulates another TM

Input (on tape) is the encoding of a TM (a program)

Halting problem: Ask if input TM halts on given input

If a machine/language can simulate a Turing machine, it is “Turing Complete.”

Church-Turing Thesis: Any problem that can be solved with a computational procedure can be solved by a Turing machine.

Reductions from HALT

A **reduction** from problem A to problem B uses a solution to B to solve A

Example: Reduce “compute multiplicative inverse” to “compute Ext-GCD”

Problem: Write a program analyzer to determine if $P(x)$ outputs “Hello world”

Call this problem HW-CHECK(P, x)

Idea: Let’s reduce HALT to HW-CHECK

Program:

$HALT(P, x)$

Transform P to P' :

Remove existing “print” statements

Add print “Hello world” at end (at program termination)

Call HW-CHECK(P', x)

Return result

By construction, $P'(x)$ prints “Hello world” if and only if P halts on x

So if HW-CHECK works correctly, then this solves HALT.

But no program can solve HALT! **Contradiction!**

$\implies$ No program can solve HW-CHECK correctly.



Reductions

A **reduction** from problem A to problem B uses a solution to B to solve A

Example: Reduce “compute multiplicative inverse” to “compute Ext-GCD”

Notation: We write $A \leq B$ to say there is a reduction from A to B

Usual way of saying: “ A is reducible to B ”

Complication: There are different kinds of reductions... not important here

Theorem: If $A \leq B$ and A is uncomputable, then B is uncomputable.

Proof idea: Assume (for contradiction) B is computable, so a program for B exists. Use reduction with B 's program to solve A . But that's impossible... $\square$

On the previous slide:

Showed $HALT \leq HW-CHECK$ (a program for $HALT$ using a program for $HW-CHECK$)

We know $HALT$ is uncomputable

$\implies$ $HW-CHECK$ is uncomputable

Undecidable Problems 1

In Theory of Computation, predicates are often called “decision problems”
An uncomputable decision problem is “undecidable”

Undecidable problems about programs:

Given program P and input x , does P halt when run on x ?

Given program P and input x , does P print “hello world” when run on x ?

Given program P and input x , does P execute “line n ” when run on x ?

Given program P and input x , does P access an array out of bounds?

All proofs are similar to $HALT \leq HW-CHECK$ reduction given earlier

Rice's Theorem: Deciding if *any* non-trivial property is met during the execution of program P on input x is undecidable.

“Static Analysis” tools check programs for bugs, security vulnerabilities, etc.
Rice's Theorem says making a perfect static analysis tool is impossible!

Undecidable Problems 2

Problems that aren't (or don't seem like!) analyzing programs.

Post Correspondence Problem

Given “domino” types with labels, unlimited copies of each

Can copies be lined up to read same top/bottom?

$$\left\{ \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\} \Rightarrow \begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} b \\ ca \end{bmatrix} \begin{bmatrix} ca \\ a \end{bmatrix} \begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} abc \\ c \end{bmatrix}$$

Solving Diophantine Equations

Algorithms exist to approximate roots of polynomial equations over $\mathbb{R}$

But what about over $\mathbb{Z}$?

Example: Are there integer roots to $x^n + y^n = 5$?

Conway's “Game of Life” (cellular automaton – pattern evolution)

Grid of locations with evolving state – based of neighbors

Stephen Wolfram's book “A New Kind of Science”

Billiard Ball Simulation

Given balls in an area with perfectly elastic collisions

Does a ball go in a pocket?

Back To Logic

Logic: Start with axioms, derive other propositions

Gödel:

Any set of axioms is either
inconsistent (can prove false statements) or
incomplete (there are true statements that cannot be proven).

Proof idea:

Formal system F : axioms, rules of inference, ...

Consider $S(F) = \text{"This statement is not provable in } F\text{"}$

Provable (that's it's not provable)? Inconsistent

Not provable? Incomplete

Can prove using undecidability of HALT too! (see notes)

Concrete example:

Continuum hypothesis: "no cardinality between naturals and reals"

Continuum hypothesis not disprovable in ZFC (Gödel 1940)

Continuum hypothesis not provable (Cohen 1963)

Lecture 15 Summary

The problem with self-reference

Liar's Paradox, Russell's Paradox, ...

Uncomputable functions

Programs as data

The halting problem

Proof of undecidability

Diagonalization view of the proof

Some History

Alan Turing

David Hilbert

Reductions: Solving one problem with another

Undecidability of basic program behavior

HALT reduction to “prints hello world”

Some additional uncomputable functions

Gödel's Incompleteness Theorem

Starting Next Week: Probability

Next up? Probability.

A bag contains:



What is the chance that a ball taken from the bag is blue?

Count blue. Count total. Divide.

It might get a little more complicated than that...

See you next time!