# Computation Theory

Anuj Dawar

12 lectures for
University of Cambridge 2026 Computer Science Tripos Part 1B
based on notes by Prof. Andrew Pitts

# Resources

## Course Webpage

https://www.cst.cam.ac.uk/current/CompTheory/

- Course Notes
- Exercise Sheets
- Additional Material

## Moodle

- Lecture Recordings
- Links to Course webpage

## Prerequisites

- This course assumes familiarity with *Part 1A Discrete Mathematics*
- This course is a prerequisite for *Part 1B Complexity Theory*

# Useful Books


- Hopcroft, J.E., Motwani, R. & Ullman, J.D. (2001). *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley.
- Hindley, J.R. & Seldin, J.P. (2008). *Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press (2nd ed.).
- Cutland, N.J. (1980) *Computability. An introduction to recursive function theory*. Cambridge University Press.
- Sipser, M. (2013) *Introduction to the Theory of Computation*. Thompson Course Technology (3rd ed.).
- Sudkamp, T.A. (1995). *Languages and Machines*, 2nd edition. Addison-Wesley.

# Outline

**Introduction: algorithmically undecidable problems**
Decision problems. The informal notion of algorithm, or effective procedure. Examples of algorithmically undecidable problems. [1 lecture]

**Register machines**
Definition and examples; graphical notation. Register machine computable functions. Doing arithmetic with register machines. [1 lecture]

**Coding programs as numbers**
Natural number encoding of pairs and lists. Coding register machine programs as numbers. [1 lecture]

**Universal register machine**
Specification and implementation of a universal register machine. [1 lecture]

**The halting problem**
Statement and proof of undecidability. Example of an uncomputable partial function. Decidable sets of numbers; examples of undecidable sets of numbers. [1 lecture]

# Outline

**Turing machines**
Informal description. Definition and examples. Turing computable functions. Equivalence of register machine computability and Turing computability. The Church-Turing Thesis. [2 lectures]

**Primitive and partial recursive functions**
Definition and examples. Existence of a recursive, but not primitive recursive function. A partial function is partial recursive if and only if it is computable. [2 lectures]

**Lambda calculus**
Alpha and beta conversion. Normalization. Encoding data. Writing recursive functions in the $\lambda$-calculus. The relationship between computable functions and $\lambda$-definable functions. [3 lectures]

# Introduction

# Algorithmically undecidable problems

Computers cannot solve all mathematical problems, even if they are given unlimited time and working space.
Three famous examples of computationally unsolvable problems are sketched in this lecture.

- Hilbert's *Entscheidungsproblem*
- The Halting Problem
- Hilbert's 10th Problem.

# Hilbert's *Entscheidungsproblem*

Is there an algorithm which when fed any statement in the formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peano's axioms for arithmetic, using the usual rules of first-order logic?

Such an algorithm would be useful! For example, by running it on

$$\forall k > 1 \, \exists p, q \, (2k = p + q \wedge \mathit{prime}(p) \wedge \mathit{prime}(q))$$

(where $\mathit{prime}(p)$ is a suitable arithmetic statement that $p$ is a prime number) we could solve *Goldbach's Conjecture* ("every even integer strictly greater than two is the sum of two primes"), a famous open problem in number theory.

# Hilbert's *Entscheidungsproblem*

Is there an algorithm which when fed any statement in the formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peano's axioms for arithmetic, using the usual rules of first-order logic?

Posed by Hilbert at the 1928 International Congress of Mathematicians. The problem was actually stated in a more ambitious form, with a more powerful formal system in place of first-order logic.

In 1928, Hilbert believed that such an algorithm could be found.
A few years later he was proved wrong by the work of Church and Turing in 1935/36, as we will see.

# Decision problems

*Entscheidungsproblem* means "decision problem". Given

- a set $S$ whose elements are finite data structures of some kind
  (e.g. formulas of first-order arithmetic)
- a property $P$ of elements of $S$
  (e.g. property of a formula that it has a proof)

we want to

find an algorithm which
terminates with result 0 or 1 when fed an element $s \in S$
and
yields result 1 when fed $s$ if and only if $s$ has property $P$.

# Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples, such as:

- Procedure for multiplying numbers in decimal place notation.
- Procedure for extracting square roots to any desired accuracy.
- Euclid's algorithm for finding highest common factors.

# Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples.
Common features of the examples:

- finite description of the procedure in terms of elementary operations
- deterministic (next step uniquely determined if there is one)
- procedure may not terminate on some input data, but we can recognize when it does terminate and what the result is.

# Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples.
In 1935/36 Turing in Cambridge and Church in Princeton independently gave negative solutions to Hilbert's *Entscheidungsproblem*.

- First step: give a precise, mathematical definition of "algorithm".
  (Turing: Turing Machines; Church: lambda-calculus.)
- Then one can regard algorithms as data on which algorithms can act and reduce the problem to...

# The Halting Problem

is the decision problem with

- set $S$ consists of all pairs $(A, D)$, where $A$ is an algorithm and $D$ is a datum on which it is designed to operate;
- property $P$ holds for $(A, D)$ if algorithm $A$ when applied to datum $D$ eventually produces a result (that is, eventually halts—we write $A(D)\downarrow$ to indicate this).

Turing and Church's work shows that the Halting Problem is undecidable, that is, there is no algorithm $H$ such that for all $(A, D) \in S$

$$H(A, D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$$

There's no $H$ such that $H(A,D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$ for all $(A,D)$.

**Informal proof**, by contradiction. If there were such an $H$, let $C$ be the algorithm:

> *"input $A$; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."*

So $\forall A\,(C(A)\downarrow \leftrightarrow H(A,A) = 0)$ (since $H$ is total)
and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A)\downarrow)$ (definition of $H$).
So $\forall A\,(C(A)\downarrow \leftrightarrow \neg A(A)\downarrow)$.
Taking $A$ to be $C$, we get $C(C)\downarrow \leftrightarrow \neg C(C)\downarrow$, contradiction!

There's no $H$ such that $H(A,D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$ for all $(A,D)$.

**Informal proof**, by contradiction. If there were such an $H$, let $C$ be the algorithm:

> *"input $A$; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."*

why is A a "datum on which A is designed to operate"?

So $\forall A\,(C(A)\downarrow \leftrightarrow H(A,A) = 0)$ (since $H$ is total)
and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A)\downarrow)$ (definition of $H$).
So $\forall A\,(C(A)\downarrow \leftrightarrow \neg A(A)\downarrow)$.
Taking $A$ to be $C$, we get $C(C)\downarrow \leftrightarrow \neg C(C)\downarrow$, contradiction!

# From HP to *Entscheidungsproblem*

Final step in Turing/Church proof of undecidability of the *Entscheidungsproblem*: they constructed an algorithm encoding instances $(A, D)$ of the Halting Problem as arithmetic statements $\Phi_{A,D}$ with the property

$$\Phi_{A,D} \text{ is provable} \leftrightarrow A(D)\downarrow$$

Thus any algorithm deciding provability of arithmetic statements could be used to decide the Halting Problem—so no such exists.

# Hilbert's *Entscheidungsproblem*

Is there an algorithm which when fed any statement in the formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peano's axioms for arithmetic, using the usual rules of first-order logic?

With hindsight, a positive solution to the *Entscheidungsproblem* would be too good to be true. However, the algorithmic unsolvability of some decision problems is much more surprising. A famous example of this is...

# Hilbert's 10th Problem

Give an algorithm which, when started with any Diophantine equation, determines in a finite number of operations whether or not there are natural numbers satisfying the equation.

One of a number of important open problems listed by Hilbert at the International Congress of Mathematicians in 1900.

# Diophantine equations

$$p(x_1, \ldots, x_n) = 0$$

where $p$ is a polynomial in unknowns $x_1, \ldots, x_n$ with coefficients from $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$.

Named after Diophantus of Alexandria (c. 250AD).
Example: "find three whole numbers such that the product of any two added to the third is a square"
[Diophantus' *Arithmetica*, Book III, Problem 7].
In modern notation: find $x_1, x_2, x_3 \in \mathbb{Z}$ for which there exists $x, y, z \in \mathbb{Z}$ with

$$(x_1x_2 + x_3 - x^2)^2 + (x_2x_3 + x_1 - y^2)^2 + (x_3x_1 + x_2 - z^2)^2 = 0$$

# Hilbert's 10th Problem

Give an algorithm which, when started with any Diophantine equation, determines in a finite number of operations whether or not there are natural numbers satisfying the equation.

- Posed in 1900, but only solved in 1970: Y Matijasevič, J Robinson, M Davis and H Putnam show it undecidable by reduction from the Halting Problem.
- Original proof used Turing machines. Later, simpler proof [JP Jones & Y Matijasevič, J. Symb. Logic 49(1984)] used Minsky and Lambek's register machines—we will use them in this course to begin with and return to Turing and Church's formulations of the notion of "algorithm" later.