Computable Functions

Tracing the Quest to Define Effective Calculability

Dr Marco Benini (`marco.benini@uninsubria.it`)

Corso di Laurea Magistrale in Matematica

a.a. 2025/26

Computable Functions: Lecture 1

Syllabus:

From intuition to rigour:

- Hilbert's program
- Kurt Gödel
- Emil Post
- Moses Schönfinkel

From abacus to computer

Tracing the quest to define effective calculability

Welcome to monographic part A!

- This 16-hour module (8 lessons) delves into a pivotal intellectual journey of the 20th century
- We will explore the mathematical quest to rigorously define “computation” and “algorithm”
- This journey not only laid the theoretical groundwork for modern computing but also revealed profound limits to what can be calculated

From abacus to computer

Key themes for this monographic part:

- The philosophical and mathematical need for a definition of "effective procedure" and "computable function"
- The development of various formal models of computation.
- The surprising discovery of uncomputable problems
- The birth of computational complexity theory, its relevance to "real" computation, and its central open questions
- The deep connections between logic, proof theory, and computation

The foundational crisis and early formalisms

What Does "Computable" Really Mean?

In this lesson, we will explore:

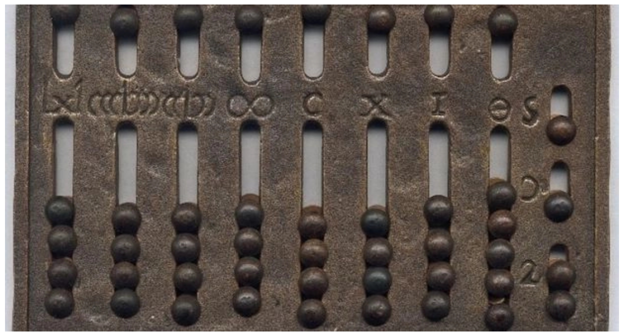
- **The problem of effective calculability**:
  The intuitive notion and the mathematical challenge
- **Hilbert's program revisited**:
  The quest for decidability and its initial setbacks
- **Emil Post's contributions**:
  Early formal systems and the concept of a "machine"
- **Kurt Gödel and incompleteness**:
  The profound limits of formal systems, driving the need for a rigorous definition of computation
- **Moses Schönfinkel's combinatory logic**:
  An early, abstract approach to function definition

Understanding the intellectual landscape that demanded a rigorous definition of computation.

From intuition to rigour

The intuitive notion of an algorithm:

- For centuries, mathematicians used "algorithms" without a formal definition:
  - Euclid's algorithm for greatest common divisor
  - Long division
  - Procedures for solving equations
- An algorithm is generally understood as a finite set of unambiguous instructions that, when applied to an input, produces a result in a finite number of steps


Abacus

From intuition to rigour

The need for formalisation in the early 20th century:

- The foundational crisis in mathematics (paradoxes in set theory, debates over intuitionism vs. formalism) demanded ultimate rigour
- **Hilbert's program** sought to formalise all of mathematics and prove its consistency and decidability. This implicitly required a precise definition of "decidable" or "computable"
- Questions arose: Are there mathematical problems that cannot be solved by **any** algorithm? How would we even prove such a thing without a rigorous definition of "algorithm"?

From intuition to rigour

Key characteristics of "effective calculability":

- **Finiteness**:
  The procedure must be describable in a finite number of instructions
- **Determinism**:
  Each step is uniquely determined by the current state and input
- **Finiteness of steps**:
  The computation terminates in a finite number of steps for any valid input
- **Mechanical execution**:
  No human intuition or creativity is required during execution
- **Bounded resources** (implicit):
  Though not explicitly part of early definitions, the idea of finite memory/time is always present

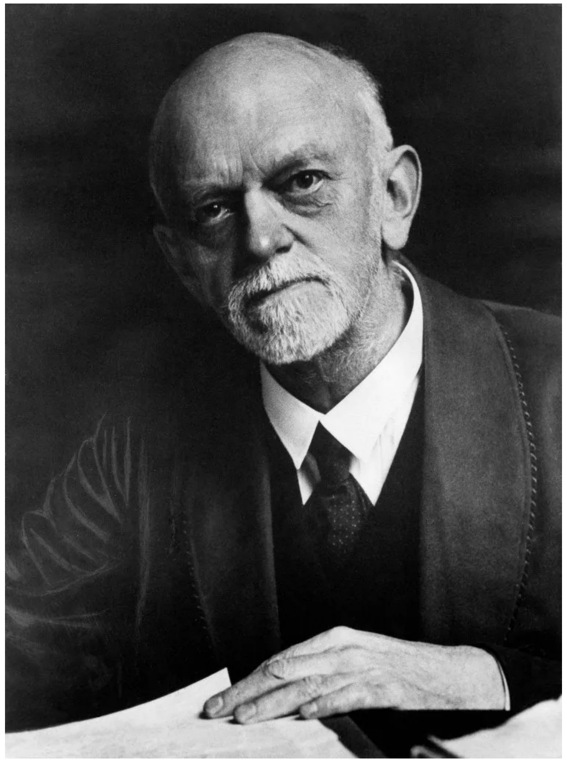
From intuition to rigour

The central question: How can we capture this intuitive notion of a "mechanical procedure" or "algorithm" within a precise, formal mathematical framework? This was the driving force behind the work of Hilbert, Gödel, Church, Turing, and others.

Hilbert's program and its echoes

David Hilbert's ambitious vision
(early 20th century):

- Recall **Hilbert's program**: a grand proposal to formalise all of mathematics using finite, axiomatic systems. Key aims:
  - **Consistency**: Prove that no contradictions could ever be derived within these systems
  - **Completeness**: Show that every true mathematical statement could be proven within the system
  - **Decidability (Entscheidungsproblem)**: Find an "effective procedure" (an algorithm) that could determine, for any given mathematical statement, whether it is true or false
- This "Entscheidungsproblem" directly necessitated a precise definition of "effective procedure" or "algorithm"


David Hilbert

Hilbert's program and its echoes

Kurt Gödel

The setback: Gödel's incompleteness theorems

- in 1931, **Kurt Gödel** delivered a severe blow to Hilbert's program
- **First incompleteness theorem**: For any finite and consistent formal system capable of expressing basic arithmetic, there will always be true statements that cannot be proven (or disproven) within the system
- **Second incompleteness theorem**: Such a system cannot prove its own consistency

- These theorems revealed fundamental limitations to purely formal, axiomatic approaches

Hilbert’s program and its echoes

The aftermath: The quest for calculability intensifies

- Gödel’s work didn’t mean mathematics was inconsistent, but it showed that Hilbert’s ambitious goal of complete decidability was unattainable
- Instead, it sharpened the focus on understanding **what could be** effectively computed or decided
- If not everything could be decided, then precisely **what** could be? This became the central driving question
- This intellectual void directly spurred the independent efforts of Post, Church, Kleene, and Turing to formalise “effective calculability”

Gödel’s legacy: Shifting the focus to computable limits: By demonstrating the inherent incompleteness of formal systems, Gödel inadvertently pushed mathematicians to rigorously define the very notion of computability, turning a crisis into a new field of inquiry.

Early formalisms and abstract machines

Emil Leon Post (1897–1954):
A pioneer in formal logic

- American mathematician and logician, one of the key independent figures in computability theory
- His work, often in parallel to Turing and Church, contributed significantly to the formalisation of effective calculability

Emil Leon Post

Early formalisms and abstract machines

Post production systems (1920s–1940s):

- Post developed formal systems based on “production rules” for manipulating strings of symbols
- These rules specify how a string can be transformed into another string
- This was an early attempt to formalise the notion of a “computation” as a sequence of symbol manipulations

Later, it has been proved that these systems are equivalent to *type-0* grammars in the Chomsky’s hierarchy, thus they recognise every *recursively enumerable* language.

Early formalisms and abstract machines

The Post-Turing machine (1936):

- Independently of Turing, Post also conceived of an abstract computing machine in 1936, remarkably similar to Turing's
- His model involved a worker moving along an infinite sequence of cells, each either marked or unmarked, following a finite set of instructions
- This independent discovery bolstered the idea that such a simple, mechanical model could capture universal computation

In his three pages article *Finite Combinatory Processes—Formulation 1*, Journal of Symbolic Logic 1(3) (1936), Post defines a machine as above. Differently from Turing, no property of his machines is proved, although

> *The writer expects the present formulation to turn out to be logically equivalent to recursiveness in the sense of the Godel-Church development.*

Early formalisms and abstract machines

Post's contributions to undecidability:

- Post explored the implications of these formal systems, proving several problems concerning them to be undecidable
- **Post correspondence problem** (PCP): A classic example of an undecidable problem. Given a collection of "dominoes" (pairs of strings), can you arrange them to form a sequence where the top string equals the bottom string?
- Proved undecidable by Post in 1946, it serves as a common tool for proving other problems undecidable in computer science

Early formalisms and abstract machines

PCP is a *decision problem*: given a set I of *instances* and a subset $A \subseteq I$, the problem asks to find an algorithm, it is exists, such that it decides whether its input $x \in I$ lies in A or not.

In PCP, fix a finite set E of symbols such that $|E| > 1$, and let

$$P = \{(x_1, y_1), \dots, (x_k, y_k) \mid k \geq 1 \text{ and } x_1, \dots, x_k, y_1, \dots y_k \in E\} \quad .$$

Let $I = P^*$, the set of finite sequences of elements in P. Define

$$A = \{[(x_1, y_1), \cdots, (x_n, y_n)] \in P^* \mid x_1 \cdots x_n = y_1 \cdots y_n\} \quad ,$$

i.e., the sequences of dominoes' tiles from P such that the string one reads on the top equals the string one reads on the bottom.

Using as tiles the intermediate states of computation of Turing machines, one shows that deciding A reduces to deciding the halting problem.

Early formalisms and abstract machines

Independent validation of the computable idea: Emil Post's parallel work, particularly his own abstract machine model and proofs of undecidability, provided powerful independent validation for the concepts emerging from Turing's and Church's research, cementing the foundations of computability.

Combinatory logic

Moses Schönfinkel

Moses Schönfinkel (1888–1942):
A precursor in logic

- Russian logician, whose work in the 1920s, though often overlooked, laid foundational ideas for later computability theories
- He sought to eliminate the need for variables in mathematical logic

Combinatory logic

Combinatory logic (1924):

- Schönfinkel introduced the fundamental concept of **combinators**, i.e., higher-order functions that operate on other functions without needing explicit variables
- The most famous is the **S, K, I combinators**:
  - **I (Identity)**: $\mathbf{I}\,x = x$
  - **K (Constant)**: $\mathbf{K}\,x\,y = x$
  - **S (Substitution/Distribution)**: $\mathbf{S}\,f\,g\,x = f\,x\,(g\,x)$
- He showed that all *functions* could be expressed using just a few such combinators, indeed, **K** and **S** only

Combinatory logic

Actually, *functions* in Schönfinkel's sense are exactly the computable functions, a result which can be easily proved using the equivalence between Church's λ-calculus and Turing's machines.

The crucial point is that the set of all the combinators coincide with the set of all *functions* in Schönfinkel's sense. Then, it is shown that all the combinators can be constructed from a very small subset of them.
Later, it has been proved by Haskell Curry that the **K** and **S** combinators suffice to define all the others.

Since $\mathbf{K} = \lambda x.\lambda y.x$ and $\mathbf{S} = \lambda f.\lambda g.\lambda x.(f\,x)(g\,x)$ in the notation of λ-calculus and, conversely, the λ operator can be simulated using **K** and **S**, it follows that the representable functions in Church's λ-calculus and the set of all the combinators are the same.

Combinatory logic

Significance for computability:

- While not a direct model of a "machine", combinatory logic provided a powerful, variable-free system for defining and manipulating functions
- It demonstrated that a very small set of primitive operations could achieve universal functional expression
- This concept of building complex operations from a minimal set of fundamental ones heavily influenced **Alonzo Church's λ-calculus**, which we will discuss next
- It is also a core concept in modern **functional programming** languages

An abstract precursor to functional computation: Schönfinkel's combinatory logic offered an early, highly abstract glimpse into how complex computational processes could be built from simple, universal functional building blocks, foreshadowing later developments in both theoretical computability and programming paradigms.

References

Post’s and Schönfinkel’s relevant articles are available on request: although they are not in the public domain, the University provides access to them:

- *Moses Schönfinkel*, Über die Bausteine der mathematischen Logik, Mathematische Annalen 92 (1924)
- *Emil Post*, Finite Combinatory Processes—Formulation 1, Journal of Symbolic Logic 1(3) (1936)
- *Emil Post*, Formal Reductions of the General Combinatorial Decision Problem, American Journal of Mathematics 65(2) (1943)
- *Emil Post*, A Variant of a Recursively Unsolvable Problem, Bulletin of the American Mathematical Society 52(4) (1946)

For the equivalence between combinatory logic, λ-calculus, Turing machines, and partial recursive functions, we suggests to start with *Barry S. Cooper*, Computability Theory, Chapman and Hall/CRC (2003)

Computable Functions: Lecture 2

Syllabus:

Lambda calculus and recursive functions

- Alonzo Church
- Stephen Kleene

Lambda calculus and recursive functions

Two more paths to define "computable"

In this lesson, we will explore:

- **Alonzo Church** and **Lambda Calculus**: A formal system for expressing computation based on function abstraction and application
- **Recursive functions**: Defining functions through recursion, building on elementary arithmetic operations
- **The equivalence of models**: How these seemingly different approaches (and those of Post and Turing) converge
- **The Church-Turing Thesis** (initial statement): The crucial hypothesis asserting that these models capture all "effective procedures"

Understanding the diverse yet convergent formalisms that shaped our definition of computability

Alonzo Church and the λ-calculus

Alonzo Church (1903–1995):
Logic, computation, and mentors

- American mathematician and logician, a central figure in the development of theoretical computer science
- Known for his work on computability theory, recursion theory, and foundational logic
- Famously advised both Alan Turing and Stephen Kleene during their PhDs
- . . . but also many other important researchers, as Martin Davis, Leon Henkin, Michael O. Rabin, J. Barkley Rosser, Dana Scott, Norman Shapiro, and Raymond Smullyan

Alonzo Church

Alonzo Church and the λ-calculus

The Lambda calculus (λ-calculus, 1930s):

- Developed by Church as a formal system for representing computation using **function abstraction** and **application**
- It's a universal model of computation, meaning any computable function can be expressed and evaluated in λ-calculus
- Unlike machine-based models, λ-calculus is purely abstract, based on symbolic manipulation of functions

Alonzo Church and the λ-calculus

Syntax of λ-calculus:

- **Variables**: $x, y, z, \ldots$
- **Function abstraction** (defining a function): $\lambda x. M$
  - “a function that takes x as an argument and evaluates to M”
- **Function application** (applying a function): $M\,N$
  - “apply function M to argument N”

Computation: by a *reduction* ($\triangleright$) relation

- β-**contraction**: $(\lambda x. M)\,N \triangleright M[x/N]$
- $\triangleright$ is a *congruence*
- $\triangleright$ is *reflexive* and *transitive*

Alonzo Church and the λ-calculus

Simple example: **the identity function**

- **Definition**: $\lambda x.x$, a function that takes x and returns x
- **Application**: $(\lambda x.x)\,\mathsf{True} \rhd \mathsf{True}$
- **Application**: $(\lambda x.x)\,5 \rhd 5$
- **Application**: $(\lambda x.x)(\lambda x.x) \rhd \lambda x.x$

Alonzo Church and the λ-calculus

Natural numbers:

- $0 \equiv \lambda x.\lambda y.x$
- $n+1 \equiv \lambda x.\lambda y.y\,(n\,x\,y)$

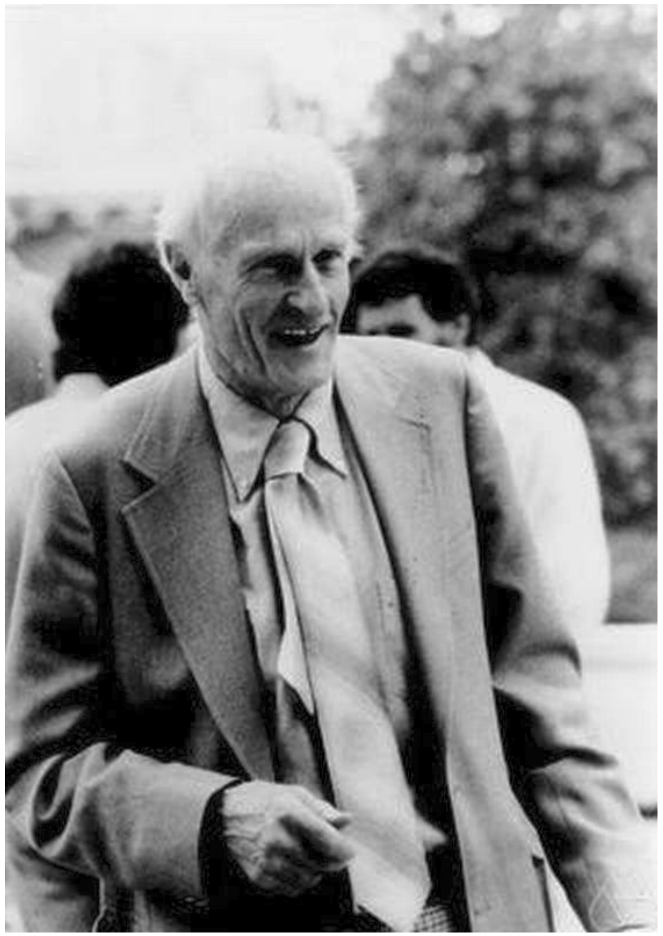
Example: **addition**

$$\lambda x.\lambda y.\lambda z.\lambda w.x\,(y\,z\,w)\,w$$

In the λ-calculus numbers and most common data structures can be represented without using additional syntax.

A foundation for functional programming: λ-calculus not only provided a formal definition of computability but also served as the theoretical bedrock for functional programming languages, which are becoming increasingly influential in modern computing

Stephen Kleene and recursive functions


Stephen Cole Kleene

Stephen Cole Kleene (1909–1994):
A central figure in recursion theory

- American mathematician and logician, a student of Alonzo Church
- Made fundamental contributions to recursion theory, laying the groundwork for what is now known as computability theory
- His work provided an arithmetic-based approach to defining computable functions: the *partial recursive functions*

Stephen Kleene and recursive functions

The concept of recursive functions (1930s):

- Kleene, building on earlier work by Gödel and Herbrand, formalised "recursive functions" as a class of functions that could be computed by purely mechanical means
- The idea is that complex functions can be built up from a small set of basic functions using specific composition rules, particularly **recursion**

Stephen Kleene and recursive functions

Building blocks: Primitive recursive functions

- **Basic functions**:
  - **Zero function** $Z(x) = 0$
  - **Successor function** $S(x) = x + 1$
  - **Projection functions** $P_i^n(x_1, \ldots, x_n) = x_i$
- **Composition rules**:
  - **Composition**: $f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots), \ldots, g_k(x_1, \ldots))$ with h and $g_1, \ldots, g_k$ recursive functions
  - **Primitive recursion**: Defining a function based on its value for $n = 0$ and a recursive step for $n + 1$ where g and h are recursive
    - $f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$
    - $f(x_1, \ldots, x_k, n+1) = h(x_1, \ldots, x_k, n, f(x_1, \ldots, x_k, n))$
- These rules allow definition of many familiar functions (addition, multiplication, exponentiation)

Stephen Kleene and recursive functions

Beyond primitive: General recursive functions

- While powerful, primitive recursive functions always terminate
- To capture **all** computable functions (including those that might not terminate or require an unbounded search), Kleene introduced the **minimisation (or unbounded search) operator** (μ): $f(x_1,\ldots,x_k) = \mu y . g(x_1,\ldots,x_k,y)$ whose value is the minimum y for which $g(x_1,\ldots,x_k,y) = 0$ with g recursive. If such a value does not exists, $f(x_1,\ldots,x_k)$ is undefined
- The set of functions obtained by combining the basic functions with composition, primitive recursion, and the minimisation operator are called **general recursive functions**

Stephen Kleene and recursive functions

Example: Addition as a primitive recursive function

- $add(x,0) = P_1^1(x) = x$ (using a projection to just return x)
- $add(x,y+1) = S(P_3^3(x,y,add(x,y))) = S(add(x,y))$
- This defines $add(x,y)$ based on x and the successor function, demonstrating how simple functions build up

Stephen Kleene and recursive functions

Example: The every undefined function

$$\perp(x) = \mu y . S(Z(x)) = 0$$

Stephen Kleene and recursive functions

An arithmetic-based definition of computability: Kleene's theory provided a rigorous, inductive definition of computable functions, showing that a vast array of complex calculations could be constructed from a minimal set of fundamental arithmetic operations and recursive rules.

The Entscheidungsproblem

Revisiting the Entscheidungsproblem (Decision problem):

- Recall Hilbert's challenge: To find an "effective procedure" to determine, for any given logical statement (in a formal system like first-order logic), whether it is universally valid (provable)
- This was a central part of Hilbert's Program

Church's Result (1936): The undecidability of λ-calculus

- While not directly proving the Entscheidungsproblem itself using λ-calculus, Church proved a crucial related result
  - There is no effective procedure (i.e., no lambda expression) that can determine, for any two given lambda expressions, if they are **equivalent** (i.e., if they compute the same function)
  - Similarly, there is no effective procedure to determine if a lambda expression has a **normal form** (i.e., if it halts or terminates its evaluation)
- This demonstrated an inherent limit to what could be computed within his formal system

The Entscheidungsproblem

Implication for the Entscheidungsproblem (Church's theorem):

- Church then showed that if the Entscheidungsproblem **were** solvable by an effective procedure, then the equivalence of lambda expressions (or their normal form existence) **would also be** solvable
- Since he had just proven that the latter is **not** solvable, it follows that the **Entscheidungsproblem itself is undecidable**
- This was a monumental result: it meant that there are fundamental questions in formal logic (and by extension, mathematics) for which no general algorithmic solution exists

The Entscheidungsproblem

A parallel discovery (Turing):

- Remarkably, later in 1936, **Alan Turing** independently arrived at the same conclusion about the undecidability of the Entscheidungsproblem using his own model of computation (the Turing Machine) and proving the undecidability of the halting problem
- This convergence of independent proofs from different formalisms provided strong evidence for the robustness of the concept of "uncomputable" — computability does not depend on the formalism: it is rather an intrinsic property which can be captured by different systems

The first rigorous demonstration of incomputability: Church's proof marked a groundbreaking moment, providing one of the first rigorous demonstrations that even with a precise definition of "computable", there exist clearly formulated mathematical problems that lie beyond the reach of any algorithm

The equivalence of computational models

Convergence: Different paths, same power

- In the 1930s, independently and using different formalisms, several mathematicians arrived at models for “ effective calculability”
  - **Alonzo Church**: Lambda calculus
  - **Stephen Kleene**: General recursive functions
  - **Emil Post**: Post Production Systems (and his abstract machine model)
  - **Alan Turing**: Turing Machines (which we’ll explore in detail next lesson)
- A remarkable discovery: It was quickly proven that all these models are **computationally equivalent**. If a function is computable in one model, it is computable in all the others
- This strong convergence from diverse starting points lent significant weight to the idea that they had successfully captured the intuitive notion of computability

The equivalence of computational models

The Church-Turing Thesis (Hypothesis):

- **Informal Statement:** *Any function that can be computed by an algorithm (an 'effective procedure') can be computed by a Turing machine (or equivalently, by a lambda expression, or a general recursive function)*
- It asserts that the intuitive notion of "effectively computable" is precisely captured by these mathematical formalisms
- **Why a "Thesis" and not a "Theorem"**?
  - Because "effectively computable" is an intuitive, pre-mathematical concept, it cannot be proven to be equivalent to a precise mathematical definition
  - Instead, it is a widely accepted hypothesis, supported by vast evidence and the absence of counterexamples

The equivalence of computational models

The cornerstone of Computability Theory: The Church-Turing Thesis is a foundational principle of computability theory, computer science, and mathematical logic. It provides a robust, universally accepted definition for what it means for a problem to be "computable", and conversely, to be "uncomputable". Its definition has been the moment in which a new discipline was born: **Computer Science**

References

The relevant literature is accessible through the university library, or ask the instructor for copies:

- *Alonzo Church*, An Unsolvable Problem of Elementary Number Theory, American Journal of Mathematics 58(2)(1936)
- *Stephen Kleene*, A Theory of Positive Integers in Formal Logic. Part I, American Journal of Mathematics. 57(1) (1935)
- *Stephen Kleene*, A Theory of Positive Integers in Formal Logic. Part II, American Journal of Mathematics. 57(2) (1935)
- *Stephen Kleene*, General recursive functions of natural numbers, Mathematische Annalen (112) (1936)
- *Alan Turing*, On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society 42(1) (1936)

Computable Functions: Lecture 3

Syllabus:

The birth of the modern computer concept

- Alan Turing

Alan Turing and the universal machine

The birth of the modern computer concept

In this lesson, we will explore:

- **Alan Turing's vision**:
  His motivation stemming from Hilbert's Entscheidungsproblem
- **The Turing machine** (TM):
  A detailed look at its simple yet powerful architecture and operation
- **The universal Turing machine** (UTM):
  The groundbreaking concept of a programmable, multi-purpose computer
- **The halting problem**:
  Turing's direct proof of its undecidability, one of the most famous limits of computation
- **Revisiting the Church-Turing thesis**:
  How Turing's model reinforces the fundamental hypothesis

From an abstract logical problem to the blueprint of the digital age

Alan Turing: A genius ahead of his time

Alan Mathison Turing (1912–1954):
A founding father

- British mathematician, logician, computer scientist, and philosopher
- Widely considered the "father of theoretical computer science and artificial intelligence"
- His work during WWII (code-breaking at Bletchley Park) cemented his practical legacy, but his theoretical work in the 1930s laid the foundation for everything

Alan Mathison Turing

Alan Turing: A genius ahead of his time

The driving question: Hilbert's Entscheidungsproblem (Again!)

- Turing, like Church, was deeply influenced by David Hilbert's challenge: was there a mechanical procedure to decide the truth of any mathematical statement?
- To answer this, he first needed a precise, unambiguous definition of what a "mechanical procedure" (or algorithm) actually was
- Unlike Church's abstract functional approach, Turing imagined a physical, albeit idealised, machine

Alan Turing: A genius ahead of his time

Turing's intuition: The Human "Computer"

- Turing observed human mathematicians (then called "computers") performing calculations:
    - They operate on symbols
    - They use a finite set of rules
    - They write on a piece of paper (or tape)
    - They only pay attention to a small part of the paper at any one time
    - Their actions are completely determined by their current state and the symbol they are observing
- He sought to distil these actions into the simplest possible formal model

From abstract problem to concrete model: Turing's genius lay in translating the abstract philosophical problem of "effective calculability" into a concrete, yet highly idealised, mechanical device, providing a robust and intuitive model for what it means to compute

The Turing machine: Simple yet universal

Defining "mechanical procedure" through a machine:

- In his seminal 1936 paper, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Turing introduced his abstract machine
- It formalised the actions of a human "computer" into the most minimal set of operations
- Despite its simplicity, the Turing Machine (TM) is posited to be able to perform **any** computation that a human or any other mechanical device can perform

The Turing machine: Simple yet universal

Core components of a Turing machine:

A Turing machine

- **Infinite tape**: A conceptual “paper” divided into cells, each capable of holding a single symbol from a finite alphabet (e.g., $\{0, 1, \text{Blank}\}$). It’s infinitely extendable in both directions
- **Read/write head**: Sits over one cell on the tape. It can:
  - **Read** the symbol in the current cell
  - **Write** a new symbol to the current cell
  - **Move** one cell to the Left (L) or Right (R)
- **State register**: Stores the current state of the machine from a finite set of internal states (e.g., $q_0, q_1, \ldots, q_n$)
- **Transition function** (table of instructions): The “program” of the TM. It dictates the machine’s behaviour:
  - Based on (current state, symbol read) → (new symbol to write, direction to move, new state)

The Turing machine: Simple yet universal

How a Turing machine computes:

- The computation starts in an initial state, with an input string on the tape
- The head begins at a designated position (e.g., leftmost input symbol)
- At each step, the machine follows the rule in its transition table corresponding to its current state and the symbol under the head
- The computation continues until the machine enters a special "halt" state, at which point the output is the contents of the tape

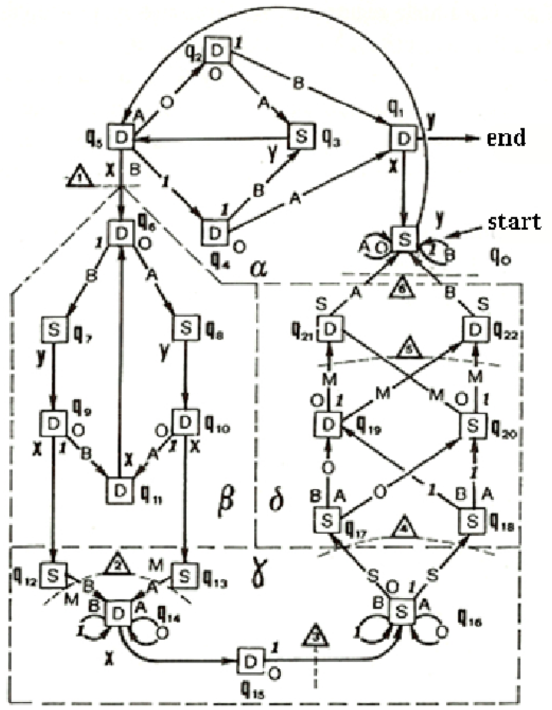
The power of simplicity: The Turing machine's elegance lies in its fundamental simplicity. With just these few components and basic operations, it can simulate any algorithm, making it the bedrock model for theoretical computer science and our definition of computability

The Universal Turing Machine (UTM)

From specific machines to a general-purpose computer:

- Any given Turing Machine (TM) is typically “hardwired” to perform one specific task (e.g., add two numbers, check for primes)
- Turing’s most profound insight was the concept of a **Universal Turing Machine** (UTM)
- A UTM is a single, fixed Turing Machine that can **simulate the behaviour of any other arbitrary Turing machine**

The Universal Turing Machine (UTM)


A universal Turing machine (Aiello et al., 1976)

How it works: Program as data

- Imagine each specific TM's description (its states and transition rules) can be encoded as a unique string of symbols. This is its "program"
- The UTM takes **two inputs** on its tape:
  - The **encoded description** of another Turing machine, *M*.
  - The **input data**, *I*, for that machine *M*
- The UTM then "reads" the description of *M* and, step-by-step, mimics exactly what *M* would do if it were running on *I*

Showing a small UTM is difficult: the smallest UTMs known up to now have been presented in Yurii Rogozhin's article (1996)

The Universal Turing Machine (UTM)

The blueprint for the digital age:

- This was the first theoretical model of a **general-purpose programmable computer**
- It demonstrated that one machine, given the right "program" (encoded as data), could perform **any** task that **any other** machine could perform
- This idea directly prefigured **John von Neumann's stored-program computer architecture**, where instructions (software) are stored in the same memory as data, allowing for flexible, re-programmable machines

Revolutionising computation: The power of programmability: The UTM transformed computation from a series of fixed-function devices to a dynamic, software-driven paradigm, laying the conceptual cornerstone for every modern computer, smartphone, and digital system we use today

The halting problem

The question: Will this program ever stop?

- The **halting problem** asks: Given an arbitrary Turing machine (or program) M and an arbitrary input I for that machine, can we determine, in a finite amount of time, whether M will eventually **halt** (terminate) or run forever (loop infinitely) when started with input I?
- This is an incredibly practical question: every programmer has faced the desire for a tool that could tell them if their code contains an infinite loop!

The halting problem

Turing's proof by contradiction (1936):

- Turing famously proved that the halting problem is **undecidable**. No general algorithm or Turing machine can solve it for all possible inputs
- The proof uses a powerful technique called **diagonalisation**, similar in spirit to Cantor's proof that real numbers are uncountable
- **Assumption** (for contradiction): Assume such a machine, let's call it H (for Halting), exists
  - $H(M, I)$ outputs "halts" if M halts on I
  - $H(M, I)$ outputs "loops" if M loops on I

The halting problem

The construction of the diagonal machine D:

- We construct a new Turing machine, D, which takes (the description of) a machine X as input
- D's behaviour on input X is defined as follows:
  - **1.** D **runs** $H(X,X)$: It uses our hypothetical halting machine H to determine if machine X halts when given its **own description** X as input
  - **2. If** $H(X,X)$ **says "halts"** then D enters an infinite loop (e.g., just keeps moving right indefinitely)
  - **3. If** $H(X,X)$ **says "loops"** then D immediately halts

The halting problem

The contradiction: What happens when D runs on itself ($D(D)$)?

- **Assume $H(D, D)$ says "halts"**
  - By rule 2 above, if $H(D, D)$ says "halts", then D **loops forever**
  - This contradicts our assumption that $H(D, D)$ says "halts"
- **Assume $H(D, D)$ says "loops"**
  - By rule 3 above, if $H(D, D)$ says "loops", then D **halts immediately**
  - This contradicts our assumption that $H(D, D)$ says "loops"

Profound implications of undecidability: Since both possibilities lead to a contradiction, our initial assumption that a general halting machine H exists must be false. The halting problem is fundamentally undecidable. This sets a crucial boundary on what can be solved by algorithms and lays the groundwork for understanding intrinsic computational limits

The Church-Turing thesis

Revisiting the core hypothesis:

- Recall the **Church-Turing thesis**: The intuitive notion of "effective calculability" is precisely captured by formal models like Lambda Calculus and Recursive Functions.
- Turing's independent work provided yet another powerful model, the Turing machine, to represent effective procedures

Turing's model: The intuitive standard:

- The Turing machine, with its simple, mechanical steps, proved to be an exceptionally intuitive and compelling model for what an "algorithm" or "computation" truly entails
- Its concrete nature (tape, head, states) resonated strongly with the human experience of performing calculations
- Church's review of Turing's article was the first to observe how natural and simple Turing's model were

The Church-Turing thesis

The confluence of ideas:

- The fact that Turing's machine model, Church's lambda calculus, Kleene's recursive functions, and Post's systems were all proven to be **computationally equivalent** provided overwhelming evidence
- This convergence from diverse mathematical and philosophical starting points greatly strengthened the belief in the Church-Turing thesis
- It suggests that there is a single, fundamental concept of "computability" independent of the specific formalism used to define it

The Church-Turing thesis

Foundation of Computer Science:

- The Church-Turing thesis, validated by Turing's work, forms the **cornerstone of theoretical computer science**
- It allows computer scientists to confidently say that if a problem cannot be solved by a Turing machine, it cannot be solved by **any** computer, regardless of its architecture or programming language
- This provides a rigorous basis for understanding both the power and the inherent limitations of computation

From intuition to universally accepted definition: Turing's work provided the most concrete and widely adopted formalisation of "effective calculability", solidifying the Church-Turing thesis as the definition of what it means for a problem to be solvable by an algorithm

References

The relevant literature is accessible through the university library, or ask the instructor for copies:

- *Alan Turing*, On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society 42(1) (1936)
- *Yurii Rogozhin*, Small universal Turing machines, Theoretical Computer Science 168(2) (1996)
- *Alonzo Church*, Review of “A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2 s. vol. 42 (1936–1937), pp. 230–265.”, Journal of Symbolic Logic 2(1) (1937)

Computable Functions: Lecture 4

Syllabus:

The limits of computation

The limits of computation: undecidability

In this lesson, we will explore:

- **Recap: The notion of undecidability**:
  Reaffirming the concept from previous lessons
- **The halting problem revisited**:
  A quick review of its significance
- **Other fundamental undecidable problems**:
  Examining more examples beyond the halting problem
- **Rice's theorem**:
  A powerful generalisation for proving properties about programs
- **Hilbert's 10th problem**:
  Its surprising connection to computability
- **Implications of undecidability**:
  What these limits mean for mathematics, computer science, and practical problem-solving

Understanding the intrinsic boundaries of what algorithms can achieve

No algorithm will suffice

What "Undecidable" truly means:

- A problem is **undecidable** if no algorithm (i.e., no Turing machine, no λ-expression, no general recursive function) exists that can solve it for **all possible inputs** in a finite amount of time
- This is a stronger statement than "we just haven't found the algorithm yet". It means one fundamentally **cannot exist**

Contrast with decidable problems:

- Most problems we encounter daily are decidable (e.g., sorting a list, finding the shortest path, checking if a number is prime)
- For these, an algorithm is guaranteed to produce an answer for every valid input in finite time

No algorithm will suffice

The halting problem: Our first encounter with undecidability

- As discussed in the previous lesson, the halting problem asks if a given program will terminate on a given input
- Turing's proof by contradiction established this problem as fundamentally undecidable: we can still decide whether some specific program on some specific input does terminate, but in general, we cannot
- This implies that no perfect universal debugger or program analyser can ever be built because it does not exist

No algorithm will suffice

Why does undecidability matter?

- **Fundamental limits**: It defines the intrinsic boundaries of what computation can achieve, regardless of technological advancements
- **Theoretical importance**: It highlights the power and limitations of formal systems and logical reasoning
- **Practical implications**: While a problem may be undecidable in general, specific instances might be solvable. It pushes us to find effective **heuristics** or **approximations** rather than perfect algorithms
- Many real-world problems (e.g., in software verification, AI safety) contain undecidable sub-problems

From "Can we solve it?" to "Can it ever be solved algorithmically?"
Undecidability forces us to ask deeper questions about the nature of problems themselves, rather than just seeking solutions

More fundamental undecidable problems

Beyond halting: Undecidability in various domains

- The halting problem is the canonical example, but undecidability permeates many areas of mathematics and computer science
- Proving a problem undecidable often involves **reduction**: showing that if you could solve the new problem, you could also solve a known undecidable problem (like the halting problem)

For example, deciding whether a program terminates for every input can be immediately reduced to the halting problem, thus it is undecidable.

More fundamental undecidable problems

The Post correspondence problem (PCP):

- Introduced by Emil Post (1946)
- **Problem**: Given a finite collection of "dominoes", where each domino has a top string and a bottom string (e.g., '[a/ab], [b/ba], [baa/a]'). Is it possible to pick a sequence of these dominoes (with repetitions allowed) such that the concatenated top strings form the same string as the concatenated bottom strings?
- **Undecidable**: There is no algorithm that can determine, for any given set of dominoes, whether such a sequence exists
- **Significance**: PCP is often used as a tool to prove the undecidability of other problems (e.g., in formal language theory)

More fundamental undecidable problems

The undecidability of first-order logic (Entscheidungsproblem):

- As discussed, formalised by Hilbert, proven undecidable by **Church** and **Turing** (1936)
- **Problem**: Given an arbitrary well-formed formula in first-order logic, is it universally valid (i.e., true in all interpretations)?
- **Undecidable**: No algorithm can solve this for all formulas. This means automatic theorem proving for full first-order logic is fundamentally limited and automatic decision within a first-order theory, e.g., arithmetic or analysis, is undecidable too

More fundamental undecidable problems

Unsolvable puzzles and tiling problems:

- Many seemingly simple tile-matching puzzles or tiling problems (e.g., given a finite set of coloured tiles, can you tile an infinite plane such that adjacent edges match colours?) are also undecidable
- The complexity arises from the infinite nature of the plane and the potential for non-repeating patterns
- Example: **Wang Tiles** (Wang, 1961) — determining if a given finite set of square tiles can tile the infinite plane

Rice's theorem

Henry Gordon Rice (1920–2003): Generalising Undecidability

- American mathematician and computer scientist
- In 1953, he proved a powerful meta-theorem that provides a very broad condition under which properties of programs are undecidable
- It generalises many individual undecidability proofs

Rice’s theorem

Rice’s theorem statement: Any **non-trivial property** of the **language accepted** (or function computed) by a Turing machine is undecidable

- **Property**: A set of Turing machines that share some characteristic
- **Language accepted/function computed**: Crucially, the property must be about the **behaviour** or **input/output relationship** of the program, not its internal structure
- **Non-trivial**: The property must not be true for **all** Turing machines, nor false for **all** Turing Machines. (If it’s trivial, you don’t need to run the program to know the answer)

Rice's theorem

Examples of properties PROVEN Undecidable by Rice's theorem:

- Does a given program halt on **all** inputs? (*The Totality Problem*)
- Does a given program halt on **any** input? (*The Emptiness Problem*)
- Does a given program compute the identity function?
- Does a given program compute a prime number?
- Is a given program a compiler?
- Is a given program equivalent to another given program?

Rice’s theorem

Examples of Properties NOT Covered by Rice’s Theorem (Decidable!):

- Does a given program contain 10 lines of code? (Structural property)
- Does a given program use a ‘FOR’ loop? (Structural property)
- Does a given program contain the substring “ Hello World”?

A powerful tool for proving incomputability: Rice’s Theorem is a cornerstone for demonstrating the undecidability of a vast array of problems in computer science. It highlights that most interesting properties about what a program **does** are beyond algorithmic determination.

Hilbert's 10th problem

David Hilbert's challenge (1900 Paris address):

- Among his famous 23 open problems, Hilbert's 10th problem asked for a general "process" (an algorithm) to determine if a Diophantine equation has integer solutions
- **Diophantine equation**: A polynomial equation where only integer solutions are sought
  - Example: $x^2 + y^2 = z^2$ (Pythagorean triples) — has integer solutions
  - Example: $x^2 = 2$ — no integer solutions
- The problem: Given $P(x_1, x_2, \ldots, x_n) = 0$, where P is a polynomial with integer coefficients, does there exist integers $x_1, \ldots, x_n$ such that the equation holds?

Hilbert's 10th problem

The unexpected answer: Undecidable!

- After decades of work by numerous mathematicians, it was definitively proven in 1970 that **no such general algorithm exists**. Hilbert's 10th problem is undecidable.

Key contributions to the solution:

- **Martin Davis** (1950): Showed the connection between Diophantine sets and recursively enumerable sets
- **Hilary Putnam & Julia Robinson** (1960s): Made significant progress, formulating a hypothesis that would imply the undecidability
- **Yurii Matiyasevich** (1970): Provided the final crucial step with **Matiyasevich's theorem** (also known as the MRDP theorem: Matiyasevich-Robinson-Davis-Putnam)

Hilbert's 10th problem

Matiyasevich's theorem and its link to computability:

- Matiyasevich proved that **Diophantine sets are precisely the recursively enumerable sets**
- **Recursively enumerable (RE) set**: A set for which there exists an algorithm that will list (enumerate) all its members
- We know that the halting problem is recursively enumerable (you can list all halting computations) but not recursive (decidable)
- Therefore, asking if a Diophantine equation has a solution is equivalent to asking if a given Turing machine halts. Since the halting problem is undecidable, so is Hilbert's 10thproblem

Profound implication: Limits within pure number theory: This result was a major shock, demonstrating that even in seemingly pure, finite areas of mathematics like number theory, fundamental questions can be intrinsically uncomputable, lying beyond the reach of any general algorithm

Why do these limits matter?

Fundamental limits of algorithms:

- Undecidability establishes that there are intrinsic boundaries to what algorithms can achieve, regardless of computational power or technological advancements
- It's not about finding a faster computer, but about the fundamental non-existence of a general solution

Why do these limits matter?

Impact on Computer Science and Software Engineering:

- **No perfect program analysers**: The undecidability of the halting problem and Rice's theorem imply that no general algorithm can perfectly detect all infinite loops, prove program correctness for all cases, or perfectly identify all malicious code
- **Design philosophy shift**: Engineers must rely on heuristics, approximations, domain-specific solutions, or interactive verification for problems with undecidable sub-components
- It guides research: Instead of seeking a universal solution, focus shifts to finding effective **partial** solutions or deciding simplified versions of undecidable problems

Why do these limits matter?

Profound philosophical & logical insights:

- Further reinforces **Gödel's incompleteness results**: Formal axiomatic systems (like mathematics itself) are inherently incomplete or limited in their decidability
- It poses questions about the nature of mathematical truth and the limits of purely mechanical reasoning

Shaping mathematical research:

- The undecidability of problems like Hilbert's 10^{th} demonstrates that not all well-posed mathematical questions have algorithmic answers
- It shifts focus from merely **finding** algorithms to proving their **non-existence**

Why do these limits matter?

Understanding the scope of AI:

- While AI aims to solve complex problems, it operates within the boundaries of computability. An AI cannot solve a truly undecidable problem in a general sense
- However, AI excels at finding patterns, making predictions, and developing heuristics for problems that are computationally hard or partially undecidable in practice

From crisis to clarification: What began as a foundational crisis in mathematics led to a profound clarification of the nature of computation itself. Undecidability isn't a dead end, but a map, showing us where the algorithmic path ends and where other forms of inquiry (intuition, approximation, human creativity) must begin

References

The relevant literature is accessible through the university library, or ask the instructor for copies:

- *Alan Turing*, On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society 42(1) (1936)
- *Hao Wang*, Proving theorems by pattern recognition—II, Bell System Technical Journal, 40(1) (1961)
- *Henry Gordon Rice*, Classes of recursively enumerable sets and their decision problems, Transactions of the American Mathematical Society, 74 (2) (1953)
- *Barry S. Cooper*, Computability Theory, Chapman and Hall/CRC (2003)

Computable Functions: Lecture 5

Syllabus:

Von Neumann and the birth of Computer Science

From abstract theory to physical machines

In this lesson, we will explore:

- **Bridging the gap**:
  Connecting the universal Turing machine to electronic computers
- **John von Neumann's vision**:
  His pivotal role in defining the architecture of modern computers
- **The stored-program concept**:
  The revolutionary idea that programs are data
- **The von Neumann architecture**:
  Key components and their function
- **Early electronic computers**:
  Pioneers like ENIAC, EDVAC, and EDSAC
- **The emergence of Computer Science**:
  Its beginnings as a distinct academic discipline

Understanding how the abstract quest for computability led to electronic computers and the digital revolution

The path to electronic computing

Human “computers” at Harvard around 1890

The pre-electronic era:
Calculation by hand and mechanical aids

- For centuries, complex calculations were performed by human “computers” (often women) or by mechanical devices (abacus, slide rules, mechanical calculators like Pascal’s or Leibniz’s)
- Even during WWII, massive calculations (e.g., ballistics tables) relied on large teams of human computers and early electromechanical machines (like the Harvard Mark I)

The path to electronic computing

The growing demand for speed and automation:

- Scientific, engineering, and military demands in the early 20th century (especially during WWII) pushed the limits of manual and human-performed mechanical computation
- Problems requiring millions of operations for a single result (e.g., simulating nuclear reactions, weather prediction) became pressing
- The discipline of **Numerical Analysis**, focused on finding approximate solutions to complex mathematical problems, underwent a profound transformation due to these demands, as it required far more computational power than was available
- The immense computational requirements for code-breaking (e.g., Bletchley Park's efforts against Enigma, involving figures like Alan Turing) also significantly spurred the quest for faster, automated calculation
- The need for machines that could perform calculations orders of magnitude faster, and with less human intervention, became critical

The path to electronic computing

The theoretical bridge: Turing's universal machine

- While a purely abstract concept, the **Universal Turing Machine** (UTM) provided the theoretical blueprint:
  - It showed that a single, programmable machine could perform any computation, not requiring a specialised machine for every specific task
  - It demonstrated that both instructions (program) and data could reside in the same format
- The challenge now was to turn this theoretical blueprint into a practical, electronic reality

From slow, dedicated calculators to fast, general-purpose machines: The drive for unprecedented computational power, combined with Turing's abstract model of universal computation, created the perfect conditions for the advent of electronic digital computers

John von Neumann

John von Neumann

János “John” Lajos Neumann
(1903–1957)

- Hungarian-American mathematician, physicist, computer scientist, and polymath
- Made fundamental contributions to quantum mechanics, functional analysis, set theory, economics (game theory), fluid dynamics, and, crucially, computer science
- He was a key figure in the Manhattan Project and recognised the immense potential of electronic computation for scientific and military problems

John von Neumann

The need for a new design paradigm:

- Early electronic computers (like ENIAC, which we'll discuss) were colossal and had to be "re-wired" or manually reconfigured for each new problem. This was incredibly time-consuming and inefficient
- The inspiration from the universal Turing machine was clear: a single, general-purpose machine could perform any possible calculation if properly instructed
- The challenge was to implement this **programmability** efficiently in an electronic device

John von Neumann

The "First Draft of a Report on the EDVAC" (1945):

- While a collaborative effort involving many brilliant minds (Mauchly, Eckert, Goldstine, et al.), von Neumann's influential report clearly articulated the fundamental concepts for the logical design of a modern computer
- This document is widely credited with outlining the **stored-program concept** and the architecture that bears his name

From theoretical universal machine to practical architecture: Von Neumann translated the abstract concept of a universal computing machine into a concrete, realisable architectural blueprint, directly leading to the first generation of general-purpose electronic digital computers

The stored-program concept

The revolution in computer design (Post-WWII):

- Before the mid-1940s, early electronic computing machines (like ENIAC) were primarily **fixed-program** machines. To change the computation, they required physical re-wiring, switch-setting, or manual patch-cord adjustments
- This process was tedious, error-prone, and could take days or even weeks for complex problems, severely limiting their versatility

The stored-program concept

The core idea: Program as data

- The **stored-program concept** (most clearly articulated by von Neumann in his EDVAC report, but ideas were also circulating among Eckert, Mauchly, Turing, and others) is deceptively simple but profoundly powerful:
  - **Instructions (the program) and data are stored together in the same memory unit**
  - They are represented in the **same binary format**
  - The machine can **execute instructions** from memory and also **treat instructions as data** (e.g., modify them)

The stored-program concept

Why it was a "game changer":

- **True programmability**: Made computers genuinely general-purpose. The hardware remained fixed; only the "software" (the stored program) needed to change for a new task
- **Flexibility**: A single machine could run any algorithm, from scientific calculations to word processing, simply by loading a different program
- **Self-modifying code**: (Though largely discouraged today for safety and readability) programs could modify their own instructions, enabling advanced techniques
- **Realisation of the UTM**: This architecture provided the practical framework for the theoretical universal Turing machine, where the TM's "program" (its transition function) is part of its "input" (on the tape)

The stored-program concept

The cornerstone of modern digital computing: The stored-program concept transformed computers from fixed-purpose calculators into flexible, re-programmable machines, directly paving the way for the entire digital age and the software industry

The von Neumann architecture

A universal blueprint for digital computers:

- The “von Neumann architecture” describes a computer design model where the program code and data are stored in the same address space
- This design facilitates the stored-program concept and forms the logical and functional foundation of almost every general-purpose computer built since the late 1940s

The von Neumann architecture

Key components and their roles:

- **Central Processing Unit** (CPU): The “brain” of the computer, responsible for executing instructions. It consists of:
    - **Arithmetic Logic Unit** (ALU): Performs arithmetic operations (addition, subtraction) and logical operations (AND, OR, NOT)
    - **Control Unit** (CU): Manages and coordinates the flow of data and instructions within the CPU and to other components. It fetches instructions from memory, decodes them, and directs the ALU
    - **Registers**: Small, fast storage locations within the CPU used for temporary data and instruction holding during processing
- **Main Memory Unit:**
    - Stores both **instructions (the program)** and **data**
    - Each storage location has a unique address for direct access

The von Neumann architecture

- **Input/Output (I/O) Devices**:
  - Allow the computer to communicate with the outside world (e.g., keyboards, screens, printers, disk drives)
- **Bus**:
  - A set of parallel wires or pathways that connect all the major internal components of a computer, allowing data, addresses, and control signals to be transferred between them

The von Neumann architecture

The Fetch-Decode-Execute cycle (The machine cycle):

- The fundamental operation of a von Neumann machine:
  1. **Fetch**: Retrieve an instruction from memory
  2. **Decode**: Interpret the instruction
  3. **Execute**: Perform the specified operation
- This cycle repeats continuously until a halt instruction is encountered

The von Neumann bottleneck: A known limitation of this architecture is the “von Neumann bottleneck” — the relatively slow speed of data transfer between the CPU and memory compared to the CPU’s processing speed, which can limit overall performance. This is why other architectures (like Harvard) were explored for specific tasks

Early electronic computers

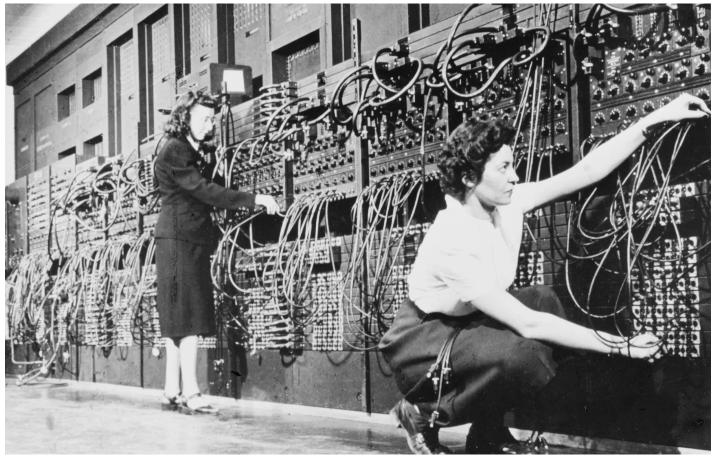
The dawn of the electronic age (1940s–1950s):

- The post-war period saw intense efforts to build functional electronic computers, driven by scientific, military, and commercial needs
- These machines were monumental in size, power consumption, and cost, often using thousands of vacuum tubes

Early electronic computers

ENIAC (Electronic Numerical Integrator and Computer, 1946):

- Built at the University of Pennsylvania by J. Presper Eckert and John Mauchly
- **Significance**: The first general-purpose electronic digital computer. Extremely fast for its time
- **Limitation**: Not a stored-program computer. Programming required tedious manual re-wiring and switch setting, taking days or weeks to change tasks



The ENIAC computer (1946)

Early electronic computers

Manchester baby (Small-Scale Experimental Machine — SSEM, 1948):

- Built at the University of Manchester, UK, by Frederic C. Williams and Tom Kilburn
- **Significance**: The **first electronic stored-program computer to run a program**, demonstrating the feasibility of the stored-program concept. A proof-of-concept

EDVAC (Electronic Discrete Variable Automatic Computer, 1949):

- Successor to ENIAC, designed with strong influence from John von Neumann's "First Draft" report
- **Significance**: One of the earliest machines designed explicitly to implement the **stored-program concept** and binary arithmetic
- Development was prolonged, and it began operating after EDSAC

Early electronic computers

EDSAC (Electronic Delay Storage Automatic Calculator, 1949):

- Built at Cambridge University, UK, by Maurice Wilkes and his team
- **Significance**: The **first practical electronic stored-program computer to actually operate and perform useful computations** (running its first program on May 6, 1949). It directly influenced early computer science education

From calculation to computation: The realisation of a vision: These pioneering machines, particularly those implementing the stored-program concept, transformed the theoretical dream of universal computation into a tangible reality, laying the physical groundwork for the entire digital revolution

The birth of Computer Science

Convergence of necessity and theory:

- The demands of World War II (ballistics, code-breaking, nuclear physics) highlighted an acute need for powerful computation
- Simultaneously, theoretical work by Church, Turing, Post, Kleene, and von Neumann provided a rigorous mathematical foundation and architectural blueprint for these machines
- This unique convergence of practical necessity and profound theory spurred the rapid development of a new field

The birth of Computer Science

Early academic centres and programs (1940s-1950s):

- Pioneers like Cambridge University (EDSAC), University of Manchester (Manchester Baby), Princeton (IAS machine), MIT, and the University of Pennsylvania (ENIAC, EDVAC) became early hubs for research and development, often driven by the needs of their scholars
- Initial programs often emerged from mathematics or electrical engineering departments of universities
- Early computer scientists were often polymaths: mathematicians, logicians, physicists, and engineers

The birth of Computer Science

From "calculators" to "information processors":

- The initial focus was on numerical computation, but the **stored-program concept** quickly revealed the machines' potential for symbolic manipulation and logical processing
- This broader scope, encompassing algorithms, data structures, and the very nature of information, progressively differentiated it from traditional engineering or mathematics

The birth of Computer Science

Formal recognition and growth:

- The 1950s and 60s saw the formal establishment of dedicated computer science departments
- Professional societies (like the ACM in 1947 and IEEE Computer Society) and specialised journals emerged, solidifying its identity
- The term "Computer Science" gained widespread acceptance, reflecting the scientific study of computation and information

The foundation of the digital age: The birth of Computer Science as a distinct academic and professional discipline laid the intellectual groundwork for the entire digital revolution, fostering research and innovation that continues to transform every aspect of modern life

References

Most relevant literature is accessible through the university library, or ask the instructor for copies:

- Report on the ENIAC (Electronic numerical integrator and computer) — available online
- *John von Neumann*, First Draft of a Report on the EDVAC (1945) — available online
- *Maurice V.Wilkes*, *David J. Wheeler*, *Stanley Gill*, The Preparation of Programs for an Electronic Digital Computer, Addison Wesley (1951)

For fun, search on Google for **OXO**, possibly the very first video-game ever developed: it was programmed as part of a thesis at the University of Cambridge in 1952

Computable Functions: Lecture 6

Syllabus:

Automata and formal languages

Automata and Formal Languages

In this lesson, we will explore:

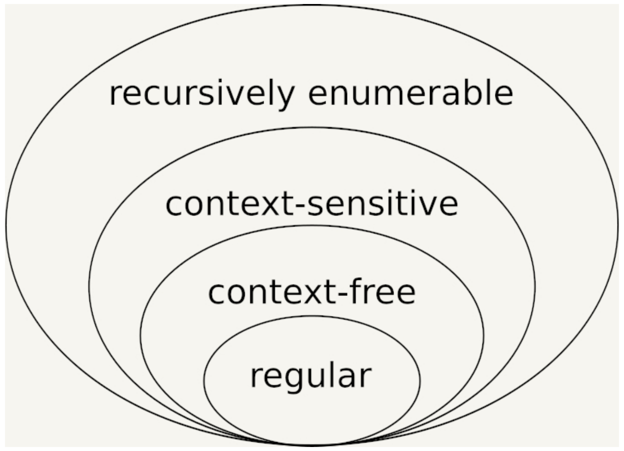
- **Beyond Turing machines**:
  Simpler computational models and their capabilities
- **Finite automata** (FSA):
  The simplest model, its components, and what it can recognise
- **Pushdown automata** (PDA):
  Adding memory to increase computational power
- **Formal grammars and languages**:
  Defining languages with rules (Chomsky hierarchy)
- **The Chomsky hierarchy**:
  A classification of languages and their corresponding automata
- **Applications**;
  Compiler design, natural language processing, pattern matching

Understanding the expressive power of computational models and the structure of languages

The hierarchy of computation

Turing machines: The universal standard

- We've established the Turing machine as the ultimate model of computation, capable of performing any algorithm
- It defines the upper bound of what is "computable"



A hierarchy of computational power

The need for simpler models:

- Not every computational task requires the full power of a Turing machine
- For many practical problems, a simpler, less powerful model might be sufficient, and often more efficient to design and analyse
- Understanding these simpler models helps us characterise the complexity and nature of different problems more precisely

The hierarchy of computation

The analogy of a toolbox:

- Imagine building. You wouldn't use a bulldozer for every task. Sometimes a hammer is enough, sometimes a screwdriver
- Different computational problems require different "tools" (models) with just the right amount of power

Key questions for each model:

- **What problems can it solve/what languages can it recognise**? (Its computational power)
- **What are its limitations**? (What problems are beyond its reach?)
- **What is its corresponding formal language class**? (The relationship between machine and language)

The hierarchy of computation

Applications across Computer Science:

- Compiler design (parsing programming languages)
- Text processing and pattern matching (regular expressions)
- Network protocols
- Artificial intelligence (state machines)
- Digital circuit design

Matching the tool to the task: By studying a hierarchy of automata, we learn to match the appropriate computational model to the complexity of a given problem, leading to more efficient and robust solutions

Finite automata: The simplest recogniser

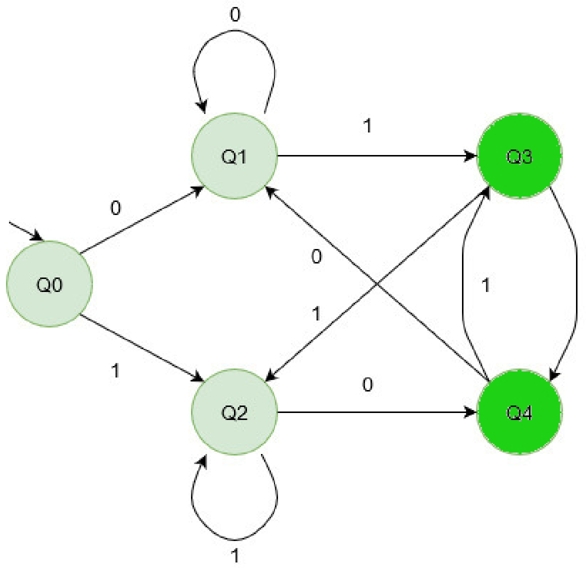
Finite State Machines (FSMs) / Finite Automata (FSA):

- The simplest model of computation, with a finite amount of memory (its current state)
- They recognise patterns in strings of symbols
- Widely used in practical applications where limited memory and quick decision-making are needed

Finite automata: The simplest recogniser

Core components of an FSA:

- **States** (Q): A finite set of discrete states. The machine is always in exactly one state
- **Input alphabet** (Σ): A finite set of symbols that the machine can read
- **Transition function** (δ): A rule that dictates the next state given the current state and the input symbol read. ($\delta : Q \times \Sigma \rightarrow Q$)
- **Start state** (q_0): The unique state where the automaton begins its computation
- **Accept (final) states** (F): A subset of states that indicate successful recognition of an input string. If the machine finishes reading the entire input and ends in an accept state, the string is "recognised" or "accepted"



Example of a finite automaton recognising strings ending with '01' or '10'

Finite automata: The simplest recogniser

How an FSA operates:

- The FSA starts in the initial state
- It reads input symbols one at a time, from left to right
- For each symbol, it transitions to a new state as defined by its transition function, following an arrow in the graphical representation
- The head only moves right; it cannot write or go back
- When the entire input string is processed, if the machine is in an accept state, the string is accepted. Otherwise, it is rejected

Finite automata: The simplest recogniser

Capabilities and limitations:

- **Can recognise**: All **regular languages** (e.g., strings ending in "00", simple keywords, valid email address formats, patterns matched by regular expressions, strings of even length)
- **Cannot recognise**: Languages requiring "memory" of arbitrary depth or counting beyond a fixed limit (e.g., correctly nested parentheses, palindromes, $a^n b^n$ where n can be arbitrarily large)

The foundation of pattern matching: Finite automata are the underlying model for regular expressions and form the basis for lexical analysis in compilers, simple protocol parsing, and basic pattern recognition

Pushdown automata: Adding memory

Beyond regular languages: The need for stack memory

- As we saw, finite automata cannot handle languages requiring “infinite” memory, such as correctly matched parentheses or strings like $a^n b^n$ (where n can be arbitrarily large). They can’t “count” or remember arbitrary nesting
- **Pushdown Automata (PDA)** extend FSAs by adding a crucial component: a **stack**

The stack: A last-in, first-out (LIFO) memory

- A stack is a data structure that supports two primary operations:
  - **Push**: Add an element to the top of the stack
  - **Pop**: Remove the top element from the stack
- This allows the PDA to remember an unbounded (but always finite at any given time) amount of information, but only the top element is directly accessible

Pushdown automata: Adding memory

Core components of a PDA:

- **States** (Q): A finite set of states (like an FSA)
- **Input alphabet** (Σ): A finite set of input symbols
- **Stack alphabet** (Γ): A finite set of symbols that can be stored on the stack on need
- **Transition function** (δ): Now considers (Current State, Input Symbol, Top of Stack Symbol) $\rightarrow$ (New State, Stack Operation). This means a transition can depend on and modify the stack
- **Start state** (q_0): The unique initial state
- **Initial stack symbol** (Z_0): A special symbol at the bottom of the stack
- **Accept (final) states** (F): States indicating acceptance (or acceptance by empty stack)

Pushdown automata: Adding memory

How a PDA operates:

- The PDA reads an input symbol, considers its current state, and the symbol on top of its stack
- Based on these three, it transitions to a new state and performs a stack operation (push, pop, or no change)
- This allows it to “remember” opening symbols (by pushing them) and check for corresponding closing symbols (by popping them)

Pushdown automata: Adding memory

Capabilities and limitations:

- **Can recognise**: All **context-free languages** (e.g., correctly nested parentheses, simple arithmetic expressions, programming language syntax like ‘if-else‘ blocks, $a^n b^n$)
- **Cannot recognise**: Languages requiring more complex memory access than LIFO (e.g., $a^n b^n c^n$, or languages that require comparing arbitrary parts of the string)

The foundation of programming language parsing: Pushdown automata are the theoretical models behind parsers, which are crucial components of compilers. They enable the syntactic analysis of programming languages

Formal languages and grammars

What is a formal language?

- In computer science and logic, a **formal language** is simply a **set of strings** composed of symbols from a finite **alphabet**
- Unlike natural languages (like English or Italian), formal languages have precise, unambiguous rules for forming valid strings
- Examples: The set of all valid Python programs, the set of all binary strings with an even number of ‘1’s, the set of all correctly formed arithmetic expressions

What is a formal grammar?

- A **formal grammar** is a set of rules that **generates** (produces) all the valid strings of a formal language, and only those strings
- It describes the syntactic structure of a language
- A common notation for grammars is **Backus-Naur Form (BNF)** or context-free grammars

Formal languages and grammars

Components of a formal grammar $(G = (V, \Sigma, R, S))$:

- **Non-terminal symbols / variables** (V): Symbols that can be replaced by other symbols according to the rules (e.g., S for Start, E for Expression, N for Number)
- **Terminal symbols / alphabet** (Σ): The actual symbols that form the strings of the language (e.g., $a, b, 0, 1, +, *, (,), \{\}$)
- **Production rules** (R): A finite set of rules (e.g., $A \to \beta$, where $A \in V$ and $\beta \in (V \cup \Sigma)^*$)
- **Start symbol (** S **):** A special non-terminal symbol from which the generation process begins

Formal languages and grammars

Example: A simple grammar for $a^n b^n$:

- Alphabet $\Sigma = \{a, b\}$
- Non-terminal $V = \{S\}$ (S is also the start symbol)
- Rules R:
  - $S \rightarrow aSb$ (Recursive rule)
  - $S \rightarrow \epsilon$ (Base case, ϵ means empty string)
- **Derivation example**:
  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \Rightarrow aabb$
- This grammar generates the language $L = \{\epsilon, ab, aabb, aaabbb, \dots\}$

Formal languages and grammars

Grammars: Generating, Automata: Recognising: Formal grammars define the syntax of languages by generating strings, while automata (like FSAs and PDAs) provide the corresponding computational models that recognise whether a given string belongs to that language. This generative-recognitive duality is fundamental

The Chomsky hierarchy

Noam Chomsky (1928–): Pioneering linguist and logician

- An American linguist, philosopher, cognitive scientist, and political activist
- In 1956, he proposed a classification of formal grammars, which became known as the Chomsky hierarchy
- This hierarchy organises formal languages into classes based on their generative power and the complexity of the automaton required to recognise them

The Chomsky hierarchy

The nested hierarchy: From simple to complex

- The hierarchy defines four major types of grammars/languages, each associated with a specific automaton model and each class being a proper superset of the one below it

Type 3: Regular languages

- **Grammar**: Regular grammars (simple rules like A → aB or A → a)
- **Automaton**: **Finite automata** (FSA)
- **Examples**: Regular expressions (e.g., for email validation, searching text patterns, defining lexical tokens in compilers)

The Chomsky hierarchy

Type 2: Context-free languages (CFL)

- **Grammar**: Context-free grammars (CFG) (e.g., A $\rightarrow$ β, where A is a single non-terminal)
- **Automaton**: **Pushdown automata** (PDA)
- **Examples**: The syntax of most programming languages, correctly nested parentheses or XML/HTML tags, arithmetic expressions

Type 1: Context-Sensitive Languages (CSL)

- **Grammar**: Context-sensitive grammars (CSG) (rules $\alpha A\beta \rightarrow \alpha\gamma\beta$; length of LHS $\leq$ RHS)
- **Automaton**: **Linear bounded automata** (LBA) (a TM with tape bounded by input length)
- **Examples**: Some aspects of natural language processing (e.g., agreement rules), non-context-free programming language features. Less common in typical Computer Science applications than CFLs

The Chomsky hierarchy

4. **Type 0: Recursively enumerable languages** (REL)
   - **Grammar**: Unrestricted Grammars (no restrictions on rules)
   - **Automaton**: **Turing machines** (TM)
   - **Examples** :Any language for which there is an algorithm that will halt and accept if the string is in the language, but may loop forever if not (e.g., the set of all halting Turing machines). This encompasses all computable problems

The fundamental framework for language and computation: The Chomsky Hierarchy provides a fundamental framework for understanding the expressive power of different formalisms, guiding the design of programming languages, compilers, and approaches to natural language processing

Applications of formal languages

Compiler Design:

- The most direct and pervasive application
- **Lexical analysis** (Scanning): Uses **Finite automata** (or regular expressions) to break source code into tokens (keywords, identifiers, operators, etc.)
- **Syntactic analysis** (Parsing): Uses **Pushdown automata** (derived from context-free grammars) to check if the sequence of tokens forms a syntactically valid program structure. This builds the parse tree

Text Processing and Pattern Matching:

- **Regular expressions** (RegEx): A powerful tool for defining and matching text patterns, directly based on **regular languages** and **finite automata**
- Used in search engines, text editors, command-line utilities (grep), scripting languages (Perl, Python), and data validation

Applications of formal languages

Network protocols and state machines:

- The behaviour of many communication protocols (e.g., TCP, Bluetooth) is modelled as **finite state machines**
- This helps in designing, implementing, and verifying the correct sequence of operations in complex systems

Natural language processing (NLP):

- While natural languages are more complex than context-free, early NLP research heavily used **context-free grammars** to model sentence structure and to generate responses
- Automata and formal languages are still fundamental for tasks like part-of-speech tagging and shallow parsing. The limitations (e.g., context-sensitivity of natural language) highlight why more advanced AI techniques are needed for full understanding

Applications of formal languages

Digital circuit design:

- Sequential logic circuits (e.g., controllers, state machines in hardware) are designed and analysed using the principles of **finite state machines**
- This ensures correct behaviour and timing in hardware

Formal verification and model checking:

- Automata are often used to model the behaviour of software and hardware systems and architectures
- Properties (e.g., “does the system ever reach an unsafe state?”) can then be checked algorithmically against these models

The unseen engines of software: Automata and formal languages, though abstract, are the unseen theoretical engines underpinning much of the software we use daily, from the compilers that translate our code to the algorithms that process text and manage network connections

References

Most relevant literature is accessible through the university library, or ask the instructor for copies:

- *Noam Chomsky*, Systems of syntactic analysis, Journal of Symbolic Logic 18(3) (1953)
- *Noam Chomsky*, Three models for the description of language, IRE Transactions on Information Theory, 2 (3) (1956)
- *Noam Chomsky*, Syntactic Structures, Mouton De Gruyter (1957)

Computable Functions: Lecture 7

Syllabus:

Computational Complexity Theory

Computational complexity theory

In this lesson, we will explore:

- **Beyond computability**:
  Introducing the concept of computational complexity
- **Measuring complexity**:
  Time and space complexity, big ‘O’ notation
- **The class P**:
  Problems solvable in polynomial time
- **The class NP**:
  Problems verifiable in polynomial time
- **The P vs. NP problem**:
  The biggest open question in computer science
- **NP-completeness**:
  The hardest problems in **NP**
- **Implications**:
  Why understanding complexity matters for algorithms and society

Quantifying the resources needed to solve computational problems

More than just “solvable”

From what to how:

- In previous lessons, we focused on **computability**: Can a problem be solved by an algorithm at all? (Decidable vs. undecidable)
- Now, we shift our focus to **computational complexity**: If a problem **can** be solved, **how efficiently** can it be solved? What resources (time, memory) does the best possible algorithm require?

Why is efficiency important?

- A problem might be theoretically solvable, but its best algorithm might take billions of years for realistic inputs.
  Such a problem is “unsolvable in practice”
- Efficiency is paramount for practical applications:
  - Fast software
  - Scalable systems
  - Real-time processing
  - Economical use of resources

More than just “solvable”

What does complexity theory measure?

- We analyse the resources required by an algorithm as a function of the **size of the input**
- **Time complexity**: The number of elementary operations (“steps”) an algorithm performs
- **Space complexity**: The amount of memory an algorithm uses
- We are typically interested in the **worst-case scenario** and the **asymptotic behaviour** (how resource usage grows for very large inputs)

From “Can it be done?” to “Can it be done practically?”
Computational complexity theory provides the tools to classify problems based on their inherent difficulty, guiding us towards what is feasible to compute in the real world

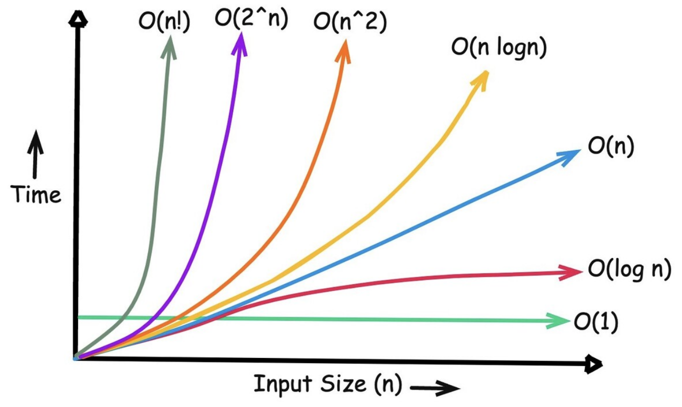
Asymptotic analysis

- Our goal is to express how the time (number of operations) or space (memory used) required by an algorithm grows as the **size of the input (n)** increases
- We want a machine-independent way to compare algorithms

Asymptotic analysis

Focusing on growth rate

- When n becomes very large, constant factors and lower-order terms in an algorithm's resource usage become insignificant compared to the highest-order term
- **Example**: If an algorithm takes $3n^2 + 5n + 100$ operations:
  - For small n, 100 might be significant
  - For $n = 1000$, $3(1000)^2 = 3{,}000{,}000$, while $5(1000) = 5{,}000$. The n^2 term dominates
- Asymptotic analysis focuses on this dominant term, representing the **long-term growth rate**



Common growth rates of algorithms

Asymptotic analysis

Big O notation: The upper bound

- **Definition**: $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- In simpler terms, $O(g(n))$ describes the **upper bound** of an algorithm's growth rate. It tells us that the algorithm will take **at most** proportional to $g(n)$ time/space for large inputs

Asymptotic analysis

Common complexity classes (from best to worst):

- $O(1)$: Constant time (accessing an array element by index)
- $O(\log n)$: Logarithmic time (binary search)
- $O(n)$: Linear time (searching an unsorted list)
- $O(n \log n)$: Log-linear time (efficient sorting algorithms like heap sort)
- $O(n^k)$: Polynomial time ($O(n^2)$ for bubble sort, $O(n^3)$ for matrix multiplication)
- $O(k^n)$: Exponential time ($O(2^n)$ for brute-force solutions to problems like Travelling Salesperson)
- $O(n!)$: Factorial time (brute-force permutations)

The language of algorithm efficiency: Big O notation is the universal language for discussing and comparing the scalability and efficiency of algorithms, focusing on how well they perform as problems grow larger

The class P

Defining "tractability":

- In complexity theory, a problem is generally considered **efficiently solvable** or **tractable** if there exists an algorithm that can solve it in polynomial time
- This means its time complexity is $O(n^k)$ for some constant $k \geq 0$

The class P:

- **P** stands for **polynomial time**
- It is the class of decision problems (problems with a YES/NO answer) that can be solved by a **deterministic Turing Machine** (or a standard, sequential computer) in time that is polynomial in the size of the input
- These are problems for which we have "good" algorithms
- **Historical Note:** The notion of polynomial time as the definition of "tractability" was first explored by **Alan Cobham** (1964) and strongly advocated by **Jack Edmonds** (1965) who laid the groundwork for this fundamental distinction

The class P

Examples of problems in P:

- **Sorting**: Given a list of numbers, can it be sorted in polynomial time? Yes, e.g., Heap sort ($O(n \log n)$)
- **Searching**: Given a list and an element, is the element in the list? Yes, e.g., Linear search ($O(n)$) or Binary search ($O(\log n)$)
- **Graph traversal**: Is there a path between two nodes in a graph? Yes, e.g., Breadth-first search ($O(V + E)$)
- **Arithmetic operations**: Multiplication of two numbers
- **Checking primality**: Determining if a given number is prime (proven in **P** in 2002 by AKS algorithm)

The set of "feasible" problems: Problems in **P** represent the set of computational problems that we consider to be practically solvable for reasonably large inputs on today's computers

The class NP

NP: Non-deterministic polynomial time

- The class **NP** includes decision problems for which a candidate solution can be **verified** in polynomial time by a **deterministic Turing machine**
- **Important distinction**: Being able to **verify** a solution quickly is not the same as being able to **find** a solution quickly

The "Non-deterministic" aspect:

- Conceptually, **NP** problems are those that can be solved in polynomial time by a hypothetical **non-deterministic Turing machine** (NTM)
- An NTM can "guess" the correct path or explore all possible computational paths simultaneously. If **any** path leads to a solution, the NTM finds it
- The polynomial time refers to the length of the shortest accepting path (if one exists)

The class NP

Historical note:

- The concept of **NP** (and **NP**-completeness) was formally introduced by **Stephen Cook** in his seminal 1971 paper “The Complexity of Theorem-Proving Procedures”
- Independently, **Leonid Levin** published similar results in 1973 in the Soviet Union

The class NP

Examples of problems in NP:

- **Boolean satisfiability problem** (SAT): Given a Boolean formula, is there an assignment to its variables that makes the formula true?
  - **Finding** an assignment can be hard. **Verifying** a proposed assignment is easy (just plug in values)
- **Travelling salesperson problem** (TSP): Given a list of cities and distances, is there a route visiting each city exactly once with total distance less than a given value K?
  - **Finding** the shortest route is hard. **Verifying** a proposed route is easy (sum distances, check visits)
- **Sudoku**: Given a partially filled Sudoku grid, does it have a solution?
  - **Finding** a solution is hard. **Verifying** a proposed solution is easy (check rows, columns, blocks)
- **Clique problem**: Given a graph G and an integer k, does G contain a clique of size at least k?

The class NP

All P problems are also in NP: If a problem can be **solved** in polynomial time (i.e., it's in **P**), then a solution can certainly verified be **in polynomial time** (just run the polynomial-time solving algorithm and check its output). Therefore, $\mathbf{P} \subseteq \mathbf{NP}$

The P vs. NP problem

The central unsolved problem of Computer Science:

- We know that $\mathbf{P} \subseteq \mathbf{NP}$ (any problem solvable in polynomial time can also be verified in polynomial time)
- The fundamental question is: **Is P = NP**?
- In simpler terms: If a solution to a problem can be **quickly verified**, can it also be **quickly found**?

Why is this question so important?

- Its answer would revolutionise or solidify vast areas of computing, mathematics, and science
- It's considered one of the seven **Millennium Prize Problems** by the Clay Mathematics Institute, with a $1,000,000 prize for its solution

The P vs. NP problem

Possible outcomes and their implications:

- **If P = NP (Highly unlikely, but possible)**:
  - Every problem whose solution can be quickly checked can be quickly solved
  - This would mean that finding optimal solutions for many currently intractable problems (e.g., drug discovery, perfect scheduling, breaking modern cryptography) would become feasible
  - The world would be fundamentally different

- **If P ≠ NP (Widely believed to be true)**:
  - There exist problems whose solutions are easy to verify but inherently difficult (require exponential time) to find
  - This would confirm the foundational difficulty of many computational tasks
  - It underpins the security of modern cryptography (e.g., public-key encryption relies on the presumed hardness of factoring large numbers, an **NP** problem)
  - Research would continue to focus on approximation algorithms and heuristics for these hard problems

The P vs. NP problem

Why is it unsolved?

- Despite decades of intense research, no one has found a polynomial-time algorithm for a known **NP**-complete problem, nor a rigorous mathematical proof that such an algorithm does not exist
- It requires a breakthrough in mathematical logic or complexity theory

The deepest mystery in computing: The **P** vs. **NP** problem captures the essence of efficient computation and represents a profound boundary in our understanding of what algorithms can truly achieve

NP-completeness

The concept of "hardest" within NP:

- If **P** $\neq$ **NP** (as widely believed), then there are problems in **NP** that are not in **P**. These are the "hard" **NP** problems
- **NP-complete** (NPC) problems are the set of decision problems within **NP** that are, in a formal sense, the **hardest possible problems** in **NP**

NP-completeness

Definition of an NP-complete problem: A decision problem L is NP-complete if:

1. L is in **NP** (meaning a given solution can be verified in polynomial time)
2. L is *complete* (meaning every other problem in **NP** can be **reduced** to L in polynomial time)

Understanding polynomial-time reduction:

- A problem A is polynomially reducible to problem B (denoted $A \leq_p B$) if an algorithm for A can be transformed into an algorithm for B in polynomial time
- Crucially, if $A \leq_p B$, and B can be solved in polynomial time, then A can also be solved in polynomial time
- If every problem in **NP** can be reduced to an **NP**-complete problem, it implies that if you find an efficient algorithm for **one NP**-complete problem, you've found an efficient algorithm for **all** problems in **NP** (and thus $\mathbf{N} = \mathbf{NP}$)

NP-completeness

Historical breakthroughs:

- In 1971, **Stephen Cook** proved that the **Boolean Satisfiability Problem** (SAT) is **NP**-complete. This was the first problem proven to be **NP**-complete (Cook-Levin Theorem)
- In 1972, **Richard Karp** published a list of 21 diverse problems that he showed were also **NP**-complete by demonstrating polynomial-time reductions from SAT or other known **NP**-complete problems. This solidified the field

NP-completeness

Examples of NP-complete problems:

- **Boolean satisfiability** (SAT): (The “first” **NP**-complete problem)
- **Travelling salesperson** (TSP): (Find the shortest route visiting all cities)
- **Knapsack** (Given items with weights and values, maximise value within a weight limit)
- **Clique**: (Finding a fully connected subgraph of a certain size)
- **Vertex cover**; (Finding a minimum set of vertices that touch all edges)
- **Sudoku**: (As discussed, finding a solution to a given puzzle)

The implications of NP-completeness: If a problem is **NP**-complete, it’s strong evidence that a polynomial-time algorithm for it is unlikely to exist. This directs research towards heuristics, approximation algorithms, or special-case solutions

Implications of computational complexity

Guiding problem-solving strategies:

- Complexity theory informs us which problems are inherently difficult. If a problem is **NP**-complete, it's strong evidence that no efficient (polynomial-time) algorithm exists
- This directs our efforts away from searching for a perfect, general polynomial-time solution for large inputs
- Instead, we focus on:
  - **Approximation algorithms**: Finding solutions that are "good enough" (provably close to optimal) in polynomial time
  - **Heuristics**: Fast methods that work well in practice for typical cases, though without optimality guarantees
  - **Special cases**: Identifying subsets of instances of the problem that **can** be solved efficiently
  - **Randomised algorithms**: Using randomness to achieve efficiency on average. These are especially relevant in AI and optimisation, where finding exact solutions is often intractable

Implications of computational complexity

Foundation of modern cryptography:

- The security of nearly all modern encryption systems (e.g., RSA public-key encryption) relies on the presumed hardness of certain computational problems (e.g., factoring large numbers, discrete logarithm problem)
- If $\mathbf{NP} = \mathbf{NP}$, then these cryptographic systems could potentially be broken efficiently, undermining digital security worldwide

Impact on artificial intelligence & optimisation:

- Many core AI problems (e.g., planning, scheduling, resource allocation, finding optimal solutions in large search spaces) are inherently **NP**-hard
- Understanding their complexity helps design effective AI algorithms that use heuristics, search strategies, or machine learning to manage computational demands

Implications of computational complexity

Scientific discovery and modelling:

- In fields like biology (e.g., protein folding, drug design), chemistry, and physics, many problems involve searching vast spaces
- Complexity theory helps identify fundamental computational barriers, guiding research to focus on new computational paradigms (like quantum computing) or simplified models

Implications of computational complexity

The future of computing:

- The **P** vs. **NP** question remains a central, driving force for theoretical computer science. Its resolution, or continued efforts to understand it, push the boundaries of our knowledge about computation
- While quantum computers might solve some currently hard problems efficiently, they are not expected to solve any **NP**-complete problems in polynomial time (i.e., they won't make **P** = **NP** for all problems)
- While quantum computers are powerful, they are not expected to solve all **NP**-complete problems in polynomial time (i.e., they don't imply **P** = **NP**). They can, however, efficiently solve some problems (like integer factorisation and discrete logarithm) that are believed to be hard for classical computers, and are in **NP** but are not known to be **NP**-complete

Implications of computational complexity

Defining the boundaries of the computable and the feasible:
Computational complexity theory provides a vital framework for classifying computational problems, informing practical algorithm design, ensuring digital security, and shaping the very landscape of what is possible in the digital age

References

Most relevant literature is accessible through the university library, or ask the instructor for copies:

- *Alan Cobham*, The intrinsic computational difficulty of functions, in Yehoshua Bar-Hillel, Logic, methodology and philosophy of science, North-Holland (1965)
- *Manindra Agrawal*, *Neeraj Kayal*, *Nitin Saxena*, **PRIMES** is in **P**, Annals of Mathematics, 160(2) (2004)
- *Stephen A. Cook*, The complexity of theorem-proving procedures, Proceedings of the Third Annual ACM Symposium on Theory of Computing (1971)
- *Richard M. Karp*, Reducibility Among Combinatorial Problems, in R.E. Miller; J.W. Thatcher, J.D. Bohlinger (eds.). Complexity of Computer Computations, Plenum (1972)

Computable Functions: Lecture 8

Syllabus:

Modern computing paradigms

Modern computing paradigms

Computing in the 21st century: Mathematical frontiers

- **Beyond the single core**: The rise of parallel and distributed computing for complex problems
- **Intelligent systems**: The profound impact of artificial intelligence and Machine Learning, with a focus on their mathematical underpinnings
- **Computing as a utility**: The role of cloud Computing in scaling mathematical and scientific endeavours
- **New frontiers**: The core concepts of quantum computing and its potential for mathematically hard problems
- **The evolving landscape**: How these paradigms push the boundaries of what is computationally and mathematically feasible

Connecting historical foundations to the contemporary mathematical challenges and opportunities in computation

Parallel and distributed computing

The limits of single-core performance:

- For decades (roughly until the mid-2000s), computer performance primarily increased due to faster clock speeds and architectural improvements in single-core CPUs. This was known as “Moore’s Law” providing “free lunch” performance gains
- However, physical limits (heat, power consumption) made further significant clock speed increases impractical
- To continue increasing computational power, the industry shifted towards parallelism in various forms

Parallel and distributed computing

Parallel computing: More cores, more threads

- Involves using multiple processing units (cores) or computational resources simultaneously to solve a problem
- **A game changer for numerical analysis**:
  - Enabled the efficient solution of complex mathematical problems and large-scale simulations (e.g., fluid dynamics, molecular modelling, weather prediction) that were previously computationally infeasible
  - Led to the development of new parallel algorithms specifically designed to exploit multi-core and many-core architectures
- **Examples**:
  - **Multi-core CPUs**: Modern processors have 2, 4, 8, or more cores executing instructions concurrently
  - **GPUs** (Graphics Processing Units): Highly parallel architectures with thousands of smaller cores, originally for graphics, now widely used for general-purpose computation (GPGPU) in AI, scientific simulations
- Requires algorithms to be designed for parallel execution.

Parallel and distributed computing

Distributed computing: Computation across networks

- Involves multiple independent computers (nodes) working together over a network to achieve a common goal
- Communication happens via message passing
- **Goals**:
  - Solving problems too large for a single machine (e.g., climate modelling, large-scale data analysis)
  - Achieving fault tolerance and high availability
- **Examples**:
  - **Supercomputers**: Clusters of thousands of interconnected processors
  - **Grid computing**: Leveraging idle computing power across a wide network
  - **Peer-to-peer networks**: (e.g., file sharing, some cryptocurrencies)

Redefining "feasible": Parallel and distributed computing overcome the physical limitations of single processors, dramatically expanding the scale and complexity of problems that are computationally feasible, pushing the practical boundaries of what can be computed

Cloud computing: Computing as a utility

The shift to on-demand services:

- Cloud computing offers on-demand access to scalable computational resources over the internet, moving away from the need for individual ownership of vast hardware
- Its primary relevance for mathematical and scientific fields lies in democratising access to immense processing power and storage

Cloud computing: Computing as a utility

Key benefits for mathematicians and researchers:

- **Unprecedented scale**: Provides virtual computing devices and high-performance computing (HPC) clusters that are essential for:
    - Running complex simulations (e.g., climate models, financial risk analysis)
    - Processing vast datasets (e.g., in statistics, data science)
    - Solving large-scale optimisation problems
- **Accessibility**: Resources are available globally, enabling collaborative research and reducing barriers for institutions without massive local infrastructure
- **Elasticity**: Resources can be rapidly scaled up or down as needed for specific projects, avoiding large upfront investments

Artificial intelligence & machine learning

Defining AI and ML:

- **Artificial intelligence** (AI): A broad field aiming to create machines that simulate human intelligence (e.g., reasoning, learning, problem-solving, perception). Pioneering ideas, like the "Turing Test" proposed by Alan Turing, laid the conceptual groundwork for machine intelligence
- **Machine learning** (ML): A subset of AI where systems learn from data, identifying patterns and making decisions with minimal explicit programming. Deep learning is a further subset using multi-layered neural networks to attain a multi-level form of learning

Artificial intelligence & machine learning

Mathematical impact and new research areas:

- AI's rapid advancements are driving new research frontiers within mathematics itself:
  - **Optimisation theory**: Development of novel algorithms for non-convex optimisation in high-dimensional spaces
  - **Information theory**: Analysing data compression, encoding, and the fundamental limits of learning
  - **Numerical analysis**: Designing efficient and stable numerical methods for large-scale computations inherent in neural networks
  - **Differential geometry & topology**: Emerging applications in understanding neural network architectures and data manifolds
  - **Statistical learning theory**: Providing theoretical guarantees and understanding the generalisation capabilities of complex models
- AI provides powerful new tools for mathematicians to explore and solve problems, leading to a symbiotic relationship

Artificial intelligence & machine learning

AI: A catalyst for mathematical innovation: Artificial intelligence is not just a consumer of computational power; it's a powerful catalyst, driving fundamental advancements and opening entirely new avenues of research across diverse branches of mathematics

Quantum computing

Beyond classical limits:

- Quantum computing represents a fundamentally different approach to computation, leveraging the principles of quantum mechanics (like **superposition** and **entanglement**)
- Unlike classical bits (0 or 1), **qubits** can exist in multiple states simultaneously and be interconnected in complex ways, allowing for vastly different computational strategies

Quantum computing

Potential for mathematically hard problems:

- This new paradigm offers the potential to efficiently solve certain problems that are currently intractable for even the most powerful classical supercomputers
- Key examples of potential "quantum advantage" include:
  - Breaking certain forms of modern **cryptography** through algorithms like **Shor's** (e.g., integer factorisation)
  - Highly accurate **simulations of molecules and materials**, which are complex quantum systems themselves
  - Solving certain classes of **optimisation problems** more efficiently.

Quantum computing

Current status and outlook:

- Quantum computing is still in its early stages of development
- It requires specialised quantum algorithms and is unlikely to replace classical computers for most tasks. Instead, it's expected to serve as a powerful accelerator for specific, extremely complex computational challenges rooted in mathematics and physics.

Redefining the bounds of "feasible computation": Quantum computing offers a glimpse into a future where the very nature of what is computationally feasible is profoundly expanded, driven by mathematical insights into the quantum world

The enduring journey of computation

Our historical trajectory:

- We began with humanity's earliest impulses to quantify and calculate, tracing the evolution from simple tools like the **abacus** to the mechanical wonders of **Pascal** and **Leibniz** (cited in the main course)
- The early 20th century then provided the essential mathematical bedrock: **Hilbert's problems**, **Gödel's incompleteness**, and the transformative work of **Turing** and **Church** defining the very essence of computability
- This theoretical framework directly led to the first electronic computers, from the groundbreaking **ENIAC** and its successors to the influential **von Neumann architecture**

The enduring journey of computation

Understanding computational boundaries:

- We delved into the profound implications of **undecidability**, recognising the inherent limits of what algorithms can achieve
- The **Chomsky hierarchy** showed us how different levels of computational power are needed for various language complexities
- **Computational complexity theory** distinguished between what is merely computable and what is **practically feasible**, bringing us to the pivotal **P vs. NP** problem

The enduring journey of computation

Modern paradigms: Scaling and intelligence:

- The need for greater power led to **parallel** and **distributed computing**, fundamentally changing how large mathematical problems are solved
- **Cloud computing** emerged as a utility, democratising access to vast computational resources for scientific and mathematical research
- We explored **artificial intelligence** and **machine learning**, recognising them not just as consumers of computing power, but as powerful drivers of new mathematical theory and algorithms, from optimisation to statistics, with potential influence on every branch of Mathematics
- Finally, we touched upon **quantum computing**, a new frontier poised to tackle specific, intractable mathematical challenges by harnessing the rules of the quantum world

The enduring journey of computation

A profound interplay: Mathematics and computation This monographic part has illuminated how the concept of “computable functions” has evolved from an abstract mathematical inquiry into the very fabric of our technological world. This journey reveals an unbreakable bond: mathematics provides the fundamental language and rigorous framework, while computation offers the means to explore, apply, and extend mathematical frontiers. The history of computing is, in essence, a dynamic chapter in the history of mathematics, continually expanding the realm of the possible

References

Most relevant literature is accessible through the university library, or ask the instructor for copies:

- *Alan M. Turing*, Computing Machinery and Intelligence, Mind 59(236) (1950)
- *Peter W. SHor*, Algorithms for quantum computation: discrete logarithms and factoring, Proceedings 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, (1994)
- *Peter W. SHor*, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, SIAM Journal on Computing 26(5) (1999)

Palazzo Davanzati, Firenze