Fundamental Algorithmic Techniques I

January 27, 2026

Outline

Finite functions & Computing

Finite functions:

$$\mathcal{F} : \{0,1\}^n \longrightarrow \{0,1\}^m$$

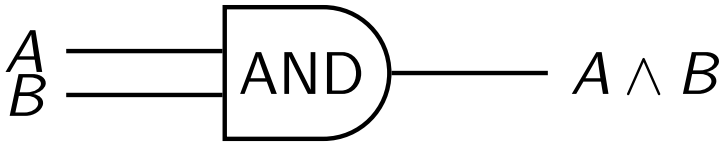
Computational Space: $\{0,1\}^n \to \{0,1\}$ with 2^{2^n} possibilities!

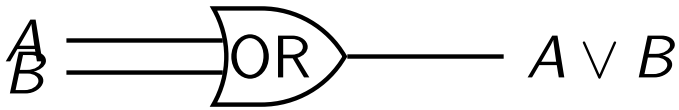
Examples: Hashing, encryption, boolean circuits

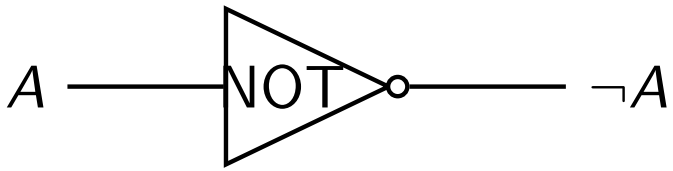
Computation:

- Circuit: $\mathcal{C}$ *computes* $\mathcal{F}$ if $\forall x \in \{0,1\}^n, \mathcal{C}(x) = \mathcal{F}(x)$
- Program: $\mathcal{P}$ *computes* $\mathcal{F}$ if $\forall x \in \{0,1\}^n, \mathcal{P}(x) = \mathcal{F}(x)$

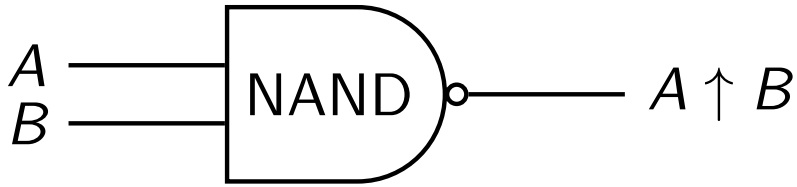
Basic Circuits: AND, OR, NOT







Basic Circuits: NAND

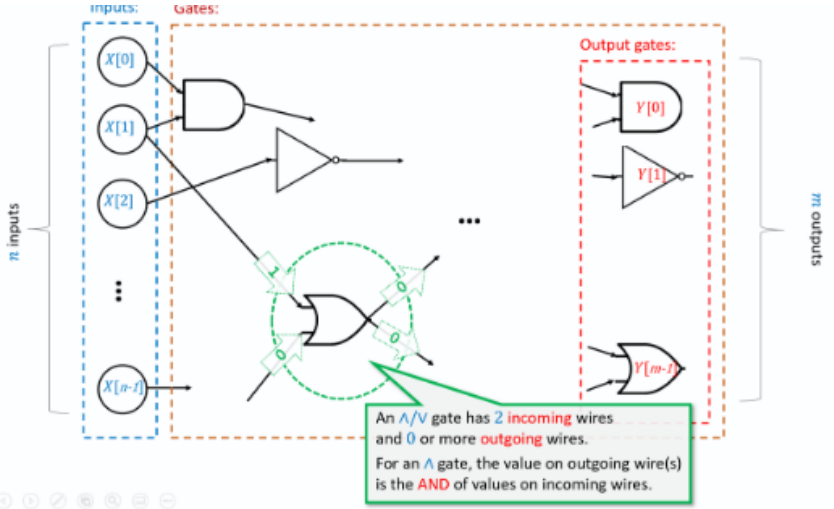


A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

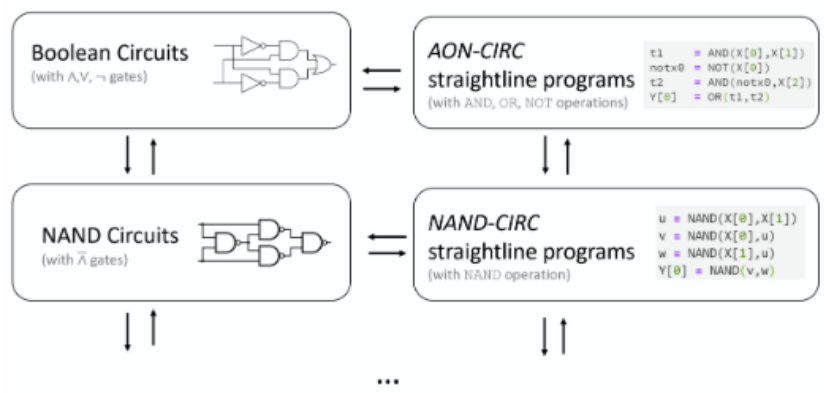
Combinations of NAND gates generate OR/AND/NOT

functionally complete Operator

Equivalence: Circuits $\Leftrightarrow$ Straight-Line Programs



A Boolean circuit is a labeled acyclic graph (DAG)



Boolean functions have straight-line program equivalents [a]

[a] AON is And, Or, Not CIRC for circuit...

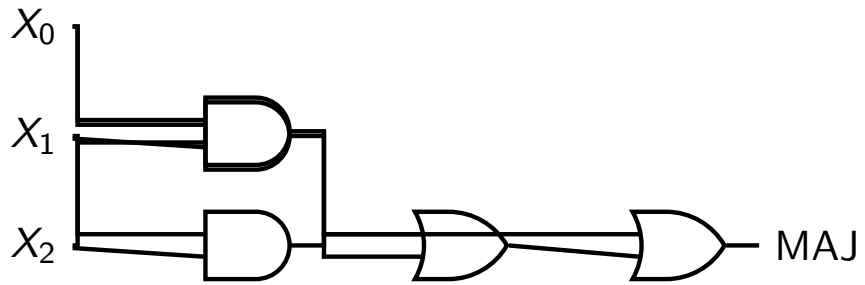
Equivalence $\Leftrightarrow$: simply via topological sorting

MAJ and XOR: Code vs Circuits

MAJ Implementation:

```
def MAJ(X[0],X[1],X[2]):
    firstpair = AND(X[0],X[1])
    secondpair = AND(X[1],X[2])
    thirdpair = AND(X[0],X[2])
    temp = OR(secondpair,
        thirdpair)
    return OR(firstpair,temp)
```

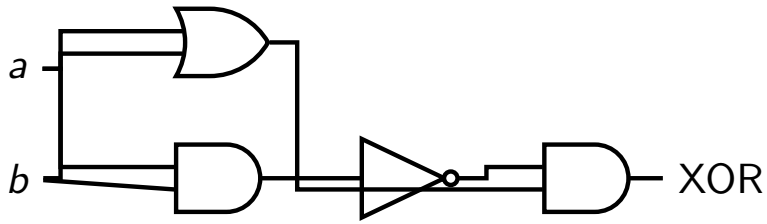
MAJ Circuit



XOR Implementation:

```
def XOR(a,b):
    w1 = AND(a,b)
    w2 = NOT(w1)
    w3 = OR(a,b)
    return AND(w2,w3)
```

XOR Circuit



Computation of Finite Functions

Theorem

Every function $f : \{0,1\}^n \to \{0,1\}^m$ *can be computed by a Boolean circuit of size at most*

$$\mathcal{O}\left(\frac{m \cdot 2^n}{n}\right)$$

using AND, OR, and NOT gates.

Corollary

Since AON computable by NAND, the same function can be computed by a NAND-only circuit of comparable size.

Corollary

Any such function can be represented by a single-line program of length $\mathcal{O}(m \cdot 2^n)$ *using truth-table enumeration (e.g., via conditional expressions or lookup tables).*

Equivalence: Circuits $\leftrightarrows$ Code $\leftrightarrows$ Binary Data

Circuit Encoding Theorem

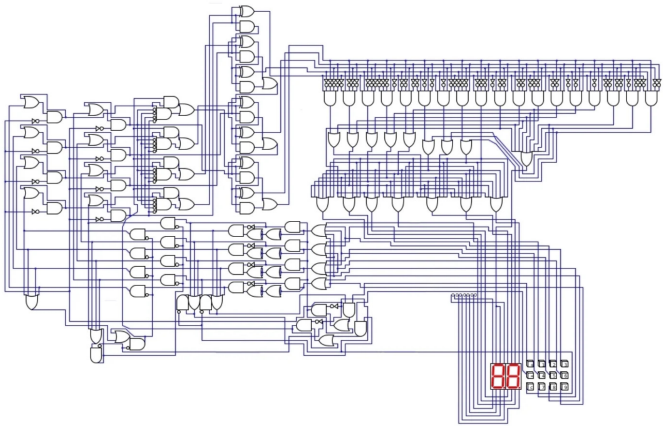
Any Boolean circuit with n gates can be represented using $\mathcal{O}(n \log n)$ bits.

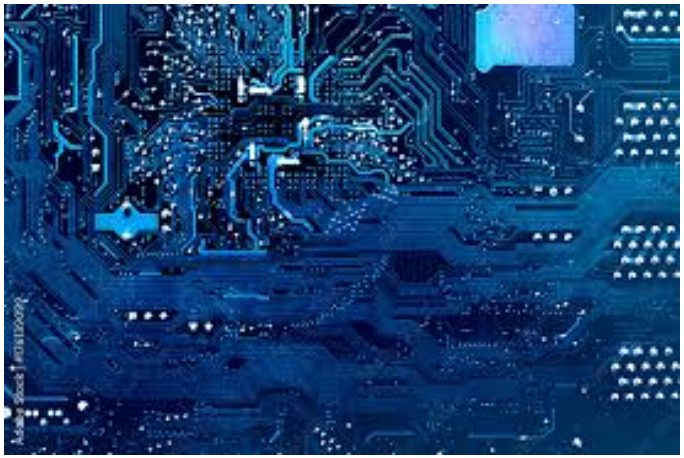
Why? Each gate requires:

- $\mathcal{O}(\log n)$ bits to specify its **type** (AND/OR/NOT)
- $\mathcal{O}(\log n)$ bits to specify its **input wires** (from $\leq n$ wires)

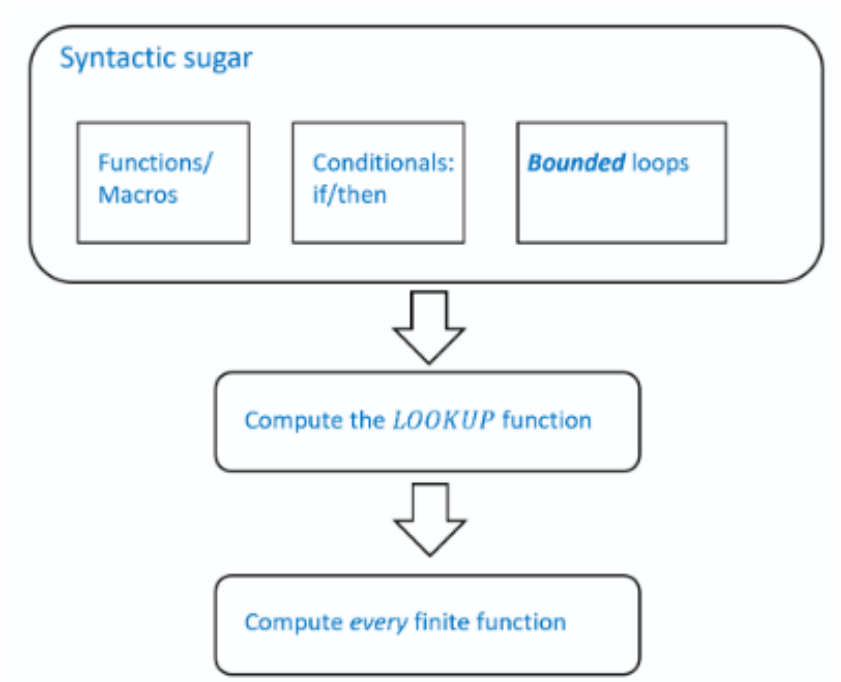
Total: $n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$ bits

Larger Code $\Leftrightarrow$ Larger Circuits!





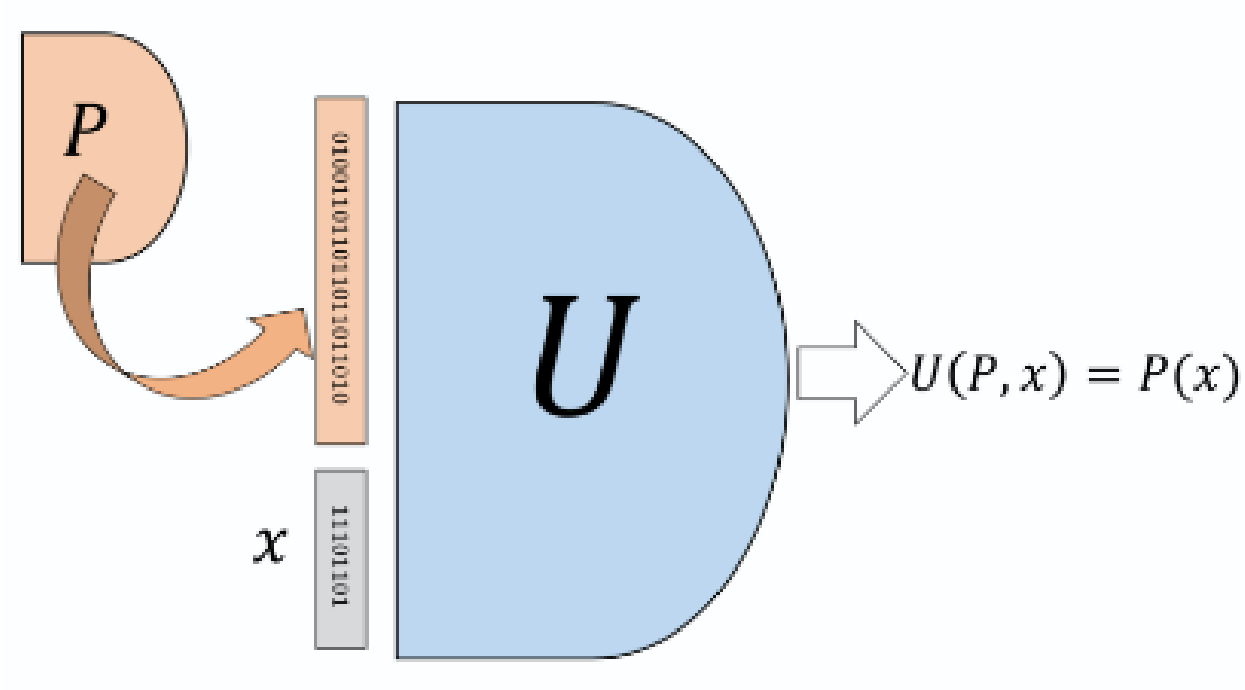
Circuits become larger and powerful via composition



Syntaxic Sugar & Composition to compute any function!

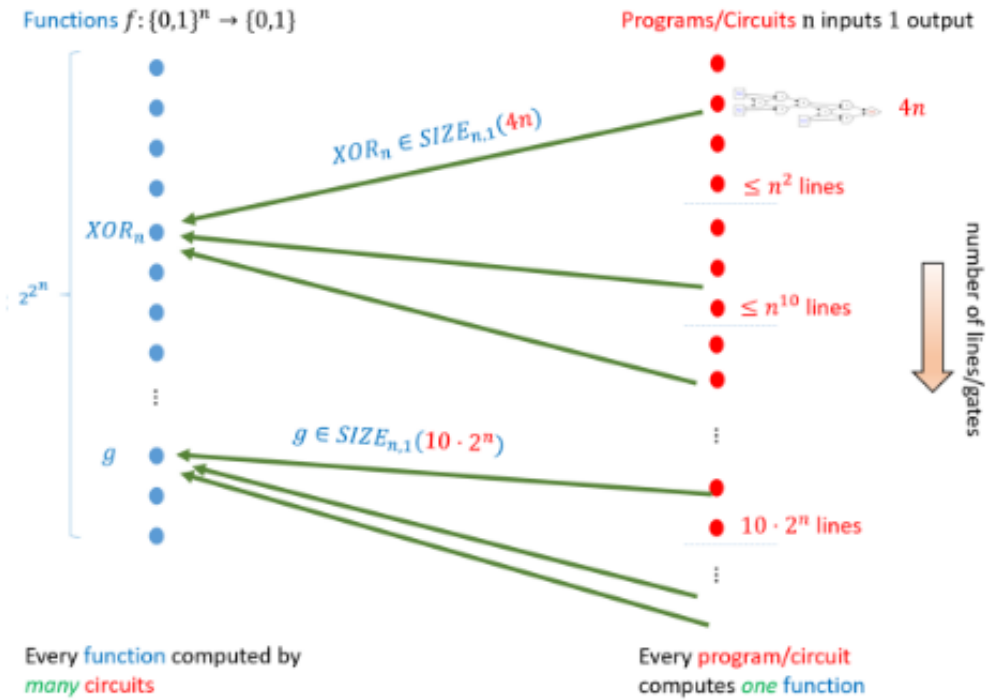
Universal Circuits & Programs

Universal circuits/Programs are able to evaluage other circuits (Compiler, Interpreters, Browser, JIT, Emulators, Universal Turing Machines, ...).



Landscape

Finite functions have a whole range of circuits/code lengths, with many rapidly requiring exponentially long circuits as complexity grows:



So we need more powerful models for real-world computing!

Introducing Infinite Functions

Infinite Functions:

$$\mathcal{F} : \{0,1\}^* \longrightarrow \{0,1\}^*,$$

with $\{0,1\}^*$ standing for arbitrary length bit-strings.

Examples: real-world computations

- Compilers (arbitrary program size)
- Network protocols (variable packet lengths)
- Compression algorithms (any input size)
- AI models (token sequences of varying length)

New notions to explore:

- Finite automaton and their limitations
- Turing machines alternative and its advantages
- Computability and Turing equivalence

From Finite to Infinite Functions

Finite functions (circuits):

- Handle fixed-size inputs: $\{0,1\}^n \to \{0,1\}^m$
- Can compute any function, but may require exponentially many gates
- **Limitation**: Cannot handle unbounded input sizes

Infinite functions (unbounded computation):

- Handle variable-length inputs: $\{0,1\}^* \to \{0,1\}^*$
- Real-world applications need this capability
- **Challenge**: Need computational models with unbounded memory

Path forward:

- Start with **Finite Automata** (limited memory)
- Progress to **Turing Machines** (unbounded memory)
- Understand **computability** and what can/cannot be computed

Deterministic Finite Automaton (DFA)

DFA Robot

- Reads input symbols (e.g., 0s and 1s) **one at a time**
- Decides to **accept** or **reject** the whole string
- Has **no memory** beyond its current state

Key components:

- **States**: Finite set (e.g., "waiting", "success", "error")
- **Start state**: Where the robot begins
- **Accept states**: Success states (double circle in diagrams)
- **Rules**: "If in state A and read symbol x, go to state B"

Efficient for Regular Expressions!
BUT Infinitely many functions non computable by DFA in $\{0,1\}^* \longrightarrow \{0,1\}$

Turing Machines: Why and How?

Finite circuits (combinational logic):

Can compute any **fixed-size** function $f : \{0,1\}^n \to \{0,1\}^m$.
But **exponentially many gates** for general functions ($\sim 2^n/n$).
Cannot handle unbounded inputs (e.g. arbitrary lengths)

Deterministic Finite Automata (DFAs):

Handle **unbounded input streams** with constant memory but are **too weak** for basic tasks.

Turing Machines add two critical capabilities:

- **Dynamic computation**: Modify state based on tape contents
- **Unbounded memory**: Read/write tape of infinite length
- **Turing Church Thesis**: Can compute any function

Computability & Turing Equivalence

Computable Functions:
Let $F : \{0,1\}^* \to \{0,1\}$. A Turing machine M *computes* F if:

$$\forall x \in \{0,1\}^*, \quad M(x) \text{ halts with output } F(x)$$

Turing Completeness

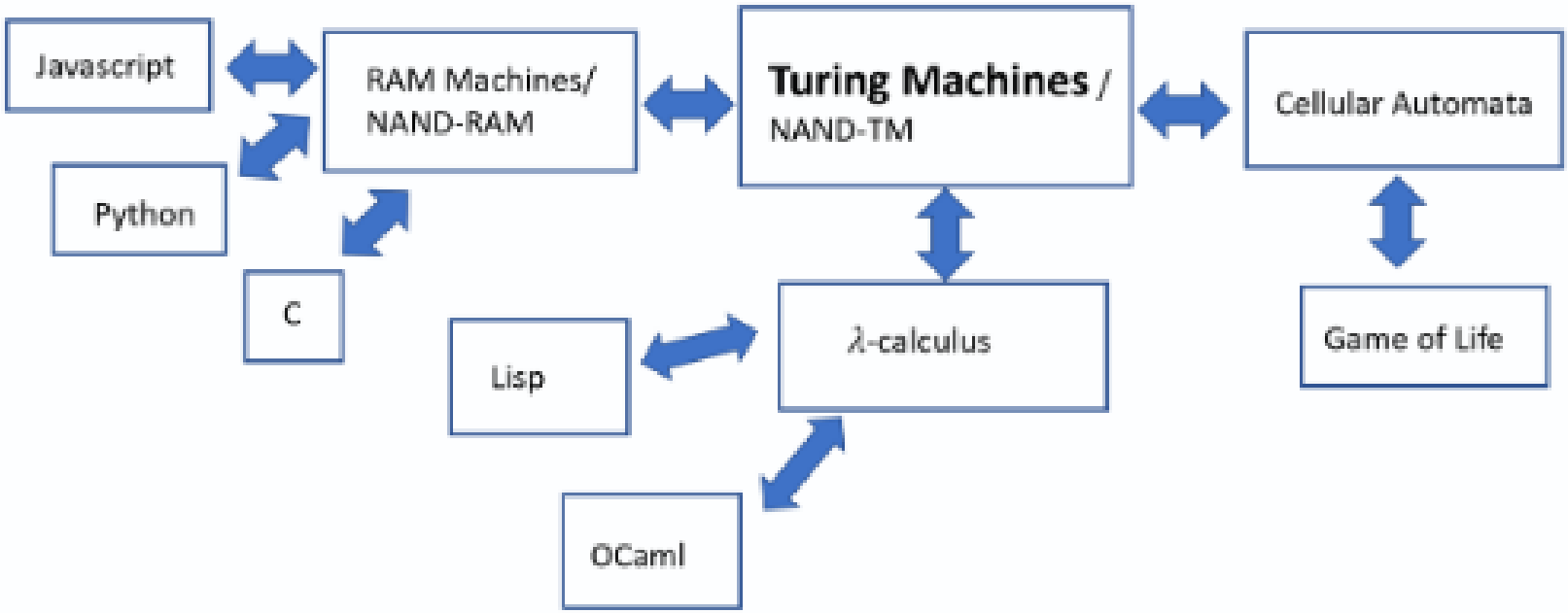
A computational model is **Turing complete** if it can compute *exactly the same functions* as (or simulate) a Turing machine.

Equivalent Models: (equivalent means Turing $\leftrightarrows$ Model simulable)

- **NAND-TM**: NAND-based Turing-complete language
- **RAM machines**: Standard computer architecture (registers, memory)
- **λ-calculus**: Function-based computation (Church's model)
- **Cellular automata**: e.g., Conway's Game of Life (2D grid rules)

Turing equivalent Models

Church-Turing Conjecture: *Any function on natural numbers can be calculated by an effective method* $\Leftrightarrow$ *can be computed by a Turing machine.*

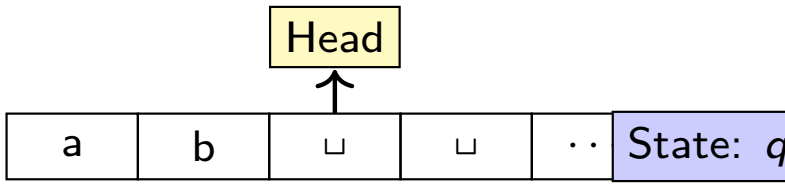


Turing Machine: Simple Definition

A **Turing Machine** is a theoretical computer with:

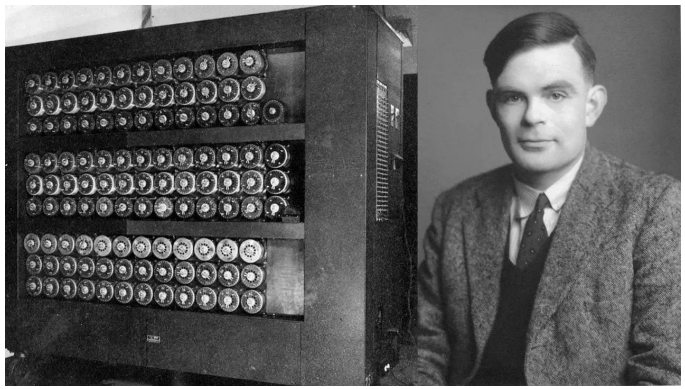
3 Key Components:

1. **Infinite tape**
   (like a never-ending strip of paper)
2. **Read/write head**
   (can read, write, and move left/right)
3. **State register**
   (remembers current "mode" of operation)



How it works:

- Starts in **initial state**
- At each step:
  1. Read symbol under head
  2. Write new symbol (or keep same)
  3. Move head **L** or **R**
  4. Switch to new state
- Halts when reaching **accept** or **reject** state



Key idea: Can compute *anything computable*!

Summary

Models of Computation

- **Circuits** $\rightarrow$ finite, fixed-size programs (exponential lengths...)
- **Automata** $\rightarrow$ handle infinite inputs/outputs, but limited to regular/simple problems
- **Turing Machine**
  - One per problem
  - Infinite, writable tape
  - Finite set of inner states
  - Transition function/table
- **Universal Turing Machine** Programmable computer!

Turing-Complete Systems

- NAND-TM language
- RAM model (e.g., Python, C)
- Lambda calculus (e.g., Lisp, OCaml, Clojure)
- Cellular automata (e.g., Conway's Game of Life)

Church–Turing Thesis:

A function on natural numbers is effectively computable

$\Leftrightarrow$

It is computable by a Turing machine.

Incomputability by Turing Machine

One can prove that **infinitely many** functions:

$$\mathcal{F} : \{0,1\}^* \to \{0,1\}$$

are **uncomputable** by a Turing Machine.

Well-known examples:

- **Halting Problem**:

  $$\mathrm{Halt}(M,x) = \begin{cases} 1 & \text{if Turing machine } M \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

  This function is uncomputable.
- **Busy Beaver**:

  For an n-state Turing machine M_n, find the maximum number of steps M_n can run before halting (over all inputs and all such machines). Grows faster than any computable function!

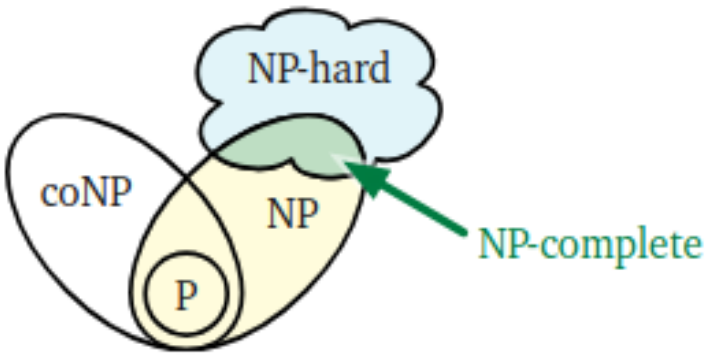
In Summary: Two kind of uncomputable functions: by logical construction or growing too fast!

Computational Complexity Classes — The Big Picture

Decision problems (answer: Yes/No)

Class	Meaning	Example
P	Solvable in poly-time	Sorting, shortest path
NP	Verifiable in poly-time	SAT, TSP (decision)
coNP	"No" answers verifiable in poly-time	Formula validity
NP-complete	In NP + NP-hard	3-SAT, Clique
NP-hard	At least as hard as any NP problem	Halting Problem, opt. TSP
Uncomputable	Uncomputable as seen above	Halting Problem, Busy Beaver, ...

NP-hard and NP-complete



Relationships among P, NP, NP-complete, and NP-hard classes. (Assuming $P \neq NP$, though unproven.)

- **NP-hard**: A problem Π is NP-hard if every problem in NP polynomial-time reduces to Π.
- **NP-complete**: A problem that is both NP-hard and in NP.

Reduction: $A \leq_p B$ means an efficient algorithm for B yields an efficient algorithm for A.
$\Rightarrow$ B is *at least as hard as* A (not necessarily equivalent).