Theorem [theorem]Corollary

Fundamental Algorithmic Techniques X

November 28, 2025

Outline

Finite functions & Computing

Finite functions:

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

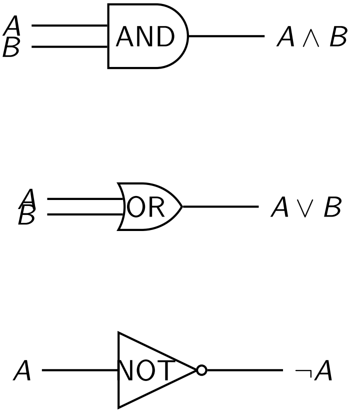
Computational Space: $\{0,1\}^n \to \{0,1\}$ with 2^{2^n} possibilities!

Examples: Hashing, encryption, boolean circuits

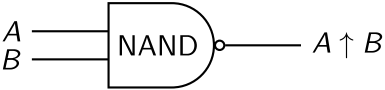
Computation:

- 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

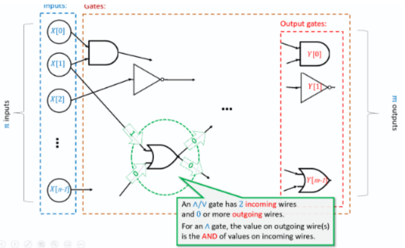


A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

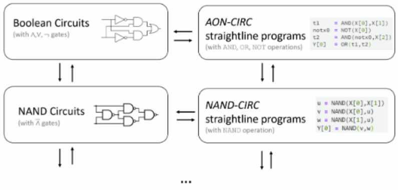
Combinations of NAND gates generate OR/AND/NOT

functionally complete Operator

Equivalence: Circuits ⇔ 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...

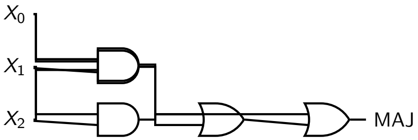
Equivalence ⇔: simply via topological sorting

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)
```

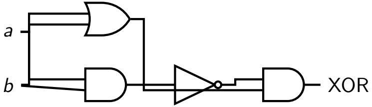
MAJ Circuit



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).

Data $\to$ Code: Circuit Representation

Circuit Encoding Theorem

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

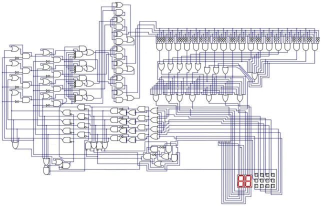
Why? Each gate requires:

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

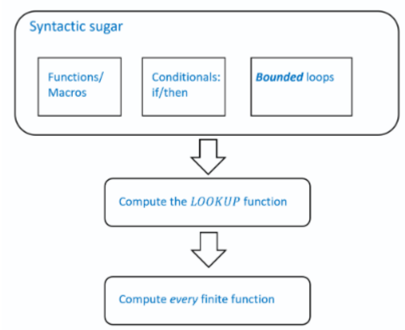
$\mathcal{O}(\log n)$ bits to specify its **input wires** (from $\leq n$ wires)

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

Larger Code $\Leftrightarrow$ Larger Circuits!



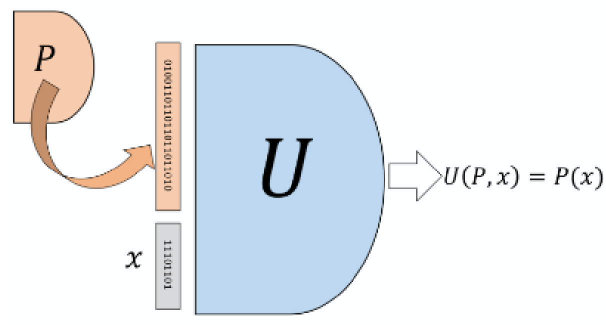
Circuits become larger via composition



Syntaxic Sugar & Composition to compute any function!

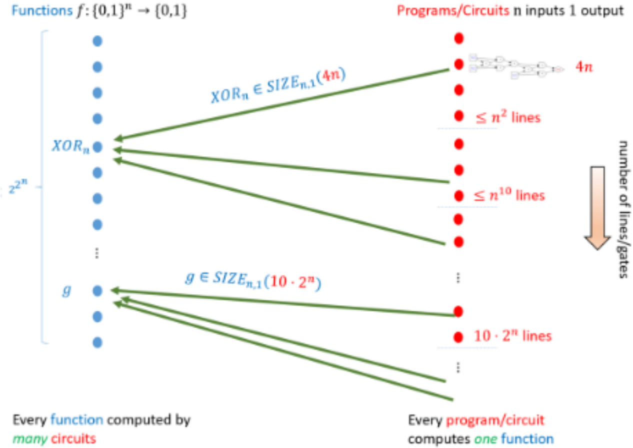
Universal Circuits & Programs

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



Landscape

A whole range of functions, some requiring exponentially long circuits:



Computing: ∞ functions

Infinite Functions:

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

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)

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 $\{0,1\}^* \longrightarrow \{0,1\}$

Why Turing Machines?

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

Computability & Turing Equivalence

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

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

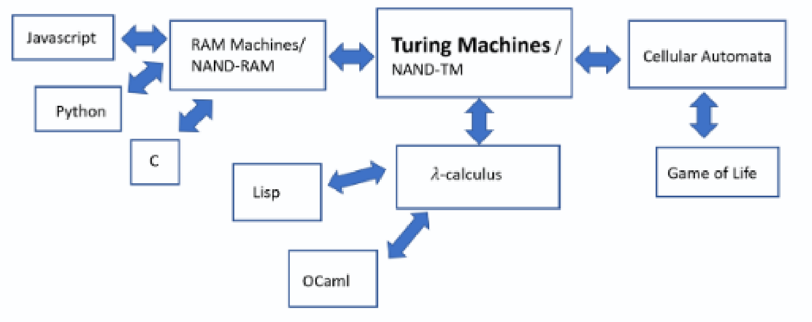
Turing Completeness
A computational model is **Turing complete** if it can compute *exactly the same functions* as (or simulate) a Turing machine.

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 that can be computed by an algorithm 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:

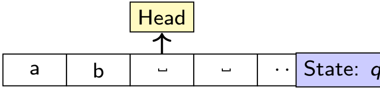
Starts in **initial state**

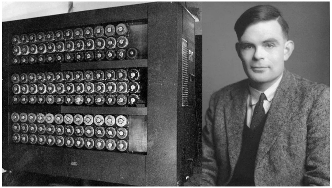
At each step:

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*!





Turing Machine: Formal Definition

A **Turing Machine** is a 7-tuple

$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where:

Q: Finite set of **states**

Σ: **Input alphabet** (does not contain blank symbol $\sqcup$)

Γ: **Tape alphabet** ($\Sigma \subseteq \Gamma$, $\sqcup \in \Gamma \setminus \Sigma$)

$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$: **Transition function**

$q_0 \in Q$: **Start state**

$q_{\text{accept}} \in Q$: **Accept state**

$q_{\text{reject}} \in Q$: **Reject state** ($q_{\text{accept}} \neq q_{\text{reject}}$)

Key properties:

Tape is infinite in one direction (to the right)

Head moves **Left** (L) or **Right** (R) at each step

Computation halts when entering q_{accept} or q_{reject}