An Introduction to the Classical Theory of Computation 3

Dr. Hugo García Tecocoatzi

Instituto Tecnológico y de Estudios Superiores de Monterrey

August 21 2025

Three fundamental questions:

1. **What is a computational problem?**
   - Examples: multiplying numbers, AI tasks.
   - Focus: *decision problems* $\rightarrow$ elegant general theory.
2. **How do we design algorithms?**
   - Given a problem, what algorithms solve it?
   - Are there general methods for broad classes?
   - How to verify correctness?
3. **What resources are needed?**
   - Algorithms consume *time, space, energy*.
   - Classify problems by minimal resource requirements.

Quantifying Computational Resources

Why do we need resource quantification?

- Different computational models (e.g., 1-tape vs 2-tape TM) may require different resources.
- We need a *model-independent* way of comparing algorithms.
- Focus: **asymptotic behavior** of algorithms, not exact step counts.

Example: Adding two n-bit numbers:

$$\text{Exact gates: } n + \log n + 16 \quad \Rightarrow \quad \text{Asymptotic: } O(n) \Rightarrow n$$

Exact vs Asymptotic Resource Analysis

Exact gate count:

$$f(n) = n + 2\log n + 16$$

- Includes constants
- Includes smaller terms
- Precise for each n

Asymptotic behavior:

$$f(n) = O(n) = n$$

- Focuses on growth rate
- Ignores constants
- Dominant term: n

For large n, only the dominant term matters.

Exact vs Asymptotic Resource Analysis

Exact gate count:

$$f(n) = n + 2\log n + 16$$

Asymptotic behavior:

$$f(n) = O(n) = n$$

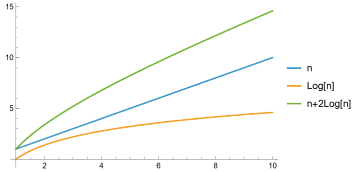


Figura: $N = 10$

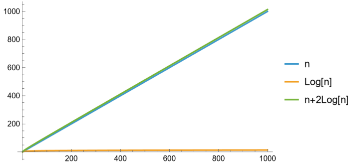


Figura: $N = 1000$

For large n, only the dominant term matters.

Big-O Notation

Definition: $f(n) \in O(g(n))$ if there exist constants c, n_0 such that:

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$$

Interpretation:

- $g(n)$ is an *upper bound* on $f(n)$ (for large n).
- Captures the **worst-case growth rate**.
- Example: $24n + 2\log n + 16 = O(n)$.

Definition: $f(n) \in \Omega(g(n))$ if there exist constants c, n_0 such that:

$$\forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$$

Interpretation:

- $g(n)$ is a *lower bound* on $f(n)$ (for large n).
- Captures the **best-case growth rate**.
- Example: $24n + 2\log n + 16 = \Omega(n)$.

Big-Theta Notation

Definition: $f(n) \in \Theta(g(n))$ if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

Interpretation:

- $g(n)$ is a **tight bound** on $f(n)$.
- Captures the **exact asymptotic growth**.
- Example: $24n + 2\log n + 16 = \Theta(n)$.

$$O \;=\text{upper bound},\quad \Omega \;=\text{lower bound},\quad \Theta \;=\text{tight bound}$$

Asymptotic Notation: Examples

- **Big-O (Upper Bound):**

$$2n \in O(n^2) \quad \text{since } 2n \leq 2n^2 \quad \forall n > 0$$

- **Big-Ω (Lower Bound):**

$$2^n \in \Omega(n^3) \quad \text{since } n^3 \leq 2^n \quad \text{for large } n$$

- **Big-Θ (Tight Bound):**

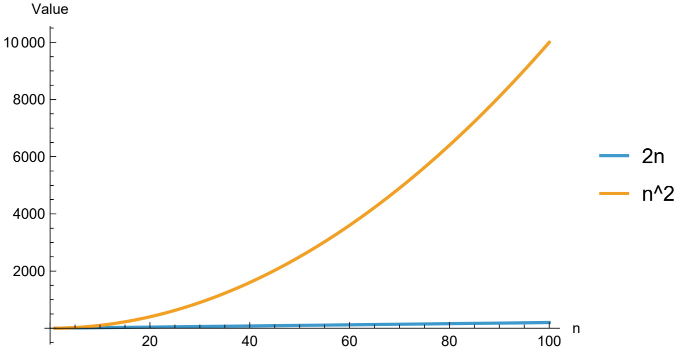
$$7n^2 + \sqrt{n}\log n \in \Theta(n^2)$$

$$7n^2 \leq 7n^2 + \sqrt{n}\log n \leq 8n^2 \quad \text{for large } n$$

Asymptotic notation captures the **growth rate**, *not the exact details.*

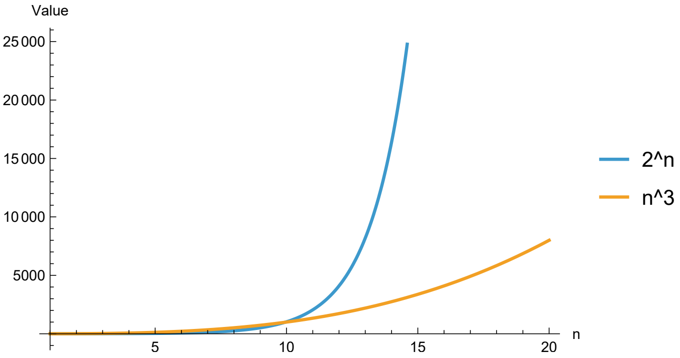
Asymptotic Notation: Big-O

$$2n \in O(n^2) \quad \text{since } 2n \leq n^2 \quad \text{for large } n$$



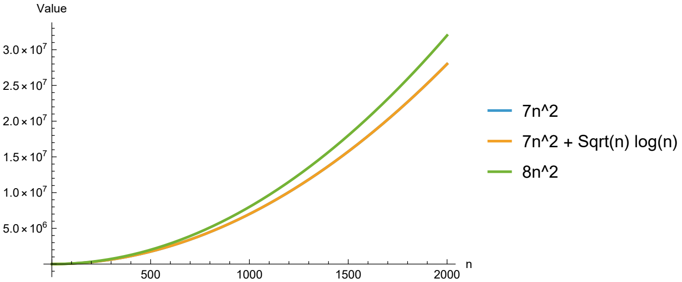
Asymptotic Notation: Big-Ω

$$2^n \in \Omega(n^3) \quad \text{since } n^3 \leq 2^n \quad \text{for large } n$$



Asymptotic Notation: Big-Θ

$$7n^2 + \sqrt{n} \log n \in \Theta(n^2)$$



Computational Complexity

- **Computational complexity** studies the **time and space resources** required to solve computational problems.
- Goal: Prove **lower bounds** on resources required by the best possible algorithm.
- Complementary to algorithm design:
  - Algorithm design: creates efficient algorithms.
  - Complexity theory: proves how efficient an algorithm *can* be.

Challenges in Defining Complexity

- Different **computational models** may require different resources:
  - Example: multi-tape Turing machines are faster than single-tape Turing machines.
- To compare models, we use **input size** n (in bits).
- Example: deciding whether an n-bit number is prime.

Measuring Computational Resources

- **Time complexity:** Number of steps as a function of input size.
- **Space complexity:** Amount of memory required.
- **Other resources:** Randomness, parallelism, energy.
- Big-O notation formalizes asymptotic growth:

$$O(f(n)) = \{\, g(n) \mid g(n) \leq cf(n) \text{ for large } n \,\}.$$

Polynomial vs. Exponential Resources

- **Polynomial time/space:** resources grow as n^k for some k.
  - Considered **efficient** (tractable, feasible).
- **Exponential time/space:** resources grow faster than any polynomial.
  - Considered **inefficient** (intractable, infeasible).
- Sometimes "exponential" includes functions like $n^{\log n}$, which grow faster than polynomials but slower than 2^n.

Decision Problems

- A **decision problem** has a **yes/no answer**.
- Example: *Is a given number m prime?*
- Importance:
  - Simpler, elegant theory.
  - Forms the foundation of complexity classes.

Decision Problems as Languages

- Formalism: decision problems $\leftrightarrow$ languages.
- A **language** L over alphabet Σ is a subset of Σ^*.
- Example: $\Sigma = \{0, 1\}$, then

$$L = \{\text{binary strings representing prime numbers}\}.$$

- A Turing machine decides L by halting in:
  - q_Y ("yes") if $x \in L$
  - q_N ("no") if $x \notin L$

Defining Complexity Classes

- For input of length n, let $\mathrm{TIME}(f(n)) =$ all problems decidable in time $O(f(n))$.
- Example: primality testing.
  - Goal: determine if n is prime in as few steps as possible.
  - If solvable in polynomial time: primality $\in$ P.
- Captures the **resources required** by the best possible algorithm.

The Complexity Class P

- $P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$
- In words: all problems solvable in **polynomial time**.
- Intuitively:
  - **Efficient, tractable, feasible** problems.
  - Algorithms scale reasonably with input size.
- Examples:
  - Sorting, shortest paths, matrix multiplication, primality testing.

Complexity Classes: P, NP,

- **P**: Problems solvable in polynomial time by a deterministic Turing machine.
  - Example: sorting ($O(n \log n)$).
- **NP**: Problems whose solutions can be *verified* in polynomial time (but cannot be found).
  - Factoring is an example of a problem in an important complexity class known as NP.

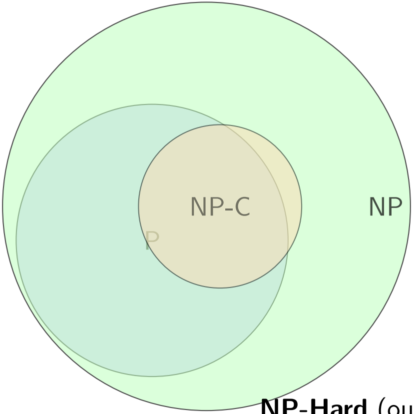
NP-Complete Problems

- A problem is **NP-Complete** if:
  1. It is in NP.
  2. It is NP-Hard.
- These are the **hardest problems in NP**.
- Key consequence:
  - If one NP-Complete problem is solved in polynomial time, then **all problems in NP** can be solved in polynomial time.
  - $\Rightarrow$ This would prove $P = NP$.

NP-Hard Problems

- A problem is **NP-Hard** if *every problem in NP* can be reduced to it in polynomial time.
- NP-Hard problems are at least as difficult as the hardest problems in NP.

Visualizing Complexity Classes



NP-Hard (outside NP)

If any NP-Complete problem is solved in polynomial time, then $P = NP$.

From P to NP

- P contains all problems efficiently decidable by a deterministic Turing machine.
- Next: class NP — problems for which a solution can be **verified** in polynomial time.
- Key question:

$$\textbf{Is } P = NP\,?$$

Discussion Point

Open Problem

Is $P = NP$? One of the Millennium Prize Problems.

- A "yes" answer: All NP problems efficiently solvable.
- A "no" answer: Inherent barrier between efficient solving and verifying.
- Practical impact: Cryptography, optimization, AI, physics simulations.

Why Complexity Theory Matters for Quantum Computing

- Complexity theory provides a framework for measuring the difficulty of computational problems.
- Classical classes (P, NP, NP-Complete, NP-Hard) serve as benchmarks.
- Quantum algorithms are compared against these classical benchmarks.
- Key question: Can quantum computers efficiently solve problems that are classically intractable?

Quantum Complexity Classes

- Just as classical computation has P and NP, quantum computation introduces new classes:
  - **BQP** (Bounded-error Quantum Polynomial time): Problems solvable by quantum computers in polynomial time with bounded error.
  - **QMA** (Quantum Merlin-Arthur): Quantum analogue of NP, where a quantum proof can be verified efficiently.
- Comparing BQP to classical P and NP highlights the potential advantages of quantum computing.

Bridging Classical and Quantum Worlds

- Understanding classical complexity is essential:
  - To define what “speedup” means.
  - To identify which classical problems solve using quantum algorithms (e.g., factoring, search).
  - To avoid overstating quantum advantages.
- This bridge sets the stage for studying algorithms like:
  - **Shor’s Algorithm** (factoring in polynomial time).
  - **Grover’s Algorithm** (quadratic speedup for search).
- Quantum computing does not replace classical computation but it extends it.