Lecture 5: Computational Models and Complexity

The Physics of Computation

Moody T. Chu

North Carolina State University

Spring 2026

Lecture Overview

1. The Turing Perspective
2. The Strong Church-Turing Thesis
3. Thermodynamics of Computation
4. Reversible Computation
5. Complexity Classes

Where We Are Headed

- Lecture 4 built quantum computing from the *bottom up*: individual gates, circuits, and protocols.
- This lecture steps back and asks a *top-down* question: what can be computed at all, and at what cost?
  - **Computability:** what is an algorithm, formally? (Turing machines)
  - **Physical cost:** does computation have an inescapable energy price? (Landauer, reversibility)
  - **Efficiency:** which problems are *tractable*, and where does quantum computing change the picture? (complexity classes)
- This framework is what lets us state precisely, later in the course, *why* algorithms like Shor's and Grover's are significant — not just that they work, but that they beat every known classical alternative.

What is an Algorithm?

- At its core, an algorithm is a precise recipe for performing some task.
- To study algorithms mathematically, we need an idealized, formal model of computation — one simple enough to reason about, yet powerful enough to capture everything we intuitively consider "computable."
- The standard such model is the *Turing machine* (TM), introduced next.

The Turing Machine: A Sketch

- A Turing machine is a theoretical, idealized computing device invented by Alan Turing in 1936. Formally, it consists of:
  - An infinite **tape** of cells, each holding a symbol from a finite alphabet (e.g., $\{0, 1, \text{blank}\}$).
  - A **head** that reads/writes one cell at a time and can move left or right.
  - A finite set of internal **states**, including a designated start state and halting states.
  - A **transition function** δ: given the current state and the symbol under the head, it specifies the symbol to write, the direction to move, and the next state.
- Despite this extreme simplicity, a Turing machine can simulate *any* algorithm that runs on real hardware — it just may do so slowly.
- **Why bother with such a primitive model?** Because its very simplicity makes it possible to *prove* theorems about what is or isn't computable, and later, how efficiently.

Why the Turing Machine?

- It ignores the messy details of hardware implementation to focus strictly on logic.
- It defines the outer limits of mechanical computation, and serves as the theoretical model underlying modern CPUs.
- *The Church-Turing Thesis*: the class of functions computable by a Turing machine corresponds exactly to the class of functions we would naturally regard as computable by *any* algorithm.
  - ◇ This is a **thesis**, not a theorem — it cannot be formally proved, since "what we'd naturally call an algorithm" is an informal notion. But every alternative model of computation ever proposed (lambda calculus, register machines, cellular automata, ...) has been proven *equivalent* in computational power to the Turing machine, which is strong evidence for the thesis.

Efficiency and the Strong Church-Turing Thesis

- The (ordinary) Church-Turing thesis says nothing about *efficiency* — only about what is computable *in principle*, with no limit on time or resources.
- A natural follow-up question: what resources are required to perform a given computational task *efficiently*?
- *The Strong Church-Turing Thesis:* any realistic model of computation can be simulated on a **probabilistic Turing machine** (PTM) with at most a polynomial increase in the number of elementary operations required.
  - A PTM is a deterministic TM equipped with one additional instruction: "flip a fair coin," and branch its behavior on the result.
  - A PTM is allowed to be wrong occasionally, provided we can run it multiple times and combine the results (e.g., by majority vote) to drive the error rate down as close to zero as we like.
- The Strong Church-Turing Thesis is widely believed by many to be false, precisely because of quantum computing — though this has not been formally proved either way.

Randomness as a Shortcut

- The key idea behind the PTM: **randomness is a shortcut** — some problems that seem to need an exhaustive, exponential-time search can be solved efficiently by a clever randomized algorithm instead.
- Every modern PC with a hardware random-number generator is a physical implementation of a PTM.
- This idea will resurface precisely when we define the complexity class **BPP** later in this lecture — BPP is, by definition, the set of problems efficiently solvable by a PTM.

Why Quantum Computing Matters Here

- The strong thesis implies that if a problem has no efficient solution on a probabilistic Turing machine, it has no efficient solution on *any* physically realizable computing device.
- If we can build a quantum computer that solves a specific, well-defined problem efficiently — while no known (or provably possible) classical algorithm can — this casts serious doubt on the strong thesis.
- Such a result would show that the physical universe is more computationally capable than classical models of computation suggest. (We will see exactly such an example: Shor's factoring algorithm, in Lecture 7–8.)

The Thermodynamic Cost of Information

- *Landauer's principle* provides a fundamental connection between computation and physics: **erasing** one bit of information necessarily dissipates at least $k_B T \ln 2$ of energy into the environment as heat, where k_B is Boltzmann's constant and T is the temperature.
- Ordinary logic gates (AND, OR, XOR) are *logically irreversible*: knowing only the output, you generally cannot reconstruct the input(s). Information about the input is discarded — erased — every time such a gate fires.
- *The Consequence:*
  - Each such erasure increases the entropy of the environment, representing a fundamental physical floor on how much energy *any* classical computer built from irreversible gates must dissipate.
  - Conversely: if a computation could be carried out using *only* logically reversible operations (no erasure, ever), Landauer's principle implies **no** fundamental lower bound on the energy it must dissipate.
- This raises the natural question pursued in the next section: *can any computation be made reversible, without changing what it computes?*

Reversible Computation: Computing for "Free"

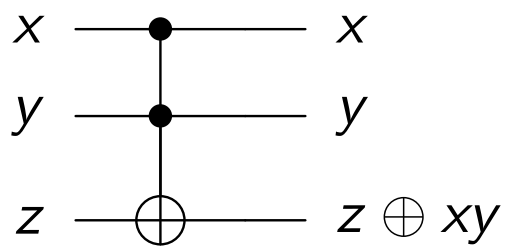
- A computation is **reversible** if and only if no information is erased at any step — equivalently, the map from inputs to outputs is one-to-one (so the input is always recoverable from the output).
- **Theorem (Bennett, 1973):** any irreversible circuit computing a function $f(x)$ can be efficiently simulated by a reversible circuit implementing

$$(x, y) \longmapsto (x,\ y \oplus f(x)), \tag{1}$$

  exactly the construction U_f from Lecture 4 — preserving the input x alongside the output is what makes the map invertible.
- This matters for quantum computing for a deeper reason than energy savings alone: a quantum circuit is built entirely out of *unitary* (hence reversible) gates, so **any classical subroutine we want to use inside a quantum algorithm must first be made reversible**.

An Example: The Toffoli Gate

- Recall from Lecture 4: the Toffoli gate leaves both control bits unchanged, and flips the target bit if and only if both control bits are 1.
- It lets us simulate *any* classical circuit without producing garbage bits that depend on the input — except the two control bits x, y themselves, preserved exactly.



- Setting $z = 0$ gives a reversible AND ($z' = xy$); with NOT (trivially reversible) and NAND's classical universality, this suffices to build *any* Boolean circuit reversibly.
- **The catch:** chaining several Toffoli gates typically leaves behind extra **garbage bits** — ancilla outputs that are neither the input nor the final answer. We address this next.

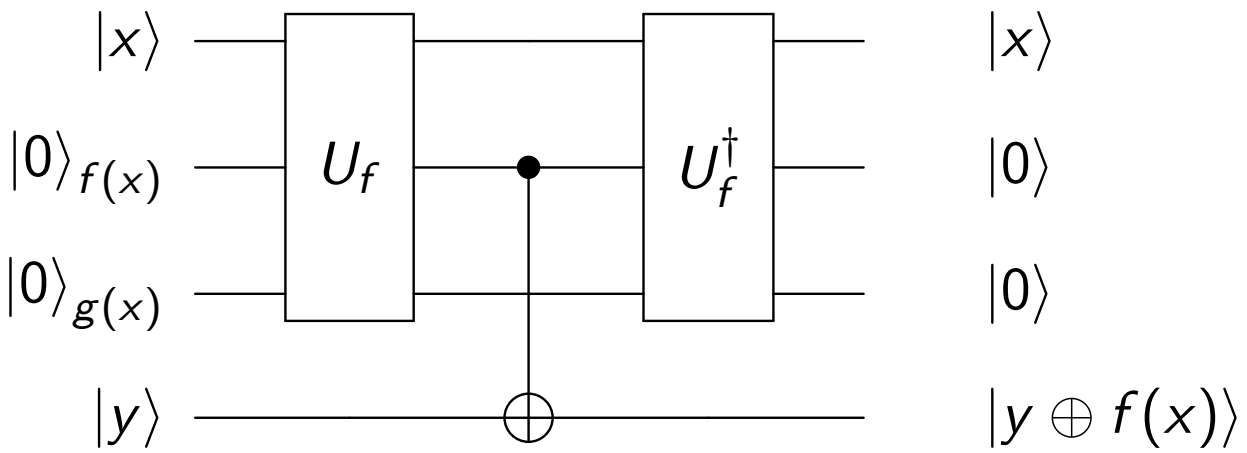
The Problem with Garbage

- Suppose U_f is built from a chain of reversible gates (like Toffolis) computing some function $f(x)$. In general, it produces *more* than just the answer:

$$U_f : |x\rangle\,|0\rangle\,|0\rangle \longmapsto |x\rangle\,|f(x)\rangle\,|g(x)\rangle\,,$$

  where $g(x)$ denotes whatever extra **garbage** (intermediate ancilla values) the circuit happened to produce along the way — entangled with x, but otherwise useless.
- **Why is garbage harmless classically, but fatal in a quantum algorithm?** Many quantum algorithms (e.g., the ones in Lectures 7–9) work by placing x in a *superposition* and relying on *interference* between different values of x to amplify the correct answer.
  - ◇ If $g(x)$ is left dangling, it acts as a "which-path" marker recording x — even if no one ever reads it, its mere existence *entangles* the register with x and destroys the interference the algorithm depends on.
- **Goal:** compute $f(x)$, copy out only the answer, then erase *all* the garbage — restoring the ancillas to $|0\rangle$ — without disturbing x or the answer. This is **Bennett's trick**, on the next slide.

Bennett's Trick (Uncomputation)



- Three stages, left to right: **compute** (U_f), **copy** (CNOT to a fresh wire $|y\rangle$), **uncompute** ($U_f^\dagger$).
- **Net result:** $|x\rangle\,|0\rangle\,|0\rangle\,|y\rangle \to |x\rangle\,|0\rangle\,|0\rangle\,|y \oplus f(x)\rangle$ — exactly the clean U_f from Lecture 4, with *zero* garbage left behind, at the cost of running U_f *twice*.

Bennett's Trick: Step by Step

1. **Compute:** apply U_f. The state of the first three wires becomes

$$|x\rangle \otimes |f(x)\rangle \otimes |g(x)\rangle .$$

2. **Copy:** apply a CNOT, controlled on the $f(x)$ register, targeting a *fresh* wire $|y\rangle$. This copies the answer out to $|y \oplus f(x)\rangle$, safely outside the part of the circuit we are about to erase.
3. **Uncompute:** apply $U_f^\dagger$. Since $U_f^\dagger U_f = I$, and x, $f(x)$, $g(x)$ are untouched by the copy step, this exactly reverses step 1, restoring $|0\rangle_{f(x)} |0\rangle_{g(x)}$.

Why this works: the copy step only reads the $f(x)$ register (via CNOT control), it never modifies x, $f(x)$, or $g(x)$ — so when we run $U_f^\dagger$ afterward, it sees exactly the same state it would have right after step 1, and correctly reverses it back to all zeros.

Bridging to Complexity

- Bennett's trick shows that *any* classical computation can be made fully reversible at only a **polynomial** overhead in time and space (roughly double the gates, plus one extra ancilla per output bit).
- This is precisely the kind of "efficient simulation" the Strong Church-Turing Thesis is concerned with — reversibility itself costs us nothing asymptotically.
- We are now ready to make the notion of "efficient" fully precise, and to classify problems by how much computational resource they truly require.

The Landscape of Difficulty: P and NP

- **Computational complexity** is the study of the time and space resources required to solve computational problems, as a function of **input size** n (e.g., the number of bits in an integer, or the number of vertices in a graph).
- A problem is considered **tractable** if it can be solved in time bounded by a polynomial in n (such as n^2 or n^3), and **intractable** if every known algorithm requires time that grows exponentially in n (such as 2^n).
  - ◇ **Why the distinction matters:** at $n = 100$, $n^3 = 10^6$ (instant on any computer), while $2^n \approx 1.3 \times 10^{30}$ (longer than the age of the universe, even at a trillion operations per second).
- We distinguish between how hard it is to *find* an answer versus how hard it is to *verify* one — this is exactly the distinction between the classes P and NP, defined next.

The Class P

- P stands for *polynomial* time — the class of "finding power."
- The class P consists of all decision problems solvable by a (deterministic) algorithm whose running time is bounded by a polynomial function of the input size n.
- Problems in P are regarded as **efficiently solvable**, making them practical for real-world computation, even for large inputs.
- *Examples:*
  - Adding or multiplying two n-digit numbers: $O(n)$ or $O(n^2)$ operations.
  - Sorting a list of n numbers: $O(n \log n)$ operations.
  - Determining whether a graph is connected: polynomial in the number of vertices and edges.

The Class NP

- NP stands for *nondeterministic polynomial* time — the class of "verification power."
- A problem is in NP if, whenever the answer is "yes," there exists a short **certificate** (or *witness*) — a piece of evidence that lets a verifier confirm the "yes" answer in polynomial time, *given* that certificate.
- *Example:* the **Hamiltonian cycle** problem.
  - Given a graph (a network of points connected by edges), does there exist a cycle that visits every vertex exactly once and returns to the start?
  - *Finding* such a cycle is believed to be hard (no known polynomial-time algorithm).
  - But *verifying* a proposed cycle is easy: check it visits every vertex once and consecutive vertices are connected — both take only polynomial time. The cycle itself is the certificate.

P versus NP

- **Key fact:** $P \subseteq NP$ always holds — any problem solvable in polynomial time is trivially also verifiable in polynomial time (just ignore the certificate and solve it directly).
  - **Open question:** is this inclusion *strict*? I.e., are there problems in NP that are *not* in P? This is the famous $P \stackrel{?}{=} NP$ problem — one of the most important open problems in all of mathematics and computer science.

NP-Completeness

- **NP-complete** problems are the "master keys" of the NP world: every problem in NP can be *reduced* to an NP-complete problem in polynomial time.
- **Consequence:** solving *any single* NP-complete problem in polynomial time would imply P = NP, and every problem in NP would suddenly become efficiently solvable.
  - ◇ This is one of the seven Clay Millennium Prize problems — still open, with a $1,000,000 prize for a resolution either way.
- *Example: Boolean Satisfiability (SAT).* Given a Boolean formula built from variables $x_1, x_2, \ldots$ combined with AND, OR, NOT, does there exist an assignment of each x_i to 0 or 1 that makes the whole formula evaluate to 1 (True)?
  - ◇ The **Cook-Levin theorem** (1971) establishes that SAT is NP-complete — the first problem ever proven so, and the seed from which thousands of other NP-completeness proofs grow (by reduction).
- Thousands of practical optimization, scheduling, and design problems reduce to SAT, which is why fast practical SAT solvers (even without a polynomial worst-case guarantee) are of enormous real-world value.

Polynomial-Time Reduction

- To compare the difficulty of two problems, we use **reduction**: problem A **reduces** to problem B (written $A \leq_p B$) if there is a polynomial-time computable function g that transforms any instance x of A into an instance $g(x)$ of B, such that

$$x \text{ is a "yes" instance of } A \iff g(x) \text{ is a "yes" instance of } B.$$

- **Intuition:** $A \leq_p B$ means "A is no harder than B" — if we had a fast algorithm for B, we could use it (plus the cheap translation g) to solve A fast too.
- Reduction is **transitive**: if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$ (compose the two translations). This is what lets NP-completeness proofs chain together.

NP-Completeness: Formal Definition

A problem L is **NP-complete** if it satisfies *both* of the following:

1. $L \in$ **NP:** a proposed "yes" answer for L can be verified in polynomial time, given a certificate (Section 5).
2. **NP-hardness:** every problem $A \in \text{NP}$ satisfies $A \leq_p L$ — that is, L is *at least as hard as every other problem in NP*.

- Condition (2) is precisely what makes L a "master key": a polynomial-time algorithm for L, combined with the reduction $A \leq_p L$, would yield a polynomial-time algorithm for *every* $A \in \text{NP}$ — forcing $\text{P} = \text{NP}$.
- A problem satisfying only condition (2) (without necessarily being in NP itself) is called **NP-hard**; NP-complete is shorthand for "NP-hard *and* in NP."

The Class BPP: Dealing with Uncertainty

- **BPP** (Bounded-error Probabilistic Polynomial time) is the class of problems solvable by a PTM (Section 2) in polynomial time, with probability of a correct answer at least $3/4$ on every run.
  - $\diamond$ $\mathrm{P} \subseteq \mathrm{BPP}$: trivially, since a deterministic algorithm is a probabilistic one that simply never uses its coin.
- **Why a 25% error rate is perfectly acceptable:** run the algorithm independently many times and take the majority answer.
  - $\diamond$ With 100 independent runs, the **Chernoff bound** guarantees the probability that the majority vote is still wrong is astronomically small: about $2 \times 10^{-6}\,\%$ (roughly 1 in 50 million).
  - $\diamond$ More runs drive this error rate down *exponentially* fast — 200 runs already pushes the failure probability below $10^{-11}\%$.

BPP in Practice

- *Example:* the **Miller-Rabin primality test** — determining whether a large integer is prime, using randomness to avoid an exhaustive trial-division search.
- BPP is generally regarded as the class of problems that are *truly* efficiently solvable on a classical computer.
  - ◇ It is widely suspected (though unproven) that $P = BPP$ — i.e., that randomness never provides more than a polynomial speedup over deterministic computation.

The Class BQP: The Quantum Frontier

- **BQP** (Bounded-error Quantum Polynomial time) is the quantum analogue of BPP: the class of problems efficiently solvable on a quantum computer, with probability of a correct answer at least $3/4$ (amplifiable exactly as in BPP).
- *Example: Integer Factoring.*
  - ◇ Finding the prime factors of a large number is believed to have no efficient (polynomial-time) classical algorithm.
  - ◇ Factoring is also believed *not* to be NP-complete — it is strongly suspected to lie in **NPI** ("NP-Intermediate"): in NP, but neither in P nor NP-complete, assuming $P \neq NP$.
  - ◇ Shor's algorithm (Lecture 7–8) solves factoring in polynomial time *on a quantum computer* — placing it in BQP.
  - ◇ This is the single clearest known example of a problem efficiently solvable quantumly but believed intractable classically.

The Complexity Hierarchy in Context

- **PSPACE** is the class of problems solvable using only a *polynomial* amount of memory (space), with *no* limitation on how much time the computation takes.
- Our current understanding of how these classes nest inside one another:

$$\mathrm{P} \subseteq \mathrm{BPP} \subseteq \mathrm{BQP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPSPACE}$$

  - $\mathrm{BPP} \subseteq \mathrm{BQP}$: any classical randomized algorithm can be simulated on a quantum computer.
  - $\mathrm{BQP} \subseteq \mathrm{PSPACE}$: a quantum circuit's amplitudes can always be computed using only polynomial memory (though possibly exponential time).
  - $\mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE}$ is one of the *few* inclusions here that is actually **proven** strict, by the space hierarchy theorem.

The Nontrivial Takeaway

- It is currently **unknown** whether any of the first four inclusions in the hierarchy

$$\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{BQP} \subseteq \mathsf{PSPACE}$$

  are strict.
  - ◇ If it turned out that $\mathsf{P} = \mathsf{PSPACE}$, quantum computers would offer *no* computational advantage over classical ones for any problem in this entire hierarchy.
  - ◇ Conversely, believing quantum computers are genuinely more powerful (as essentially all evidence suggests) means believing at least one of these inclusions is strict — we just cannot yet prove which one.

Summary of Lecture 5

- **Turing machines** formalize the notion of an algorithm; the **Church-Turing thesis** asserts this captures everything intuitively computable.
- The **Strong Church-Turing Thesis** extends this to efficiency, conjecturing PTMs capture everything *efficiently* computable — a claim quantum computing directly challenges.
- **Landauer's principle:** erasing information costs energy ($\geq k_B T \ln 2$ per bit); **reversible** computation sidesteps this floor, and is mandatory inside any quantum circuit.
- **Bennett's trick:** compute, copy the answer out, then uncompute — a clean, garbage-free U_f preserving quantum interference.
- **Complexity classes:** P (solvable) $\subseteq$ NP (verifiable); NP-complete problems (like SAT) are the hardest in NP; BPP, BQP add bounded-error randomness and quantum resources.
- Full picture: P $\subseteq$ BPP $\subseteq$ BQP $\subseteq$ PSPACE $\subseteq$ EXPSPACE, with only the last inclusion proven strict.

Looking Ahead: Lecture 6

Having established *what* efficient computation means, we return to building the quantum toolbox needed to achieve it. In Lecture 6 we will cover:

- **Decomposition of unitaries:** breaking any quantum operation into elementary gates.
- **Controlled-U gates** and multi-controlled operations.
- **Quantum simulation:** using a quantum computer to simulate quantum physics itself — historically, the very first proposed application of quantum computing.