Theory of Computer Science Notes

A Long-Term Study Plan in Theoretical Computer Science

Shen Han

December 23, 2025

Contents

Part I

Proof Techniques and Discrete Foundations

Chapter 1

What Is a Rigorous Proof?

Chapter 2

Sets, Functions, and Relations

Chapter 3

Equivalence Relations and Quotients

Chapter 4

Partial Orders and Lattices

4.1 Partial Orders: Allowing Incomparability

4.1.1 Posets and the idea of incomparability

A *partial order* on a set P is a relation $\leq$ satisfying:

1. **Reflexive:** $\forall x \in P,\ x \leq x$.
2. **Antisymmetric:** $(x \leq y\ \wedge\ y \leq x) \Rightarrow x = y$.
3. **Transitive:** $(x \leq y\ \wedge\ y \leq z) \Rightarrow x \leq z$.

A set equipped with a partial order is a *poset*.

The key new feature (compared to total orders) is **incomparability**: for some $a, b \in P$, it may happen that neither $a \leq b$ nor $b \leq a$ holds. For example, in $(\mathcal{P}(X), \subseteq)$, $\{1\}$ and $\{2\}$ are incomparable.

4.1.2 A quick non-example

Define on Z:

$$a \preceq b \iff a - b \text{ is even.}$$

Then $1 \preceq 3$ and $3 \preceq 1$ but $1 \neq 3$, so antisymmetry fails. (So this is closer to an equivalence relation than a partial order.)

4.2 Hasse Diagrams and Order-Theoretic Extremes

4.2.1 Cover relation and Hasse diagrams

In a finite poset, y *covers* x (written $x \prec y$) if $x < y$ and there is no z with $x < z < y$. A *Hasse diagram* draws only cover relations (omitting reflexive and transitive edges).

4.2.2 Least vs. minimal (and greatest vs. maximal)

In a poset $(P, \leq)$:

- m is a *least element* if $\forall x \in P,\ m \leq x$.
- a is *minimal* if there is no $x \neq a$ with $x \leq a$.

A poset may have many minimal elements but has at most one least element (if it exists). Dual notions: *greatest* vs. *maximal*.

4.3 Thin Categories: The Categorical View of Posets

4.3.1 From posets to thin categories

A poset can be viewed as a *thin category* (a category with at most one morphism between any two objects). We fix the standard convention:

$$x \to y \iff x \leq y.$$

Under this convention:

- A least element $\bot$ becomes an **initial object**.
- A greatest element $\top$ becomes a **terminal object**.

If one reverses arrows, initial/terminal swap; this matches the intuition that "universal object" depends on arrow direction.

4.3.2 Why morphisms are unique

In a thin category arising from a poset, $\mathrm{Hom}(x, y)$ is either empty (if xy) or a singleton (if $x \leq y$). So whenever an arrow exists, it is *unique by construction.*

4.4 Meets, Joins, and Lattices

4.4.1 Bounds, meet, and join

Given $A \subseteq P$:

- u is an *upper bound* if $\forall a \in A,\ a \leq u$.
- $\sup A$ is the *least upper bound* (if it exists).
- $\inf A$ is the *greatest lower bound* (if it exists).

For two elements x, y:

$$x \vee y := \sup\{x, y\}, \qquad x \wedge y := \inf\{x, y\}.$$

4.4.2 Meet/product and join/coproduct in the thin-category convention

With $x \to y \iff x \leq y$:

- $x \wedge y$ is the **product** of x and y (greatest lower bound).
- $x \vee y$ is the **coproduct** of x and y (least upper bound).

Example in $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$:

$$A = \{1, 2\},\ B = \{2, 3\} \quad \Rightarrow \quad A \wedge B = A \cap B = \{2\}, \quad A \vee B = A \cup B = \{1, 2, 3\}.$$

4.4.3 Lattices and bounded lattices

A poset is a *lattice* if every pair (x, y) has both $x \wedge y$ and $x \vee y$. It is *bounded* if it also has $\bot$ and $\top$.

In $(\mathcal{P}(X), \subseteq)$:

$$\bot = \emptyset, \qquad \top = X.$$

We explicitly verified $\bot = \emptyset$ is least (hence initial in the thin-category convention).

4.4.4 Complete lattices

A poset (often a lattice) is *complete* if every subset $S \subseteq L$ has both $\sup S$ and $\inf S$. In $\mathcal{P}(X)$:

$$\sup \mathcal{F} = \bigcup \mathcal{F}, \qquad \inf \mathcal{F} = \bigcap \mathcal{F},$$

so $\mathcal{P}(X)$ is a complete lattice.

4.5 Distributive Lattices, Boolean Algebras, and Stone Duality (Outline)

4.5.1 Distributive lattices

A lattice is *distributive* if

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$

(and dually for $\vee$ over $\wedge$). In $\mathcal{P}(X)$, these are the usual distributive laws of $\cap$ and $\cup$.

We also noted a typical *non-lattice* pattern: if a, b have no common lower bound, then $a \wedge b$ does not exist, so the poset cannot be a lattice.

4.5.2 Boolean algebras

A *Boolean algebra* is a bounded distributive lattice in which every element has a complement:

$$x \wedge \neg x = \bot, \qquad x \vee \neg x = \top.$$

In $\mathcal{P}(X)$, $\neg A = X \setminus A$. For $X = \{1, 2, 3\}$ and $A = \{1, 2\}$, $\neg A = \{3\}$.

4.5.3 Ultrafilters and the bivalence intuition

A *filter* on $\mathcal{P}(X)$ is a family $\mathcal{F} \subseteq \mathcal{P}(X)$ such that:

1. $\emptyset \notin \mathcal{F}$,
2. upward closed: $A \in \mathcal{F},\ A \subseteq B \Rightarrow B \in \mathcal{F}$,
3. closed under finite intersection: $A, B \in \mathcal{F} \Rightarrow A \cap B \in \mathcal{F}$.

We checked the *principal filter* at a point $x \in X$:

$$\mathcal{F}_x := \{A \subseteq X :\ x \in A\}$$

satisfies these axioms.

An *ultrafilter* is a maximal filter; in $\mathcal{P}(X)$ (and in general Boolean algebras) it corresponds to a **bivalent choice principle**: for each $A \subseteq X$, exactly one of A or $X \setminus A$ is "accepted" (and never both). We also observed why a and $\neg a$ cannot both belong to an ultrafilter: closure under $\wedge$ would force $\bot$.

4.5.4 Stone duality (conceptual outline)

Stone duality relates Boolean algebras to Stone spaces. The core dictionary (informal):

- Points $\leftrightarrow$ ultrafilters.
- Boolean element b determines a clopen set

$$\widehat{b} := \{\mathcal{U} :\ b \in \mathcal{U}\}.$$

- The clopen sets form a Boolean algebra, recovering the original Boolean algebra up to duality.

(We keep this as an outline; details depend on the topology generated by the sets $\widehat{b}$.)

4.6 Why Nets and Filters: Beyond Sequences

4.6.1 Why sequences can be insufficient

A sequence uses the index set N, so it can only express "eventually" along a *countable* direction. In some spaces, convergence/closure phenomena require more general directed indexing.

4.6.2 Directed sets and nets

A *directed set* $(I, \preceq)$ satisfies: for any $i, j \in I$, there exists $k \in I$ with $i \preceq k$ and $j \preceq k$. A *net* in X is a function $x : I \to X$, written $(x_i)_{i \in I}$.

Convergence of a net (x_i) to x: for every neighborhood $U \ni x$, there exists i_0 such that $i \succeq i_0 \Rightarrow x_i \in U$.

A standard directed index for approaching a point is the neighborhood system:

$$I = \mathcal{N}(x), \qquad U \preceq V \iff U \supseteq V,$$

using that neighborhoods are closed under finite intersections.

4.6.3 Closure via nets

Key fact (used as motivation): $x \in \overline{A}$ iff there exists a net in A converging to x. A canonical construction: for each neighborhood $U \ni x$, choose $a_U \in A \cap U$ (possible exactly when $x \in \overline{A}$).

4.6.4 From nets to filters: the tail filter

Given a net indexed by $(I, \preceq)$, define tails

$$T_{i_0} := \{i \in I : \ i \succeq i_0\},$$

and the tail filter on I:

$$\mathcal{F}_{\text{tail}} := \{A \subseteq I : \exists i_0, \ T_{i_0} \subseteq A\}.$$

For sequences ($I = N$), this is the usual Fréchet "eventually" filter.

4.7 Galois Connections (TCS + Relation-Induced View)

4.7.1 Abstract interpretation: α and γ

Let $(C, \leq_C)$ be a concrete domain and $(A, \leq_A)$ an abstract domain. A pair of monotone maps

$$\alpha : C \to A, \qquad \gamma : A \to C$$

forms a (covariant) Galois connection when:

$$\alpha(c) \leq_A a \iff c \leq_C \gamma(a).$$

Intuitively, α is the best abstraction and γ is concretization.

4.7.2 Relation-induced (contravariant) Galois connection

Given a relation $R \subseteq X \times Y$ ("x has attribute y"), define for $A \subseteq X$ and $B \subseteq Y$:

$$A' := \{y \in Y : \ \forall x \in A, \ xRy\}, \qquad B' := \{x \in X : \ \forall y \in B, \ xRy\}.$$

Then

$$A \subseteq B' \iff B \subseteq A'.$$

This captures the familiar *order-reversing* behavior of classical Galois correspondences (e.g. "smaller subgroup $\leftrightarrow$ larger fixed field").

4.7.3 Many-semantics perspective (family of contexts)

We also discussed a "multi-context" viewpoint: instead of one fixed satisfaction relation, consider a family $(R_i)_{i\in I}$ (different semantic assignments). One can compute closures per context and then aggregate (e.g. by unions) to form a panoramic family of definable consequences. This emphasizes the *structure of closures* rather than forcing a single unified universe.

Mini-Recap

- Posets generalize order by allowing incomparability; $\mathcal{P}(X)$ with $\subseteq$ is the guiding example.
- With $x \to y \iff x \leq y$, $\bot$ is initial, $\top$ is terminal, meet is product, join is coproduct.
- Lattices require pairwise meets/joins; complete lattices require arbitrary sups/infs.
- Boolean algebras add complements; ultrafilters encode bivalent "truth assignments" and lead into Stone duality.
- Nets generalize sequences via directed index sets; filters repackage "eventually" as a family of large sets.
- Galois connections appear in TCS via $\alpha \dashv \gamma$ and in algebra/logic via relation-induced closures.

Chapter 5

Graph Theory Basics

5.1 Graphs: Basic Definitions

A *graph* is a discrete structure consisting of *vertices* (or *nodes*) together with *edges* describing which vertices are related. Graphs serve as a uniform language for many TCS objects: state-transition systems, dependency graphs, networks, control-flow graphs, etc.

5.1.1 Graphs as Pairs $G = (V, E)$

We typically write a graph as

$$G = (V, E),$$

where V is a set of vertices and E is a set of edges. The precise meaning of an "edge" depends on the graph variant (directed vs. undirected, allowing loops, etc.).

5.1.2 Undirected vs. Directed Graphs

Undirected graphs. An *undirected* graph has edges with no orientation. Each edge connects two distinct vertices and is represented as an unordered pair:

$$E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}.$$

Intuitively, $\{u, v\}$ indicates a symmetric relationship between u and v.

Directed graphs (digraphs). A *directed* graph has oriented edges (arcs). Each edge is an ordered pair:

$$E \subseteq V \times V.$$

An edge (u, v) points from u to v, capturing an asymmetric relationship (e.g. "u can transition to v").

5.1.3 Simple Graphs, Multigraphs, and Loops

Simple graphs. A *simple* graph (the default in many theorems and counting arguments) has *no parallel edges* and typically *no self-loops*. Concretely:

- no parallel edges: at most one edge between any two distinct vertices;
- no self-loop: no edge of the form (v, v) (directed) or $\{v, v\}$ (undirected).

Multigraphs and pseudographs. In modeling, we may allow richer edge sets:

- a *multigraph* allows *parallel edges* (multiple edges between the same pair of vertices);
- a *pseudograph* (terminology varies) may allow *self-loops* and sometimes parallel edges as well.

When parallel edges are allowed, an upper bound on $|E|$ is no longer determined solely by $|V|$ unless we impose additional constraints (e.g. at most k parallel edges).

5.1.4 Weighted Graphs

A *weighted* graph is a graph equipped with a weight function on edges:

$$G = (V, E, w), \qquad w : E \to R$$

(or more generally $w : E \to W$ for some weight set W). We interpret $w(e)$ as a cost, length, capacity, similarity score, multiplicity aggregate, etc., depending on context. Weighted graphs are common in algorithmic problems such as shortest paths and network flows.

5.1.5 Maximum Number of Edges (Simple Case)

Let $|V| = n$.

- Simple undirected (no loops): $|E| \leq \binom{n}{2} = \frac{n(n-1)}{2}$.
- Simple directed (no loops): $|E| \leq n(n-1)$.
- Simple directed (loops allowed): $|E| \leq n^2$.

Remark. For proofs and clean complexity bounds, we often assume "simple" graphs. For realistic modeling, multiedges and loops can be meaningful; sometimes they can be compressed into edge weights, but doing so may discard edge-level identity or distributional information.

5.2 Graph Representations and Basic Invariants

In this subsection we introduce two standard data representations of a graph and the most basic numerical invariants. Throughout, let $G = (V, E)$ be a (possibly directed and/or weighted) graph, with $n := |V|$ and $m := |E|$.

5.2.1 Adjacency list (sparse representation)

For each vertex $v \in V$, the *adjacency list* stores the set (or list) of its neighbors:

$$\mathrm{Adj}(v) = \{\, u \in V : (v, u) \in E \,\} \quad \text{(directed)}, \qquad \mathrm{Adj}(v) = \{\, u \in V : \{v, u\} \in E \,\} \quad \text{(undirected)}.$$

For a weighted graph, we store pairs (u, w_{vu}) for each edge (v, u) (or $\{v, u\}$). For an undirected graph, each edge is stored twice: once in $\mathrm{Adj}(v)$ and once in $\mathrm{Adj}(u)$.

Space and access. Adjacency lists use $\Theta(n + m)$ memory. Enumerating all neighbors of a vertex v costs $\Theta(\deg(v))$ time (or $\Theta(\deg^+(v))$ in a directed graph).

5.2.2 Adjacency matrix (dense representation)

Fix an ordering of $V = \{1, 2, \ldots, n\}$. The *adjacency matrix* is the $n \times n$ matrix $A = (A_{ij})$ defined by

$$A_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \text{ (unweighted)} \\ w_{ij}, & \text{if } (i,j) \in E \text{ (weighted)} \\ 0 \text{ or } +\infty, & \text{if } (i,j) \notin E \text{ (convention).} \end{cases}$$

For undirected graphs, A is symmetric ($A_{ij} = A_{ji}$).

Convention (missing edges). When A is used as a *0–1 adjacency indicator*, it is natural to set $A_{ij} = 0$ for "no edge". When A is used to encode *edge costs/lengths* (e.g. shortest-path problems), it is common to set $A_{ij} = +\infty$ for "no edge" so that nonexistent edges are never chosen as finite-length moves. For *capacity* (flow) networks, one often uses $A_{ij} = 0$ to mean "zero capacity".

Space and access. Adjacency matrices use $\Theta(n^2)$ memory. Testing whether an edge exists is $O(1)$ by reading A_{ij}, but enumerating all neighbors of i typically costs $\Theta(n)$ (scan row i).

5.2.3 Degrees and basic counts

Undirected graphs. The *degree* of v is

$$\deg(v) = |\{\, u \in V : \{v, u\} \in E \,\}|.$$

Directed graphs. The *out-degree* and *in-degree* are

$$\deg^+(v) = |\{\, u \in V : (v,u) \in E \,\}|, \qquad \deg^-(v) = |\{\, u \in V : (u,v) \in E \,\}|.$$

5.2.4 Handshaking lemma (undirected)

For any undirected graph $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2m.$$

Reason (intuition). Each undirected edge contributes 1 to the degree count of each of its two endpoints, hence contributes 2 to the total sum.

Remark (why adjacency lists are common in algorithms). Many graph algorithms repeatedly *enumerate neighbors*. With adjacency lists this costs $\sum_v \deg(v) = 2m$ overall, whereas with adjacency matrices it often costs $\Theta(n^2)$ due to scanning many non-edges.

5.3 Walks, Paths, Cycles, and Connectivity

Let $G = (V, E)$ be a graph. For an undirected graph, edges are unordered pairs $\{u, v\}$; for a directed graph, edges are ordered pairs (u, v).

5.3.1 Walks, trails, and (simple) paths

A *walk* of length k is a sequence of vertices

$$v_0, v_1, \ldots, v_k$$

such that each consecutive pair forms an edge:

$$(v_{i-1}, v_i) \in E \quad \text{(directed)}, \qquad \{v_{i-1}, v_i\} \in E \quad \text{(undirected)}.$$

Walks may repeat vertices and edges.

A *trail* is a walk in which no edge is repeated. A *(simple) path* is a walk in which no vertex is repeated. (In particular, every path is a trail, and every trail is a walk.)

Length and weight. The (unweighted) *length* of a walk is the number of steps k. In a weighted graph, one often defines the *cost/length* of a walk as $\sum_{i=1}^{k} w(v_{i-1}, v_i)$, where w is the edge-weight function.

Cycles and acyclicity

A *cycle* is a closed walk $v_0, v_1, \ldots, v_k$ with $v_0 = v_k$. A *simple cycle* typically additionally requires $v_0, \ldots, v_{k-1}$ to be distinct.

A graph is *acyclic* if it contains no (simple) cycles. In undirected graphs, acyclicity is the key structural condition behind the notion of a *tree* (cf. §5.4).

Connectivity in undirected graphs

In an undirected graph, two vertices $u, v \in V$ are *connected* if there exists a path from u to v. The graph is *connected* if every pair of vertices is connected.

The relation "u is connected to v" is an equivalence relation on V (reflexive via the length-0 path, symmetric by reversing paths, and transitive by concatenation). Its equivalence classes are called the *connected components*. Equivalently, a connected component is a maximal (by inclusion) connected subgraph.

Reachability and strong connectivity in directed graphs

In a directed graph, *reachability* is directional: we say v is *reachable* from u (denoted uv) if there exists a directed path from u to v. In general, reachability is *not* symmetric.

Two vertices u, v are *strongly connected* if uv and vu. This defines an equivalence relation on V; its equivalence classes are the *strongly connected components* (SCCs).

5.3.2 DAGs and topological order

A *directed acyclic graph* (DAG) is a directed graph with no directed cycle.

A *topological order* of a directed graph is a linear ordering of vertices

$$v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$$

such that for every edge $(u, v) \in E$, the vertex u appears before v in the order. Intuitively, a topological order is possible precisely when the directed graph has no directed cycles (i.e. it is a DAG); algorithmic constructions will appear later in the graph-algorithms chapter.

Remark (glossary hierarchy).

path $\subseteq$ trail $\subseteq$ walk, (directed) reachability is not symmetric, while strong connectivity is.

5.4 Trees and Spanning Trees

In this subsection we focus on undirected graphs. Trees provide the simplest connected structures (no redundancy), and spanning trees extract a minimal "backbone" from an arbitrary connected graph.

Trees

An undirected graph $T = (V, E)$ is called a *tree* if it is *connected* and *acyclic* (contains no cycle).

Equivalent characterizations (intuition)

For finite undirected graphs, the following viewpoints are fundamental and (with standard proofs) equivalent:

1. **(Minimal connectivity)** T is connected, and removing any edge disconnects it.
2. **(Maximal acyclicity)** T is acyclic, and adding any new edge creates a cycle.
3. **(Unique simple paths)** Between any two distinct vertices $u, v \in V$, there is a *unique* simple path.

Remark. The presence of a cycle means "redundancy": there exist two different routes between some vertices, so one can delete an edge on the cycle while keeping the graph connected.

Edge count of a tree

If T is a tree on $n = |V|$ vertices, then

$$|E| = n - 1.$$

Proof idea (leaf-stripping). A finite tree always has a *leaf* (a vertex of degree 1). Removing a leaf and its incident edge yields a smaller tree. Repeating until one vertex remains (with 0 edges) shows that each removed vertex corresponds to exactly one removed edge, hence $|E| = n - 1$.

5.4.1 Spanning trees

Let $G = (V, E)$ be a connected undirected graph. A *spanning tree* of G is a subgraph

$$T = (V, E_T) \quad \text{with } E_T \subseteq E$$

such that T is a tree. Equivalently, T is a cycle-free connected subgraph that *spans* all vertices of G.

Basic consequence. Any spanning tree on n vertices has exactly $n - 1$ edges, so a spanning tree can be viewed as a choice of $n - 1$ edges from E that keeps all vertices connected and removes all cycles.

Remark (algorithmic motivation). Many graph procedures (e.g. traversal, connectivity certificates, and minimum spanning tree problems) work by maintaining or extracting a spanning tree as a minimal connected "skeleton" of the original graph.

5.5 Bipartite Graphs and Matchings (Intro)

Bipartite graphs are graphs whose vertices can be separated into two "types" so that every edge goes across the two types. They are central in allocation/assignment problems, and they form the natural setting for matchings.

Bipartite graphs and 2-coloring

An undirected graph $G = (V, E)$ is *bipartite* if there exists a partition

$$V = L \sqcup R$$

such that every edge has one endpoint in L and the other in R. Equivalently, G is bipartite if and only if its vertices admit a *2-coloring* (say red/blue) such that each edge connects vertices of different colors.

Odd cycles and a standard characterization

A fundamental (and very useful) characterization is:

$$G \text{ is bipartite } \iff G \text{ contains no odd cycle.}$$

Intuition. Along any walk around a cycle, a valid 2-coloring must alternate colors at each step. Returning to the starting vertex after an odd number of steps would force the starting vertex to have two different colors, a contradiction. Even cycles (e.g. a 4-cycle) do not cause this issue.

5.5.1 Matchings

A *matching* is a set of edges $M \subseteq E$ such that no two edges in M share a common endpoint. In other words, each vertex is incident to at most one edge of M.

- A *maximum matching* is a matching of largest possible size $|M|$.
- A *perfect matching* is a matching that covers every vertex (equivalently, every vertex is incident to exactly one matching edge).

Remark (parity and perfect matchings). A perfect matching can exist only if $|V|$ is even. This is a constraint on perfect matchings, not on bipartiteness: bipartite graphs may have odd or even numbers of vertices.

5.5.2 Alternating and augmenting paths (intuition)

Fix a matching M. A path is called *alternating* (with respect to M) if its edges alternate between non-matching edges ($\notin M$) and matching edges ($\in M$).

An alternating path whose two endpoints are *unmatched* vertices is called an *augmenting path*. Along such a path, we can *flip* the status of edges (matching $\leftrightarrow$ non-matching), which produces a new matching of size $|M| + 1$.

Why alternation matters. The alternation ensures that after flipping, every internal vertex of the path remains incident to exactly one matching edge, so the matching constraint (no shared endpoints) is preserved, while the two previously-unmatched endpoints become matched.

Remark (algorithmic outlook). Many matching algorithms repeatedly search for augmenting paths and flip them until no further augmentation is possible. In later chapters, matchings in bipartite graphs will also be related to flow networks.

5.6 Directed Graph Structure: SCCs and DAGs

This subsection organizes the basic structural notions for directed graphs. The key idea is that directed cycles represent *feedback*, while acyclic structure admits a *consistent ordering*.

5.6.1 Reachability and strong connectivity

Let $G = (V, E)$ be a directed graph. We say v is *reachable* from u (denoted uv) if there exists a directed path from u to v. Reachability is generally not symmetric.

Vertices u, v are *strongly connected* if uv and vu. This is an equivalence relation on V; its equivalence classes are the *strongly connected components* (SCCs). Intuitively, an SCC is a maximal region of mutual reachability (a "feedback cluster").

5.6.2 Condensation graph and acyclicity

Contract each SCC into a single *super-vertex*. The resulting directed graph is called the *condensation graph* (or SCC quotient graph) of G.

Key fact. The condensation graph is always a *DAG* (it has no directed cycle).
Intuition. If there were a directed cycle among SCCs, then each SCC on the cycle would be reachable from the others, so they would merge into one larger SCC, contradicting maximality.

5.6.3 DAGs and topological order

A *directed acyclic graph* (DAG) is a directed graph with no directed cycle.

A *topological order* is a linear ordering of vertices $v_{\sigma(1)}, \ldots, v_{\sigma(n)}$ such that for every edge $(u, v) \in E$, the vertex u appears before v. Equivalently, a topological order is a total order extending the "must-come-before" constraints given by directed edges.

Key equivalence. A directed graph admits a topological order *if and only if* it is a DAG.

Remark (from feedback to hierarchy). SCCs capture the intrinsically cyclic/feedback part of a directed graph. After contracting SCCs, what remains is a DAG, which can be arranged into layers via a topological order.

5.7 Flows and Cuts (Conceptual)

Flow networks encode transport under capacity constraints. They form a bridge between graph structure and optimization, and they will reappear later in the max-flow/min-cut chapter.

5.7.1 Flow networks

A *flow network* is a directed graph $G = (V, E)$ together with a *capacity* function $c : E \to R_{\geq 0}$, and two distinguished vertices: a *source* s and a *sink* t. Intuitively, each directed edge is a pipe of maximum throughput $c(e)$.

5.7.2 Flows

A (feasible) *flow* is a function $f : E \to R_{\geq 0}$ satisfying:

1. **Capacity constraints:** $0 \leq f(e) \leq c(e)$ for all $e \in E$.

2. **Flow conservation:** for every vertex $v \in V \setminus \{s, t\}$,

$$\sum_{(u,v)\in E} f(u, v) \;=\; \sum_{(v,w)\in E} f(v, w).$$

Interpretation. Conservation is imposed only on intermediate vertices: the source may have net outflow (injection), and the sink may have net inflow (absorption/drain).

5.7.3 Value of a flow

The *value* of a flow f is the net amount sent from s (equivalently, received at t):

$$|f| \;:=\; \sum_{(s,w)\in E} f(s, w) \;-\; \sum_{(u,s)\in E} f(u, s).$$

In many standard networks one assumes no incoming edges to s (or $f(u, s) = 0$), so $|f|$ is simply the total outflow from s.

5.7.4 cuts

An s–t *cut* is a partition $V = S \sqcup \bar{S}$ with $s \in S$ and $t \in \bar{S}$. Its *capacity* is

$$c(S, \bar{S}) \;:=\; \sum_{(u,v)\in E,\; u\in S,\; v\in\bar{S}} c(u, v),$$

i.e. the total capacity of edges crossing the boundary from the s-side to the t-side. (Edges oriented from $\bar{S}$ back to S do not directly limit transport from s to t.)

5.7.5 Max-flow / min-cut principle (intuition)

For any feasible flow f and any s–t cut $(S, \bar{S})$, one has the upper bound

$$|f| \le c(S, \bar{S}).$$

Intuition. Any amount transported from s to t must cross from S to $\bar{S}$ at least once, and the total throughput across that boundary is limited by the sum of capacities of the crossing edges.

A deep and central theorem states that this bound is tight at optimum:

$$\max\{|f| :\; f \text{ is a feasible flow}\} \;=\; \min\{c(S, \bar{S}) :\; (S, \bar{S}) \text{ is an } s\text{–}t \text{ cut}\}.$$

We will not prove this here; algorithmic proofs and constructions appear in later chapters.

5.8 Diffusion and Random Walks on Graphs (Motivation)

Although a full probabilistic treatment will appear later (see the probability chapter), it is already useful to view a graph as a *state space* on which a simple dynamics evolves. The guiding example is a random walk, which can be interpreted as a discrete diffusion process.

5.8.1 Unweighted random walk

Let $G = (V, E)$ be an undirected graph. The (simple) random walk moves from a current vertex v to a uniformly random neighbor $u \in \mathrm{Adj}(v)$. Equivalently, the one-step transition rule is

$$P(v \to u) = \begin{cases} \frac{1}{\deg(v)}, & \text{if } \{v, u\} \in E, \\ 0, & \text{otherwise.} \end{cases}$$

One can view P as an *update operator* describing how a "mass" or "belief" distribution over vertices changes in one step.

5.8.2 Weighted random walk as a biased local measure

If the graph is weighted with $w : E \to R_{\geq 0}$, a natural generalization is to bias moves according to weights. Define the *weighted degree* (also called the *strength*)

$$s(v) := \sum_{u:\{v,u\}\in E} w_{vu}.$$

Then the weighted random walk uses the transition rule

$$P(v \to u) = \begin{cases} \frac{w_{vu}}{s(v)}, & \text{if } \{v,u\} \in E, \\ 0, & \text{otherwise.} \end{cases}$$

Intuitively, weights replace the uniform choice on $\mathrm{Adj}(v)$ by a *local weighted measure*: larger w_{vu} means the edge is traversed more often.

5.8.3 Why "clusters" can appear (intuition)

Random walks do not create or destroy total mass; instead they *redistribute* it. Nevertheless, graphs with strong community structure often exhibit a *trapping* (metastability) phenomenon: if a vertex set $S \subseteq V$ has many internal connections but only weak/rare connections to $V \setminus S$, then once the walk enters S it tends to remain there for a long time before "leaking" out. In weighted graphs, the same intuition is captured by comparing total internal weight versus boundary (cut) weight.

5.8.4 Bridge to Laplacians and diffusion

Averaging behavior of random walks is closely related to discrete diffusion and to graph Laplacians. The next subsection introduces the graph Laplacian as the canonical operator governing smoothing, flow, and propagation on graphs.

5.9 Graph Laplacians and Diffusion

This final subsection introduces the graph Laplacian, the canonical operator for "smoothing" (diffusion) on graphs. It unifies ideas from flows/cuts and random walks by measuring local discrepancies and how they propagate along edges.

5.9.1 Adjacency, degree, and the Laplacian

Let $G = (V, E)$ be an undirected graph with $|V| = n$. Let A be its adjacency matrix ($A_{uv} = 1$ if $\{u, v\} \in E$, else 0), and let D be the degree matrix

$$D = \mathrm{diag}(\deg(1), \deg(2), \ldots, \deg(n)).$$

The *(combinatorial) graph Laplacian* is

$$L := D - A.$$

5.9.2 Laplacian as a discrete difference operator

Given a signal $x \in R^n$ on vertices (e.g. temperature, potential, mass, etc.), the v-th coordinate of Lx is

$$(Lx)_v \;=\; \deg(v)\, x_v \;-\; \sum_{u\sim v} x_u \;=\; \sum_{u\sim v} (x_v - x_u).$$

Thus Lx measures how different a vertex value is from its neighbors: if $x_v = x_u$ for all neighbors $u \sim v$, then $(Lx)_v = 0$.

5.9.3 Diffusion / smoothing dynamics (discrete-time)

A simple diffusion (smoothing) update on the graph can be written as

$$x^{(t+1)} \;=\; x^{(t)} - \alpha\, L x^{(t)},$$

where $\alpha > 0$ is a small step size. Intuitively, vertices higher than their neighbors decrease, and vertices lower than their neighbors increase, so the signal becomes progressively smoother.

5.9.4 Energy interpretation

The Laplacian induces a nonnegative quadratic form:

$$x^\top L x \;=\; \frac{1}{2} \sum_{\{u,v\}\in E} (x_u - x_v)^2 \;\geq\; 0.$$

This quantity measures total variation across edges; diffusion can be viewed as a process that reduces this edge-based discrepancy over time.

5.9.5 Relation to random walks (bridge to probability)

For an unweighted undirected graph, the random-walk transition matrix is

$$P = D^{-1} A,$$

which averages values over neighbors. One also defines the *random-walk Laplacian*

$$L_{\mathrm{rw}} = I - D^{-1} A = I - P,$$

making explicit the connection between diffusion, averaging, and one-step transitions. A full probabilistic treatment (Markov chains, stationarity, mixing, etc.) is deferred to the probability chapter.

5.9.6 Weighted graphs (brief remark)

For a weighted graph with weights $w_{uv} \geq 0$, one replaces A by the weighted adjacency matrix, and $\deg(v)$ by the weighted degree $s(v) = \sum_{u\sim v} w_{uv}$, yielding the weighted Laplacian $L = D - A$ in the same form.

Chapter 6

Counting and the Pigeonhole Principle

6.1 Goals of This Chapter

This chapter builds a small toolkit for turning qualitative statements (*existence*, *impossibility*, *collision*) into quantitative arguments. We will focus on:

- Addition rule (disjoint union counting),
- Multiplication rule (step-by-step counting),
- Permutations vs. combinations (order matters vs. does not matter),
- Pigeonhole principle (basic and strong forms),
- Two classic applications: modular prefix sums and graph theory,
- Handshaking lemma and degree-based structural consequences.

6.2 Two Fundamental Counting Rules

6.2.1 Addition Rule (Disjoint Cases)

If a set of outcomes is partitioned into *mutually exclusive* cases $A_1, \ldots, A_k$ (pairwise disjoint), then

$$|A_1 \cup \cdots \cup A_k| = |A_1| + \cdots + |A_k|.$$

Key diagnostic: "Either case 1 *or* case 2 *or* ..." and these cases cannot happen simultaneously.

6.2.2 Multiplication Rule (Sequential Steps)

If a process consists of sequential steps, and step i has c_i choices (given the previous steps), then the total number of outcomes is

$$c_1 \cdot c_2 \cdots c_r.$$

Key diagnostic: "First do this, *then* do that, *then* ..."

6.2.3 Minimal Inclusion–Exclusion (When Cases Overlap)

For two sets A, B (not necessarily disjoint),

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Interpretation: $|A| + |B|$ double-counts the overlap once, so we subtract it once.

6.3 Permutations and Combinations via Equivalence Classes

6.3.1 Permutation (Order/Role Matters)

If we assign distinct roles (or positions) to chosen people/objects, order matters. Example: from n people, choose and assign k distinct roles:

$$n \cdot (n-1) \cdots (n-k+1) = \frac{n!}{(n-k)!}.$$

6.3.2 Combination (Order Does Not Matter)

If only the chosen set matters (no roles/ordering), then we quotient by internal reorderings:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k!} = \frac{n!}{k!(n-k)!}.$$

Conceptual takeaway (from our discussion): a k-subset is an *equivalence class* of k-tuples under reordering, and each equivalence class has size $k!$.

6.4 Pigeonhole Principle

6.4.1 Basic Form

If N objects are placed into k boxes and $N > k$, then at least one box contains ≥ 2 objects.

6.4.2 Strong Form

If N objects are placed into k boxes, then some box contains at least

$$\left\lceil \frac{N}{k} \right\rceil$$

objects.

6.4.3 Hash Collisions as Pigeonholes

A hash function $h : \mathcal{X} \to \{0, 1, \ldots, m-1\}$ maps inputs to m possible outputs (boxes). Selecting $m+1$ distinct inputs guarantees a collision: some output value contains at least two inputs.

6.5 A Classic Modular Application: Prefix Sums

Let $a_1, \ldots, a_n$ be integers and define prefix sums $S_k = a_1 + \cdots + a_k$.

[Prefix-sum divisibility] There exists a nonempty consecutive block $a_{i+1} + \cdots + a_j$ whose sum is divisible by n.

[Proof sketch (our two-case pigeonhole argument)] Consider residues $S_1 \bmod n, \ldots, S_n \bmod n$.

- If some $S_k \equiv 0 \pmod{n}$, then $a_1 + \cdots + a_k$ is divisible by n.
- Otherwise all S_k lie in $\{1, 2, \ldots, n-1\}$ (only $n-1$ boxes), but we have n residues (objects), so two are equal: $S_i \equiv S_j \pmod{n}$ with $i < j$. Then $S_j - S_i = a_{i+1} + \cdots + a_j$ is divisible by n.

6.6 Graph Theory Applications

6.6.1 Degree Pigeonhole

Let $G = (V, E)$ be a simple undirected graph with $|V| = n$. Each vertex degree lies in $\{0, 1, \ldots, n-1\}$.

In any simple undirected graph on n vertices, there exist two vertices with the same degree.

[Proof sketch (our "forbidden endpoints" argument)] It is impossible to have both a vertex of degree 0 and a vertex of degree $n-1$: if $\deg(v) = n-1$, then v is adjacent to every other vertex, so no vertex can have degree 0. Hence, the set of degrees realized by the graph is contained in either $\{0, 1, \ldots, n-2\}$ or $\{1, 2, \ldots, n-1\}$, which has at most $n-1$ values (boxes). With n vertices (objects), pigeonhole implies two vertices share a degree.

6.7 Handshaking Lemma

[Handshaking lemma] For any finite undirected graph $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2|E|.$$

[Proof idea] Each edge contributes 1 to the degree of each of its two endpoints, hence contributes 2 in total. Summing over all edges gives $2|E|$.

6.7.1 Corollary: Even Number of Odd-Degree Vertices

The number of vertices of odd degree in an undirected graph is even.

[One-line parity argument] The left-hand side $\sum \deg(v) = 2|E|$ is even. The sum of even degrees is even, so the sum of odd degrees must be even, which happens iff there are an even number of odd terms.

6.7.2 A Connectivity Criterion from Minimum Degree

Let $\delta(G)$ denote the minimum degree of G.

If $\delta(G) \geq \frac{n}{2}$, then G is connected.

[Proof sketch (component-size contradiction)] Assume G is disconnected. Then some connected component has size $k \leq n/2$. Any vertex in that component has degree at most $k - 1 \leq n/2 - 1$, contradicting $\delta(G) \geq n/2$.

6.7.3 Important Correction We Noticed

Because degrees are integers, the condition $\delta(G) \geq \frac{n-1}{2}$ actually still forces connectivity:

- If n is even, $\frac{n-1}{2}$ is not an integer, so $\deg(v) \geq \frac{n-1}{2}$ implies $\deg(v) \geq \frac{n}{2}$.
- If n is odd, a disconnected graph would have a component of size $k \leq \frac{n-1}{2}$, forcing some degree $\leq k - 1 \leq \frac{n-3}{2}$, contradicting $\delta(G) \geq \frac{n-1}{2}$.

6.8 Study Notes: How We Decided "Add or Multiply"

- **Multiply** when you have sequential steps ("first..., then..., then...").
- **Add** when you split into mutually exclusive cases ("either..., or..., or...").
- **Inclusion–exclusion** when cases overlap ("or" but not disjoint).

- **Combination** is “permutation modulo internal reordering” (equivalence classes).
- **Pigeonhole** is “even the sparsest distribution forces collision”.

6.9 Mini Glossary

Permutation Choosing with order/roles.

Combination Choosing without order; quotient by reorderings.

Pigeonhole principle $N > k$ forces a repeated box; strong form uses $\lceil N/k \rceil$.

Collision Two different inputs mapping to the same output (hashing viewpoint).

Degree Number of neighbors of a vertex in an undirected graph.

Minimum degree $\delta(G)$ The smallest vertex degree in G.

Chapter 7

Probability for TCS

Chapter 8

Asymptotic Notation and Growth Rates

Chapter 9

Invariants and Potential Functions

Chapter 10

Reductions: The Core Technique

Part II

Formal Languages and Automata

Chapter 11

Alphabets, Languages, and Operations

Chapter 12

Deterministic Finite Automata (DFA)

Chapter 13

NFA and the Subset Construction

Chapter 14

Regular Expressions and Thompson Construction

Chapter 15

Minimization and Myhill–Nerode

Chapter 16

Closure Properties and Decidability for Regular Languages

Chapter 17

Context-Free Grammars (CFG)

Chapter 18

Pushdown Automata (PDA)

Chapter 19

Pumping Lemmas and Non-Regularity Proofs

Chapter 20

Parsing Algorithms: CYK / Earley Overview

Part III

Computability

Chapter 21

Turing Machines as a Model of Computation

Chapter 22

Decidable vs. Recognizable Languages

Chapter 23

The Halting Problem and Diagonalization

Chapter 24

Many-One Reductions and Undecidability Proofs

Chapter 25

Rice's Theorem

Chapter 26

Post Correspondence Problem (PCP)

Chapter 27

Kleene's Recursion Theorem (Self-Reference)

Chapter 28

Lambda Calculus Perspective

Chapter 29

Interfaces with Logic (A Preview)

Chapter 30

Oracles and Relative Computability

Part IV

Computational Complexity

Chapter 31

Time and Space Complexity Classes

Chapter 32

Hierarchy Theorems

Chapter 33

P, NP, and the Verification View

Chapter 34

SAT and the Cook–Levin Theorem

Chapter 35

NP-Completeness Toolkit and Canonical Problems

Chapter 36

coNP and Complement Classes

Chapter 37

The Polynomial Hierarchy

Chapter 38

Randomized Complexity (RP, BPP) and Amplification

Chapter 39

Circuit Complexity: An Entry Point

Chapter 40

Communication Complexity: An Entry Point

Part V

Algorithms and Data Structures

Chapter 41

Correctness Proofs and Complexity Analysis

Chapter 42

Divide and Conquer

Chapter 43

Greedy Algorithms and Exchange Arguments

Chapter 44

Dynamic Programming and State Design

Chapter 45

Hashing and Randomized Data Structures

Chapter 46

Core Graph Algorithms

Chapter 47

Max Flow / Min Cut and Duality Intuition

Chapter 48

Matchings and Bipartite Graphs

Chapter 49

Amortized Analysis

Chapter 50

Approximation Algorithms for NP-Hard Problems

Part VI

Logic, Semantics, Types, and Verification

Chapter 51

Propositional Logic and SAT Basics

Chapter 52

First-Order Logic: Structures and Models

Chapter 53

Soundness, Completeness, and Compactness (Intuition)

Chapter 54

Hoare Logic and Program Correctness

Chapter 55

Operational Semantics (Small-Step / Big-Step)

Chapter 56

Denotational Semantics and Fixed Points

Chapter 57

Type Systems: Progress and Preservation

Chapter 58

Curry–Howard Correspondence

Chapter 59

Category-Theoretic Interfaces (CCC, Adjunctions, Monads)

Chapter 60

Model Checking and Temporal Logics (LTL/CTL)

Part VII

Advanced Topics and Research Readiness

Chapter 61

Cryptography Foundations and Security Notions

Chapter 62

Pseudorandomness and Complexity Connections

Chapter 63

Distributed Consensus and Impossibility Intuition

Chapter 64

Concurrency and Consistency Models

Chapter 65

PAC Learning and VC Dimension

Chapter 66

Online Learning and Regret Bounds

Chapter 67

Algorithmic Information Theory (Kolmogorov Complexity)

Chapter 68

Game Theory and Algorithmic Mechanism Design

Chapter 69

Optional: Dependent Types / HoTT / Higher-Categorical Semantics

Chapter 70

How to Read Papers and Write Proofs (Research Workflow)