Introduction to the Theory of Computation

Mathematical Foundations and Preliminaries

Theory of Computation Slide Sets

Table of Contents

Learning Outcomes and Notation

By the end of this set, you should be able to:

- use sets, functions, relations, and quantifiers precisely;
- distinguish walks, cycles, rooted trees, depth, and height;
- manipulate strings and languages, including their empty cases;
- distinguish a problem, an instance, a solution, and an algorithm, and encode a finite instance as a string;
- distinguish generating a language from recognizing or deciding it;
- view computation as a string-to-string transformation and explain why different models help us study it;
- choose and write a direct, contrapositive, contradiction, or induction proof.

Learning Outcomes and Notation

By the end of this set, you should be able to:

- use sets, functions, relations, and quantifiers precisely;
- distinguish walks, cycles, rooted trees, depth, and height;
- manipulate strings and languages, including their empty cases;
- distinguish a problem, an instance, a solution, and an algorithm, and encode a finite instance as a string;
- distinguish generating a language from recognizing or deciding it;
- view computation as a string-to-string transformation and explain why different models help us study it;
- choose and write a direct, contrapositive, contradiction, or induction proof.

Conventions used throughout these slide sets

$\mathbb{N} = \{0, 1, 2, \ldots\}$; $\mathbb{Z}$ denotes the integers. $|S|$ is a set's cardinality, whereas $|w|$ is a string's length. An alphabet is finite and nonempty. A word means a finite string. Unless stated otherwise, graphs and trees here are finite.

Definitions and proof patterns are the prerequisites; the appendix supplies worked answers and further examples for independent practice.

Part 1: The Big Question

The Fundamental Question

What are the fundamental capabilities and limitations of computers?

Part 1: The Big Question

The Fundamental Question

What are the fundamental capabilities and limitations of computers?

To answer this rigorously, theoretical computer science rests on three pillars:

Part 1: The Big Question

The Fundamental Question

What are the fundamental capabilities and limitations of computers?

To answer this rigorously, theoretical computer science rests on three pillars:

- **Automata Theory:** How do we model computation? Finite automata, pushdown automata, and Turing machines provide precise models with different memory capabilities.

Part 1: The Big Question

The Fundamental Question

What are the fundamental capabilities and limitations of computers?

To answer this rigorously, theoretical computer science rests on three pillars:

- **Automata Theory:** How do we model computation? Finite automata, pushdown automata, and Turing machines provide precise models with different memory capabilities.
- **Computability Theory:** What is solvable vs. unsolvable? We determine if an algorithm can exist for a given problem (e.g., the Halting Problem).

Part 1: The Big Question

The Fundamental Question

What are the fundamental capabilities and limitations of computers?

To answer this rigorously, theoretical computer science rests on three pillars:

- **Automata Theory:** How do we model computation? Finite automata, pushdown automata, and Turing machines provide precise models with different memory capabilities.
- **Computability Theory:** What is solvable vs. unsolvable? We determine if an algorithm can exist for a given problem (e.g., the Halting Problem).
- **Complexity Theory:** What is computationally easy vs. hard? We classify problems based on resource requirements like time and space (e.g., P vs. NP).

Part 2: Sets and Basics

A **set** is a collection of distinct objects, called its **elements**. Order and repeated listings do not matter: $\{a, b, a\} = \{b, a\}$.

Part 2: Sets and Basics

A **set** is a collection of distinct objects, called its **elements**. Order and repeated listings do not matter: $\{a, b, a\} = \{b, a\}$.

- **Membership:** $x \in S$ means x is an element of S. $x \notin S$ means it is not.
- **Subset:** $A \subseteq B$ means every element of A is also in B.
- **Empty Set:** $\emptyset$ is the set with no elements.
- **Set-builder notation:** $\{n \in \mathbb{N} : n \text{ is even}\} = \{0, 2, 4, \ldots\}$.

Part 2: Sets and Basics

A **set** is a collection of distinct objects, called its **elements**. Order and repeated listings do not matter: $\{a, b, a\} = \{b, a\}$.

- **Membership:** $x \in S$ means x is an element of S. $x \notin S$ means it is not.
- **Subset:** $A \subseteq B$ means every element of A is also in B.
- **Empty Set:** $\emptyset$ is the set with no elements.
- **Set-builder notation:** $\{n \in \mathbb{N} : n \text{ is even}\} = \{0, 2, 4, \ldots\}$.

Membership is not containment

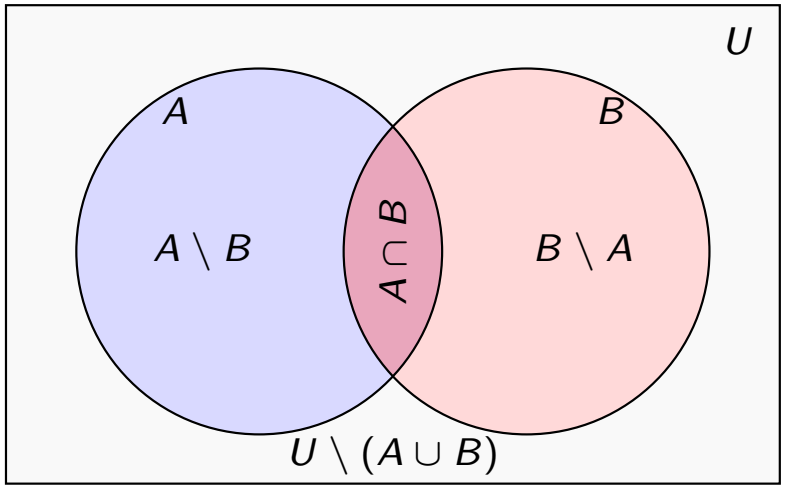
For $S = \{a, b\}$, $a \in S$, but $\{a\} \subseteq S$. Also $\emptyset \subseteq S$, while $\emptyset \notin S$. The set $\{\emptyset\}$ has one element, so it is not empty.

To prove $A = B$, prove both $A \subseteq B$ and $B \subseteq A$.

Set Operations and Venn Diagrams

Let $A, B \subseteq U$, where U is the fixed universe.

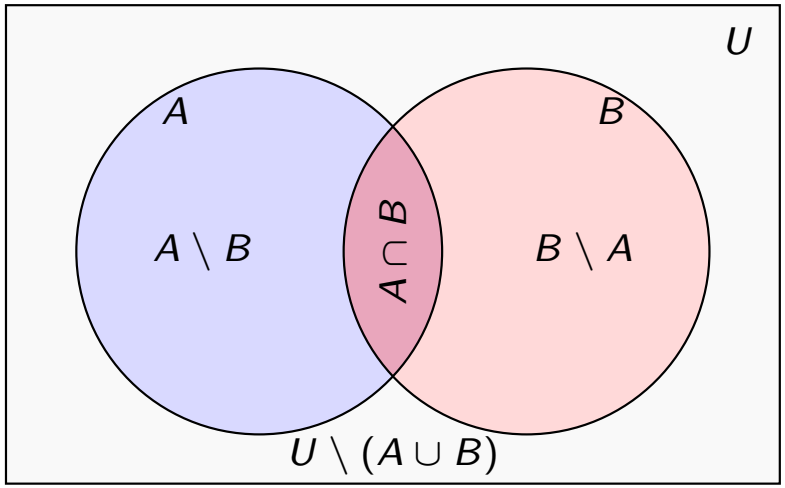
- **Union** ($A \cup B$): Elements in A or B or both.
- **Intersection** ($A \cap B$): Elements in both.
- **Difference** ($A - B$ or $A \setminus B$): Elements in A but not B.
- **Complement**: $\overline{A} = U \setminus A$.



Set Operations and Venn Diagrams

Let $A, B \subseteq U$, where U is the fixed universe.

- **Union** ($A \cup B$): Elements in A or B or both.
- **Intersection** ($A \cap B$): Elements in both.
- **Difference** ($A - B$ or $A \setminus B$): Elements in A but not B.
- **Complement**: $\overline{A} = U \setminus A$.



De Morgan's laws: negate the membership condition

$\overline{A \cup B} = \overline{A} \cap \overline{B}, \qquad \overline{A \cap B} = \overline{A} \cup \overline{B}.$

All complements must use the same universe.

Power Sets and Cartesian Products

Power Set

The **power set** of S, denoted 2^S (or $\mathcal{P}(S)$), is the set of all subsets of S.
Example: If $S = \{a, b, c\}$, then:

$$2^S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

For finite S, $|2^S| = 2^{|S|}$: independently include or exclude each element. In particular, $2^\emptyset = \{\emptyset\}$ has size 1.

Power Sets and Cartesian Products

Power Set

The **power set** of S, denoted 2^S (or $\mathcal{P}(S)$), is the set of all subsets of S.
Example: If $S = \{a, b, c\}$, then:

$$2^S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

For finite S, $|2^S| = 2^{|S|}$: independently include or exclude each element. In particular, $2^\emptyset = \{\emptyset\}$ has size 1.

Sequences, Tuples, and Cartesian Products

A **sequence** is an ordered list; repetitions are allowed. A finite sequence is a **tuple**; $(a, b) = (c, d)$ iff $a = c$ and $b = d$.
Cartesian Product $(A \times B)$: The set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.
Example: $A = \{1, 2\}$, $B = \{x, y, z\}$

$$A \times B = \{(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)\}$$

For finite sets, $|A \times B| = |A||B|$. If either factor is empty, the product is empty.

Part 3: Functions

A **function** (or mapping) $f : D \to C$ assigns exactly one element in the **codomain** C to each element in the **domain** D.

Part 3: Functions

A **function** (or mapping) $f : D \to C$ assigns exactly one element in the **codomain** C to each element in the **domain** D.

- **Range:** The subset of the codomain containing all actual outputs, defined as $f(D) = \{f(x) \mid x \in D\} \subseteq C$.

Part 3: Functions

A **function** (or mapping) $f : D \to C$ assigns exactly one element in the **codomain** C to each element in the **domain** D.

- **Range:** The subset of the codomain containing all actual outputs, defined as $f(D) = \{f(x) \mid x \in D\} \subseteq C$.
- **Total Function:** Defined for *every* element in the domain D.

Part 3: Functions

A **function** (or mapping) $f : D \to C$ assigns exactly one element in the **codomain** C to each element in the **domain** D.

- **Range:** The subset of the codomain containing all actual outputs, defined as $f(D) = \{f(x) \mid x \in D\} \subseteq C$.
- **Total Function:** Defined for *every* element in the domain D.
- **Partial Function:** $f : D \rightharpoonup C$ assigns an output to each element of some subset of D; other inputs have no defined output. Ordinary $f : D \to C$ means a total function here.

Part 3: Functions

A **function** (or mapping) $f : D \to C$ assigns exactly one element in the **codomain** C to each element in the **domain** D.

- **Range:** The subset of the codomain containing all actual outputs, defined as $f(D) = \{f(x) \mid x \in D\} \subseteq C$.
- **Total Function:** Defined for *every* element in the domain D.
- **Partial Function:** $f : D \rightharpoonup C$ assigns an output to each element of some subset of D; other inputs have no defined output. Ordinary $f : D \to C$ means a total function here.

Example

For $f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = |x|$, the range is $\mathbb{N} = \{0, 1, 2, \ldots\}$, not all of the codomain $\mathbb{Z}$.

Bridge to deterministic finite automata (DFAs): their transition function is total. A missing arrow must be handled explicitly, usually by a rejecting sink state. A nondeterministic finite automaton (NFA) can instead return the empty *set* of successor states.

Part 3: Properties of Functions

Functions are classified by how elements in the domain map to the codomain:

Part 3: Properties of Functions

Functions are classified by how elements in the domain map to the codomain:

- **Injective (One-to-One):** Distinct inputs always map to distinct outputs. If $f(a) = f(b)$, then $a = b$. No two arrows point to the same target.

Part 3: Properties of Functions

Functions are classified by how elements in the domain map to the codomain:

- **Injective (One-to-One):** Distinct inputs always map to distinct outputs. If $f(a) = f(b)$, then $a = b$. No two arrows point to the same target.
- **Surjective (Onto):** Every element in the codomain is mapped to by at least one element in the domain. The **range equals the codomain**.

Part 3: Properties of Functions

Functions are classified by how elements in the domain map to the codomain:

- **Injective (One-to-One):** Distinct inputs always map to distinct outputs. If $f(a) = f(b)$, then $a = b$. No two arrows point to the same target.
- **Surjective (Onto):** Every element in the codomain is mapped to by at least one element in the domain. The **range equals the codomain**.
- **Bijective (One-to-One Correspondence):** Both injective and surjective. Every input has a unique output, and every output has a unique input. Essential for defining **inverse functions**.

Part 3: Properties of Functions

Functions are classified by how elements in the domain map to the codomain:

- **Injective (One-to-One):** Distinct inputs always map to distinct outputs. If $f(a) = f(b)$, then $a = b$. No two arrows point to the same target.
- **Surjective (Onto):** Every element in the codomain is mapped to by at least one element in the domain. The **range equals the codomain**.
- **Bijective (One-to-One Correspondence):** Both injective and surjective. Every input has a unique output, and every output has a unique input. Essential for defining **inverse functions**.

Quick Check

- $f : \mathbb{R} \to \mathbb{R}, f(x) = x^3$ is **bijective**.
- $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2$ is **neither**: $f(-1) = f(1)$, and no input maps to -1.

Changing domain and codomain matters: $x \mapsto x^2$ from $[0, \infty)$ to $[0, \infty)$ is bijective, with inverse $x \mapsto \sqrt{x}$. For $f : D \to C$ and $g : C \to E$, composition is read right to left: $(g \circ f)(x) = g(f(x))$.

Relations and Equivalence

A k-ary **relation** is a subset of a Cartesian product of k sets. A binary relation R on A is a subset of $A \times A$. Write aRb for $(a, b) \in R$. A relation need not be a function.

Relations and Equivalence

A k-ary **relation** is a subset of a Cartesian product of k sets. A binary relation R on A is a subset of $A \times A$. Write aRb for $(a, b) \in R$. A relation need not be a function.

Equivalence Relation

A binary relation R is an **equivalence relation** if it satisfies three conditions:

1. **Reflexivity:** xRx for every x.
2. **Symmetry:** If xRy, then yRx.
3. **Transitivity:** If xRy and yRz, then xRz.

Relations and Equivalence

A k-ary **relation** is a subset of a Cartesian product of k sets. A binary relation R on A is a subset of $A \times A$. Write aRb for $(a, b) \in R$. A relation need not be a function.

Equivalence Relation

A binary relation R is an **equivalence relation** if it satisfies three conditions:

1. **Reflexivity:** xRx for every x.
2. **Symmetry:** If xRy, then yRx.
3. **Transitivity:** If xRy and yRz, then xRz.

Equivalence Classes

Define $[a]_R = \{b \in A : aRb\}$. These classes cover A; any two are equal or disjoint. Thus the distinct classes form a partition. State minimization later groups states with the same future behavior.

Worked Equivalence Relation: Remainders Modulo Three

On $\mathbb{Z}$, define $x \sim y$ iff 3 divides $x - y$. Equivalently, x and y have the same remainder modulo 3.

- **Reflexive:** $x - x = 0$ is divisible by 3.
- **Symmetric:** if $x - y = 3k$, then $y - x = 3(-k)$.
- **Transitive:** if $x - y = 3k$ and $y - z = 3\ell$, then $x - z = 3(k + \ell)$.

Worked Equivalence Relation: Remainders Modulo Three

On $\mathbb{Z}$, define $x \sim y$ iff 3 divides $x - y$. Equivalently, x and y have the same remainder modulo 3.

- **Reflexive:** $x - x = 0$ is divisible by 3.
- **Symmetric:** if $x - y = 3k$, then $y - x = 3(-k)$.
- **Transitive:** if $x - y = 3k$ and $y - z = 3\ell$, then $x - z = 3(k + \ell)$.

There are exactly three classes:

$$[0] = \{\ldots, -6, -3, 0, 3, 6, \ldots\},$$
$$[1] = \{\ldots, -5, -2, 1, 4, 7, \ldots\},$$
$$[2] = \{\ldots, -4, -1, 2, 5, 8, \ldots\}.$$

Worked Equivalence Relation: Remainders Modulo Three

On $\mathbb{Z}$, define $x \sim y$ iff 3 divides $x - y$. Equivalently, x and y have the same remainder modulo 3.

- **Reflexive:** $x - x = 0$ is divisible by 3.
- **Symmetric:** if $x - y = 3k$, then $y - x = 3(-k)$.
- **Transitive:** if $x - y = 3k$ and $y - z = 3\ell$, then $x - z = 3(k + \ell)$.

There are exactly three classes:

$$[0] = \{\ldots, -6, -3, 0, 3, 6, \ldots\},$$
$$[1] = \{\ldots, -5, -2, 1, 4, 7, \ldots\},$$
$$[2] = \{\ldots, -4, -1, 2, 5, 8, \ldots\}.$$

For example, $[4] = [1]$, not a fourth class. A machine checking a count modulo three can remember its class instead of the entire count.

Boolean Logic

Boolean logic uses values TRUE (1) and FALSE (0). We build complex expressions using boolean operations:

Boolean Logic

Boolean logic uses values TRUE (1) and FALSE (0). We build complex expressions using boolean operations:

- **NOT ($\neg$):** $\neg 0 = 1$ and $\neg 1 = 0$.
- **AND ($\wedge$):** $1 \wedge 1 = 1$, all other combinations are 0.
- **OR ($\vee$):** $0 \vee 0 = 0$, all other combinations are 1.

Boolean Logic

Boolean logic uses values TRUE (1) and FALSE (0). We build complex expressions using boolean operations:

- **NOT ($\neg$):** $\neg 0 = 1$ and $\neg 1 = 0$.
- **AND ($\wedge$):** $1 \wedge 1 = 1$, all other combinations are 0.
- **OR ($\vee$):** $0 \vee 0 = 0$, all other combinations are 1.

Precedence

Just like arithmetic ($\times$ before $+$), boolean operations have an evaluation order:

NOT precedes AND, and AND precedes OR.

Example: $P \vee Q \wedge \neg R$ is evaluated as $P \vee (Q \wedge (\neg R))$.

Implication, Equivalence, and Counterexamples

Implication is not a claim of causation

$P \Rightarrow Q$ means "if P, then Q" and is equivalent to $\neg P \vee Q$. It is false only when P is true and Q is false. $P \Leftrightarrow Q$ requires both $P \Rightarrow Q$ and $Q \Rightarrow P$.

Implication, Equivalence, and Counterexamples

Implication is not a claim of causation

$P \Rightarrow Q$ means "if P, then Q" and is equivalent to $\neg P \vee Q$. It is false only when P is true and Q is false. $P \Leftrightarrow Q$ requires both $P \Rightarrow Q$ and $Q \Rightarrow P$.

- The **contrapositive** $\neg Q \Rightarrow \neg P$ is equivalent to $P \Rightarrow Q$.
- The **converse** $Q \Rightarrow P$ need not follow. Divisibility by 4 implies evenness, but 6 refutes the converse.
- To refute a universal implication, find one case where the hypothesis holds and the conclusion fails. Positive examples alone do not prove a universal claim.

Implication, Equivalence, and Counterexamples

Implication is not a claim of causation

$P \Rightarrow Q$ means "if P, then Q" and is equivalent to $\neg P \vee Q$. It is false only when P is true and Q is false. $P \Leftrightarrow Q$ requires both $P \Rightarrow Q$ and $Q \Rightarrow P$.

- The **contrapositive** $\neg Q \Rightarrow \neg P$ is equivalent to $P \Rightarrow Q$.
- The **converse** $Q \Rightarrow P$ need not follow. Divisibility by 4 implies evenness, but 6 refutes the converse.
- To refute a universal implication, find one case where the hypothesis holds and the conclusion fails. Positive examples alone do not prove a universal claim.

Vacuous truth: if no object satisfies the hypothesis, there is no counterexample. This is why $\emptyset \subseteq A$ for every set A.

Quantifiers: Who May Depend on Whom?

$\forall x \in D$ means "for every x in D"; $\exists x \in D$ means "for at least one x in D." Always state the domain.

Changing the order changes the claim

$\forall n \in \mathbb{N}\, \exists m \in \mathbb{N} : m > n$ is true; choose $m = n + 1$, depending on n.

$\exists m \in \mathbb{N}\, \forall n \in \mathbb{N} : m > n$ is false; the proposed fixed m fails when $n = m$.

Quantifiers: Who May Depend on Whom?

$\forall x \in D$ means "for every x in D"; $\exists x \in D$ means "for at least one x in D." Always state the domain.

Changing the order changes the claim

$\forall n \in \mathbb{N}\, \exists m \in \mathbb{N} : m > n$ is true; choose $m = n + 1$, depending on n.

$\exists m \in \mathbb{N}\, \forall n \in \mathbb{N} : m > n$ is false; the proposed fixed m fails when $n = m$.

Negation swaps the quantifier and negates its body

$$\neg(\forall x \in D\ P(x)) \iff \exists x \in D\ \neg P(x),$$
$$\neg(\exists x \in D\ P(x)) \iff \forall x \in D\ \neg P(x).$$

Over an empty domain, a universal statement is true and an existential statement is false. Quantifier order will be central to pumping arguments.

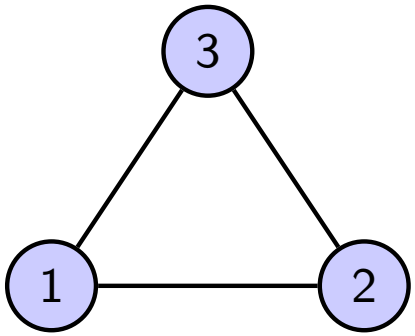
Part 4: Graph Theory Basics

A **graph** $G = (V, E)$ consists of a set of vertices (nodes) V and a set of edges E.

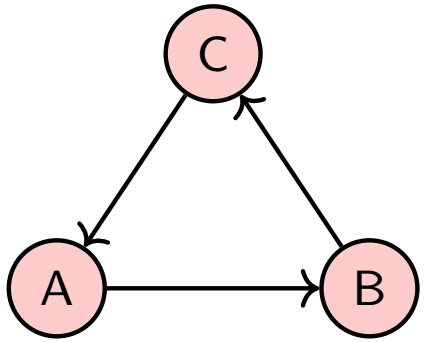
Part 4: Graph Theory Basics

A **graph** $G = (V, E)$ consists of a set of vertices (nodes) V and a set of edges E.

Simple undirected graph: E contains unordered pairs $\{u, v\}$ with $u \neq v$; no loops or parallel edges.



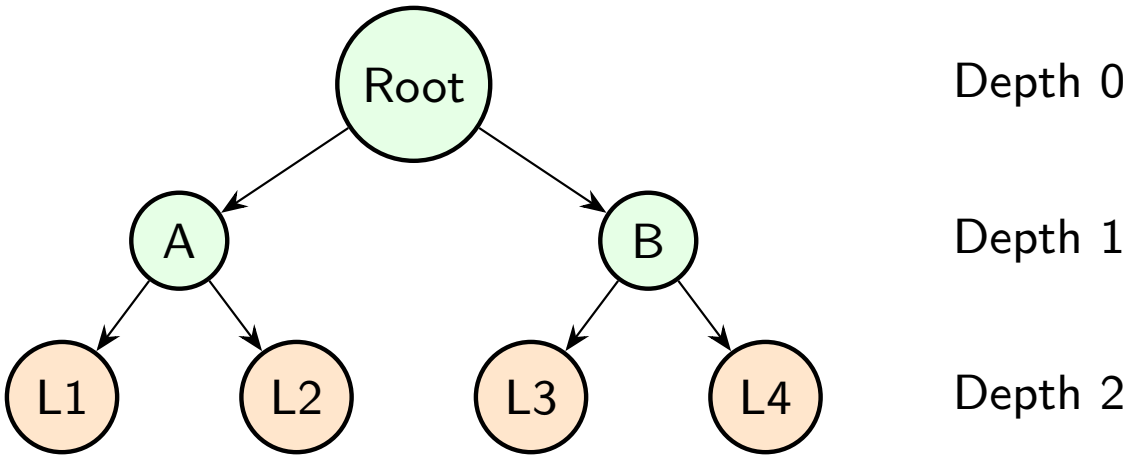
Directed Graph (Digraph): Edges have direction. E consists of ordered pairs (u, v).



A **walk** follows adjacent edges (respecting direction in a digraph); its length is its number of edges. A **simple path** repeats no vertex. A **cycle** is a nonempty closed walk with no repeated edge and no repeated vertex other than its initial/final vertex. Automata diagrams may have loops and labels; their computations are walks, often called paths in automata texts.

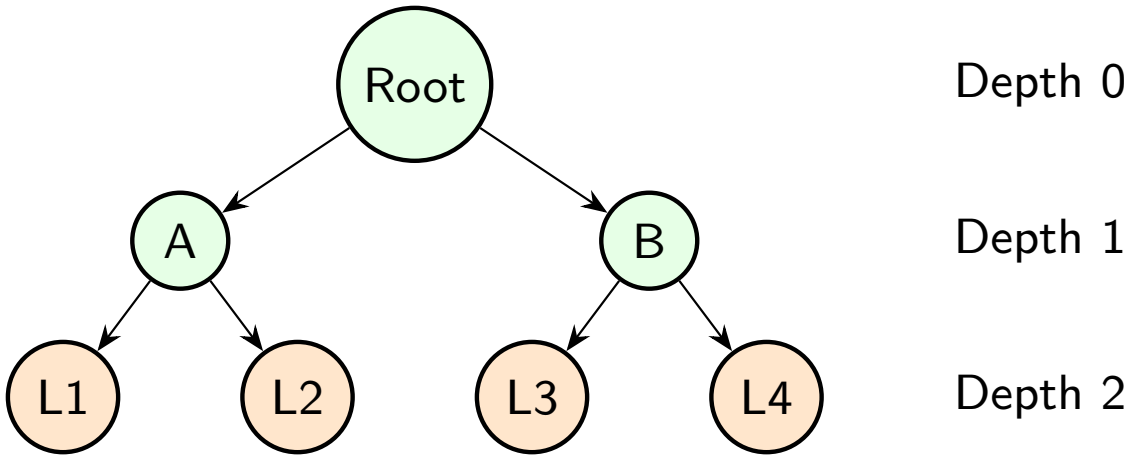
Tree Structures

An undirected **tree** is a connected graph with no cycles. Choose a **root** and orient each edge away from it: the root has no parent, every other node has exactly one parent, and nodes with no children are **leaves**. A single root is also a tree and a leaf.



Tree Structures

An undirected **tree** is a connected graph with no cycles. Choose a **root** and orient each edge away from it: the root has no parent, every other node has exactly one parent, and nodes with no children are **leaves**. A single root is also a tree and a leaf.



Depth is the number of edges from the root to a node. **Height** of a node is the longest downward path to a leaf; the tree's height is its root's height (here 2). A **binary tree** has at most two children per node. In an **ordered tree**, the left-to-right child order matters.

- **Alphabet (Σ):** A finite, nonempty set of symbols.
  Example: $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.

- **Alphabet (Σ):** A finite, nonempty set of symbols.
  Example: $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.
- **String:** A finite sequence of symbols from Σ.
- **Length ($|w|$):** The number of symbols in string w. If $w = 1011$, $|w| = 4$.

- **Alphabet (Σ):** A finite, nonempty set of symbols.
  Example: $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.
- **String:** A finite sequence of symbols from Σ.
- **Length ($|w|$):** The number of symbols in string w. If $w = 1011$, $|w| = 4$.
- **Empty String (ε or λ):** The string of length zero. $|\varepsilon| = 0$.
- **Concatenation:** Appending strings. If $x = 01$ and $y = 11$, $xy = 0111$.
- **Reversal (w^R):** The string written backwards. $(011)^R = 110$.

The symbol ε denotes a word with no symbols; it is not an extra input symbol in Σ.

String Operations and Positions

Concatenation is associative: $(xy)z = x(yz)$, with identity $x\varepsilon = \varepsilon x = x$. It is not generally commutative: $01 \neq 10$.

$$|xy| = |x| + |y|, \qquad (xy)^R = y^R x^R, \qquad (w^R)^R = w.$$

Define $w^0 = \varepsilon$ and $w^{n+1} = w^n w$; thus $(01)^3 = 010101$.

String Operations and Positions

Concatenation is associative: $(xy)z = x(yz)$, with identity $x\varepsilon = \varepsilon x = x$. It is not generally commutative: $01 \neq 10$.

$$|xy| = |x| + |y|, \qquad (xy)^R = y^R x^R, \qquad (w^R)^R = w.$$

Define $w^0 = \varepsilon$ and $w^{n+1} = w^n w$; thus $(01)^3 = 010101$.

Prefix, suffix, substring, subsequence

u is a **prefix** of w if $w = uv$ for some v; a **suffix** if $w = vu$; a **substring** if $w = xuy$. A **subsequence** is obtained by deleting zero or more symbols, without changing the order of those retained.

In $w = 0101$, 010 is a prefix, 101 a suffix, and 10 a substring. The word 00 is a subsequence but not a substring.

String Operations and Positions

Concatenation is associative: $(xy)z = x(yz)$, with identity $x\varepsilon = \varepsilon x = x$. It is not generally commutative: $01 \neq 10$.

$$|xy| = |x| + |y|, \qquad (xy)^R = y^R x^R, \qquad (w^R)^R = w.$$

Define $w^0 = \varepsilon$ and $w^{n+1} = w^n w$; thus $(01)^3 = 010101$.

Prefix, suffix, substring, subsequence

u is a **prefix** of w if $w = uv$ for some v; a **suffix** if $w = vu$; a **substring** if $w = xuy$. A **subsequence** is obtained by deleting zero or more symbols, without changing the order of those retained.

In $w = 0101$, 010 is a prefix, 101 a suffix, and 10 a substring. The word 00 is a subsequence but not a substring.

The empty word is a prefix, suffix, substring, and subsequence of every word. A substring must be contiguous; a subsequence need not be.

Languages and the Kleene Star

Kleene Star (Σ^*)

Let Σ^n be the set of strings of length n, with $\Sigma^0 = \{\varepsilon\}$. Then $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ and $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n = \Sigma^* \setminus \{\varepsilon\}$.
Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Languages and the Kleene Star

Kleene Star (Σ^*)

Let Σ^n be the set of strings of length n, with $\Sigma^0 = \{\varepsilon\}$. Then $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ and $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n = \Sigma^* \setminus \{\varepsilon\}$.
Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Formal Language

A **language** L is simply a subset of Σ^*.

$$L \subseteq \Sigma^*$$

A language can be finite or infinite. Even the empty set $\emptyset$ is a language.

Languages and the Kleene Star

Kleene Star (Σ^*)

Let Σ^n be the set of strings of length n, with $\Sigma^0 = \{\varepsilon\}$. Then $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ and $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n = \Sigma^* \setminus \{\varepsilon\}$.
Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Formal Language

A **language** L is simply a subset of Σ^*.

$$L \subseteq \Sigma^*$$

A language can be finite or infinite. Even the empty set $\emptyset$ is a language.

Keep the empty cases separate

$\emptyset$ is a language with zero words; $\{\varepsilon\}$ is a language with one word; ε is a word of length zero. Thus $|\emptyset| = 0$, $|\{\varepsilon\}| = 1$, and $|\varepsilon| = 0$. A language is a set: listing a word twice does not create another element.

Operations on Languages

For $L, K \subseteq \Sigma^*$, union and intersection are ordinary set operations; complement means $\Sigma^* \setminus L$ for the fixed alphabet.

Concatenation and repetition

$$LK = \{xy : x \in L,\ y \in K\}, \qquad L^R = \{w^R : w \in L\},$$

$$L^0 = \{\varepsilon\}, \quad L^{n+1} = L^n L, \quad L^* = \bigcup_{n \geq 0} L^n, \quad L^+ = \bigcup_{n \geq 1} L^n.$$

Each factor may be a *different* word of L.

Operations on Languages

For $L, K \subseteq \Sigma^*$, union and intersection are ordinary set operations; complement means $\Sigma^* \setminus L$ for the fixed alphabet.

Concatenation and repetition

$$LK = \{xy : x \in L,\ y \in K\}, \qquad L^R = \{w^R : w \in L\},$$

$$L^0 = \{\varepsilon\}, \quad L^{n+1} = L^n L, \quad L^* = \bigcup_{n \geq 0} L^n, \quad L^+ = \bigcup_{n \geq 1} L^n.$$

Each factor may be a *different* word of L.

For $L = \{a, bb\}$, $L^2 = \{aa, abb, bba, bbbb\}$.

$$L\{\varepsilon\} = \{\varepsilon\}L = L, \qquad L\emptyset = \emptyset L = \emptyset, \qquad \emptyset^* = \{\varepsilon\}.$$

Operations on Languages

For $L, K \subseteq \Sigma^*$, union and intersection are ordinary set operations; complement means $\Sigma^* \setminus L$ for the fixed alphabet.

Concatenation and repetition

$$LK = \{xy : x \in L,\ y \in K\}, \qquad L^R = \{w^R : w \in L\},$$

$$L^0 = \{\varepsilon\}, \quad L^{n+1} = L^n L, \quad L^* = \bigcup_{n \geq 0} L^n, \quad L^+ = \bigcup_{n \geq 1} L^n.$$

Each factor may be a *different* word of L.

For $L = \{a, bb\}$, $L^2 = \{aa, abb, bba, bbbb\}$.

$$L\{\varepsilon\} = \{\varepsilon\}L = L, \qquad L\emptyset = \emptyset L = \emptyset, \qquad \emptyset^* = \{\varepsilon\}.$$

A plus does not always exclude the empty word

L^+ means one or more *factors*. If $\varepsilon \in L$, then $\varepsilon \in L^+$ too. The identity $L^+ = L^* \setminus \{\varepsilon\}$ holds when $\varepsilon \notin L$, as for an alphabet's one-symbol words.

Counting Words and the Pigeonhole Principle

If $|\Sigma| = k$, then $|\Sigma^n| = k^n$: each of the n positions has k independent choices. For $n = 0$, there is one word, not zero words.

Binary words

Length three: $000, 001, 010, 011, 100, 101, 110, 111$. Length at most three: $1 + 2 + 4 + 8 = 15$ words, including ε.

Counting Words and the Pigeonhole Principle

If $|\Sigma| = k$, then $|\Sigma^n| = k^n$: each of the n positions has k independent choices. For $n = 0$, there is one word, not zero words.

Binary words

Length three: $000, 001, 010, 011, 100, 101, 110, 111$. Length at most three: $1 + 2 + 4 + 8 = 15$ words, including ε.

Pigeonhole principle

Placing more than r objects into r boxes forces some box to contain at least two objects. Otherwise the total could not exceed r.

Counting Words and the Pigeonhole Principle

If $|\Sigma| = k$, then $|\Sigma^n| = k^n$: each of the n positions has k independent choices. For $n = 0$, there is one word, not zero words.

Binary words

Length three: $000, 001, 010, 011, 100, 101, 110, 111$. Length at most three: $1 + 2 + 4 + 8 = 15$ words, including ε.

Pigeonhole principle

Placing more than r objects into r boxes forces some box to contain at least two objects. Otherwise the total could not exceed r.

A run that visits $r + 1$ states in a machine with only r states repeats a state. A DFA reading r symbols has $r + 1$ visits, counting its start. This does not itself imply acceptance; it identifies repeated behavior that later proofs can exploit.

Problems, Instances, Solutions, and Algorithms

A **computational problem** specifies allowed inputs and the outputs considered correct. An **instance** is one particular input. A **solution to an instance** is a correct output, not the procedure used to obtain it.

Sorting as a running example

Problem: given a finite list of integers, return the same elements, with the same multiplicities, in nondecreasing order.

Instance: $(3, 1, 3, 2)$. **Solution:** $(1, 2, 3, 3)$. Removing a repeated 3 would give an incorrect solution.

Problems, Instances, Solutions, and Algorithms

A **computational problem** specifies allowed inputs and the outputs considered correct. An **instance** is one particular input. A **solution to an instance** is a correct output, not the procedure used to obtain it.

Sorting as a running example

Problem: given a finite list of integers, return the same elements, with the same multiplicities, in nondecreasing order.

Instance: $(3, 1, 3, 2)$. **Solution:** $(1, 2, 3, 3)$. Removing a repeated 3 would give an incorrect solution.

A specification is not an algorithm

With one specified answer per input, the problem is a function $f : I \to O$: it states *what* output belongs to each input. An **algorithm computing** f is a finite, effective procedure that halts with output $f(x)$ for every permitted input x. It states *how* to obtain the answer.

Some tasks allow several outputs (for example, any valid route). Specify the allowed answers, or fix a selection rule to obtain a function. Defining an input–output requirement does not guarantee an algorithm exists [DKS20].

What Does Encoding an Instance as a String Mean?

An **encoding** is an agreed representation of structured data by symbols from a finite alphabet. Writing $\langle x \rangle$ means "the chosen string representation of the instance x."

What the format must specify

- How numbers, vertices, lists, and other components are written.
- How component boundaries and their order can be recovered.
- Which strings are well-formed and how to decode them.

We choose an effective, unambiguous format. A canonical encoding $\text{enc} : I \to \Sigma^*$ is injective and satisfies $\text{decode}(\text{enc}(x)) = x$.

What Does Encoding an Instance as a String Mean?

An **encoding** is an agreed representation of structured data by symbols from a finite alphabet. Writing $\langle x \rangle$ means "the chosen string representation of the instance x."

What the format must specify

- How numbers, vertices, lists, and other components are written.
- How component boundaries and their order can be recovered.
- Which strings are well-formed and how to decode them.

We choose an effective, unambiguous format. A canonical encoding $\text{enc} : I \to \Sigma^*$ is injective and satisfies $\text{decode}(\text{enc}(x)) = x$.

Example: the number 13 can be represented in binary as `1101`. The number is the object; the four-symbol word is its encoding. Encoding need not be encryption or compression.

What Does Encoding an Instance as a String Mean?

An **encoding** is an agreed representation of structured data by symbols from a finite alphabet. Writing $\langle x \rangle$ means "the chosen string representation of the instance x."

What the format must specify

- How numbers, vertices, lists, and other components are written.
- How component boundaries and their order can be recovered.
- Which strings are well-formed and how to decode them.

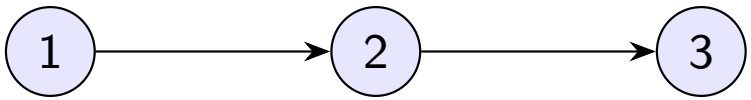
We choose an effective, unambiguous format. A canonical encoding $\text{enc} : I \to \Sigma^*$ is injective and satisfies $\text{decode}(\text{enc}(x)) = x$.

Example: the number 13 can be represented in binary as 1101. The number is the object; the four-symbol word is its encoding. Encoding need not be encryption or compression.

The **input length** is $m = |\langle x \rangle|$, not necessarily the numerical value of x or the number of objects in it. Algorithms read representations; a decoder connects those strings to their meaning.

Worked Encoding: A Graph and a Reachability Question

An instance is (G, s, t): a labeled directed graph and two designated vertices. Does a directed path lead from s to t?



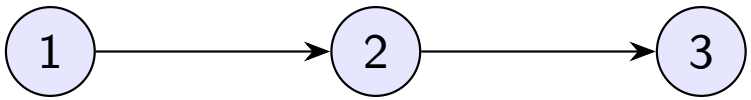
$n = 3,\ s = 1,\ t = 3$

$A_{ij} = 1$ iff $i \to j$ is an edge:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

Worked Encoding: A Graph and a Reachability Question

An instance is (G, s, t): a labeled directed graph and two designated vertices. Does a directed path lead from s to t?



$n = 3,\ s = 1,\ t = 3$

$A_{ij} = 1$ iff $i \to j$ is an edge:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

Use alphabet $\{0, 1, \#\}$ and the format

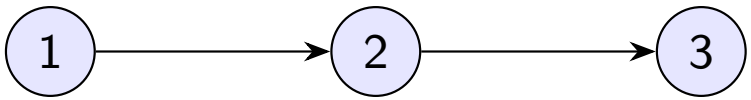
$$\langle G, s, t \rangle = \text{bin}(n)\#\text{rows}(A)\#\text{bin}(s)\#\text{bin}(t).$$

Here vertices are $1, \ldots, n$; positive integers use binary without leading zeros; rows(A) lists exactly n^2 bits row by row.

11#010001000#1#11

Worked Encoding: A Graph and a Reachability Question

An instance is (G, s, t): a labeled directed graph and two designated vertices. Does a directed path lead from s to t?



$n = 3,\ s = 1,\ t = 3$

$A_{ij} = 1$ iff $i \to j$ is an edge:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

Use alphabet $\{0, 1, \#\}$ and the format

$$\langle G, s, t \rangle = \text{bin}(n)\#\text{rows}(A)\#\text{bin}(s)\#\text{bin}(t).$$

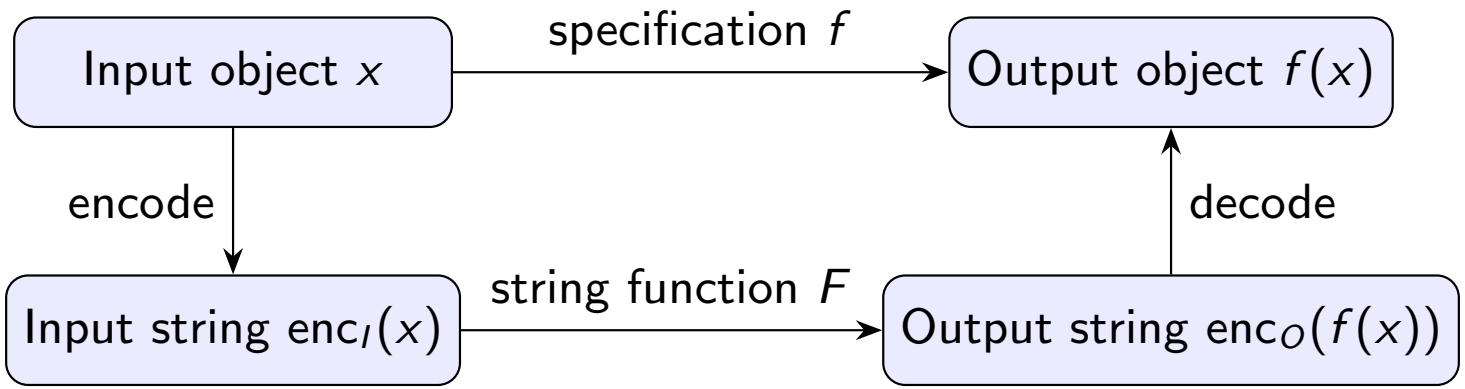
Here vertices are $1, \ldots, n$; positive integers use binary without leading zeros; rows(A) lists exactly n^2 bits row by row.

`11#010001000#1#11`

Decoding recovers the graph and endpoints, not the answer. The answer is `YES`, with path $(1, 2, 3)$. A valid encoding with endpoints reversed is a *no-instance*; it is not a malformed string.

Computation: Encoded Input to Encoded Output

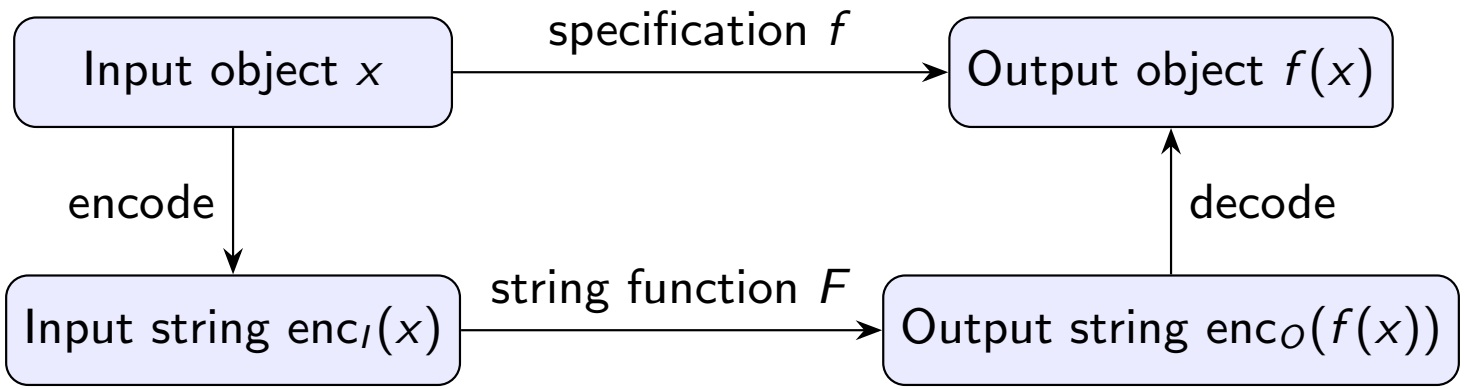
Suppose $f : I \to O$ specifies the desired result. Choose effective, unambiguous encodings for *both* inputs and outputs. The machine's task is the corresponding string transformation F:



$$F(\text{enc}_I(x)) = \text{enc}_O(f(x)).$$

Computation: Encoded Input to Encoded Output

Suppose $f : I \to O$ specifies the desired result. Choose effective, unambiguous encodings for *both* inputs and outputs. The machine's task is the corresponding string transformation F:

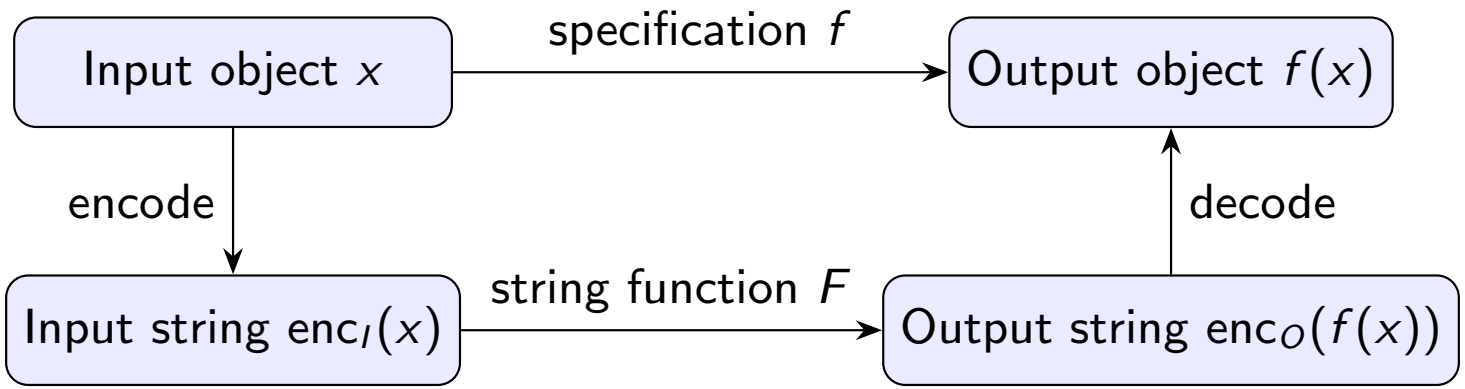


$$F(\text{enc}_I(x)) = \text{enc}_O(f(x)).$$

A Turing machine **computes** this transformation if, on each valid input encoding, it halts with the required output encoding in a designated tape region. Turing machines can produce outputs, not only accept/reject [C19].

Computation: Encoded Input to Encoded Output

Suppose $f : I \to O$ specifies the desired result. Choose effective, unambiguous encodings for *both* inputs and outputs. The machine's task is the corresponding string transformation F:



$$F(\mathrm{enc}_I(x)) = \mathrm{enc}_O(f(x)).$$

A Turing machine **computes** this transformation if, on each valid input encoding, it halts with the required output encoding in a designated tape region. Turing machines can produce outputs, not only accept/reject [C19].

This viewpoint covers tasks with finite, effectively representable data. It does *not* assert that every specified F is computable, or that arbitrary infinite objects have exact finite encodings.

Why This Viewpoint Belongs in a CSE Curriculum

Lists, graphs, documents, and programs look different to an application, but their finite representations let us study them as string-processing tasks.

Three questions beyond implementing one algorithm

- **Specification:** which input–output behavior is required? Correctness compares an implementation with that requirement.

Why This Viewpoint Belongs in a CSE Curriculum

Lists, graphs, documents, and programs look different to an application, but their finite representations let us study them as string-processing tasks.

Three questions beyond implementing one algorithm

- **Specification:** which input–output behavior is required? Correctness compares an implementation with that requirement.
- **Computability:** can *any* algorithm meet it on every valid input? Encoding a task does not answer this question.

Why This Viewpoint Belongs in a CSE Curriculum

Lists, graphs, documents, and programs look different to an application, but their finite representations let us study them as string-processing tasks.

Three questions beyond implementing one algorithm

- **Specification:** which input–output behavior is required? Correctness compares an implementation with that requirement.
- **Computability:** can *any* algorithm meet it on every valid input? Encoding a task does not answer this question.
- **Complexity and models:** what resources are needed, and how does the chosen model affect the analysis?

Why This Viewpoint Belongs in a CSE Curriculum

Lists, graphs, documents, and programs look different to an application, but their finite representations let us study them as string-processing tasks.

Three questions beyond implementing one algorithm

- **Specification:** which input–output behavior is required? Correctness compares an implementation with that requirement.
- **Computability:** can *any* algorithm meet it on every valid input? Encoding a task does not answer this question.
- **Complexity and models:** what resources are needed, and how does the chosen model affect the analysis?

One sorting function, many implementations

Insertion sort and merge sort can implement the same mapping from lists to sorted lists. The specification stays fixed while the method and resource use change. ToC also asks what constraints apply to *all* possible implementations, not only to these two.

Studying strings is therefore an abstraction of computational tasks, not a restriction of CSE to text editing.

Decision Problems as Languages

A **decision problem** asks only for `YES` or `NO`. Encode its instances using a specified format. Its **language of yes-instances** is

$$L = \{w \in \Sigma^* : w \text{ encodes an instance whose answer is yes}\}.$$

Fix the encoding and how malformed inputs are handled (here, reject them). A general problem such as sorting produces an output object, not just a membership bit; decision languages capture its yes/no variants. Equivalently, a decider computes the characteristic function $\chi_L(w) = 1$ when $w \in L$, and $\chi_L(w) = 0$ otherwise.

Decision Problems as Languages

A **decision problem** asks only for `YES` or `NO`. Encode its instances using a specified format. Its **language of yes-instances** is

$$L = \{w \in \Sigma^* : w \text{ encodes an instance whose answer is yes}\}.$$

Fix the encoding and how malformed inputs are handled (here, reject them). A general problem such as sorting produces an output object, not just a membership bit; decision languages capture its yes/no variants. Equivalently, a decider computes the characteristic function $\chi_L(w) = 1$ when $w \in L$, and $\chi_L(w) = 0$ otherwise.

Examples

Binary strings with an odd number of 1's form one language. Encoded graphs having a path from a designated s to t form another. Asking for membership in these sets expresses the corresponding decision problems; the sets alone do not supply algorithms.

Decision Problems as Languages

A **decision problem** asks only for `YES` or `NO`. Encode its instances using a specified format. Its **language of yes-instances** is

$$L = \{w \in \Sigma^* : w \text{ encodes an instance whose answer is yes}\}.$$

Fix the encoding and how malformed inputs are handled (here, reject them). A general problem such as sorting produces an output object, not just a membership bit; decision languages capture its yes/no variants. Equivalently, a decider computes the characteristic function $\chi_L(w) = 1$ when $w \in L$, and $\chi_L(w) = 0$ otherwise.

Examples

Binary strings with an odd number of 1's form one language. Encoded graphs having a path from a designated s to t form another. Asking for membership in these sets expresses the corresponding decision problems; the sets alone do not supply algorithms.

A relevant compiler connection

Lexing and parsing check aspects of a source program's form. A compiler also performs semantic checks and translates the program; its entire job is not just a syntax-membership test. Valid syntax does not by itself guarantee correct behavior.

Part 6: Grammars as Generators

A **grammar** specifies how to generate words. We preview a *context-free grammar* (CFG); its full study comes in Slide Set 5.

Part 6: Grammars as Generators

A **grammar** specifies how to generate words. We preview a *context-free grammar* (CFG); its full study comes in Slide Set 5.

Grammar Quadruple

A CFG is a 4-tuple $G = (V, T, S, P)$:

- V: Finite set of **Variables** (Non-terminals).
- T: Finite set of **Terminals** (symbols from the alphabet). $V \cap T = \emptyset$.
- $S \in V$: The **Start Variable**.
- P: Finite set of **productions** $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

Part 6: Grammars as Generators

A **grammar** specifies how to generate words. We preview a *context-free grammar* (CFG); its full study comes in Slide Set 5.

Grammar Quadruple

A CFG is a 4-tuple $G = (V, T, S, P)$:

- V: Finite set of **Variables** (Non-terminals).
- T: Finite set of **Terminals** (symbols from the alphabet). $V \cap T = \emptyset$.
- $S \in V$: The **Start Variable**.
- P: Finite set of **productions** $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

If $A \to \alpha \in P$, one step replaces uAv by $u\alpha v$: $uAv \Rightarrow u\alpha v$. The notation $\Rightarrow^*$ allows zero or more steps. A generated word contains only terminals:

$$L(G) = \{w \in T^* : S \Rightarrow^* w\}.$$

The rule arrow $\to$ and derivation arrow $\Rightarrow$ have different roles. More general grammars may replace longer left-hand sides; the one-variable restriction here is what makes the grammar context-free.

A Grammar Example and Its Correctness

Use $V = \{S\}$, $T = \{a, b\}$ and productions $S \to aSb \mid \varepsilon$. The bar abbreviates two alternatives. A derivation is

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

The intermediate *aaSbb* is a *sentential form*, not a terminal word.

A Grammar Example and Its Correctness

Use $V = \{S\}$, $T = \{a, b\}$ and productions $S \to aSb \mid \varepsilon$. The bar abbreviates two alternatives. A derivation is

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

The intermediate *aaSbb* is a *sentential form*, not a terminal word.

Claim: $L(G) = \{a^n b^n : n \geq 0\}$

Soundness. After k uses of $S \to aSb$, the form is $a^k S b^k$. Any terminating derivation then uses $S \to \varepsilon$, yielding $a^k b^k$.

Completeness. For any $n \geq 0$, use the recursive rule exactly n times and the empty rule once. This generates $a^n b^n$.

A Grammar Example and Its Correctness

Use $V = \{S\}$, $T = \{a, b\}$ and productions $S \to aSb \mid \varepsilon$. The bar abbreviates two alternatives. A derivation is

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

The intermediate *aaSbb* is a *sentential form*, not a terminal word.

Claim: $L(G) = \{a^n b^n : n \geq 0\}$

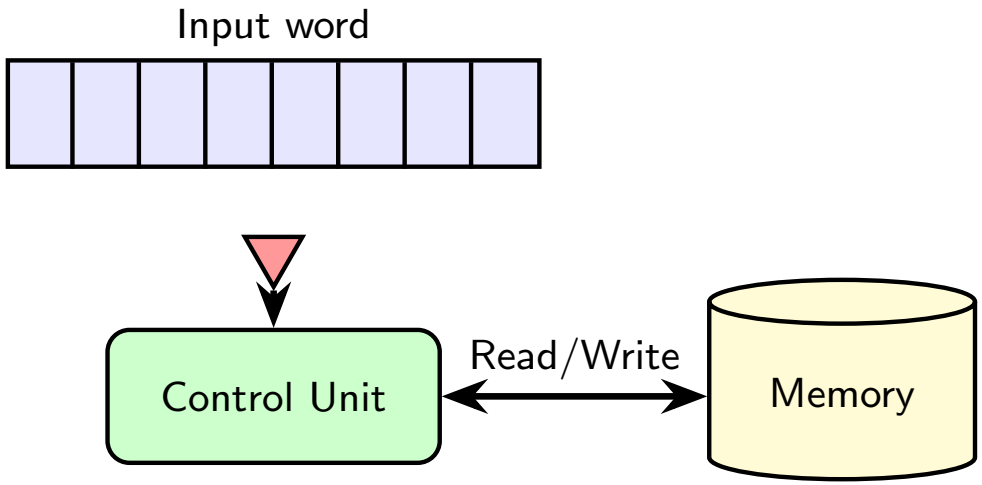
Soundness. After k uses of $S \to aSb$, the form is $a^k S b^k$. Any terminating derivation then uses $S \to \varepsilon$, yielding $a^k b^k$.

Completeness. For any $n \geq 0$, use the recursive rule exactly n times and the empty rule once. This generates $a^n b^n$.

The case $n = 0$ gives $S \Rightarrow \varepsilon$. Examples suggest the pattern; the two inclusions prove it for every word. Without the terminating rule, this grammar would generate no terminal words.

Automata as Recognizers

An **automaton** processes an input according to precise rules. Its model determines the available memory and allowed operations.



- **Finite automata:** finite-state memory only; no separate unbounded work storage.
- **Pushdown automata:** finite control plus a stack (last in, first out).
- **Turing machines:** finite control plus an unbounded read/write tape; the head moves locally, not by random access.

The diagram is schematic. In a standard one-tape Turing machine, the input is initially on the work tape itself.

Several Models, Different Views of Computation

A **model of computation** fixes the machine's configurations, allowed steps, and input/output conventions. No single model serves every explanatory purpose.

Finite automata (DFA/NFA): finite-state control; study what can be recognized with only finite memory.

Pushdown automata (PDA): finite control plus a stack; model nested structure and stack-based recognition.

Turing machines (TM): control plus an unbounded tape whose head moves locally; a simple general-purpose model for computability proofs.

Several Models, Different Views of Computation

A **model of computation** fixes the machine's configurations, allowed steps, and input/output conventions. No single model serves every explanatory purpose.

Finite automata (DFA/NFA): finite-state control; study what can be recognized with only finite memory.

Pushdown automata (PDA): finite control plus a stack; model nested structure and stack-based recognition.

Turing machines (TM): control plus an unbounded tape whose head moves locally; a simple general-purpose model for computability proofs.

Random Access Machine (RAM): the programming viewpoint

A program uses registers, addressed memory, arithmetic, and branches. "Random access" means access by address, not randomness. A word-RAM specifies word size and primitive-operation costs, making it useful for ordinary algorithm analysis [M].

Several Models, Different Views of Computation

A **model of computation** fixes the machine's configurations, allowed steps, and input/output conventions. No single model serves every explanatory purpose.

Finite automata (DFA/NFA): finite-state control; study what can be recognized with only finite memory.

Pushdown automata (PDA): finite control plus a stack; model nested structure and stack-based recognition.

Turing machines (TM): control plus an unbounded tape whose head moves locally; a simple general-purpose model for computability proofs.

Random Access Machine (RAM): the programming viewpoint

A program uses registers, addressed memory, arithmetic, and branches. "Random access" means access by address, not randomness. A word-RAM specifies word size and primitive-operation costs, making it useful for ordinary algorithm analysis [M].

RAM is a model *alongside* TM, not the hidden basis of every ToC model. Standard effective RAM and TM models simulate each other; runtime comparisons require explicit cost assumptions [B].

Other Models You Will Encounter in CSE

Lambda calculus: computation by function abstraction, application, and substitution; a foundation for functional programming.

Other Models You Will Encounter in CSE

Lambda calculus: computation by function abstraction, application, and substitution; a foundation for functional programming.

Boolean circuits: networks of logic gates transform input bits into output bits; connect computation to digital hardware. One fixed circuit has fixed input and output sizes; varying input lengths require a circuit family [BC].

Other Models You Will Encounter in CSE

Lambda calculus: computation by function abstraction, application, and substitution; a foundation for functional programming.

Boolean circuits: networks of logic gates transform input bits into output bits; connect computation to digital hardware. One fixed circuit has fixed input and output sizes; varying input lengths require a circuit family [BC].

Cellular automata: cells repeatedly update their states using local neighbors; model parallel, spatial computation. Some rule systems can simulate a Turing machine, but not every rule system can do so [BM].

Other Models You Will Encounter in CSE

Lambda calculus: computation by function abstraction, application, and substitution; a foundation for functional programming.

Boolean circuits: networks of logic gates transform input bits into output bits; connect computation to digital hardware. One fixed circuit has fixed input and output sizes; varying input lengths require a circuit family [BC].

Cellular automata: cells repeatedly update their states using local neighbors; model parallel, spatial computation. Some rule systems can simulate a Turing machine, but not every rule system can do so [BM].

Different presentations do not always mean different power

The untyped lambda calculus and standard general-purpose TM/RAM models describe the same computable functions, under effective encodings [BM]. Finite automata and one-stack PDAs are more restricted. A single fixed Boolean circuit is not a general-purpose machine for arbitrarily long inputs.

Model equivalence concerns what can be computed; it does not automatically equate time, memory, or implementation convenience.

Recognizing Is Not Always Deciding

Two different requirements for a language L

A **decider** halts on every input, accepting exactly the words in L and rejecting all others.

A **recognizer** accepts every word in L. On a word outside L, it never accepts, but may reject or run forever.

Recognizing Is Not Always Deciding

Two different requirements for a language L

A **decider** halts on every input, accepting exactly the words in L and rejecting all others.

A **recognizer** accepts every word in L. On a word outside L, it never accepts, but may reject or run forever.

- Every decider is a recognizer; a recognizer need not be a decider.
- A DFA always finishes after reading its finite input, so it decides its language. General Turing-machine recognizers may not halt.
- A grammar generates words; a machine tests input words. These are different descriptions, and equivalence requires a theorem.

Recognizing Is Not Always Deciding

Two different requirements for a language L

A **decider** halts on every input, accepting exactly the words in L and rejecting all others.

A **recognizer** accepts every word in L. On a word outside L, it never accepts, but may reject or run forever.

- Every decider is a recognizer; a recognizer need not be a decider.
- A DFA always finishes after reading its finite input, so it decides its language. General Turing-machine recognizers may not halt.
- A grammar generates words; a machine tests input words. These are different descriptions, and equivalence requires a theorem.

Road ahead: finite automata and regular grammars describe the same languages; pushdown automata and CFGs describe the same languages. Neither statement says that every language has such a description.

Part 7: Choosing a Proof Method

- **Direct proof of $P \Rightarrow Q$:** assume P and derive Q.
- **Contrapositive:** instead prove $\neg Q \Rightarrow \neg P$.
- **Contradiction:** assume the claim is false and derive an impossibility. For $P \Rightarrow Q$, assume P and $\neg Q$.
- **Equality of sets or languages:** prove both inclusions.
- **Induction:** establish a base case and a step that covers every larger case.

Part 7: Choosing a Proof Method

- **Direct proof of $P \Rightarrow Q$:** assume P and derive Q.
- **Contrapositive:** instead prove $\neg Q \Rightarrow \neg P$.
- **Contradiction:** assume the claim is false and derive an impossibility. For $P \Rightarrow Q$, assume P and $\neg Q$.
- **Equality of sets or languages:** prove both inclusions.
- **Induction:** establish a base case and a step that covers every larger case.

Contrapositive: if n^2 is even, then n is even

If n is odd, write $n = 2k + 1$ for some integer k. Then $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$ is odd. This proves the contrapositive.

Part 7: Choosing a Proof Method

- **Direct proof of $P \Rightarrow Q$:** assume P and derive Q.
- **Contrapositive:** instead prove $\neg Q \Rightarrow \neg P$.
- **Contradiction:** assume the claim is false and derive an impossibility. For $P \Rightarrow Q$, assume P and $\neg Q$.
- **Equality of sets or languages:** prove both inclusions.
- **Induction:** establish a base case and a step that covers every larger case.

Contrapositive: if n^2 is even, then n is even

If n is odd, write $n = 2k + 1$ for some integer k. Then $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$ is odd. This proves the contrapositive.

State the claim and assumptions before calculating. A diagram or a few test inputs may guide a proof, but do not settle an infinite family of cases.

Proof by Contradiction

In a proof by contradiction, we assume the statement is *false*, and show that this assumption leads to a logical impossibility.

Proof by Contradiction

In a proof by contradiction, we assume the statement is *false*, and show that this assumption leads to a logical impossibility.

Theorem: $\sqrt{2}$ is irrational

Proof: Assume for contradiction that $\sqrt{2}$ is rational.

1. Write $\sqrt{2} = m/n$ with positive integers m, n in lowest terms: $\gcd(m, n) = 1$.
2. Squaring both sides: $2 = \frac{m^2}{n^2} \implies 2n^2 = m^2$.
3. By the parity fact just proved, m is even. Write $m = 2k$.

Proof by Contradiction

In a proof by contradiction, we assume the statement is *false*, and show that this assumption leads to a logical impossibility.

Theorem: $\sqrt{2}$ is irrational

Proof: Assume for contradiction that $\sqrt{2}$ is rational.

1. Write $\sqrt{2} = m/n$ with positive integers m, n in lowest terms: $\gcd(m, n) = 1$.
2. Squaring both sides: $2 = \frac{m^2}{n^2} \implies 2n^2 = m^2$.
3. By the parity fact just proved, m is even. Write $m = 2k$.
4. Substitute back: $2n^2 = (2k)^2 = 4k^2 \implies n^2 = 2k^2$.
5. This means n^2 is even, so n must be even.
6. If m and n are both even, they share a common factor of 2. This contradicts our assumption! Therefore, $\sqrt{2}$ must be irrational. ■

Proof by Induction

To prove a property $P(n)$ holds for all integers $n \geq 0$:

- **Basis:** Prove $P(0)$ is true.
- **Inductive step:** for an arbitrary $k \geq 0$, assume $P(k)$ and prove $P(k+1)$. Do not assume $P(k+1)$ while proving it.

Proof by Induction

To prove a property $P(n)$ holds for all integers $n \geq 0$:

- **Basis:** Prove $P(0)$ is true.
- **Inductive step:** for an arbitrary $k \geq 0$, assume $P(k)$ and prove $P(k+1)$. Do not assume $P(k+1)$ while proving it.

Theorem: a binary tree of height at most n has at most 2^n leaves

Use height measured in edges and at most two children per node.

- **Base:** height at most 0 means a single root, with one leaf, and $1 = 2^0$.
- **Hypothesis:** every binary tree of height at most k has at most 2^k leaves.

Proof by Induction

To prove a property $P(n)$ holds for all integers $n \geq 0$:

- **Basis:** Prove $P(0)$ is true.
- **Inductive step:** for an arbitrary $k \geq 0$, assume $P(k)$ and prove $P(k+1)$. Do not assume $P(k+1)$ while proving it.

Theorem: a binary tree of height at most n has at most 2^n leaves

Use height measured in edges and at most two children per node.

- **Base:** height at most 0 means a single root, with one leaf, and $1 = 2^0$.
- **Hypothesis:** every binary tree of height at most k has at most 2^k leaves.
- **Step:** a tree of height at most $k+1$ is either a single leaf or has one or two child subtrees, each of height at most k. Their leaves total at most $2 \cdot 2^k = 2^{k+1}$. The single-leaf case also satisfies the bound.

The phrase "at most" lets the hypothesis apply to unequal-height subtrees. Equality is attained when every internal node has two children and every leaf is at depth n.

Structural Induction: A String Identity

Finite strings are built from ε by repeatedly appending one symbol. Proving a base case and an append-symbol case is induction on length.

Example: $(xy)^R = y^R x^R$ for all strings x, y

Fix an arbitrary x and induct on y.

Base: $y = \varepsilon$ gives $(x\varepsilon)^R = x^R = \varepsilon^R x^R$.

Structural Induction: A String Identity

Finite strings are built from ε by repeatedly appending one symbol. Proving a base case and an append-symbol case is induction on length.

Example: $(xy)^R = y^R x^R$ for all strings x, y

Fix an arbitrary x and induct on y.

Base: $y = \varepsilon$ gives $(x\varepsilon)^R = x^R = \varepsilon^R x^R$.

Step: let $y = za$, where a is one symbol, and assume $(xz)^R = z^R x^R$. By the definition $(wa)^R = aw^R$,

$$(xza)^R = a(xz)^R = az^R x^R = (za)^R x^R.$$

Thus the claim holds for every y, and x was arbitrary too.

Structural Induction: A String Identity

Finite strings are built from ε by repeatedly appending one symbol. Proving a base case and an append-symbol case is induction on length.

Example: $(xy)^R = y^R x^R$ for all strings x, y

Fix an arbitrary x and induct on y.

Base: $y = \varepsilon$ gives $(x\varepsilon)^R = x^R = \varepsilon^R x^R$.

Step: let $y = za$, where a is one symbol, and assume $(xz)^R = z^R x^R$. By the definition $(wa)^R = aw^R$,

$$(xza)^R = a(xz)^R = az^R x^R = (za)^R x^R.$$

Thus the claim holds for every y, and x was arbitrary too.

The same pattern proves properties of recursively built expressions or derivation trees: cover every base constructor and every recursive constructor.

Checkpoint Exercises 1: Sets, Relations, and Logic

1. **Sets:** Let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$.
   What is $(A \cup B) - (A \cap B)$?

2. **Relations:** A relation R on $\mathbb{Z}$ is defined as xRy if $x \leq y$.
   Which equivalence property does this relation *fail*? (Reflexivity, Symmetry, or Transitivity?)

3. **Boolean Logic:** Evaluate the following expression, given $P = 1, Q = 0, R = 1$:

$$\neg P \vee Q \wedge R$$

Checkpoint Exercises 2: Strings and Grammars

4. **Strings:** Let $\Sigma = \{0, 1\}$. How many strings are in Σ^* of length exactly 3? List them.

5. **Grammars:** Consider the grammar G:

$$S \to aA$$
$$A \to bS \mid \varepsilon$$

Show the derivation sequence for the string *ababa*. What is the general language $L(G)$ described in English?

Checkpoint Exercises 3: Proofs and Graphs

6. **Graphs:** A finite simple undirected graph has v vertices and e edges. The degree of a vertex is the number of incident edges. Prove directly that the sum of all vertex degrees is $2e$.

7. **Induction:** We want to prove by induction that $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$. State explicitly what the **Basis** and the **Inductive Hypothesis** are for this proof.

Checkpoint Exercises 4: Boundaries and Quantifiers

8. Negate $\forall x \in D\ \exists y \in D\ R(x, y)$, showing both quantifiers. Explain why choosing a single y in advance is not generally allowed in the original statement.

Checkpoint Exercises 4: Boundaries and Quantifiers

8. Negate $\forall x \in D\ \exists y \in D\ R(x, y)$, showing both quantifiers. Explain why choosing a single y in advance is not generally allowed in the original statement.
9. For $L = \{\varepsilon, a\}$, compute L^2, and decide whether $L^+ = L^* \setminus \{\varepsilon\}$. Explain rather than just answer yes/no.

Checkpoint Exercises 4: Boundaries and Quantifiers

8. Negate $\forall x \in D\ \exists y \in D\ R(x, y)$, showing both quantifiers. Explain why choosing a single y in advance is not generally allowed in the original statement.
9. For $L = \{\varepsilon, a\}$, compute L^2, and decide whether $L^+ = L^* \setminus \{\varepsilon\}$. Explain rather than just answer yes/no.
10. A rooted tree has a root with two children; the left child is a leaf and the right child has two leaf children. Find the tree's height, its number of leaves, and the depth and height of the left child.

Self-check before submitting a proof

Have you fixed the universe or alphabet? Included the empty case? Proved the stated direction? Used a hypothesis strong enough for every recursive subcase?

Summary and Bridge to Finite Automata

- Sets ignore order; tuples and strings do not. A language is a set of finite strings, not a single string.
- Functions have specified domains and codomains. Equivalence relations partition a set into classes.
- Quantifier order determines what choices may depend on what; negation reverses universal and existential quantifiers.
- Graphs describe connections; rooted trees describe hierarchy. Depth and height count edges in different directions.
- Grammars generate words; recognizers accept words; deciders additionally halt on every input.
- A problem specifies correct outputs; an encoding represents its instances. Runtime depends on input length and the chosen machine model.
- Proofs need explicit assumptions, all required directions, and boundary cases—not just convincing examples.

Summary and Bridge to Finite Automata

- Sets ignore order; tuples and strings do not. A language is a set of finite strings, not a single string.
- Functions have specified domains and codomains. Equivalence relations partition a set into classes.
- Quantifier order determines what choices may depend on what; negation reverses universal and existential quantifiers.
- Graphs describe connections; rooted trees describe hierarchy. Depth and height count edges in different directions.
- Grammars generate words; recognizers accept words; deciders additionally halt on every input.
- A problem specifies correct outputs; an encoding represents its instances. Runtime depends on input length and the chosen machine model.
- Proofs need explicit assumptions, all required directions, and boundary cases—not just convincing examples.

Next: Slide Set 2

A finite automaton combines sets, functions, and a labeled directed graph. Its states remember input-prefix information; induction justifies this meaning.

Solutions 1–3: Sets, Relations, and Logic

1. $A \cup B = \{1, 2, 3, 4, 5\}$ and $A \cap B = \{3\}$, so $(A \cup B) \setminus (A \cap B) = \{1, 2, 4, 5\}$. This operation is the *symmetric difference*: membership in exactly one set.

Solutions 1–3: Sets, Relations, and Logic

1. $A \cup B = \{1, 2, 3, 4, 5\}$ and $A \cap B = \{3\}$, so $(A \cup B) \setminus (A \cap B) = \{1, 2, 4, 5\}$. This operation is the *symmetric difference*: membership in exactly one set.
2. The relation $x \leq y$ on $\mathbb{Z}$ is reflexive and transitive, but not symmetric: $3 \leq 4$ while $4 \not\leq 3$. Thus it is not an equivalence relation. A relation may satisfy some properties without satisfying all of them.

Solutions 1–3: Sets, Relations, and Logic

1. $A \cup B = \{1, 2, 3, 4, 5\}$ and $A \cap B = \{3\}$, so $(A \cup B) \setminus (A \cap B) = \{1, 2, 4, 5\}$. This operation is the *symmetric difference*: membership in exactly one set.
2. The relation $x \leq y$ on $\mathbb{Z}$ is reflexive and transitive, but not symmetric: $3 \leq 4$ while $4 \not\leq 3$. Thus it is not an equivalence relation. A relation may satisfy some properties without satisfying all of them.
3. For $P = 1$, $Q = 0$, $R = 1$,

$$\neg P \vee (Q \wedge R) = 0 \vee (0 \wedge 1) = 0.$$

The parentheses show the intended precedence explicitly.

Solutions 4–5: Strings and a Grammar

4. There are $2^3 = 8$ length-three binary words: $000, 001, 010, 011, 100, 101, 110, 111$.

Solutions 4–5: Strings and a Grammar

4. There are $2^3 = 8$ length-three binary words: $000, 001, 010, 011, 100, 101, 110, 111$.

5. For $S \to aA$, $A \to bS \mid \varepsilon$, a derivation is

$$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow ababS \Rightarrow ababaA \Rightarrow ababa.$$

The full language, not just one example

$L(G) = \{(ab)^n a : n \geq 0\}$: alternating a's and b's beginning and ending with a.

Each cycle $S \Rightarrow aA \Rightarrow abS$ adds one ab. Termination uses $S \Rightarrow aA \Rightarrow a$, so every terminal derivation has the stated form. Conversely, take n cycles and then terminate to generate any $(ab)^n a$.

Solutions 4–5: Strings and a Grammar

4. There are $2^3 = 8$ length-three binary words: $000, 001, 010, 011, 100, 101, 110, 111$.

5. For $S \to aA$, $A \to bS \mid \varepsilon$, a derivation is

$$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow ababS \Rightarrow ababaA \Rightarrow ababa.$$

The full language, not just one example

$L(G) = \{(ab)^n a : n \geq 0\}$: alternating a's and b's beginning and ending with a.

Each cycle $S \Rightarrow aA \Rightarrow abS$ adds one ab. Termination uses $S \Rightarrow aA \Rightarrow a$, so every terminal derivation has the stated form. Conversely, take n cycles and then terminate to generate any $(ab)^n a$.

The shortest word is a, not ε. Only A can derive ε here; the start variable still forces an a before termination.

Solutions 6–7: Double Counting and Induction

6. Handshaking argument. Count vertex–edge incidences. Summing vertex degrees counts each incidence once. Each edge of a simple undirected graph has two distinct endpoints, so contributes two incidences. Hence $\sum_{v \in V} \deg(v) = 2|E|$.

Solutions 6–7: Double Counting and Induction

6. Handshaking argument. Count vertex–edge incidences. Summing vertex degrees counts each incidence once. Each edge of a simple undirected graph has two distinct endpoints, so contributes two incidences. Hence $\sum_{v \in V} \deg(v) = 2|E|$.

7. Sum of the first n positive integers, $n \geq 1$

Base: $n = 1$ gives $1 = 1(1+1)/2$.

Hypothesis: for an arbitrary $k \geq 1$, assume $1 + \cdots + k = k(k+1)/2$.

Step:

$$1 + \cdots + k + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}.$$

This is the formula for $k+1$, completing the induction.

Starting at $n = 0$ also works if the empty sum is defined to be zero; make that convention explicit.

Solutions 8–10: Boundary Cases Matter

8. The negation is $\exists x \in D\, \forall y \in D\, \neg R(x, y)$. In the original statement, a suitable y may depend on x; requiring one common witness changes the assertion.

Solutions 8–10: Boundary Cases Matter

8. The negation is $\exists x \in D\, \forall y \in D\, \neg R(x, y)$. In the original statement, a suitable y may depend on x; requiring one common witness changes the assertion.
9. $L^2 = \{\varepsilon, a, aa\}$. Duplicate ways to form a do not produce duplicate set elements. Both L^+ and L^* equal $\{a^n : n \geq 0\}$, since an empty factor is allowed. Thus L^+ contains ε, whereas $L^* \setminus \{\varepsilon\}$ does not.

Solutions 8–10: Boundary Cases Matter

8. The negation is $\exists x \in D\, \forall y \in D\, \neg R(x, y)$. In the original statement, a suitable y may depend on x; requiring one common witness changes the assertion.
9. $L^2 = \{\varepsilon, a, aa\}$. Duplicate ways to form a do not produce duplicate set elements. Both L^+ and L^* equal $\{a^n : n \geq 0\}$, since an empty factor is allowed. Thus L^+ contains ε, whereas $L^* \setminus \{\varepsilon\}$ does not.
10. The tree has height 2 and three leaves. The left child has depth 1 and height 0. The leaf bound gives $3 \leq 2^2 = 4$; it is an upper bound, not a formula for the exact number of leaves.

One Function, Two Representations: Sorting

Return to the sorting problem. For a simple format, let list entries be single decimal digits. Encode a list using brackets and comma-separated digits; the empty list is `[]`.

Object-level specification

$$f((3,1,3,2)) = (1,2,3,3).$$

The output is nondecreasing and preserves every element's multiplicity.

One Function, Two Representations: Sorting

Return to the sorting problem. For a simple format, let list entries be single decimal digits. Encode a list using brackets and comma-separated digits; the empty list is `[]`.

Object-level specification

$$f((3,1,3,2)) = (1,2,3,3).$$

The output is nondecreasing and preserves every element's multiplicity.

The corresponding string function

$$F(\texttt{[3,1,3,2]}) = \texttt{[1,2,3,3]}, \qquad F(\texttt{[]}) = \texttt{[]}.$$

Decoding the output string gives exactly the required output list. This is the same task expressed at a different level, not a new problem.

One Function, Two Representations: Sorting

Return to the sorting problem. For a simple format, let list entries be single decimal digits. Encode a list using brackets and comma-separated digits; the empty list is `[]`.

Object-level specification

$$f((3,1,3,2)) = (1,2,3,3).$$

The output is nondecreasing and preserves every element's multiplicity.

The corresponding string function

$$F(\texttt{[3,1,3,2]}) = \texttt{[1,2,3,3]}, \qquad F(\texttt{[]}) = \texttt{[]}.$$

Decoding the output string gives exactly the required output list. This is the same task expressed at a different level, not a new problem.

A RAM program may manipulate an array; a Turing machine may manipulate a tape representation. If both meet this specification on every valid list, they compute the same function despite different internal steps. Correctness concerns the input–output relationship, not identical execution.

A Turing Machine Producing an Output: Increment

The numerical task is $f(n) = n + 1$ for $n \in \mathbb{N}$. Use binary, with 0 encoded as 0 and no leading zeros otherwise. The string task is $F(\text{bin}(n)) = \text{bin}(n + 1)$.

$$13 \mapsto 14 \qquad \text{becomes} \qquad 1101 \mapsto 1110.$$

A Turing Machine Producing an Output: Increment

The numerical task is $f(n) = n + 1$ for $n \in \mathbb{N}$. Use binary, with 0 encoded as 0 and no leading zeros otherwise. The string task is $F(\text{bin}(n)) = \text{bin}(n + 1)$.

$$13 \mapsto 14 \qquad \text{becomes} \qquad 1101 \mapsto 1110.$$

A tape-level strategy, not yet a formal transition table

On a tape with blank cells on both sides of the input:

1. Scan right to the first blank, then move left to the last bit.
2. Carry a 1: replace each encountered 1 by 0 and move left.
3. At a 0, write 1 and halt. If the carry reaches the blank before the word, write 1 there and halt instead.

A Turing Machine Producing an Output: Increment

The numerical task is $f(n) = n + 1$ for $n \in \mathbb{N}$. Use binary, with 0 encoded as 0 and no leading zeros otherwise. The string task is $F(\text{bin}(n)) = \text{bin}(n + 1)$.

$$13 \mapsto 14 \qquad \text{becomes} \qquad 1101 \mapsto 1110.$$

A tape-level strategy, not yet a formal transition table

On a tape with blank cells on both sides of the input:

1. Scan right to the first blank, then move left to the last bit.
2. Carry a 1: replace each encountered 1 by 0 and move left.
3. At a 0, write 1 and halt. If the carry reaches the blank before the word, write 1 there and halt instead.

For 1101, the changed tape word goes $1101 \to 1100 \to 1110$. For 111, the final output is 1000. Read the final contiguous block of bits as the output.

The finite scan and carry always terminate on valid input. The machine has *computed a value*, not merely answered a membership question.

The Graph Example: A Bit Answer or a Structured Answer?

Reuse the main-slide instance: edges $1 \to 2 \to 3$, with $s = 1, t = 3$. Its input encoding is

$$\texttt{11\#010001000\#1\#11}.$$

The *output specification* determines which function we want.

Reachability as a decision function

Output `1` if a path exists, and `0` otherwise. For this input, the output string is `1`. This is the language-membership viewpoint.

The Graph Example: A Bit Answer or a Structured Answer?

Reuse the main-slide instance: edges $1 \to 2 \to 3$, with $s = 1, t = 3$. Its input encoding is

$$\texttt{11\#010001000\#1\#11}.$$

The *output specification* determines which function we want.

Reachability as a decision function

Output `1` if a path exists, and `0` otherwise. For this input, the output string is `1`. This is the language-membership viewpoint.

Route production as a string-output function

Specify the shortest path, breaking ties lexicographically by vertex sequence; return `NONE` if there is no path. Encode a path as its vertex numbers in binary, separated by `#`. Here the answer $(1, 2, 3)$ becomes `1#10#11`.

The Graph Example: A Bit Answer or a Structured Answer?

Reuse the main-slide instance: edges $1 \to 2 \to 3$, with $s = 1, t = 3$. Its input encoding is

$$\texttt{11\#010001000\#1\#11}.$$

The *output specification* determines which function we want.

Reachability as a decision function

Output `1` if a path exists, and `0` otherwise. For this input, the output string is `1`. This is the language-membership viewpoint.

Route production as a string-output function

Specify the shortest path, breaking ties lexicographically by vertex sequence; return `NONE` if there is no path. Encode a path as its vertex numbers in binary, separated by `#`. Here the answer $(1, 2, 3)$ becomes `1#10#11`.

Both tasks process the *same input representation*, but compute different outputs. The tie-breaking rule makes the second answer unique. Without it, "return any route" specifies allowed input–output pairs rather than a unique function.

Programs Are Data Too: The Interpreter Perspective

A program has a finite description, so it too can be encoded as a string. An interpreter takes *two pieces of data*: a program description and that program's input, packaged in an unambiguous pair encoding.

$$\langle P, x \rangle \xrightarrow{\text{interpreter or universal machine}} \langle y \rangle \qquad \text{if } P(x) \text{ halts with } y.$$

Programs Are Data Too: The Interpreter Perspective

A program has a finite description, so it too can be encoded as a string. An interpreter takes *two pieces of data*: a program description and that program's input, packaged in an unambiguous pair encoding.

$$\langle P, x \rangle \xrightarrow{\text{interpreter or universal machine}} \langle y \rangle \qquad \text{if } P(x) \text{ halts with } y.$$

A connection to everyday programming

Let P be a program for incrementing a natural number, and let $x = 13$. The interpreter receives their encodings and eventually produces the encoding of 14. Changing P changes the transformation without changing the interpreter itself.

Programs Are Data Too: The Interpreter Perspective

A program has a finite description, so it too can be encoded as a string. An interpreter takes *two pieces of data*: a program description and that program's input, packaged in an unambiguous pair encoding.

$$\langle P, x \rangle \xrightarrow{\text{interpreter or universal machine}} \langle y \rangle \qquad \text{if } P(x) \text{ halts with } y.$$

A connection to everyday programming

Let P be a program for incrementing a natural number, and let $x = 13$. The interpreter receives their encodings and eventually produces the encoding of 14. Changing P changes the transformation without changing the interpreter itself.

Why termination now matters

If $P(x)$ runs forever, a faithful interpreter may do so too. Its behavior is a *partial* string function: some inputs have no returned output. A universal Turing machine formalizes this idea [C17].

This connects strings, programs, simulation between models, and the question of which computational tasks can be solved at all.

Further Reading: Problems, Encodings, and RAM Models

[DKS20] Erik Demaine, Jason Ku, and Justin Solomon, MIT 6.006, Lecture 1: Introduction, Spring 2020. Input–output problem specifications, algorithms, and word-RAM costs.

[M] Pat Morin, *Open Data Structures*, Section 1.4: The Model of Computation. Word size, addressable memory, and primitive operations.

[B] Boaz Barak, *Introduction to Theoretical Computer Science*, Modeling Running Time. Formal RAM/Turing-machine simulations and their cost assumptions.

The list and graph formats illustrate input–output representations; they are teaching examples, not universal serialization standards.

Further Reading: Functions and Alternative Models

[C19] Cornell CS 4820, 2019, *Limits of Computability*. Turing machines computing partial functions from input strings to output strings.

[BC] Boaz Barak, *Introduction to Theoretical Computer Science*, Defining Computation. Functions as specifications and circuits as implementations for fixed input/output sizes.

[BM] Boaz Barak, same text, Equivalent Models of Computation. Lambda calculus, cellular automata, and simulation between models.

[C17] Cornell CS 2800, Spring 2017, Turing Machines. Encoding machines and universal simulation.

The purpose here is to connect computational specifications to CSE practice; formal machine definitions and equivalence proofs come later.

References and Suggested Teaching Use

[S13] Michael Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage, 2013. Chapter 0: mathematical notions, definitions, theorems, and proofs.

[LR23] Peter Linz and Susan H. Rodger, *An Introduction to Formal Languages and Automata*, 7th ed., Jones & Bartlett Learning, 2023. Introductory mathematical preliminaries, languages, grammars, and automata.

[LLM15] Eric Lehman, F. Thomson Leighton, and Albert R. Meyer, *Mathematics for Computer Science*, MIT, 2015. Supplementary practice in logic, proof methods, relations, and graphs.

Use the main sequence as a prerequisite review and the appendix for worked answers and deeper encoding/model examples. These exercises are course practice, not claimed to be numbered textbook problems.