Introduction to the Theory of Computation

Mathematical Foundations and Preliminaries

Theory of Computation Slide Sets

Table of Contents

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 mathematically model computation? We study abstract machines (finite automata, pushdown automata, Turing machines) to define what a "computer" is.

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 mathematically model computation? We study abstract machines (finite automata, pushdown automata, Turing machines) to define what a "computer" is.
- **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 mathematically model computation? We study abstract machines (finite automata, pushdown automata, Turing machines) to define what a "computer" is.
- **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 group of objects represented as a unit. The objects are called **elements** or **members**.

Part 2: Sets and Basics

A **set** is a group of objects represented as a unit. The objects are called **elements** or **members**.

- **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.

Part 2: Sets and Basics

A **set** is a group of objects represented as a unit. The objects are called **elements** or **members**.

- **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.

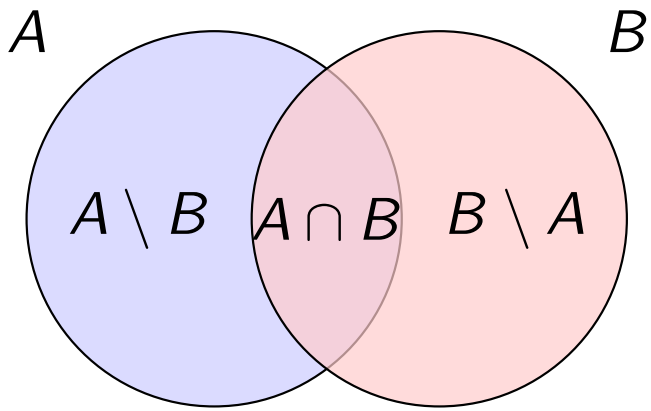
Teacher's Corner: A Common Pitfall

Students frequently confuse the empty set $\emptyset$ with the empty string ε.

- $\emptyset$ is a *set* that contains absolutely nothing: $|\emptyset| = 0$.
- ε is a *string* of length zero.
- The set $\{\varepsilon\}$ is *not* empty! It is a set containing exactly one element (the empty string).

Set Operations and Venn Diagrams

- **Union** ($A \cup B$): Elements in A or B.
- **Intersection** ($A \cap B$): Elements in both.
- **Difference** ($A - B$ or $A \setminus B$): Elements in A but not B.
- **Complement** ($\overline{A}$): Elements not in A (relative to a universal set).



$A \cup B$ is the entire colored region

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:

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

Note: $|2^S| = 2^{|S|}$.

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:

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

Note: $|2^S| = 2^{|S|}$.

Sequences, Tuples, and Cartesian Products

A **sequence** is an ordered list. A finite sequence is a **tuple**. An ordered pair is a 2-tuple.
Cartesian Product ($A \times B$): The set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.
Example: $A = \{1, 2\}$, $B = \{x, y, z\}$

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

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:** Defined for *some* (but not necessarily all) elements in D. In automata theory, transition functions are often partial (e.g., the machine crashes if no transition is defined).

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:** Defined for *some* (but not necessarily all) elements in D. In automata theory, transition functions are often partial (e.g., the machine crashes if no transition is defined).

Example

Let $f : \mathbb{Z} \to \mathbb{Z}$ be $f(x) = |x|$. The domain is $\mathbb{Z}$ and the codomain is $\mathbb{Z}$. The actual **range** is the set of non-negative integers ($\mathbb{Z}^+$), meaning the function is not *onto* the codomain.

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** (fails injective for negative inputs, fails surjective for negative outputs).

Relations and Equivalence

A **relation** is a property whose domain is a set of k-tuples. A binary relation R on a set A is a subset of $A \times A$. We often write aRb instead of $(a, b) \in R$.

Relations and Equivalence

A **relation** is a property whose domain is a set of k-tuples. A binary relation R on a set A is a subset of $A \times A$. We often write aRb instead of $(a, b) \in R$.

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 **relation** is a property whose domain is a set of k-tuples. A binary relation R on a set A is a subset of $A \times A$. We often write aRb instead of $(a, b) \in R$.

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

An equivalence relation partitions a set into disjoint **equivalence classes**. Elements in the same class are deemed "equivalent." (Crucial later for state minimization in automata).

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 $P \vee (Q \wedge (\neg R))$.

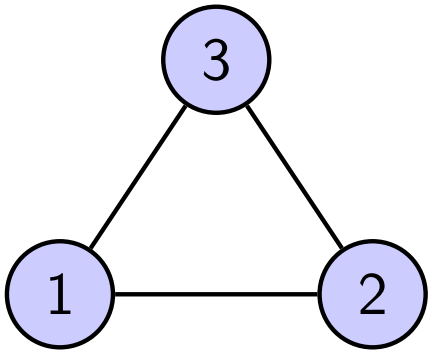
Part 4: Graph Theory Basics

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

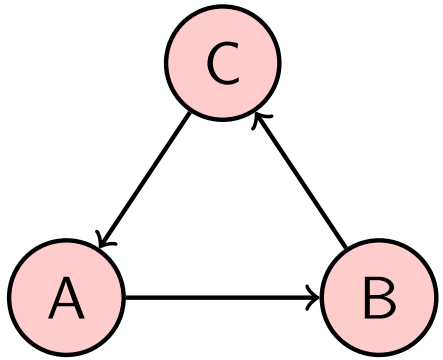
Part 4: Graph Theory Basics

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

Undirected Graph: Edges have no direction. E consists of unordered pairs $\{u, v\}$.



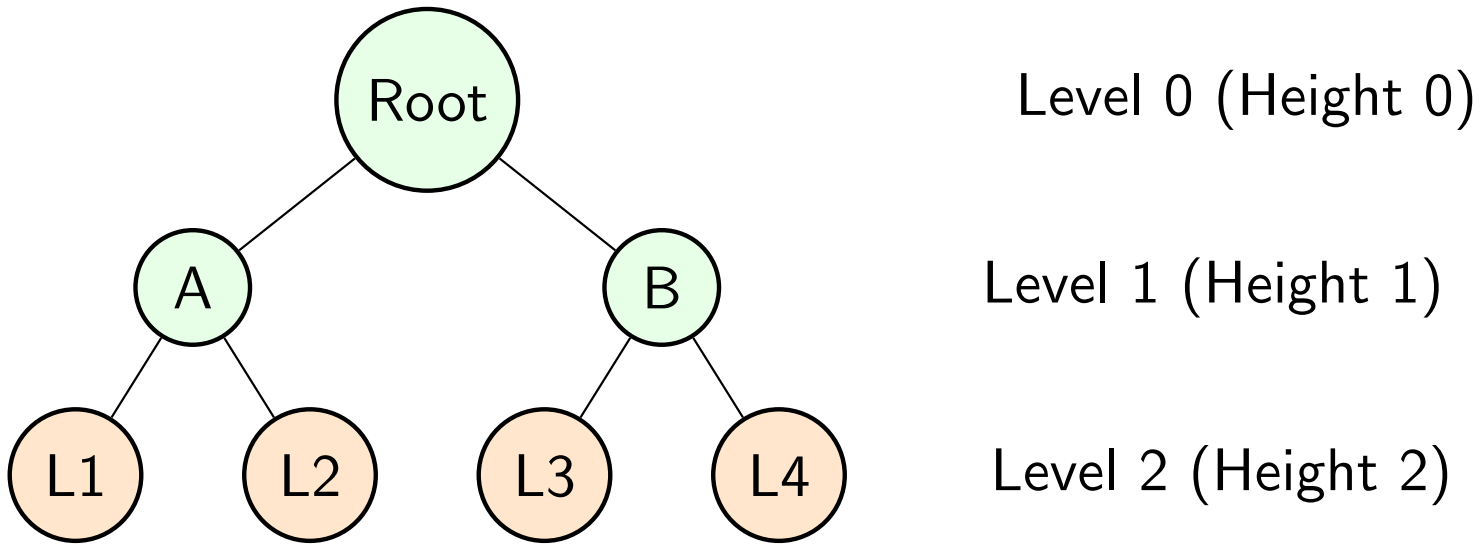
Directed Graph (Digraph): Edges have direction. E consists of ordered pairs (u, v).



A **path** is a sequence of nodes connected by edges. A **cycle** is a path that starts and ends at the same node.

Tree Structures

A **tree** is a connected directed graph with no simple cycles, where one node is designated the **root**, and every other node has exactly one incoming edge (from its **parent**). Nodes with no outgoing edges are **leaves**.



- **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. $(011)^R = 110$.

Languages and the Kleene Star

Kleene Star (Σ^*)

The set of *all possible strings* over an alphabet Σ, including the empty string ε.

Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Languages and the Kleene Star

Kleene Star (Σ^*)

The set of *all possible strings* over an alphabet Σ, including the empty string ε.
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^*)

The set of *all possible strings* over an alphabet Σ, including the empty string ε.
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.

Pedagogical Note

In computer science, a "language" is just a set of strings. The strings don't need to mean anything! A compiler's job is essentially answering the membership question: *Is the string (source code) a member of the language (valid C++ syntax)?*

Part 6: Grammars as Generators

To study languages mathematically, we need mechanisms to describe them. A **grammar** is a generating mechanism.

Part 6: Grammars as Generators

To study languages mathematically, we need mechanisms to describe them. A **grammar** is a generating mechanism.

Grammar Quadruple

A grammar 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** (substitution rules).

Part 6: Grammars as Generators

To study languages mathematically, we need mechanisms to describe them. A **grammar** is a generating mechanism.

Grammar Quadruple

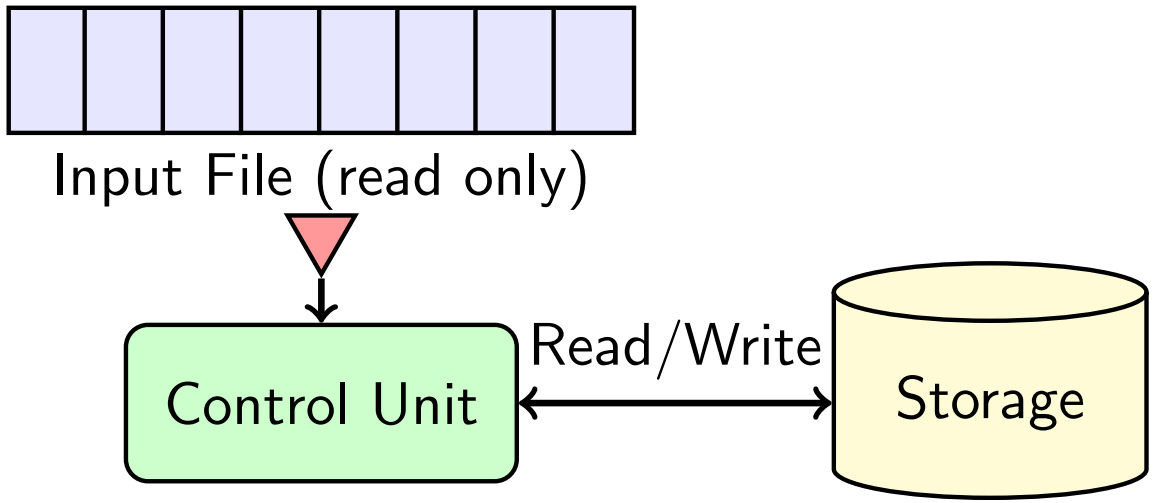
A grammar is a 4-tuple $G = (V, T, S, P)$:

- 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** (substitution rules).

Derivations ($\Rightarrow$): We apply rules from P to replace variables.
Example: $S \to aSb \mid \varepsilon$.
Derivation: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.
The generated language is $L(G) = \{a^n b^n \mid n \geq 0\}$.

Automata as Recognizers

An **automaton** is an abstract model of a digital computer. It processes an input string and decides whether to "Accept" or "Reject".



- **Finite Automata:** No temporary storage.
- **Pushdown Automata:** Storage is a Stack (LIFO).
- **Turing Machines:** Infinite random-access tape.

Part 7: Proof by Contradiction

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

Part 7: 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. Then $\sqrt{2} = \frac{m}{n}$ where m, n are integers with no common factors.
2. Squaring both sides: $2 = \frac{m^2}{n^2} \implies 2n^2 = m^2$.
3. This means m^2 is even, so m must be even. Let $m = 2k$.
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:** Assume $P(k)$ is true (Inductive Hypothesis), and use it to prove $P(k+1)$ is true.

Proof by Induction

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

- **Basis:** Prove $P(0)$ is true.
- **Inductive Step:** Assume $P(k)$ is true (Inductive Hypothesis), and use it to prove $P(k+1)$ is true.

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

Proof: Let $l(n)$ be the max number of leaves at height n.

- **Basis:** For $n = 0$, the tree is just the root. Leaves $= 1$. $2^0 = 1$. True.
- **Inductive Step:** Assume $l(k) \leq 2^k$. A tree of height $k+1$ is formed by taking a tree of height k and adding at most 2 children to each leaf.
- Thus, $l(k+1) \leq 2 \times l(k)$.
- By the hypothesis, $l(k+1) \leq 2 \times 2^k = 2^{k+1}$. ■

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:** An undirected graph has v vertices and e edges. Prove using a direct argument that the sum of the degrees of all vertices is exactly $2e$.

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.

Hints for Checkpoint Exercises

Slide 16 Hints:

- 1: Find the union first, then find the intersection. Remove the intersection elements from the union.
- 2: Try swapping the variables. If $3 \leq 4$, does $4 \leq 3$?
- 3: Remember precedence: NOT precedes AND, which precedes OR.

Slide 17 Hints:

- 4: The number of strings of length n over an alphabet of size k is k^n.
- 5: Substitute S, then A, then S again until you generate *ababa*, finishing with the ε rule to clear the variable.

Slide 18 Hints:

- 6: Every edge connects exactly two nodes. How many times is an edge counted when you sum the degrees?
- 7: The basis is $n = 1$. The inductive hypothesis assumes the formula holds for an arbitrary k.